POLITECNICO DI MILANO

μ-LAB

High Performance Processors and Systems

Exceptions and interrupts

Donatella Sciuto: sciuto@elet.polimi.it

HPPS

Dealing with exceptions

- Harder to handle in a pipelined CPU
  - Overlapping makes more difficult to know whether an instruction can safely change the state of the CPU
- Types of exceptions:
  - I/O device request
  - OS system call
  - Integer arithmetic overflow
  - Memory protection access
  - Page fault
  - Hardware malfunction
  - .....

Exception/Interrupt classifications

- *Exceptions:* relevant to the current process
  - Faults, arithmetic traps, and synchronous traps
  - Invoke software on behalf of the currently executing process
- *Interrupts:* caused by asynchronous, outside events
  - I/O devices requiring service (DISK, network)
  - Clock interrupts (real time scheduling)
- *Machine Checks:* caused by serious hardware failure
  - Not always restartable
  - Indicate that bad things have happened.
    - Non-recoverable ECC error
    - Machine room fire
    - Power outage

A related classification: Synchronous vs. Asynchronous

- *Synchronous:* means related to the instruction stream, i.e. during the execution of an instruction
  - Must stop an instruction that is currently executing
  - Page fault on load or store instruction
  - Arithmetic exception
  - Software Trap Instructions
- *Asynchronous:* means unrelated to the instruction stream, i.e. caused by an outside event.
  - Does not have to disrupt instructions that are already executing
  - Interrupts are asynchronous
  - Machine checks are asynchronous
- *SemiSynchronous (or high-availability interrupts):*
  - Caused by external event but may have to disrupt current instructions in order to guarantee service

Classes of exceptions

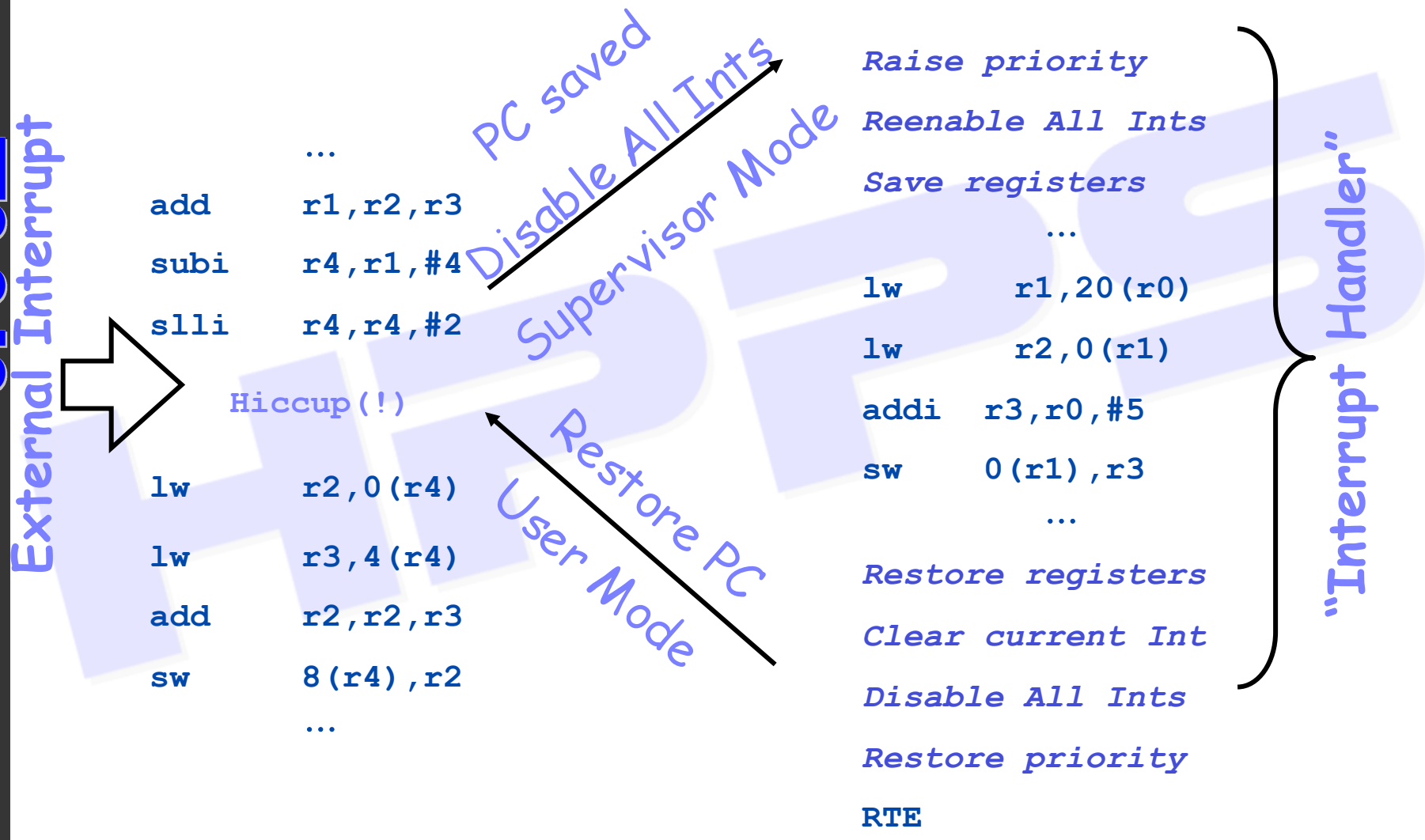
- Synchronous vs asynchronous
  - Asynchronous events are caused by devices external to the CPU and memory and can be handled after the completion of the current instruction (easier to handle)
- User requested vs coerced
  - User requested are predictable: treated as exceptions because they use the same 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

Example: Device Interrupt

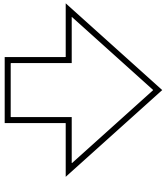
(Say, arrival of network message)



Alternative: Polling

(again, for arrival of network message)

External Interrupt



```
        Disable Network Intr
        ...
        subi    r4,r1,#4
        slli    r4,r4,#2
        lw      r2,0(r4)
        lw      r3,4(r4)
        add     r2,r2,r3
        sw      8(r4),r2
        lw      r1,12(r0)
        beq     r1,no_mess
        lw      r1,20(r0)
        lw      r2,0(r1)
        addi    r3,r0,#5
        sw      0(r1),r3
        Clear Network Intr
no_mess:        ...
```

Polling Point

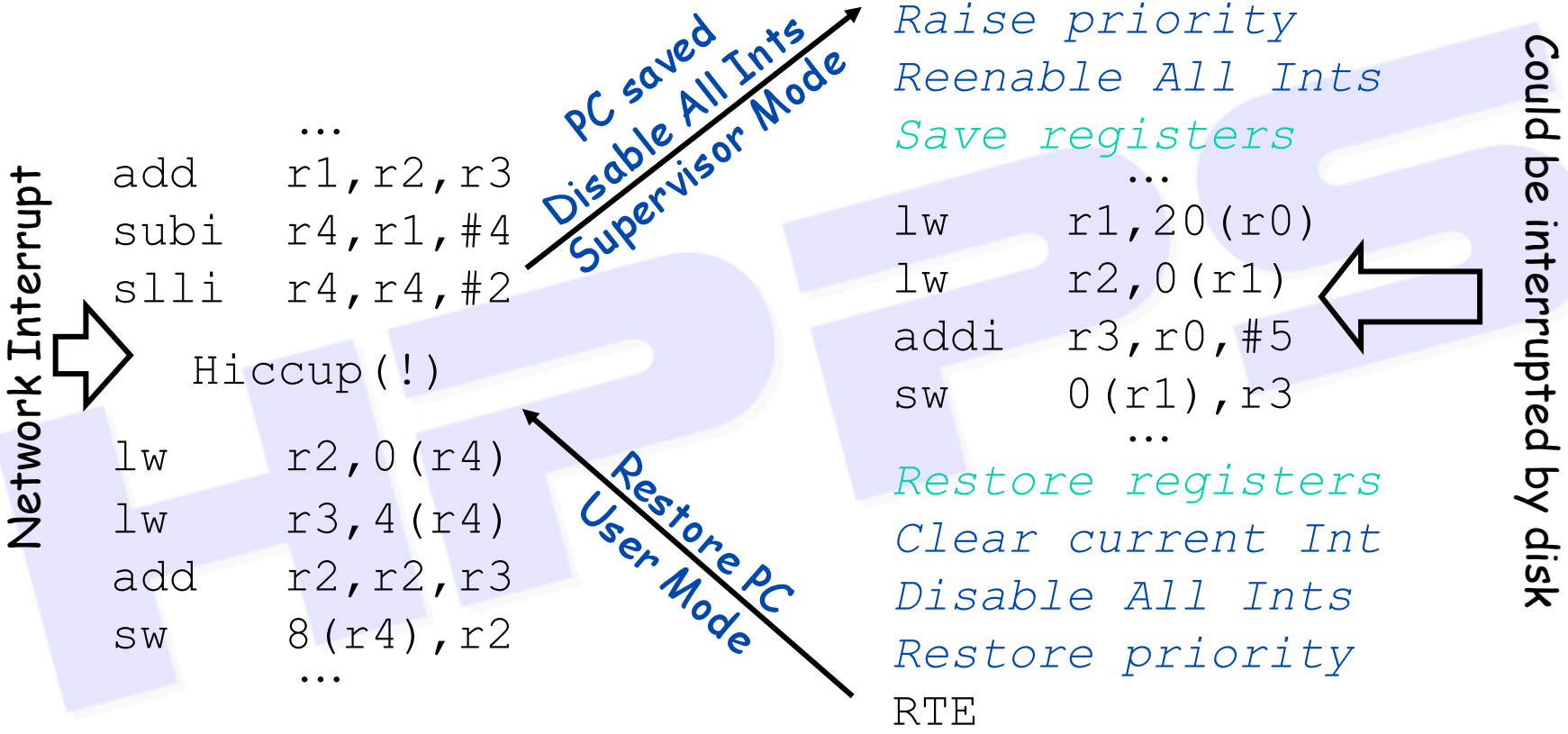
(check device register)

"Handler"

Polling is faster/slower than Interrupts.

- Polling is faster than interrupts because
  - Compiler knows which registers in use at polling point. Hence, do not need to save and restore registers (or not as many).
  - Other interrupt overhead avoided (pipeline flush, trap priorities, etc).
- Polling is slower than interrupts because
  - Overhead of polling instructions is incurred regardless of whether or not handler is run. This could add to inner-loop delay.
  - Device may have to wait for service for a long time.
- When to use one or the other?
  - Multi-axis tradeoff
    - Frequent/regular events good for polling, **as long as device can be controlled at user level.**
    - Interrupts good for infrequent/irregular events
    - Interrupts good for ensuring regular/predictable service of events.

Interrupt Priorities Must be Handled



Note that priority must be raised to avoid recursive interrupts!

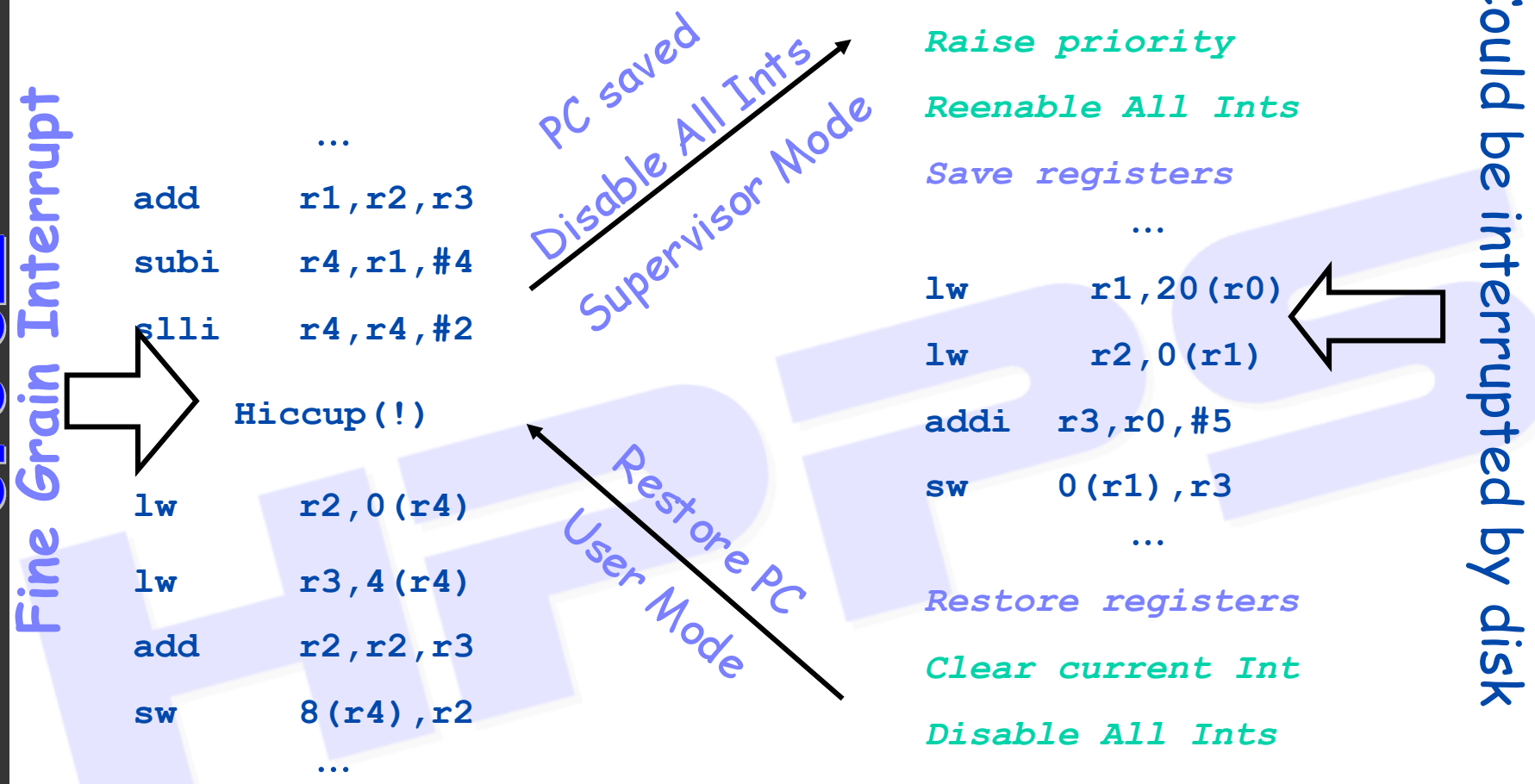
Interrupt controller hardware and mask levels

- Interrupt disable mask may be multi-bit word accessed through some special memory address
- Operating system constructs a hierarchy of masks that reflects some form of interrupt priority.
- For instance:

Priority	Examples
0	Software interrupts
2	Network Interrupts
4	Sound card
5	Disk Interrupt
6	Real Time clock

  - This reflects the an order of urgency to interrupts
  - For instance, this ordering says that disk events can interrupt the interrupt handlers for network interrupts.

Can we have fast interrupts?



- Pipeline Drain: Can be very Expensive
- Priority Manipulations
- Register Save/Restore
  - 128 registers + cache misses + etc.

SPARC (and RISC I) had register windows

- On interrupt or procedure call, simply switch to a different set of registers
- Really saves on interrupt overhead
  - Interrupts can happen at any point in the execution, so compiler cannot help with knowledge of live registers.
  - Conservative handlers must save all registers
  - Short handlers might be able to save only a few, but this analysis is complicated
- Not as big a deal with procedure calls
  - Original statement by Patterson was that Berkeley didn't have a compiler team, so they used a hardware solution
  - Good compilers can allocate registers across procedure boundaries
  - Good compilers know what registers are live at any one time

Supervisor State

- Typically, processors have some amount of state that user programs are not allowed to touch.
  - Page mapping hardware/TLB
    - TLB prevents one user from accessing memory of another
    - TLB protection prevents user from modifying mappings
  - Interrupt controllers -- User code prevented from crashing machine by disabling interrupts. Ignoring device interrupts, etc.
  - Real-time clock interrupts ensure that users cannot lockup/crash machine even if they run code that goes into a loop:
    - "Preemptive Multitasking" vs "non-preemptive multitasking"
- Access to hardware devices restricted
  - Prevents malicious user from stealing network packets
  - Prevents user from writing over disk blocks
- Distinction made with at least two-levels: USER/SYSTEM (one hardware mode-bit)
  - x86 architectures actually provide 4 different levels, only two usually used by OS (or only 1 in older Microsoft OSs)

Entry into Supervisor Mode

- Entry into supervisor mode typically happens on interrupts, exceptions, and special trap instructions.
- Entry goes through kernel instructions:
  - interrupts, exceptions, and trap instructions change to supervisor mode, then jump (indirectly) through table of instructions in kernel

```
intvec:  j      handle_int0
         j      handle_int1
                …
         j      handle_fp_except0
                …
         j      handle_trap0
         j      handle_trap1
```

Entry into Supervisor Mode

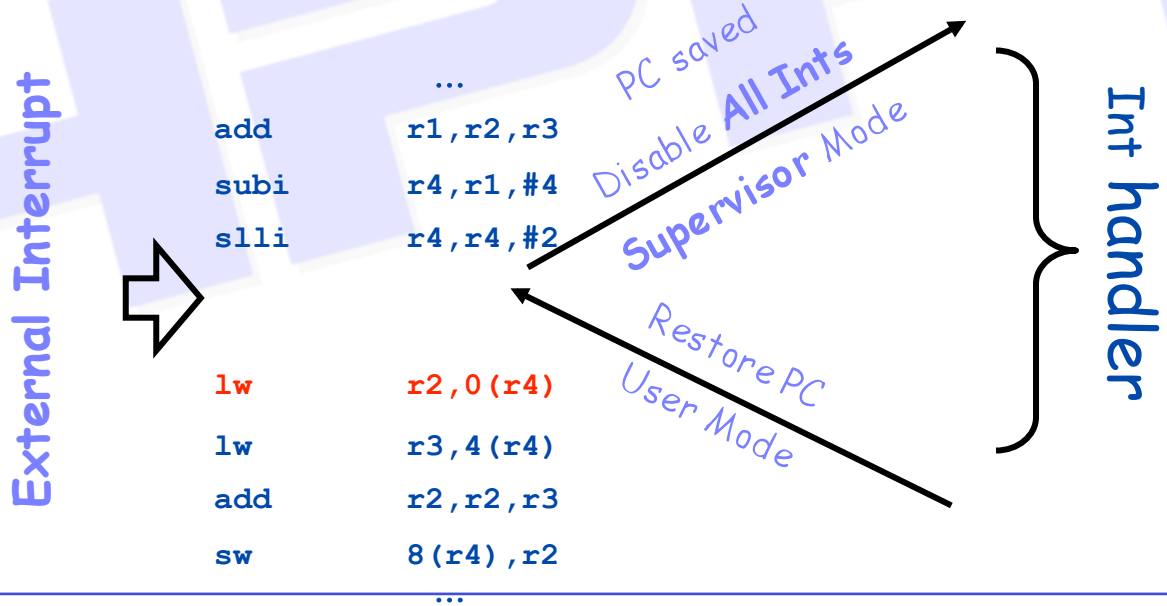
- OS “System Calls” are just trap instructions:

```
read(fd,buffer,count)  ⇨  st     20(r0),r1
                          st     24(r0),r2
                          st     28(r0),r3
                          trap    $READ
```

- OS overhead can be serious concern for achieving fast interrupt behavior.

Precise Interrupts/Exceptions

- 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



Precise interrupt point requires multiple PCs to describe in presence of delayed branches

```
        addi   r4,r3,#4
        subaddi      r4,r3,#4
        sub    r1,r2,r3
PC:     bne    r1,there
PC+4:   and    r2,r3,r5
        <other insts>
               r1,r2,r3
PC:     bne    r1,there
PC+4:   and    r2,r3,r5
        <other insts>
```

⇦ Interrupt point described as <PC,PC+4>

Interrupt point described as:

⇦

<PC+4,there> (branch was taken)
or
<PC+4,PC+8> (branch was not taken)

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: $f(x) = \frac{\sin(x)}{x}$

Then, for $x \longrightarrow 0$

$$f(0) = \frac{0}{0} \Rightarrow NaN + illegal\_operation$$

Want to take exception, replace **NaN** with 1, then restart.

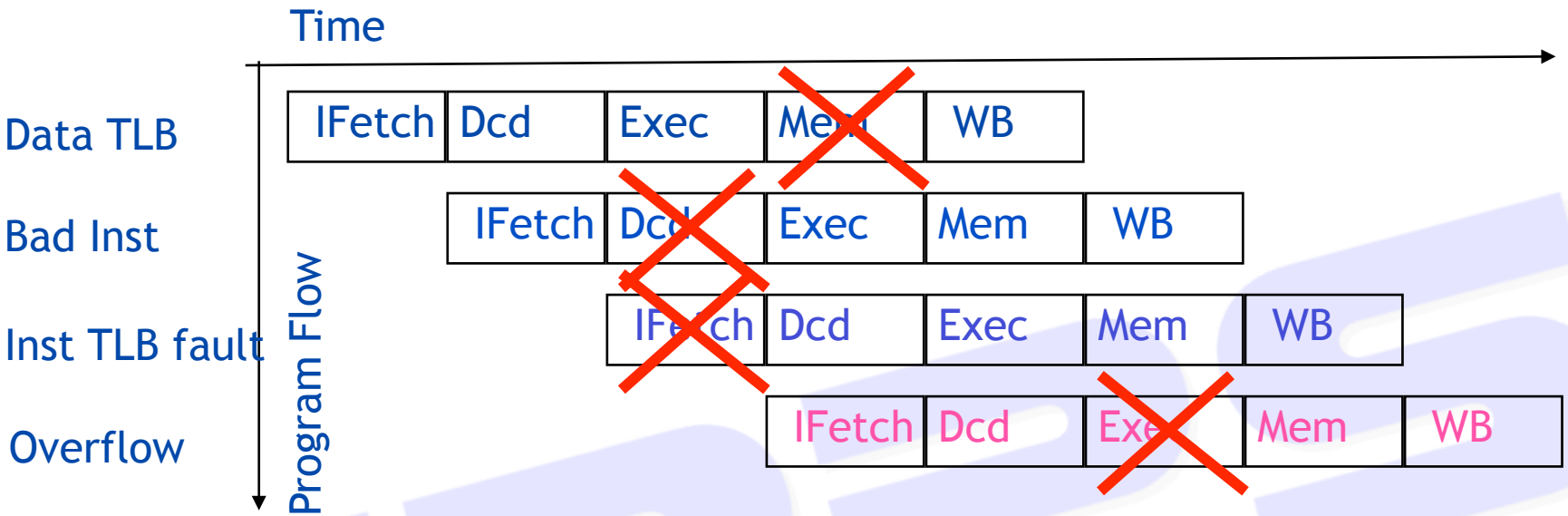
Why are precise interrupts desirable?

- 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)

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

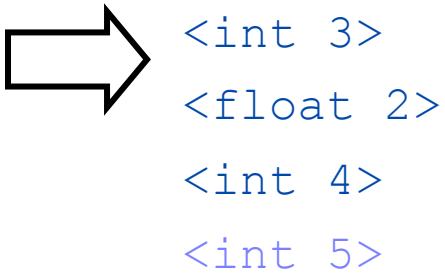
Another look at the exception problem



- 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.

How to achieve precise interrupts when instructions executing in arbitrary order?

- Jim Smith's classic paper (will read next time) 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.

Review: Summary of Pipelining Basics

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
- Today: Longer pipelines (R4000) => Better branch prediction, more instruction parallelism?

Summary

- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with *precise exceptions* provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  - Important enabling factor for out-of-order execution