

Exception handling

Donatella Sciuto: donatella.sciuto@polimi.it

Types of Exceptions

- We use the term 'exception' to cover not only exceptions but also interrupts and faults. More in general, we consider the following type of events:
 - I/O device request;
 - Invoking OS system call from a user program;
 - Tracing instruction execution;
 - Integer arithmetic overflow/underflow;
 - Floating point arithmetic anomaly;
 - Page fault;
 - Misaligned memory access;
 - Memory protection violation;
 - Hardware / power failure.

Causes of Interrupts / Exceptions

Interrupt: an *event* that requests the attention of the processor

- ***Asynchronous:*** an *external event, such as:*
 - input/output device service-request
 - timer expiration
 - power disruptions, hardware failure
- ***Synchronous:*** an *internal event (a.k.a. exceptions)*
 - undefined opcode, privileged instruction
 - Integer arithmetic overflow, FPU exception
 - misaligned memory access
 - *virtual memory exceptions:* page faults, TLB misses, protection violations
 - *traps:* system calls, e.g., jumps into kernel

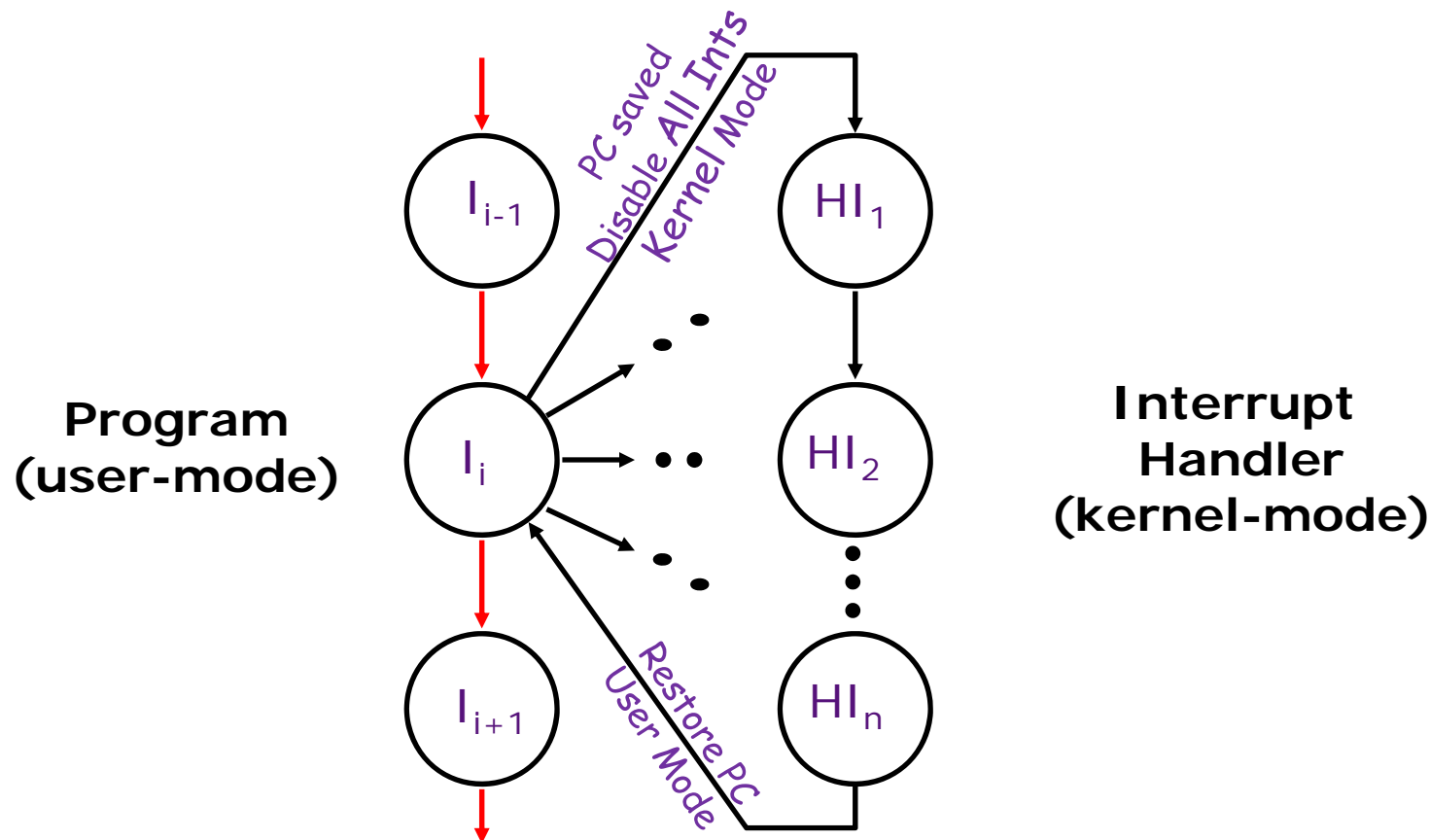
Classes of exceptions

- 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* (such as I/O events) 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 (easier to handle)
 - *Coerced* are caused by some HW event not under control of the user program; hard to implement because they are **unpredictable**;
- User maskable vs user nonmaskable
 - The mask simply controls whether the HW responds to the exception or not

Classes of exceptions (cont'd)

- 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 such as HW malfunctions and always cause program termination.
- Resume vs terminate
 - Terminating event: program's execution always stops after the interrupt/exception;
 - Resuming event: program's execution continues after the interrupt/exception;

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.

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} (*precise interrupt*)
 - It saves the PC of instruction I_i in a special register Exception Program Counter: **PC -> EPC**
 - It disables interrupts and transfers control to a designated interrupt handler running in the **kernel mode**:
Int. Vector Address -> PC

Interrupt Handler

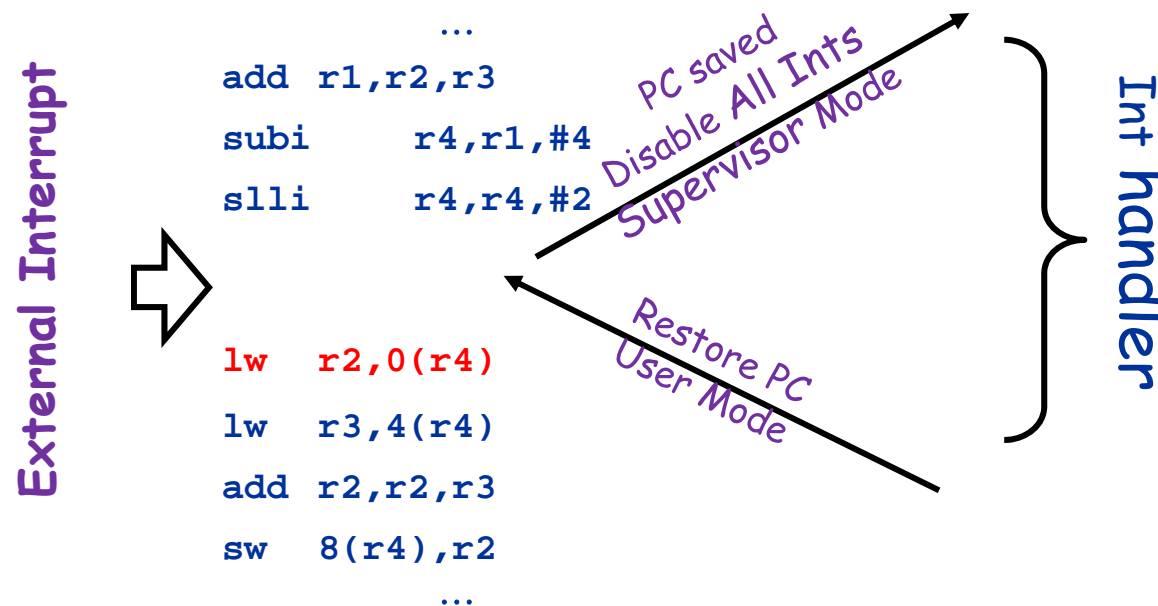
- To allow nested interrupts, we need to save PC before enabling interrupts \Rightarrow
 - need an instruction to move PC into GPRs
 - need a way to mask further interrupts at least until PC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) which **restore the PC** and:
 - enables interrupts
 - restores the processor to the **user mode**
 - restores hardware status and control state
- The instruction *li* and the next instructions (*li+1, ...*) are restarted

Synchronous Interrupts

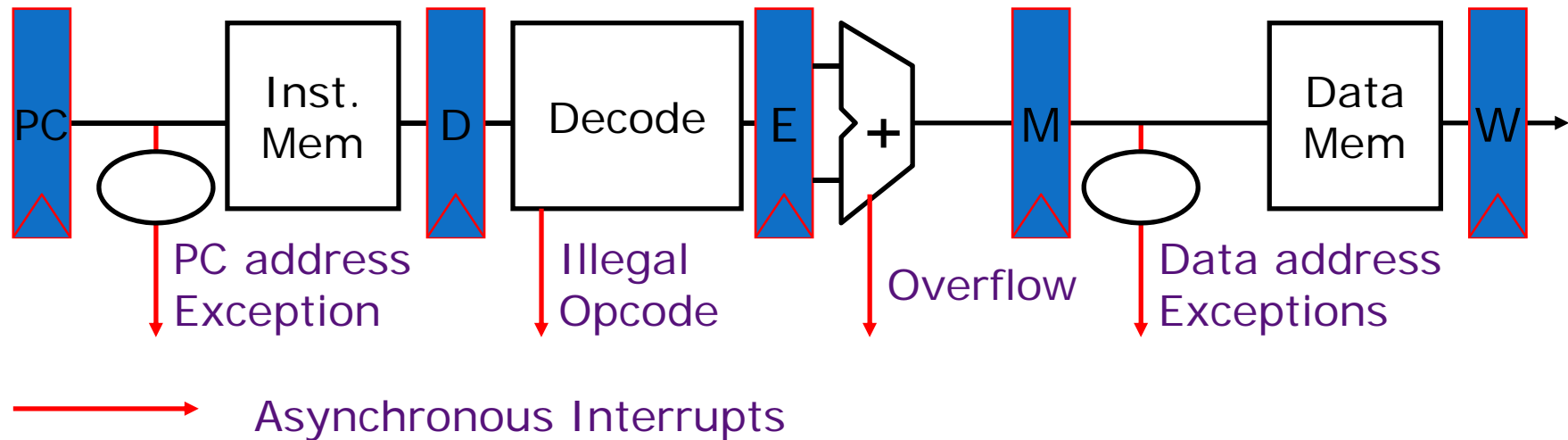
- A synchronous interrupt (exception) is caused by a *particular instruction*
- In general, the instruction *li* cannot be completed and needs to be *restarted* after the exception has been handled
 - In the pipeline this would require undoing the effect of one or more partially executed instructions

Precise Interrupts/Exceptions

- An interrupt or exception is *precise* if there is a single instruction (or interrupt point) for which all instructions before have committed their state and no following instructions (including the interrupting instruction *li*) have modified any state.
 - This means, effectively, that we 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 **red lw** instruction *li*)



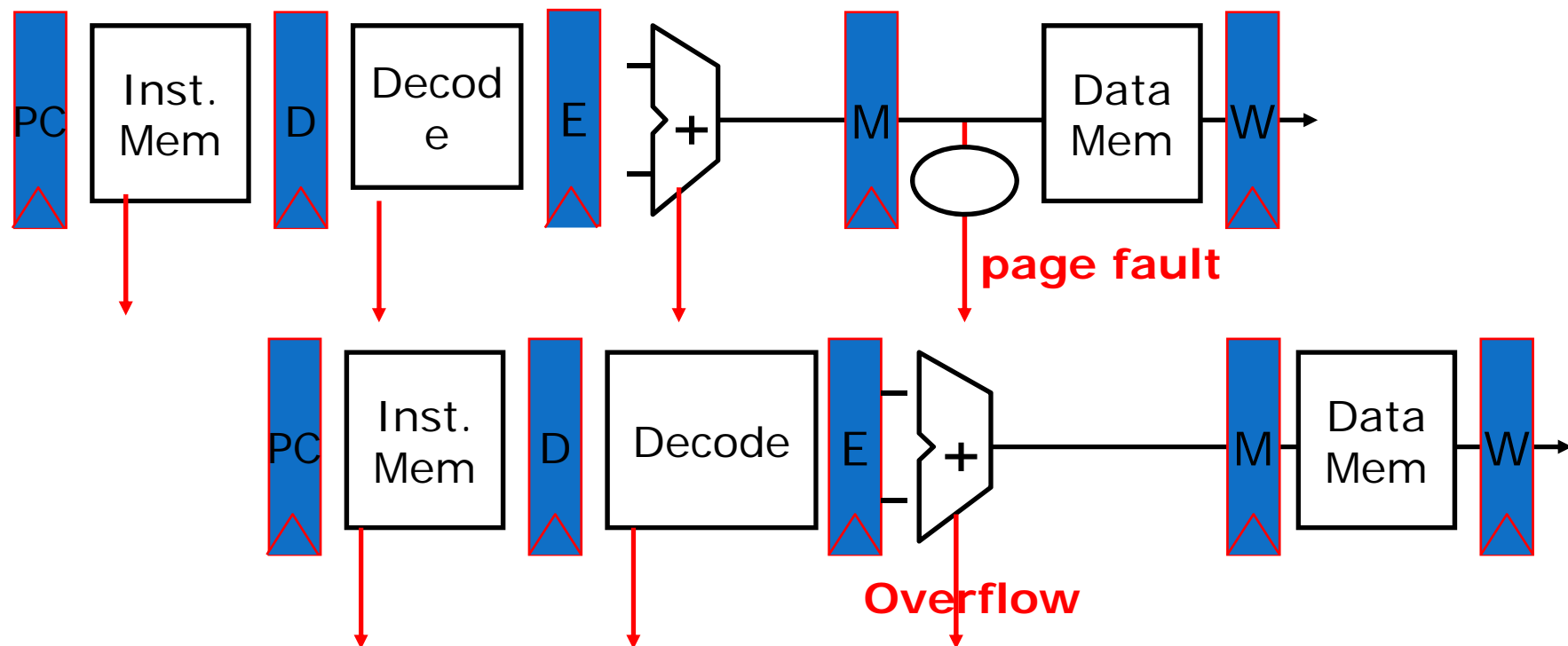
Exception Handling: 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

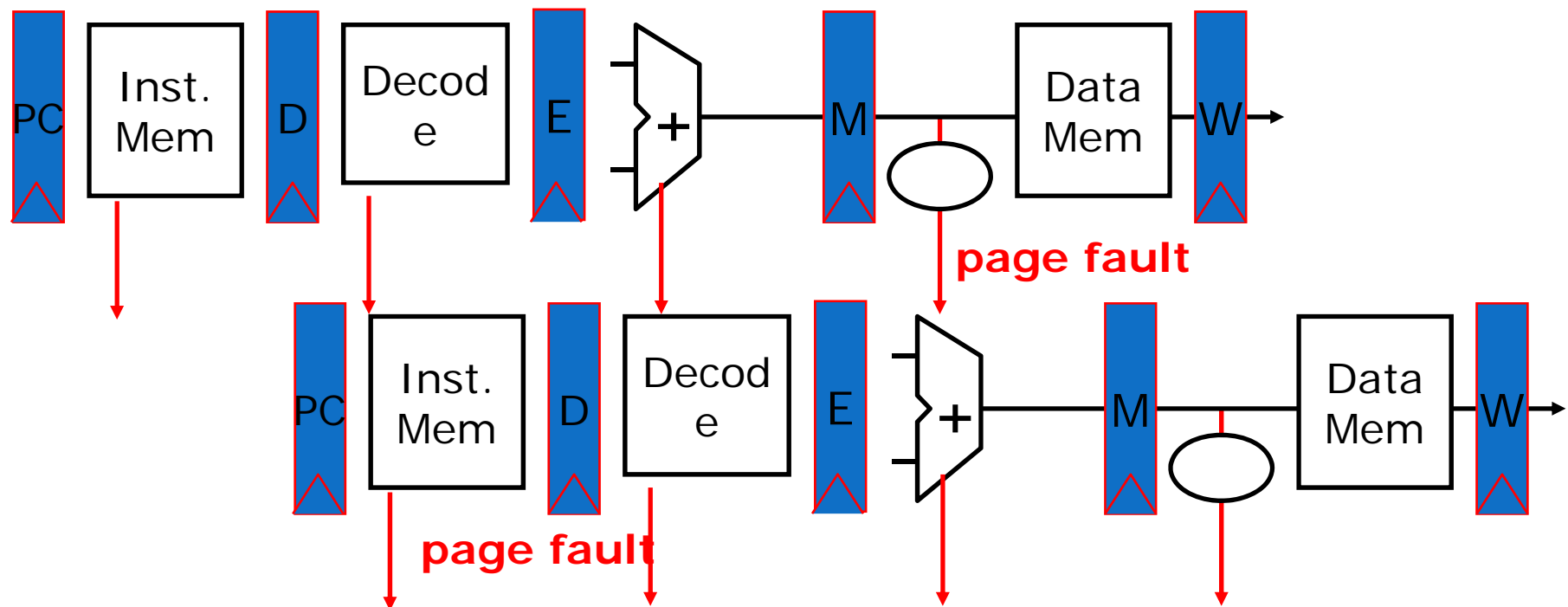
Precise Exceptions in simple 5-stage pipeline

- Exceptions may occur at different stages in pipeline
-> Exceptions may be raised *out of order*
- Let's consider this **first** example:
 - Data page fault occurs in memory stage of first instruction
 - Arithmetic exception occurs in execution stage of second instruction
- Data page fault is handled first: OK!

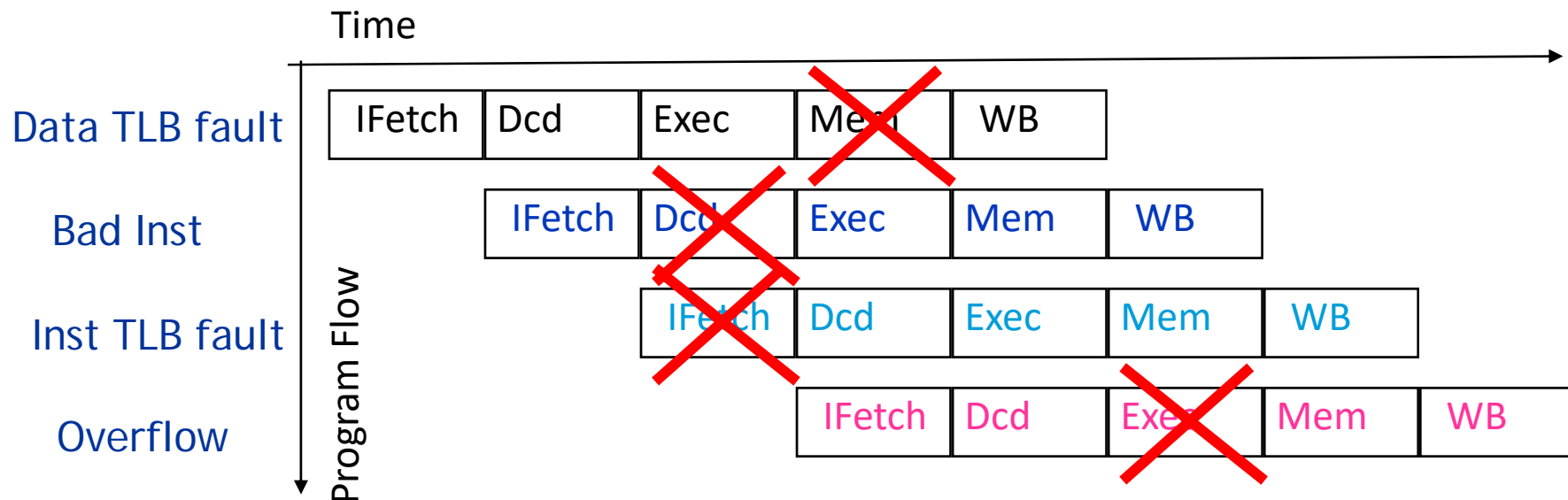


Precise Exceptions in simple 5-stage pipeline

- Exceptions may occur at different stages in pipeline
-> Exceptions may be raised *out of order*
- Let's consider this second example:
 - Instruction page fault occurs in Instruction Memory stage of first instruction
 - Data page fault occurs in memory stage of second instruction
- Instruction page fault is handled first!!!

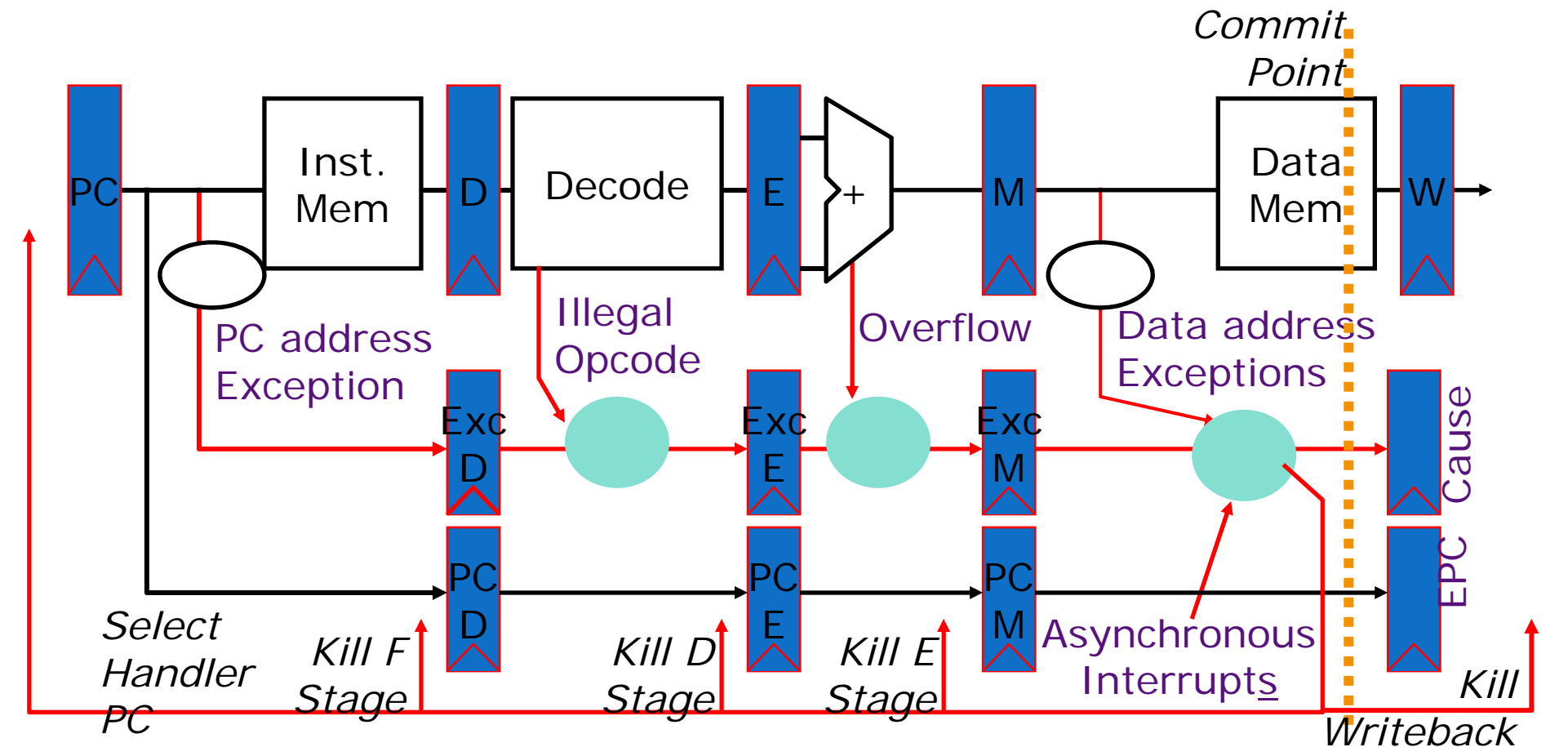


Another look at the exception problem



- Use pipeline to sort this out!
 - Pass exception status along with instruction.
 - Keep track of PCs for every instruction in pipeline.
 - Wait until the end of MEM stage to flag an exception
 - Don't act on exception until it reaches WB stage
- Handle interrupts through “faulting noop” in IF stage
- When instruction reaches Commit Point entering WB stage:
 - Save PC \Rightarrow EPC, Interrupt Handler Addr \Rightarrow PC
 - Turn all next instructions in earlier stages into NOPs!

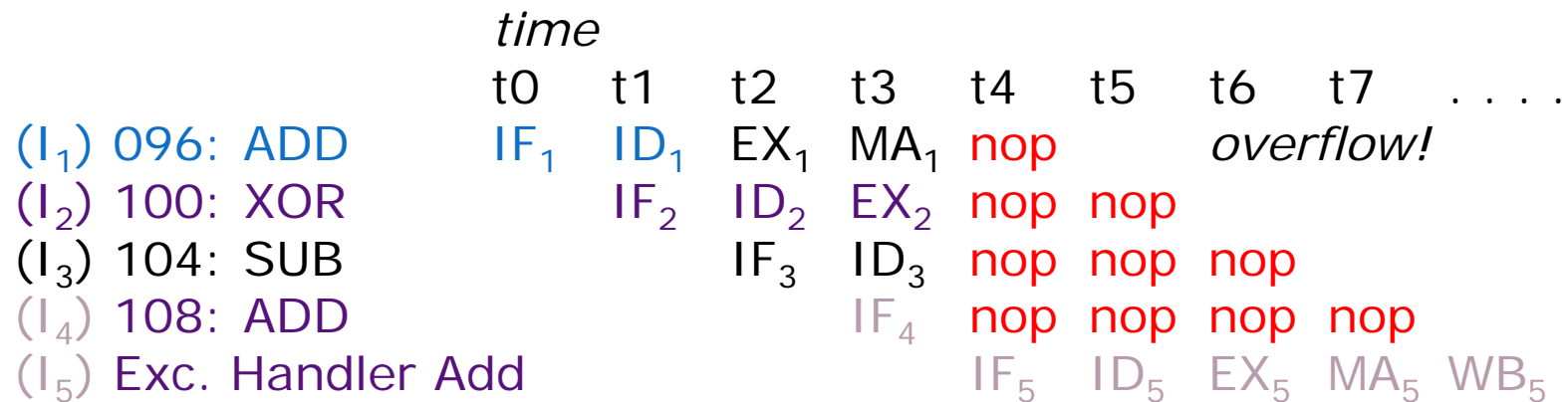
Exception Handling: 5-Stage Pipeline



- Write in Data Memory and Write Back in RF are disabled to guarantee a precise interrupt model
- Hold exception flags in pipeline until commit point (M stage)

Exception Pipeline Diagram

- When instruction ADD at Commit Point before entering in WB stage:
Save PC and Exc. Handler Address \Rightarrow PC
- Turn all next instructions in earlier stages into NOPs!



- After the end of the Exception Handler Routine, the ADD instruction will be re-executed and the instruction flow will continue

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)
- Later on we will discuss again exception handling in ***out-of-order execution processors.***