# 05.Exceptions

We talked about the data but there is also the control of the system and the execution. There could be a standard program flow but there there can be an interrrupt/exception. These problems have to be handled. Every that you see it seems to be executed in parallel but it is executed serially by the CPU. An Interrupt Handler is the component that handles the exception. An interrupt requires the transfer of the control flow to the handler but you have to not affect the program so you save the state, execute the new handler program and then reload the initial program data. These events are considered rare respect the number of instructions.

## Causes of interrupt

1. 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, something executed inside the program but it is not expected)
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception(aritmetic errors)
  - misaligned memory access(accessing an element with the incorrect address)
  - virtual memory exceptions: page faults, TLB misses, protection violations(want to access something not in your process)
  - traps: system calls, e.g., jumps into kernel

## Exception Handling

Concept started in the fifties to handles overflows principally then also for I/O.
Mobile computer have the agility pros but they have to compute before the battery runs out.

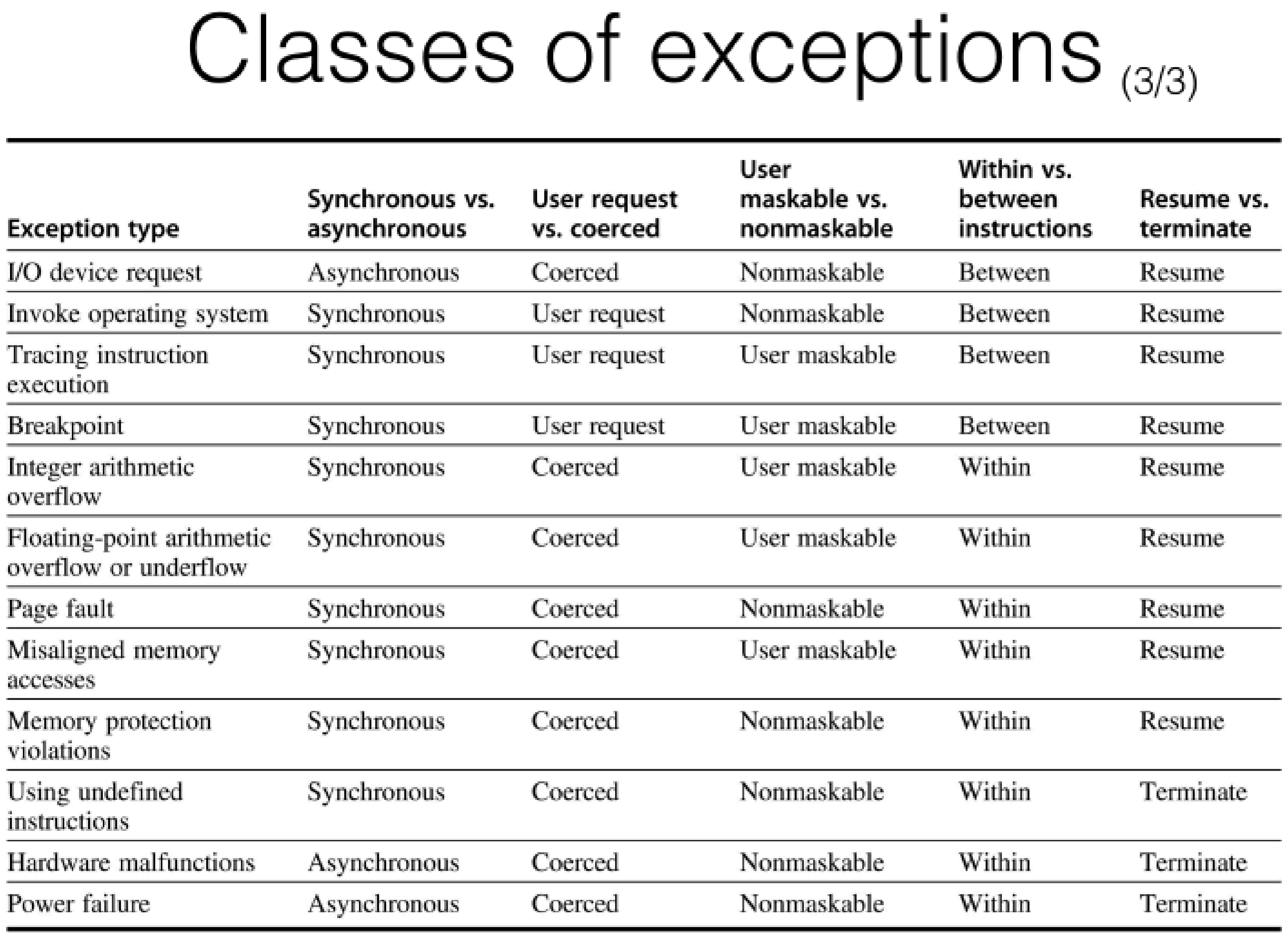
## Classes of excetions

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). You try to at least complete the instruction currently executing at the time(a few cycles delay).
- User Requested vs Coerced(Who ask the event)
  - 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(Cannot delay too much the execution)

- User maskable vs User nonmaskable(Having an event and during this event you have another event to manage. You have to set a priority for the events)
  - The mask simply controls whether the hardware responds to the exception or not. Not maskable it means it cannot manage more than one exception at a time.
- 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 cause program termination(easiest and CPU conservative solution).
- Resume vs Terminate
  - Terminating event: program's execution always stops after the interrupt(Don't know how to manage what it is happening)
  - Resuming event: program's execution continues after the interrupt



Classes of exceptions (3/3)

| Exception type | Synchronous vs. asynchronous | User request vs. coerced | User maskable vs. nonmaskable | Within vs. between instructions | Resume vs. terminate |
|---|---|---|---|---|---|
| I/O device request | Asynchronous | Coerced | Nonmaskable | Between | Resume |
| Invoke operating system | Synchronous | User request | Nonmaskable | Between | Resume |
| Tracing instruction execution | Synchronous | User request | User maskable | Between | Resume |
| Breakpoint | Synchronous | User request | User maskable | Between | Resume |
| Integer arithmetic overflow | Synchronous | Coerced | User maskable | Within | Resume |
| Floating-point arithmetic overflow or underflow | Synchronous | Coerced | User maskable | Within | Resume |
| Page fault | Synchronous | Coerced | Nonmaskable | Within | Resume |
| Misaligned memory accesses | Synchronous | Coerced | User maskable | Within | Resume |
| Memory protection violations | Synchronous | Coerced | Nonmaskable | Within | Resume |
| Using undefined instructions | Synchronous | Coerced | Nonmaskable | Within | Terminate |
| Hardware malfunctions | Asynchronous | Coerced | Nonmaskable | Within | Terminate |
| Power failure | Asynchronous | Coerced | Nonmaskable | Within | Terminate |

## 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). This is to restart from the next instruction before the interrupt
- It saves the PC of instruction Ii in a special register (EPC)
- It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

## Interrupt Handler

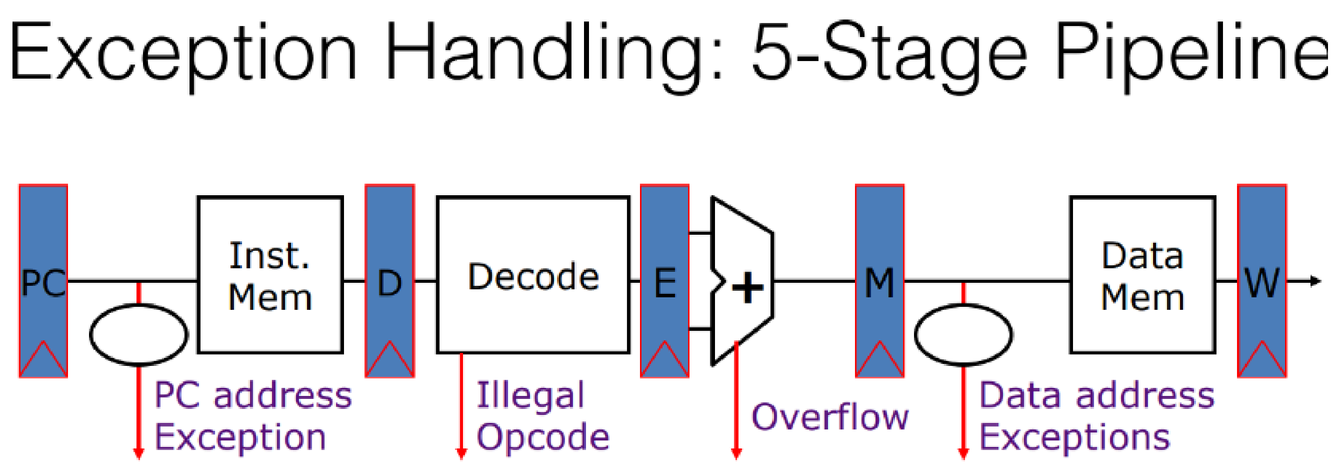
If you have an interrupt you could stop the execution. You want to mask further interrupt. You need a status register of the interrupt to determine what causes the interrupt. You need a special instruction to return from the exception(Jump instruction Return From Exception)

## Synchronous Interrupts

They are internal and in general are special calls to some function.
A synchronous interrupt (a.k.a. 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
Sometimes the exception arise from the will of undoing an instruction.

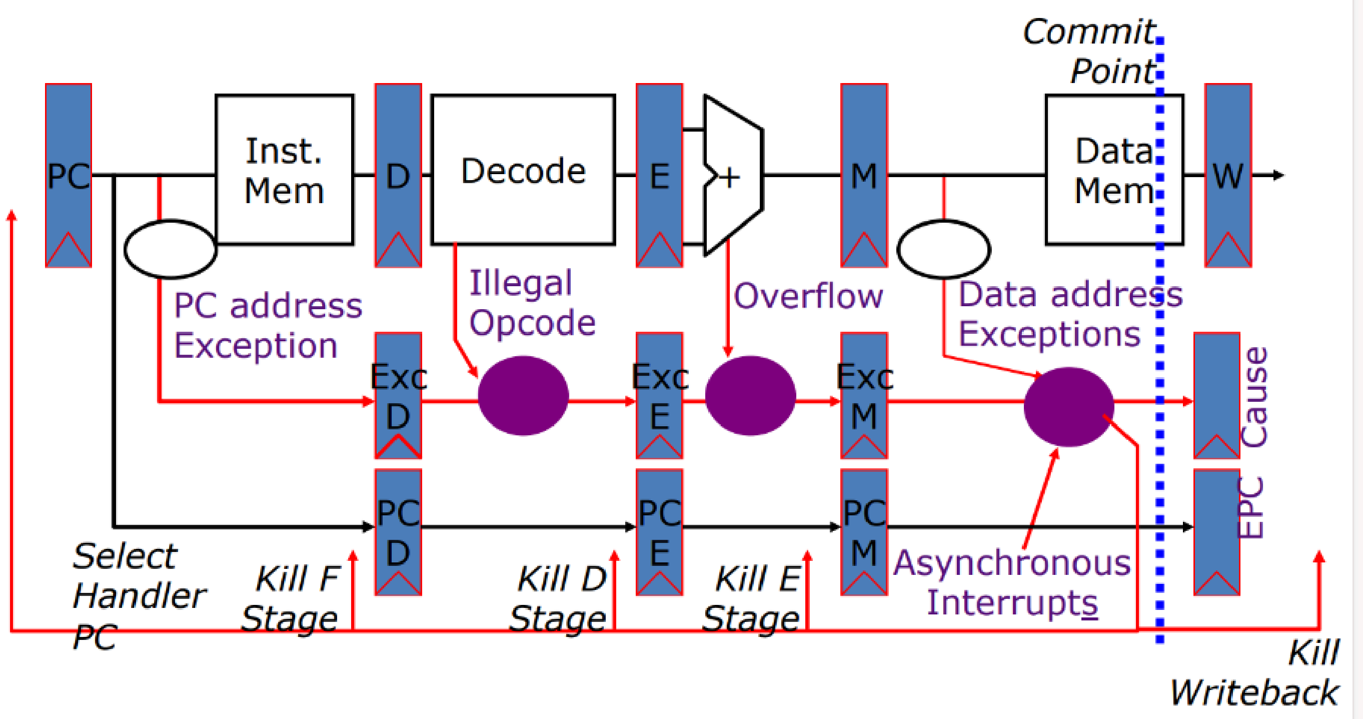
## Precise Interrupts/Exceptions

If there is a single instruction/point where the instruction before are completed and the next instructions don't have committed to a state(In the restart you don't have problems at all). These interrupts are desiderable as it is easier to figure out what happened, precision isn't needed in Restartability. However, preciseness makes it a lot easier to restart.

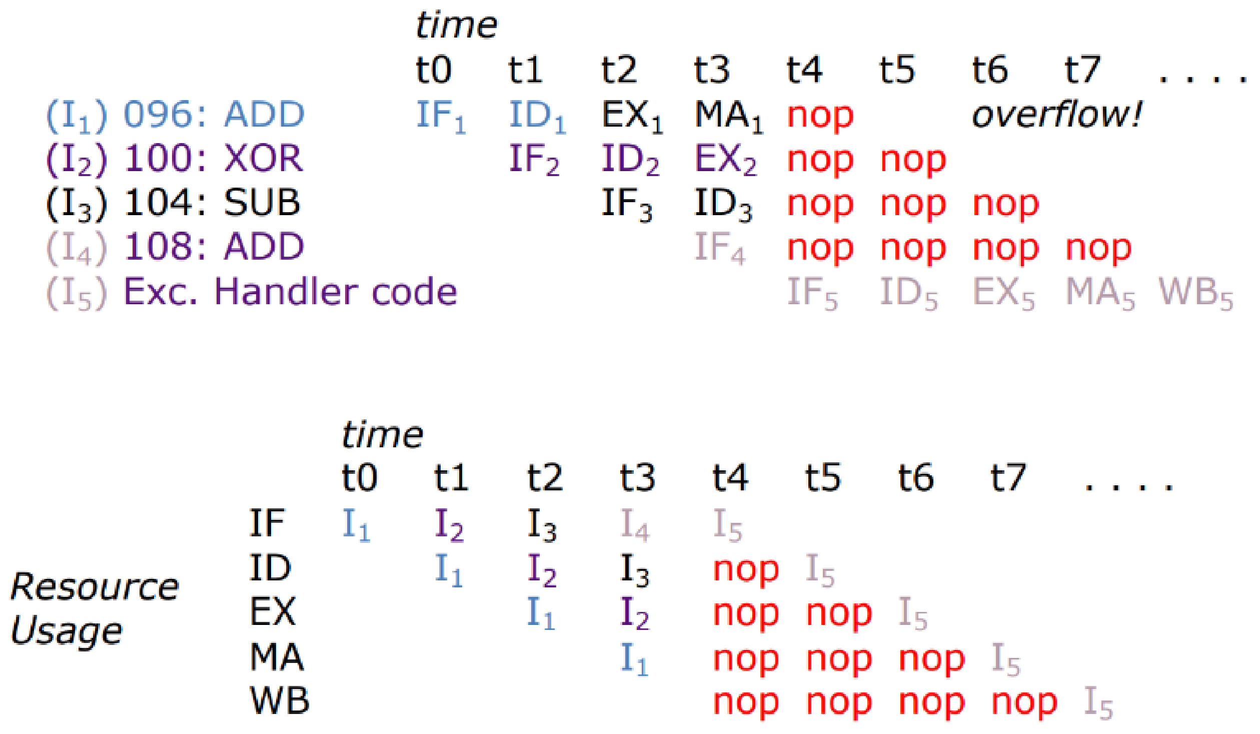


For every stage is better to detect the exception the state before as you don't propagate error and interrupts. There can be multiple interrupts at the same point. In these situations you want to interrupt the pipeline the less possible. In general you can tag an instruction as causing an exception and wait until end of memory stage to flag exception where the interrupt is marked as NOPs placed in the pipeline instead of an instruction(Assume that interrupt condition persists in case NOP flushed)

## Exception Handling: 5-Stage Pipeline



## Exception Pipeline Diagram



*time*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| ($I_1$) 096: ADD | $IF_1$ | $ID_1$ | $EX_1$ | $MA_1$ | nop | | *overflow!* | | |
| ($I_2$) 100: XOR | | $IF_2$ | $ID_2$ | $EX_2$ | nop | nop | | | |
| ($I_3$) 104: SUB | | | $IF_3$ | $ID_3$ | nop | nop | nop | | |
| ($I_4$) 108: ADD | | | | $IF_4$ | nop | nop | nop | nop | |
| ($I_5$) Exc. Handler code | | | | | $IF_5$ | $ID_5$ | $EX_5$ | $MA_5$ | $WB_5$ |

*time*

| *Resource Usage* | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| IF | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | | | |
| ID | | $I_1$ | $I_2$ | $I_3$ | nop | $I_5$ | | | |
| EX | | | $I_1$ | $I_2$ | nop | nop | $I_5$ | | |
| MA | | | | $I_1$ | nop | nop | nop | $I_5$ | |
| WB | | | | | nop | nop | nop | nop | $I_5$ |

## 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
  Jim Smith's classic paper discusses several methods for getting precise interrupts:
- In-order instruction completion: you can have the instruction in a different order BUT you have to complete the instruction in order
- Reorder buffer
- History buffer

## Speculating on Exceptions

You want to anticipate the decision and then commit to the evaluation after you resolve the condition.

- 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