Lecture-10 (Exception/Interrupts)
CS422-Spring 2020

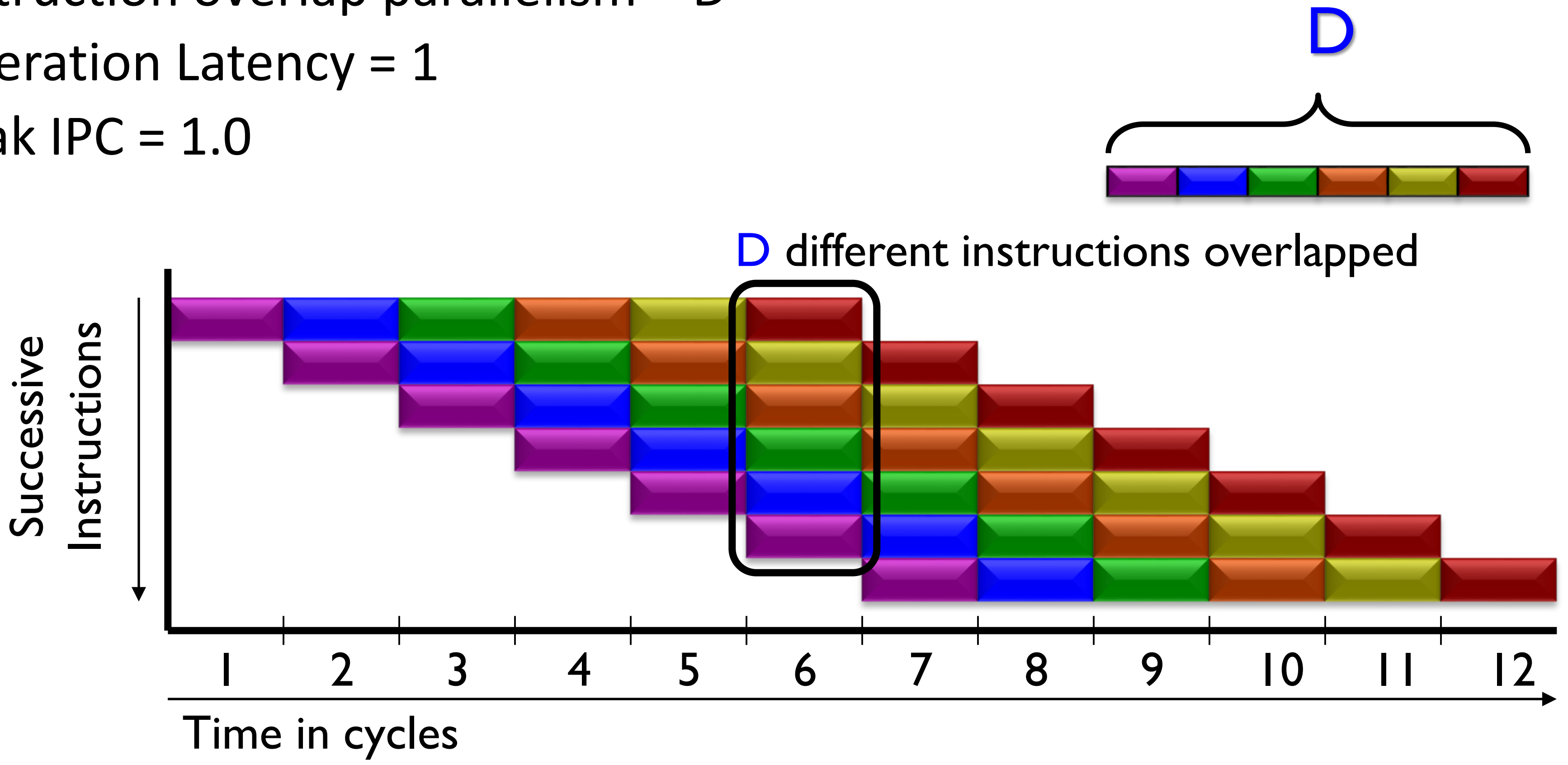
Biswa@CSE-IITK

Going Beyond Scalar

- Scalar pipeline limited to CPI ≥ 1.0
  - Can never run more than 1 insn per cycle
- “Superscalar” can achieve CPI ≤ 1.0 (i.e., IPC ≥ 1.0)
  - *Superscalar* means executing multiple insns in parallel

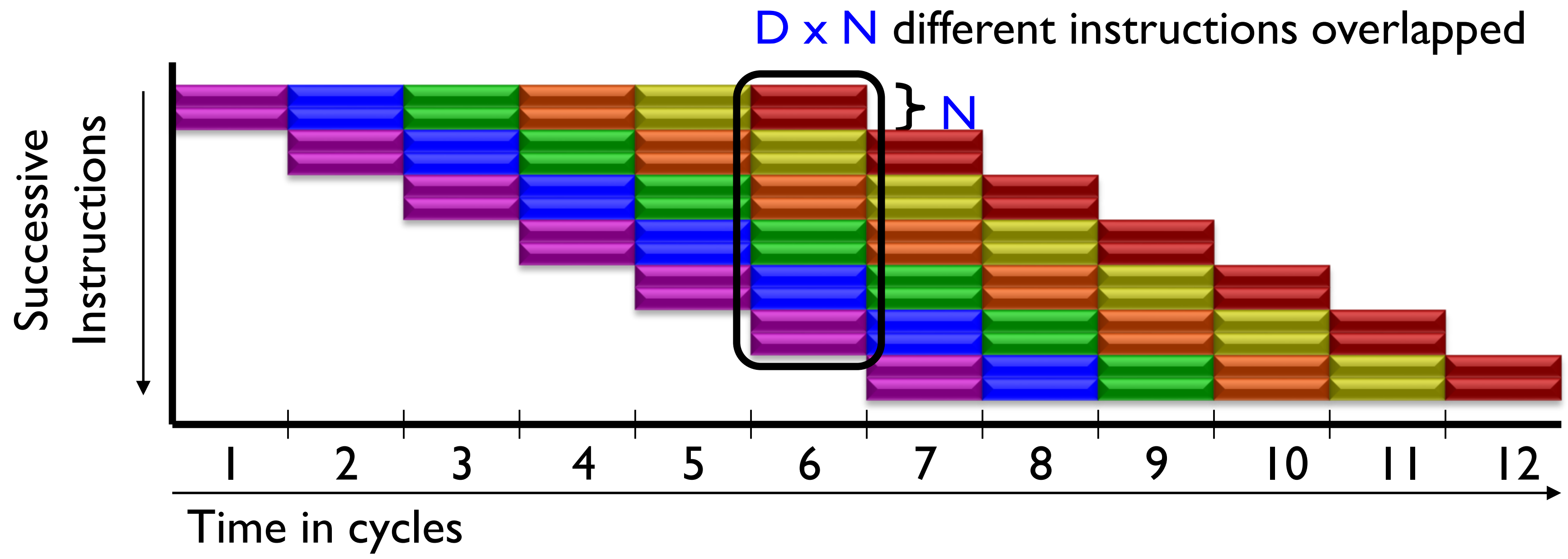
Architectures for Instruction Parallelism

- Scalar pipeline (baseline)
  - Instruction overlap parallelism = D
  - Operation Latency = 1
  - Peak IPC = 1.0



Superscalar Machine

- Superscalar (pipelined) Execution
  - Instruction parallelism = D x N
  - Operation Latency = 1
  - Peak IPC = N per cycle



Problems with Pipelining

- Exception: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode

- Interrupt: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)

- Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
  - The effect of all instructions up to and including I_i is totaling complete
  - No effect of any instruction after I_i can take place

- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

World of Faults, interrupts, aborts

- SYNC Interrupts (Exceptions)

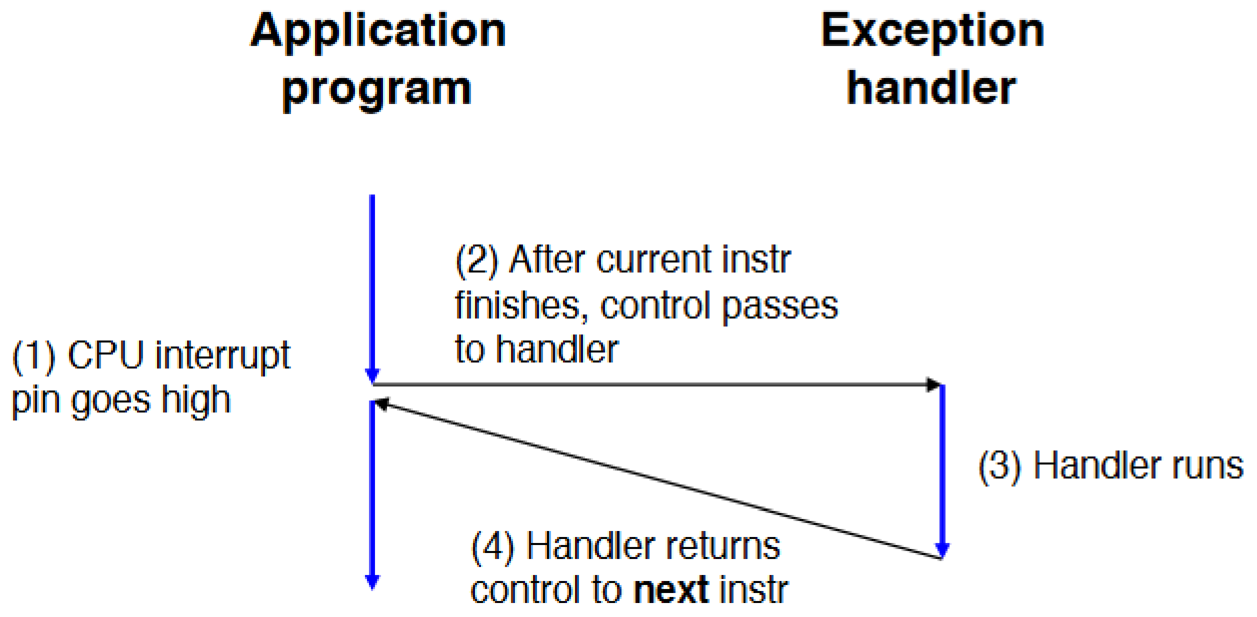
Faults
Traps

- ASYNC Interrupts

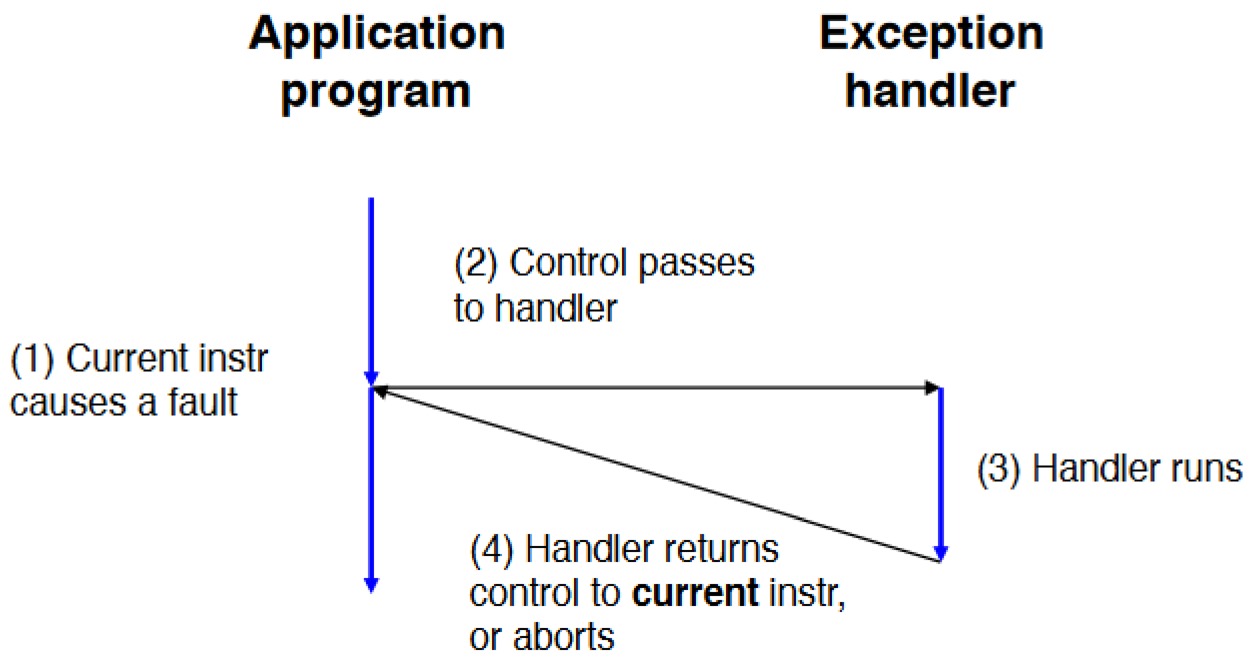
Actual interrupts

Others:
ABORTS

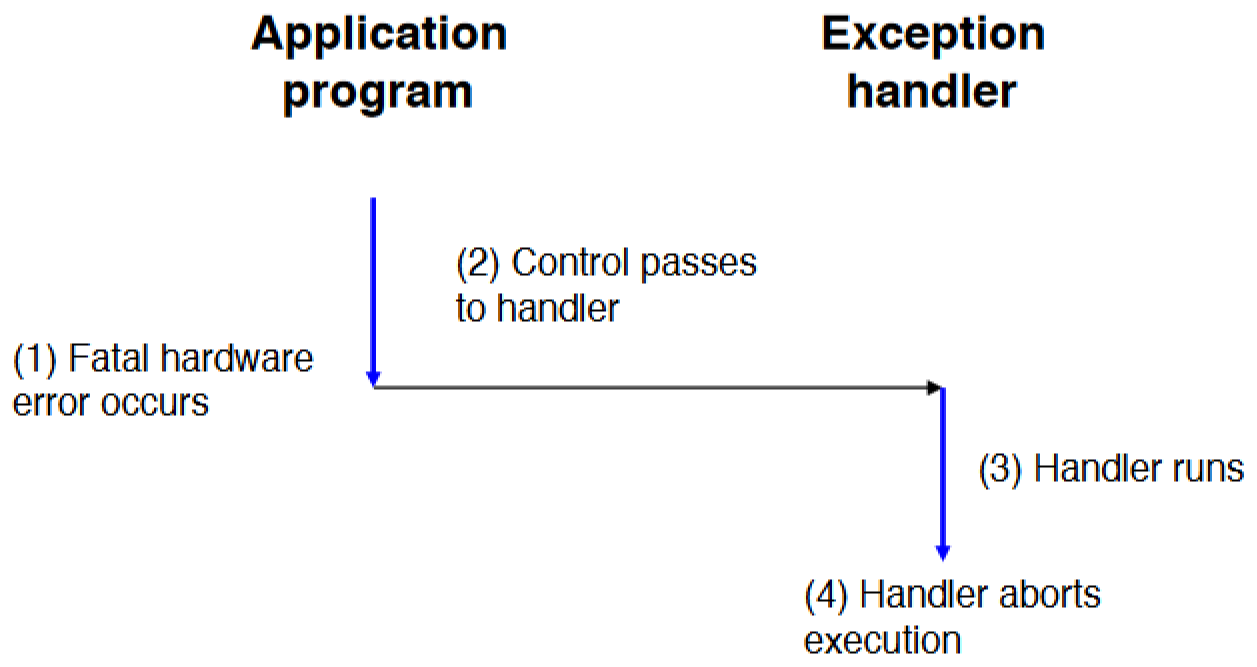
Interrupts


Application program
Exception handler
(1) CPU interrupt pin goes high
(2) After current instr finishes, control passes to handler
(3) Handler runs
(4) Handler returns control to **next** instr

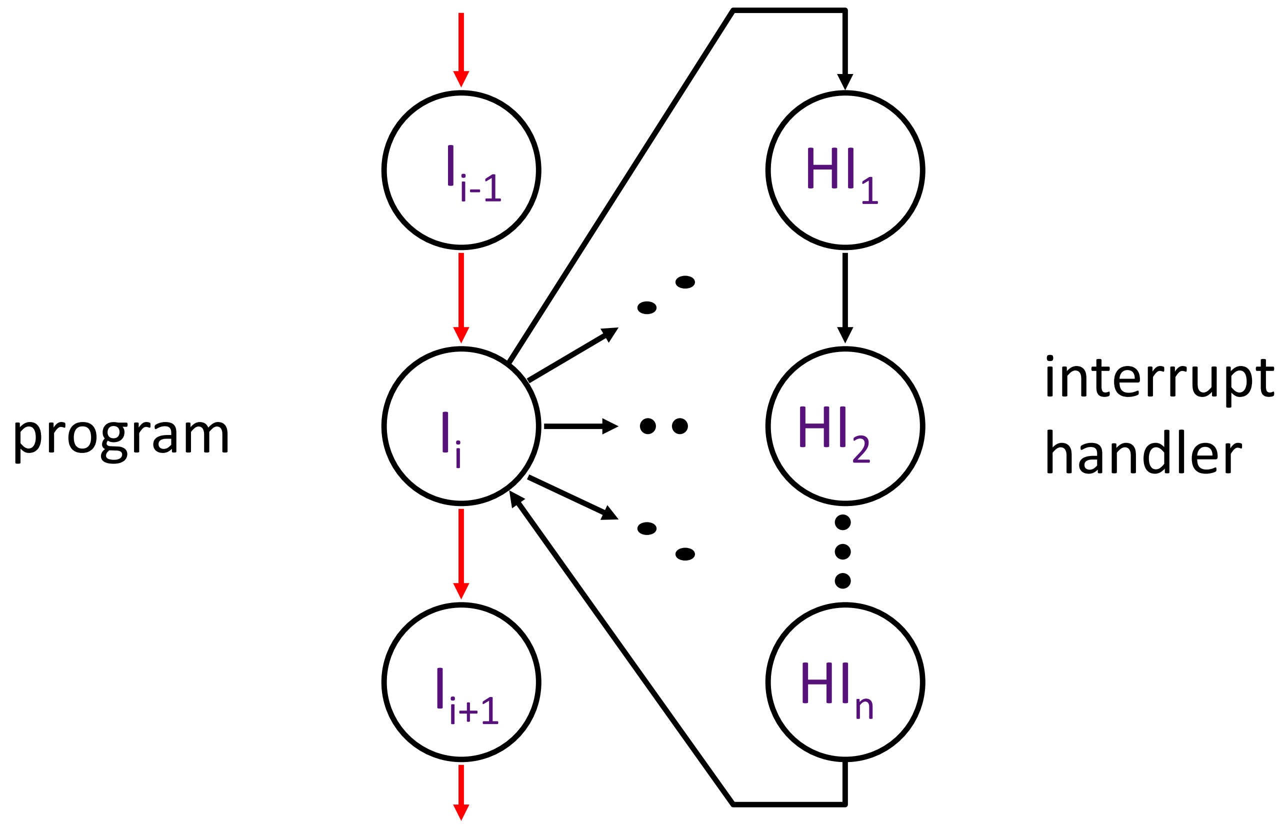
Faults


Application program
Exception handler
(1) Current instr causes a fault
(2) Control passes to handler
(3) Handler runs
(4) Handler returns control to **current** instr, or aborts

Aborts


Application program
Exception handler
(2) Control passes to handler
(1) Fatal hardware error occurs
(3) Handler runs
(4) Handler aborts execution

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.

Causes

- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure

- Synchronous: an *internal event (a.k.a. traps or exceptions)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults, protection violations
  - system calls, e.g., jumps into kernel

In General

- 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 (*precise interrupt)*

- It saves the PC of instruction I_{i+1} in a special register (EPC)

- It disables interrupts and transfers control to a designated interrupt handler running in the kernel mode

Interrupt Handler

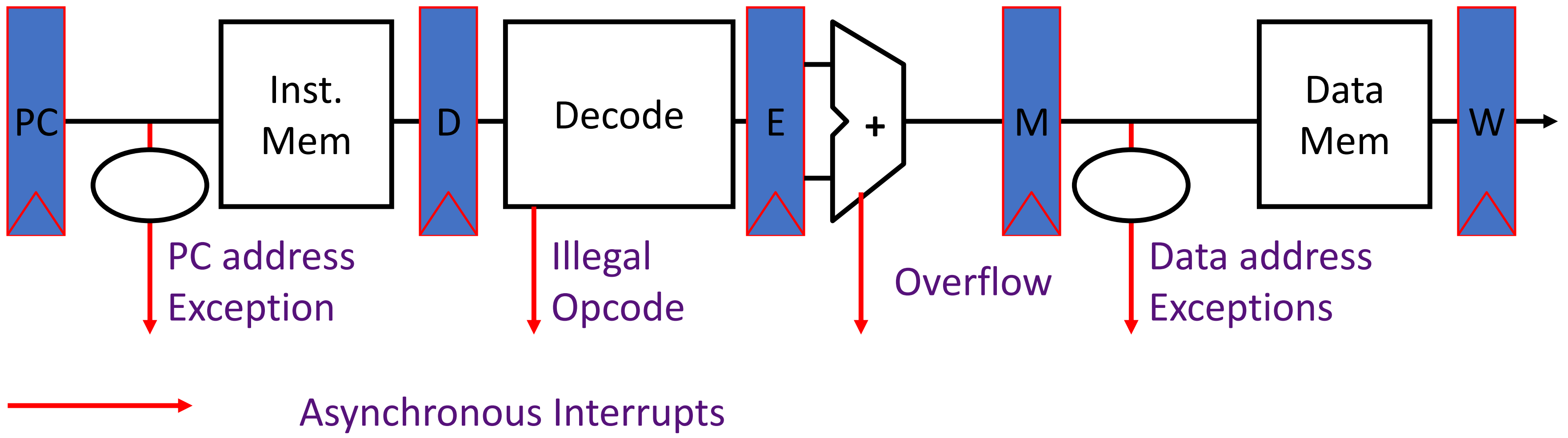
- Saves EPC before enabling interrupts to allow nested interrupts
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved

- Needs to read a *status register* that indicates the cause of the interrupt

Syn. Interrupts

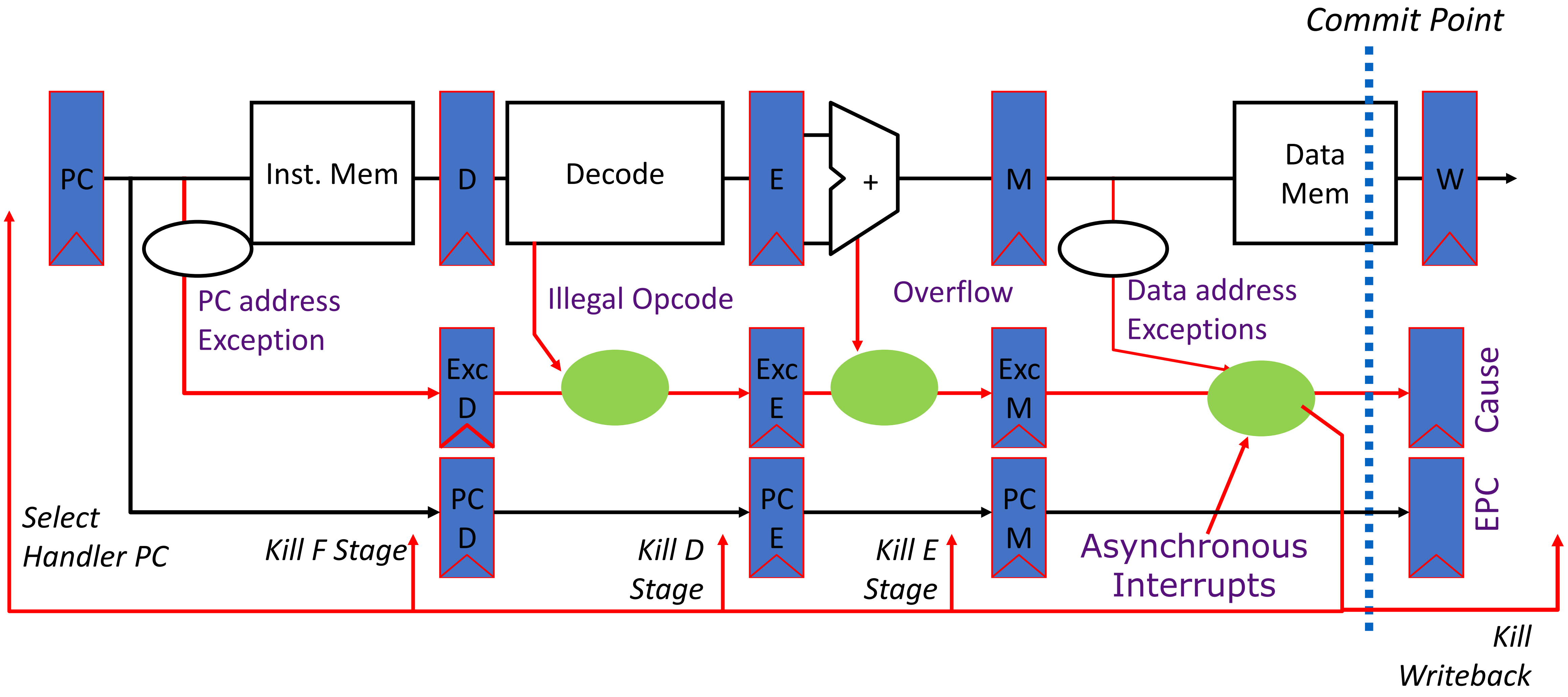
- A synchronous interrupt (exception) is caused by a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - requires undoing the effect of one or more partially executed instructions
- In the case of a system call trap, the instruction is considered to have been completed
  - a special jump instruction involving a change to privileged kernel mode

Exception Handling



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

Exception Handling



Exception Handling

- Hold exception flags in pipeline until commit point for instructions that will be killed (M stage): Why? To ensure precise exception so that all the previous instructions should commit before triggering the exception.

- Exceptions in earlier pipe stages override later exceptions *for a given instruction*

- If exception at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage

Pipeline with Exception

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

		t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Resource Usage	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	nop
	WB					nop	nop	nop	I_5	