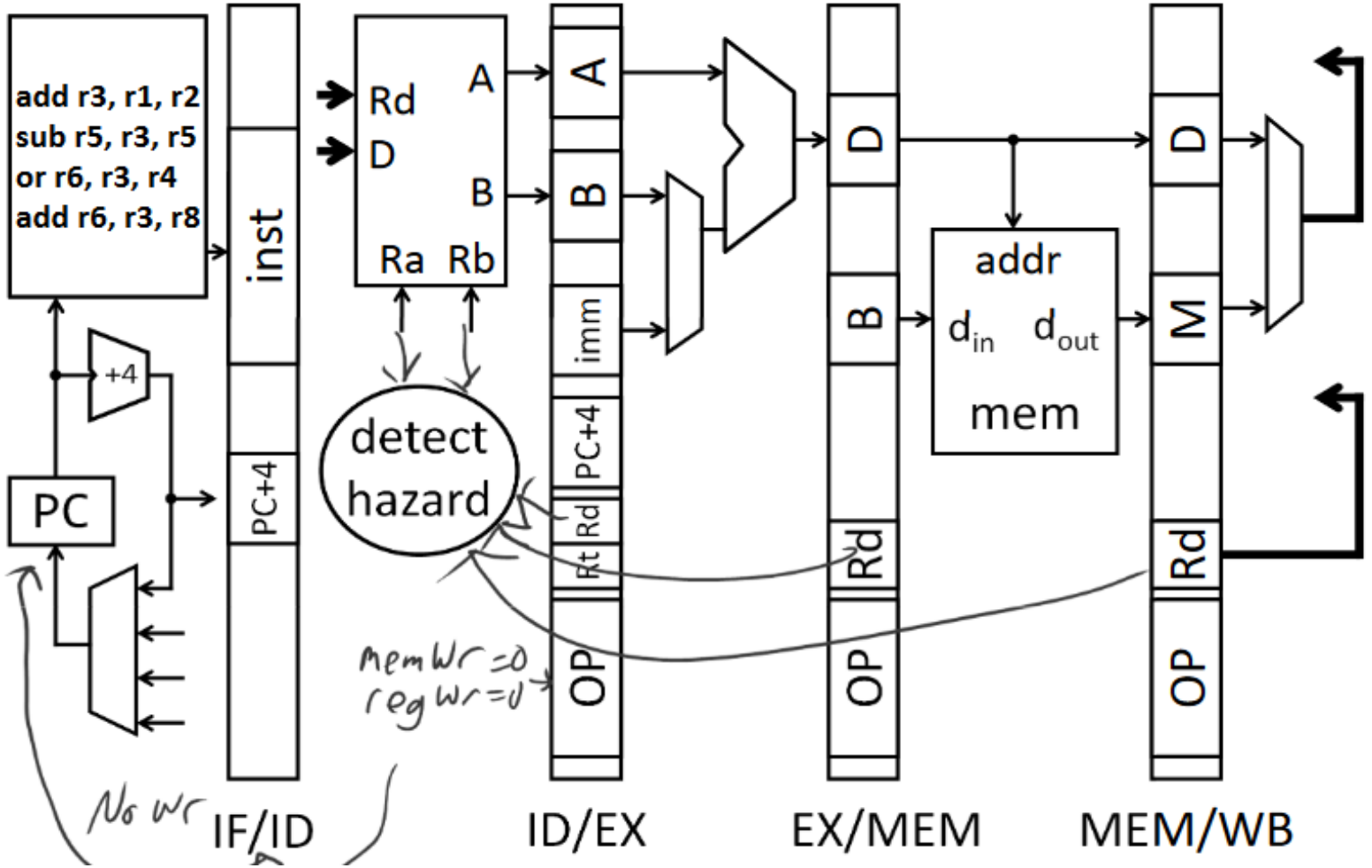
Lecture-7 (Branches)
CS422-Spring 2018

Biswa@CSE-IITK

Revisiting Hazard Detection



How to detect? Logic in ID stage:

```
stall = (IF/ID.rA != 0 && (IF/ID.rA == ID/EX.rD ||
                           IF/ID.rA == EX/M.rD ||
                           IF/ID.rA == M/WB.rD))
        || (same for rB)
```

Branches

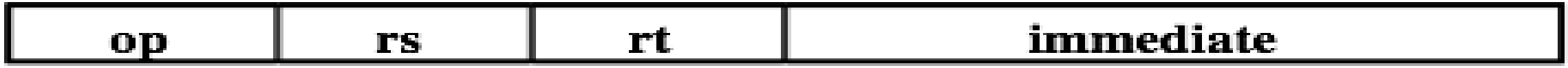
Conditional

- the target address is close to the current PC location
- branch distance from the incremented PC value fits into the immediate field
- for example: loops, if statements

Unconditional (jumps)

- transfers of control
- the target address is far away from the current PC location
- for example: subroutine calls

Branches

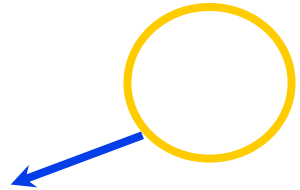


Syntax: BEQ $1, $2, 12

Action: If ($1 != $2), PC = PC + 4

Action: If ($1 == $2), PC = PC + 4 + 48

Immediate field codes # words, not # bytes.
Why is this encoding a good idea?

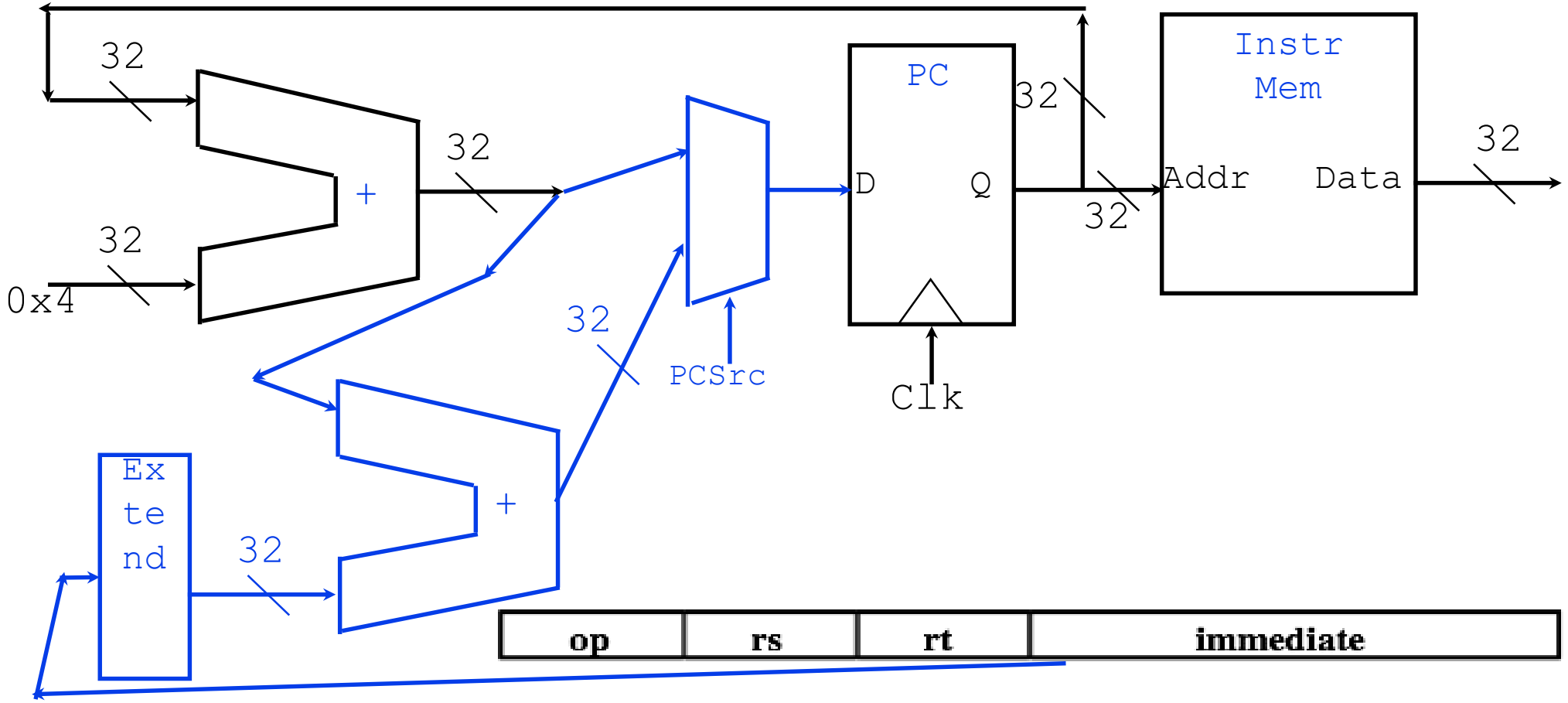


Increases branch range to 128 KB.

Zero-extend or sign-extend immediate field? Sign-extend.
Why is this extension method a good idea?
Supports forward and backward branches.

Datapath

```
Syntax: BEQ $1, $2, 12
Action: If ($1 != $2), PC = PC + 4
Action: If ($1 == $2), PC = PC + 4 + 48
```



Control Hazard Alternatives

#1: Stall until branch direction is clear (in MIPS: sll 0 0)

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
  - MIPS still incurs 1 cycle branch penalty

Delayed Branch

#4: Delayed Branch

- Define branch to take place AFTER a following instruction

  branch instruction

  sequential successor$_1$

  sequential successor$_2$

  .....

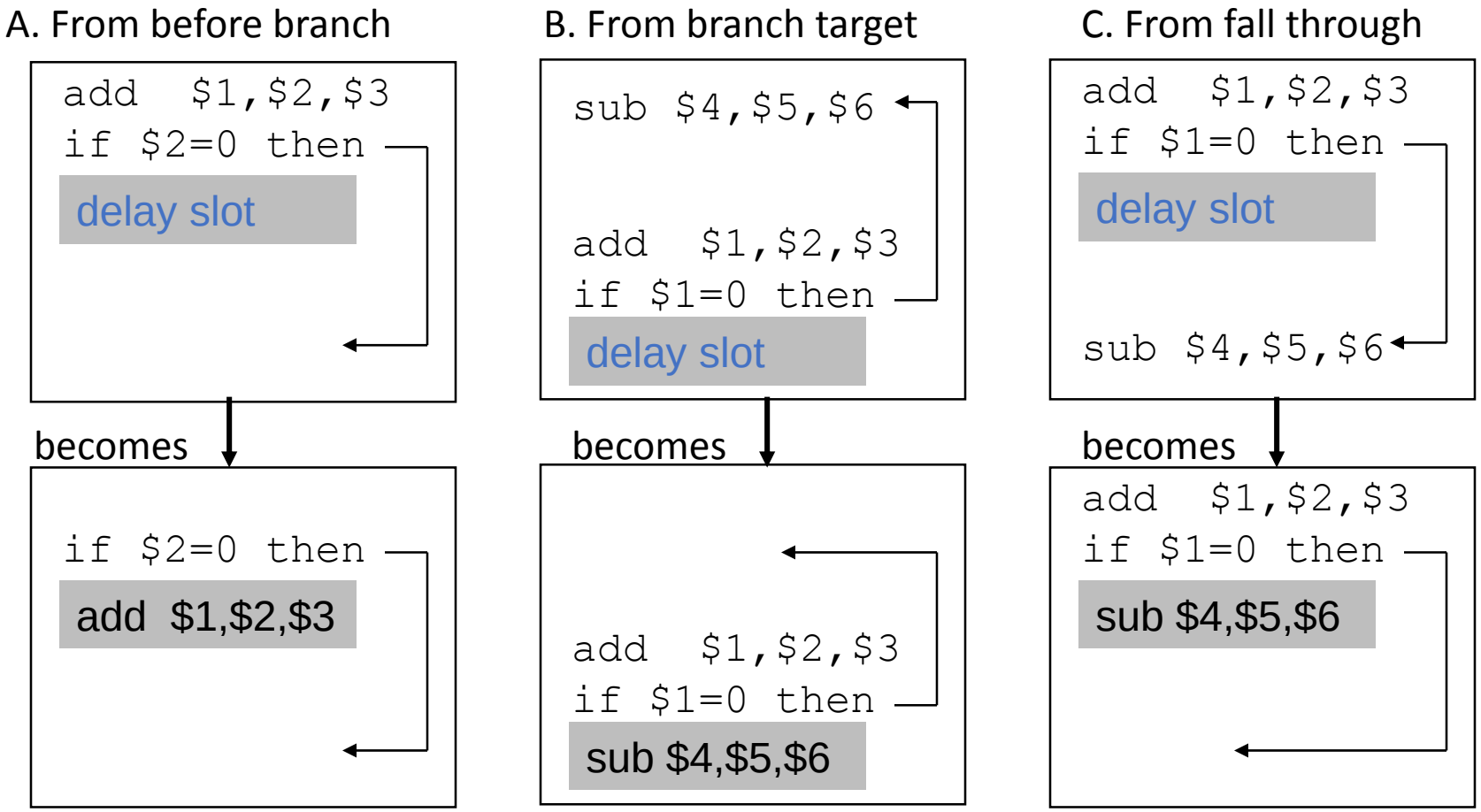
  sequential successor$_n$

  branch target if taken

  Branch delay of length *n*

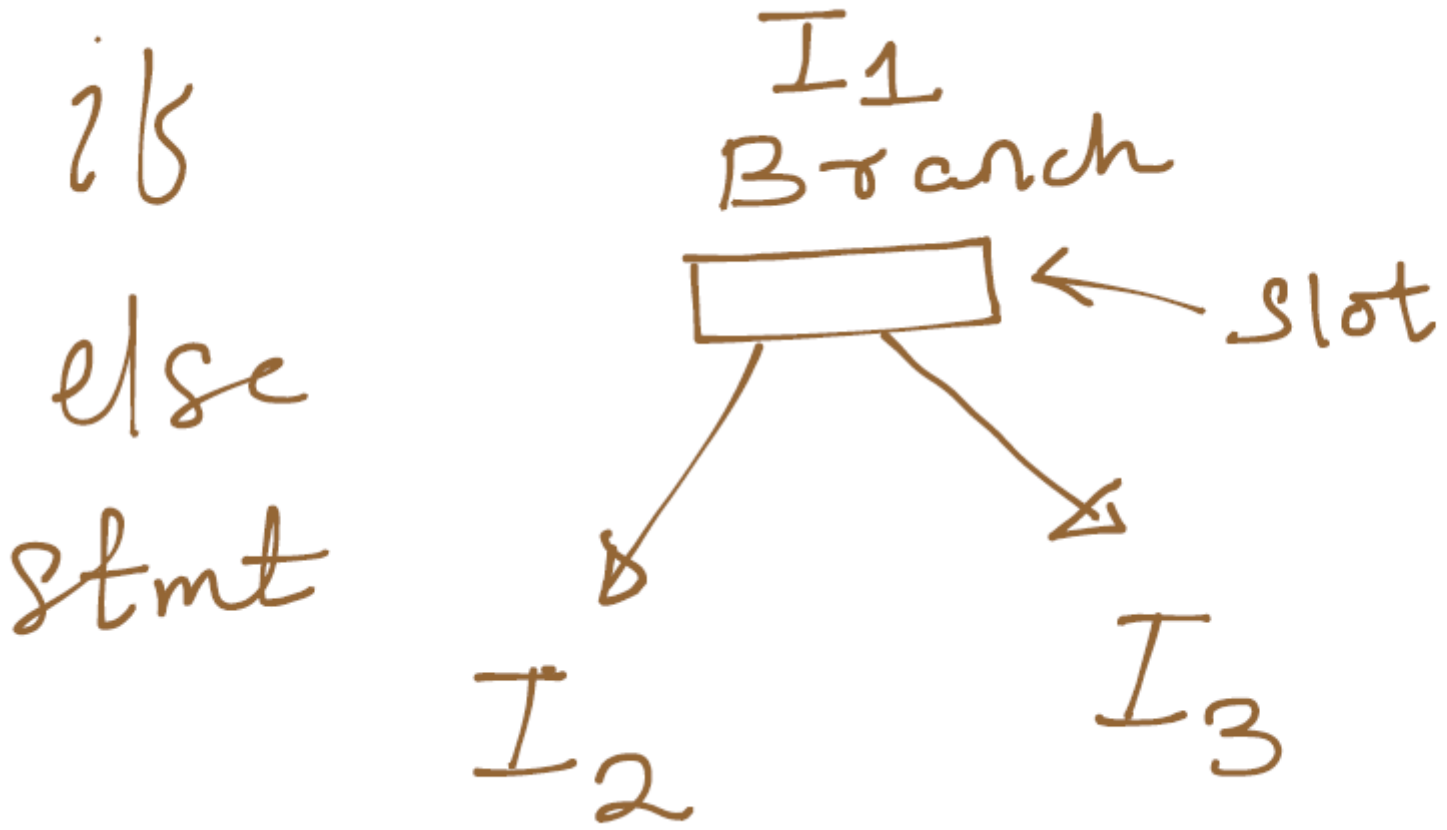
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Confused?



How to Handle Control Dependences?

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Guessing Next PC = PC+4

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction and branch prediction

- How can you make this more effective?

- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
  - Hardware: ??? (how can you do this in hardware…)

Impact of Stall on Performance

- Each stall cycle corresponds to *one lost cycle* in which no instruction can be completed

- For a program with N instructions and S stall cycles,
- Average CPI=(N+S)/N
- S depends on
  - frequency of RAW dependences
  - exact distance between the dependent instructions
  - distance between dependences

  suppose i_1,i_2 and i_3 all depend on i_0, once i_1's dependence is resolved, i_2 and i_3 must be okay too

“Iron Law” of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends on ISA and µarchitecture
- Time per cycle depends upon the µarchitecture and base technology

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

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, Traps, interrupts, aborts (Piazza +1)