

CASPER

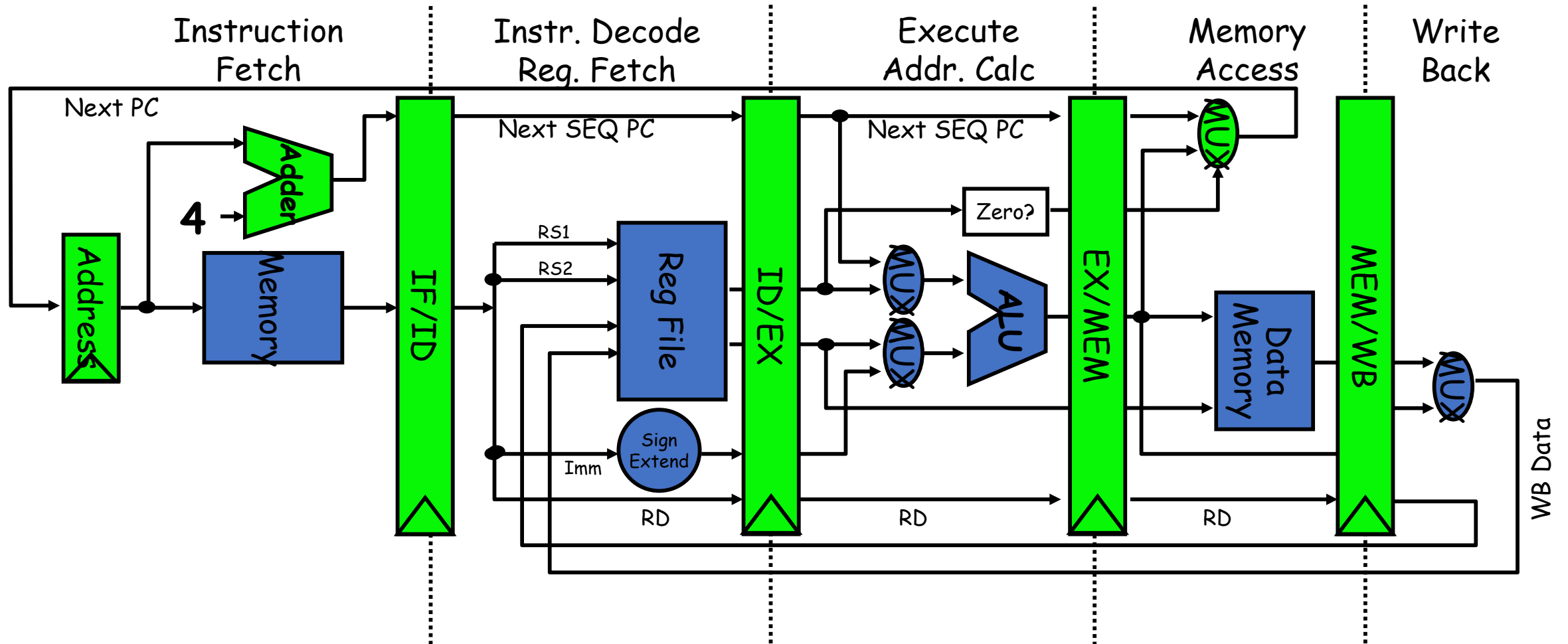
CS773-2025-Spring: Computer Architecture for Performance and Security

Lecture 7: 10K feet View of CPUs 😊




ON SILENT MODE PLEASE

Vanilla 5-stage pipeline



Data dependences (hazards)


```
add  R1, R2, R3
sub  R2, R4, R1
or   R1, R6, R3
```



read-after-write
(RAW)

True dependence


```
add  R1, R2, R3
sub  R2, R4, R1
or   R1, R6, R3
```



write-after-read
(WAR)

anti dependence

```
add  R1, R2, R3
sub  R2, R4, R1
or   R1, R6, R3
```



write-after-write
(WAW)

output dependence

Read-After-Write (RAW)

- Read must wait until earlier write finishes

Anti-Dependence (WAR)

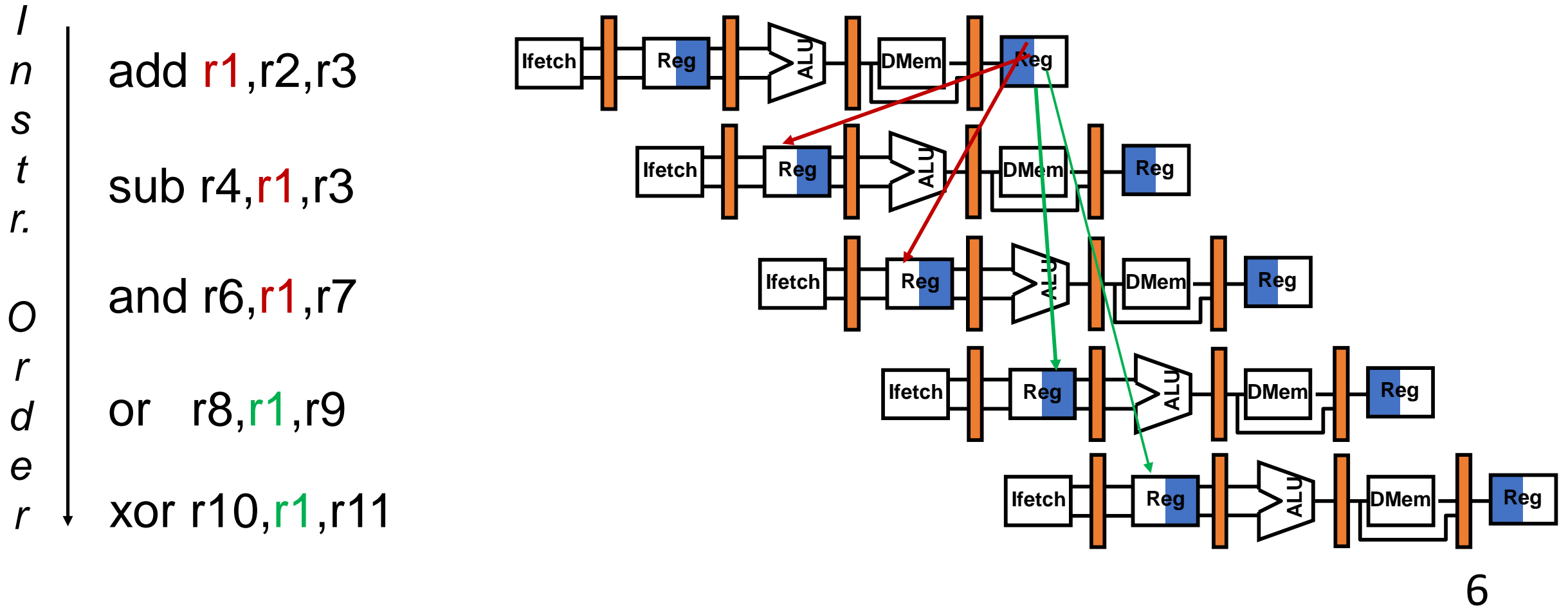
- Write must wait until earlier read finishes

Output Dependence (WAW)

- Earlier write can't overwrite later write
- (WAW hazard: not possible with vanilla 5-stage pipeline)

Data Hazards (Examples)

Time (clock cycles)



Control Hazards: An Example

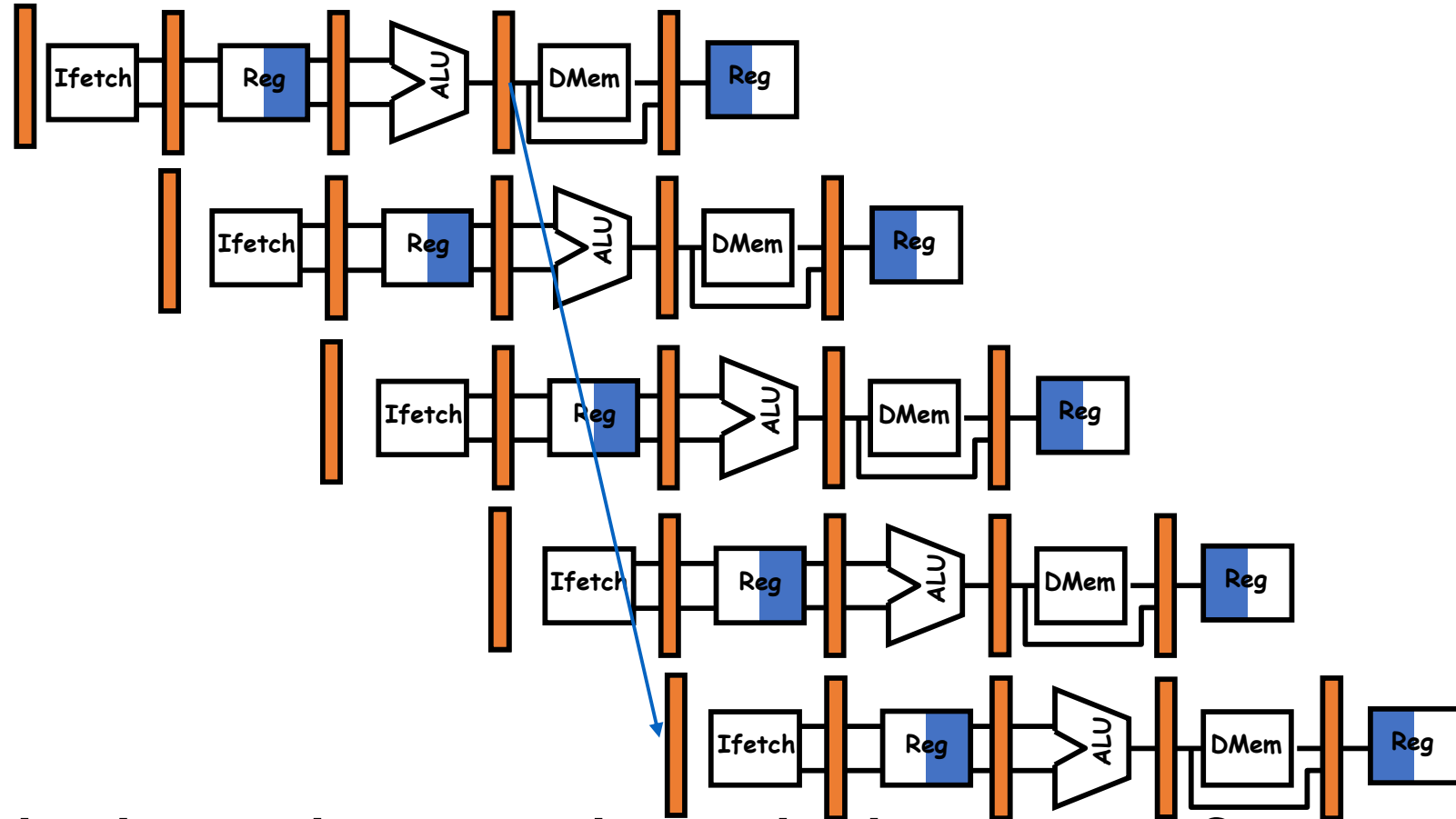
10: beq r1,r3,36

14: and r2,r3,r5 ☹️

18: or r6,r1,r7 ☹️

22: add r8,r1,r9 ☹️

50: xor r10,r1,r11



What do you do with the 3 instructions in between?

How do you do it?

What and Where? Control Hazard

What do we need to calculate next PC?

- For Jumps
 - Opcode, offset, and PC
- For Jump Register
 - Opcode and register value
- For Conditional Branches
 - Opcode, offset, PC, and register (for condition)
- For all others
 - Opcode and PC

In what stage do we know these?

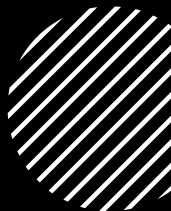
- PC - Fetch
- Opcode, offset - Decode (or Fetch?)
- Register value - Decode
- Branch condition $((rs) == 0)$ - Execute (or Decode?)

Branches: Taken/Not Taken and Target

<i>Instruction</i>	<i>Taken known?</i>	<i>Target known?</i>
J	After Inst. Decode	After Inst. Decode
BEQZ/BNEZ	After Inst. Execute	After Inst. Execute



Branch Prediction: 10K Feet View



Predict whether the next PC is a branch PC, at the fetch stage?



If branch, will it be taken?



If taken, what is the target address?



How?



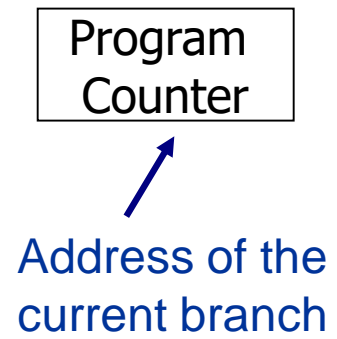
We know whether it is a branch PC or not in the decode stage. Oh no 😞

Branch Predictor: A bit deeper

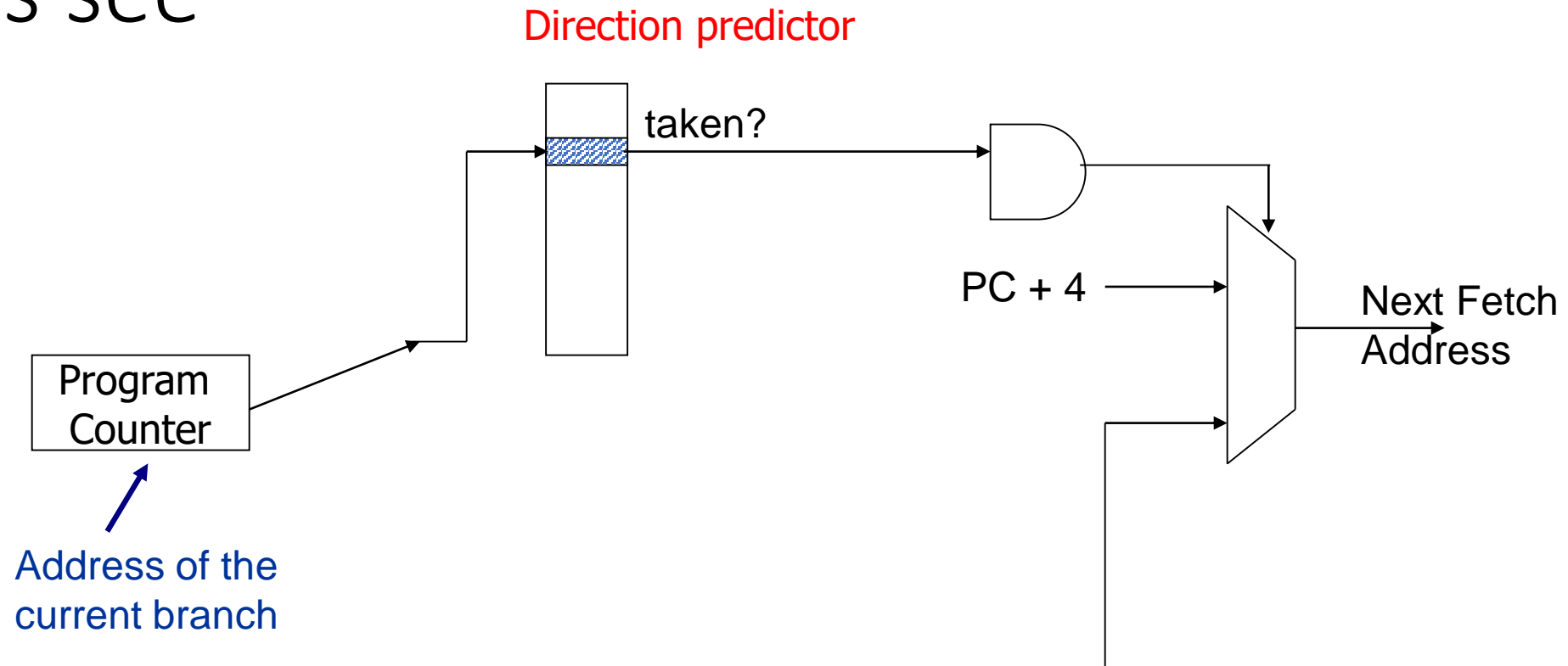
Three tasks

1. Is the PC a branch/jump? YES/NO
2. If Yes, can we predict the direction? Taken or not-taken
3. If taken, can we predict the target address?

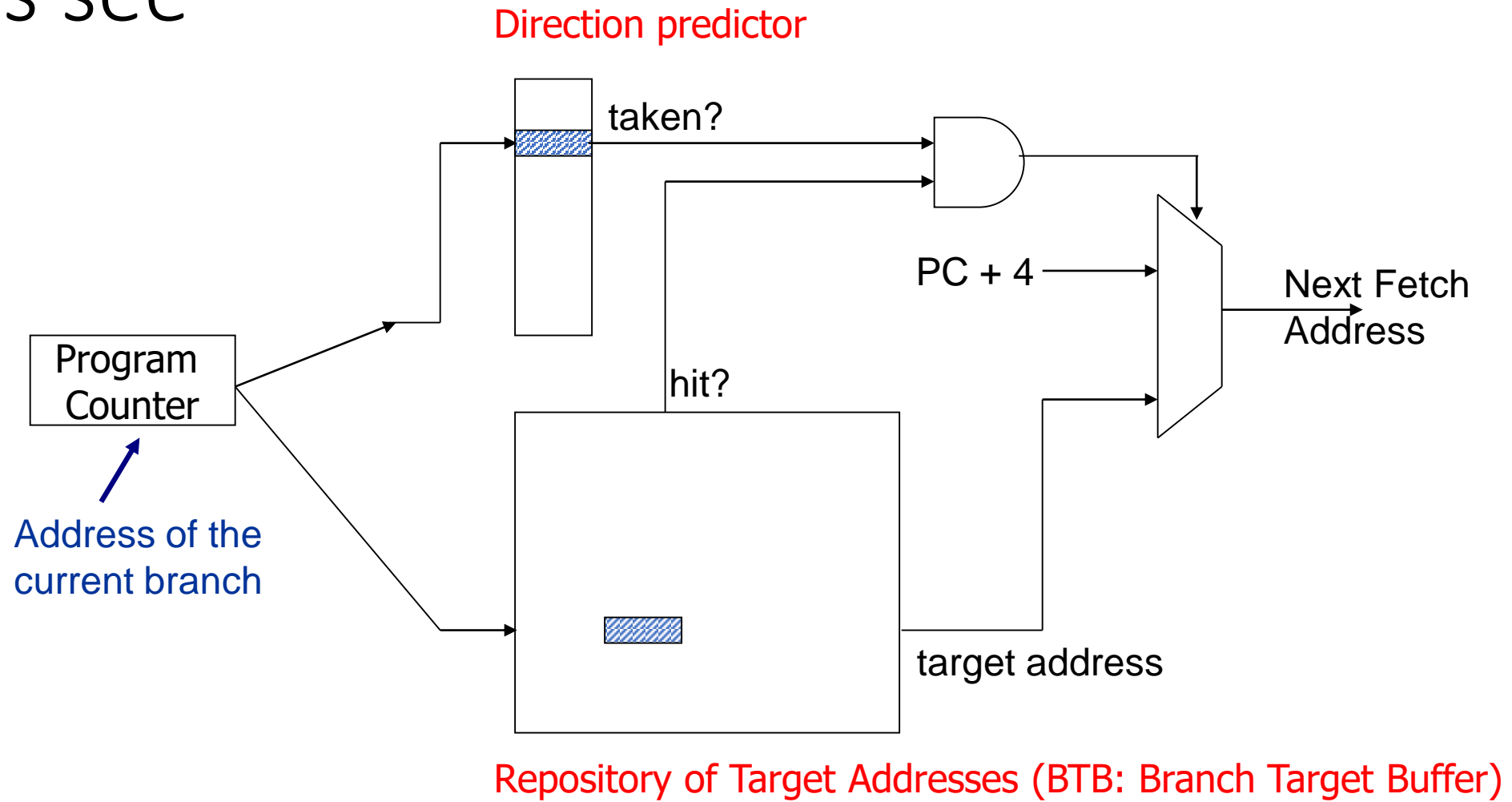
Let's see



Let's see



Let's see



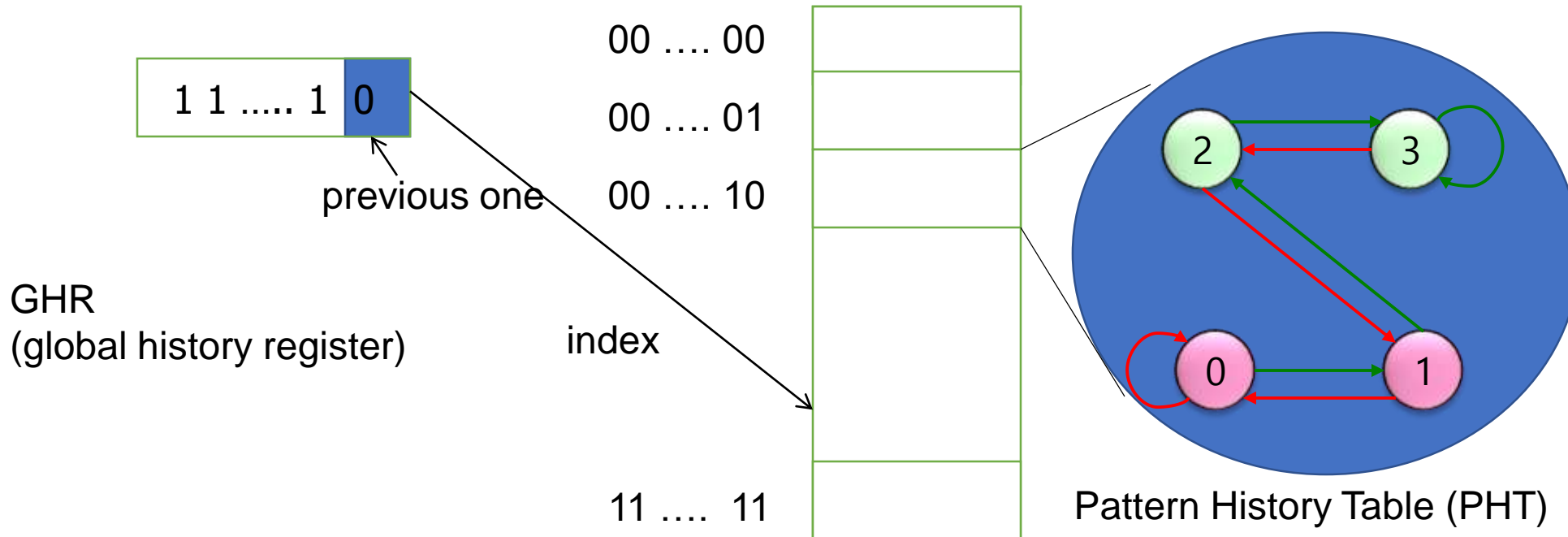
Two Level Branch Predictors

First level: **Global branch history register** (N bits)

The direction of last N branches

Second level: **Table of saturating counters for each history entry**

The direction the branch took the last time the same history was seen



BTB (Target Address Predictor)

Address of branch instruction

0b0110[...]01001000

Branch instruction

BNEZ R1 Loop

Branch Target Buffer (BTB)

30-bit address tag

target address

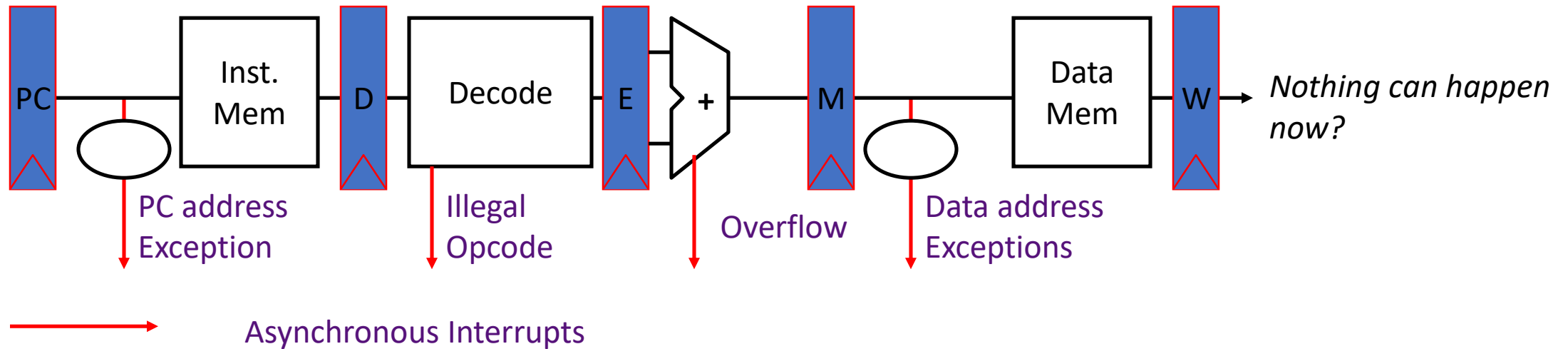
0b0110[...]0010	PC + 4 + Loop

Branch History Table (BHT)

2 state bits

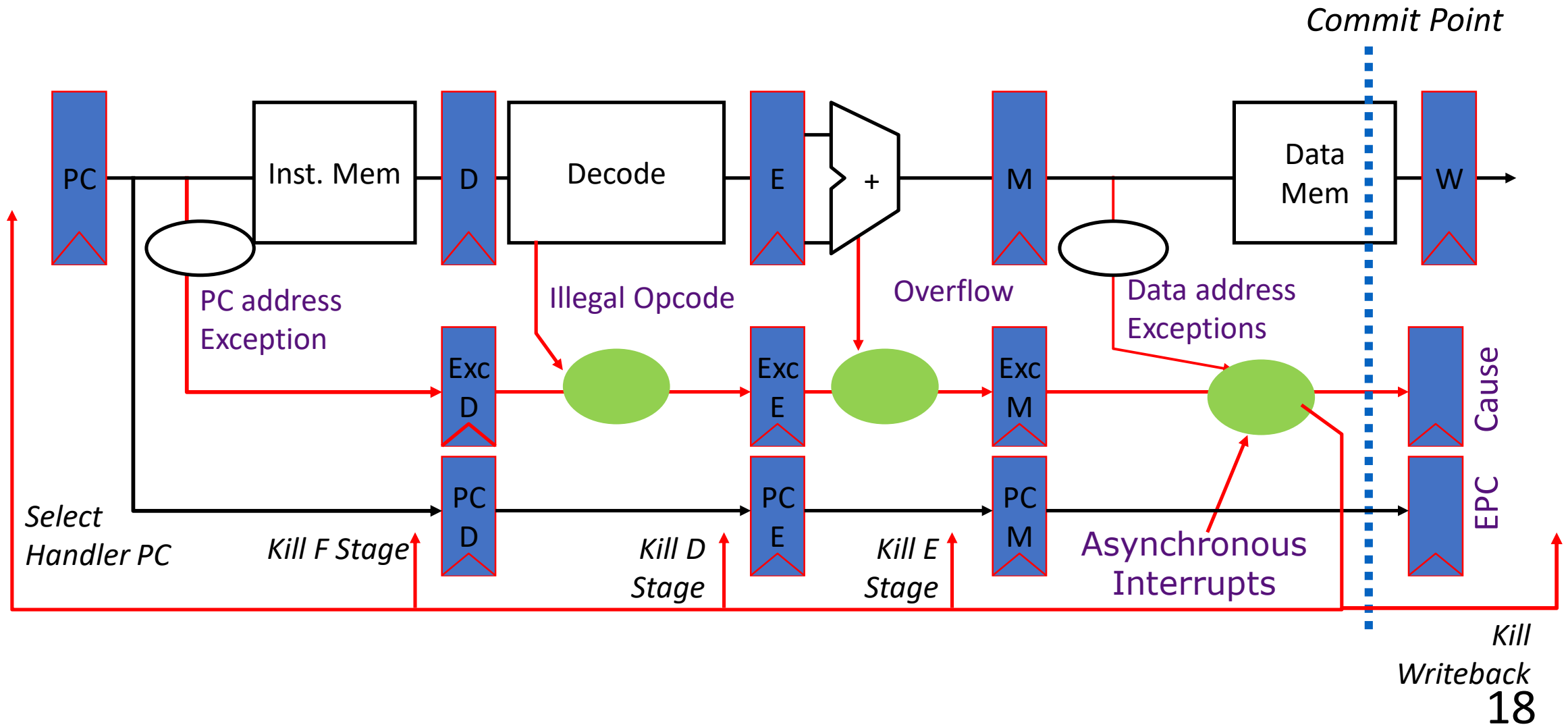
BTB is probed in
the fetch stage
along with the
direction predictor.
A hit in the BTB
means the PC is a
branch PC.

Exception handling and Pipelining



- When do we stop the pipeline for *precise* interrupts or exceptions?
- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Contd.



Contd.

- Hold exception flags in pipeline until commit point for instructions that will be killed
- 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

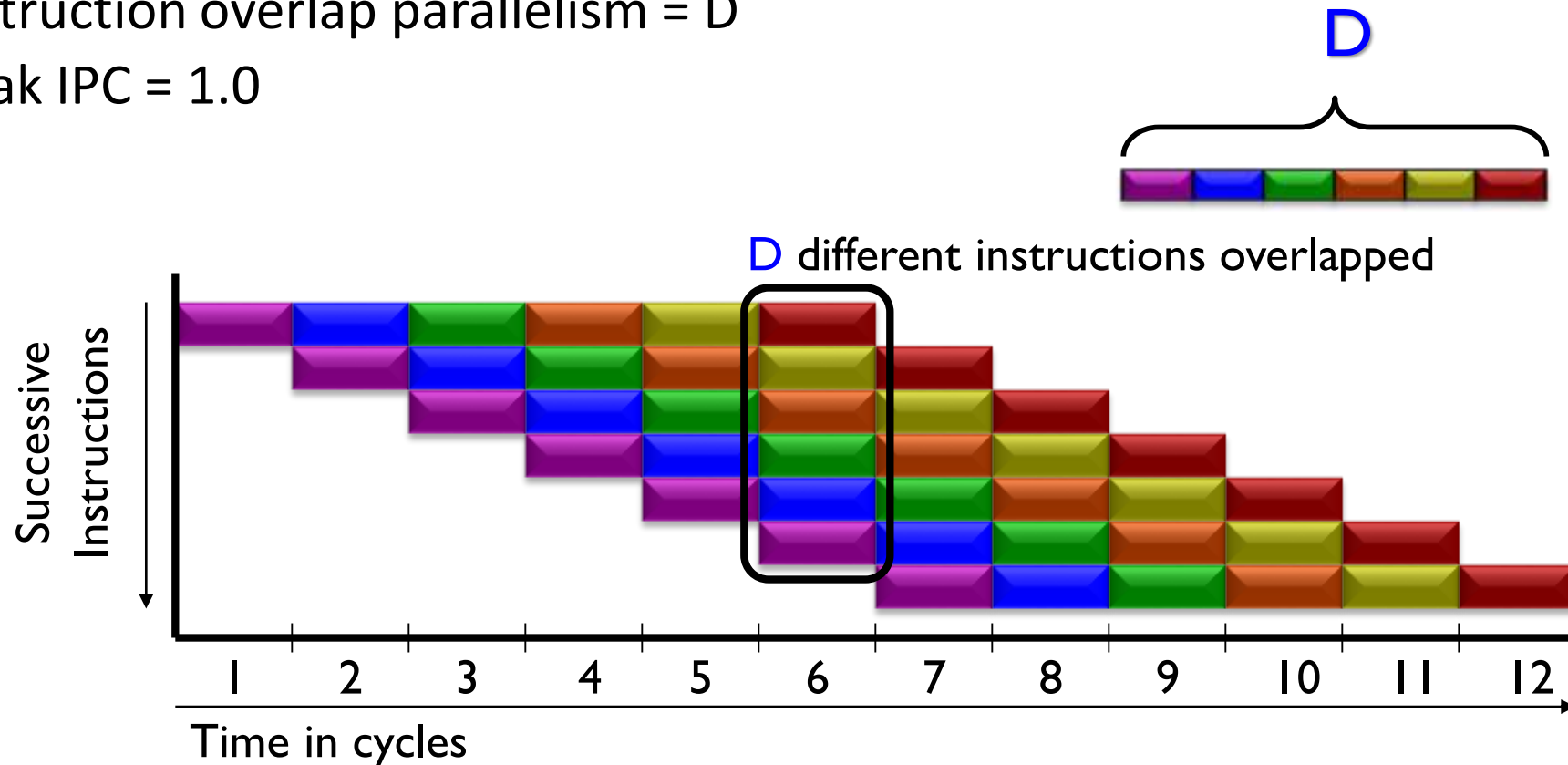
Beyond Scalar

- Scalar pipeline limited to $\text{CPI} \geq 1.0$
 - Can never run more than 1 insn per cycle
- “Superscalar” can achieve $\text{CPI} \leq 1.0$ (i.e., $\text{IPC} \geq 1.0$)
 - Superscalar means executing multiple insns in parallel

Instruction Level Parallelism (ILP)

- Scalar pipeline (baseline)

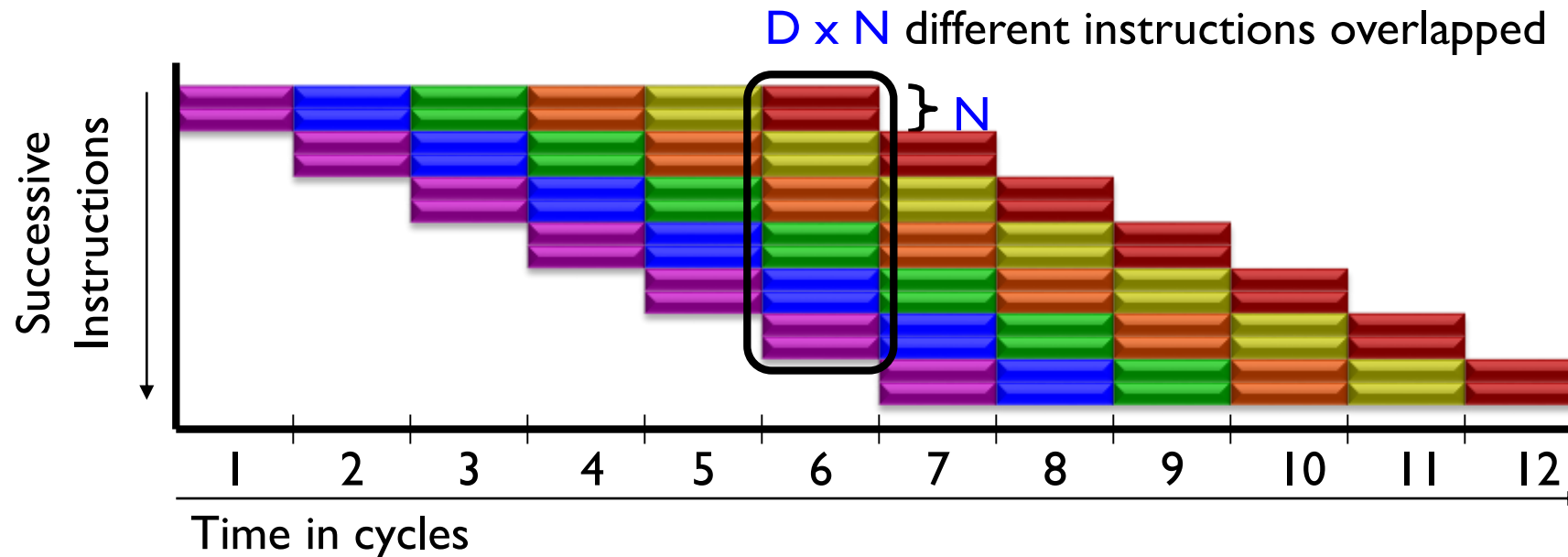
- Instruction overlap parallelism = D
- Peak IPC = 1.0



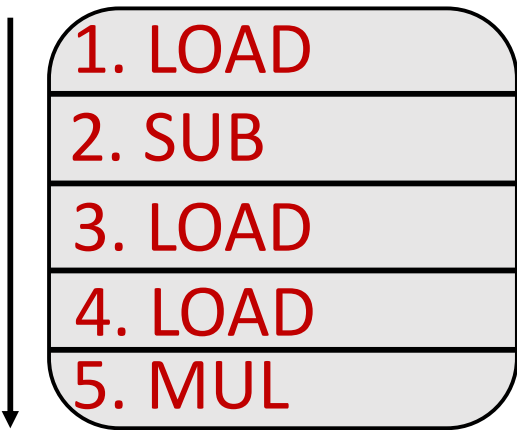
Superscalar Processor

- Superscalar (pipelined) Execution

- Instruction parallelism = $D \times N$
- Peak IPC = N per cycle

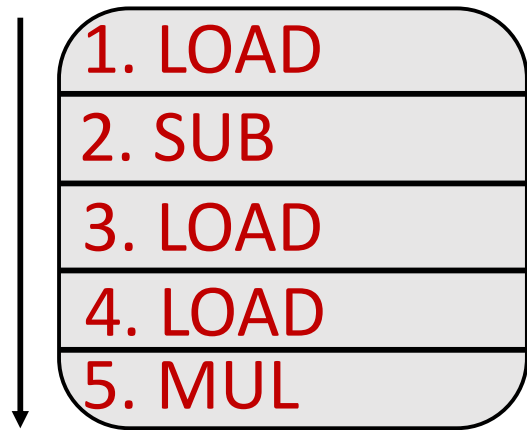


Modern Processors: In-order fetch

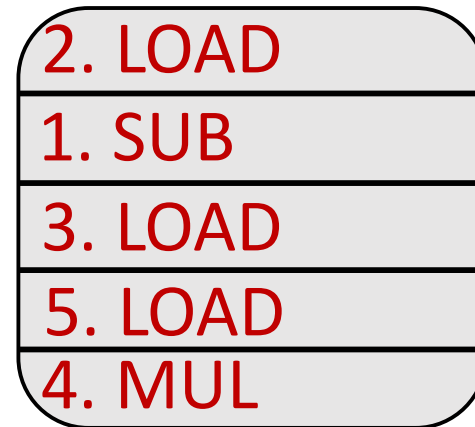


In-order Instruction Fetch
(Multiple fetch in one cycle)

Modern Processors: Out-of-order Execute

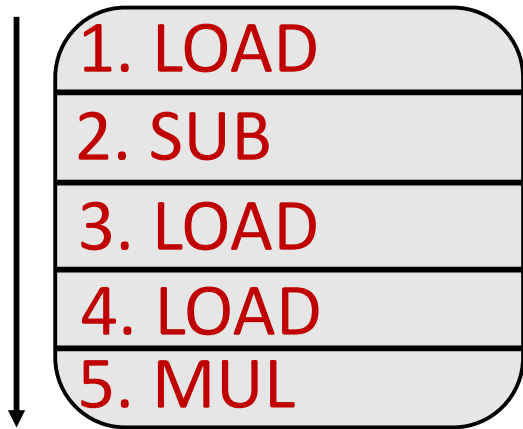


In-order Instruction Fetch
(Multiple fetch in one cycle)

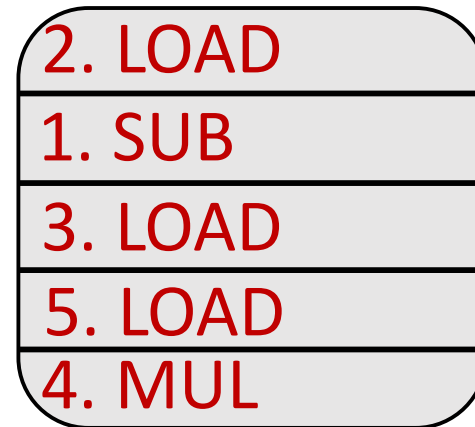


Out of order execute

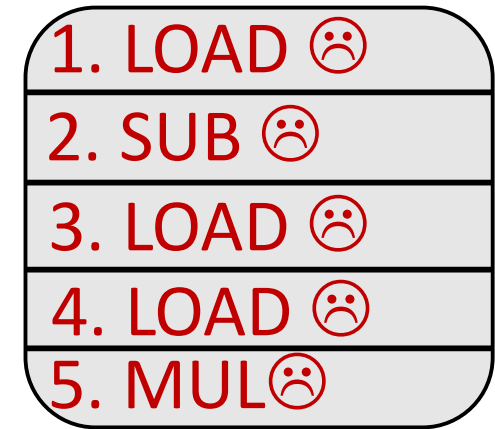
Modern Processors: In-order Commit



In-order Instruction Fetch
(Multiple fetch in one cycle)

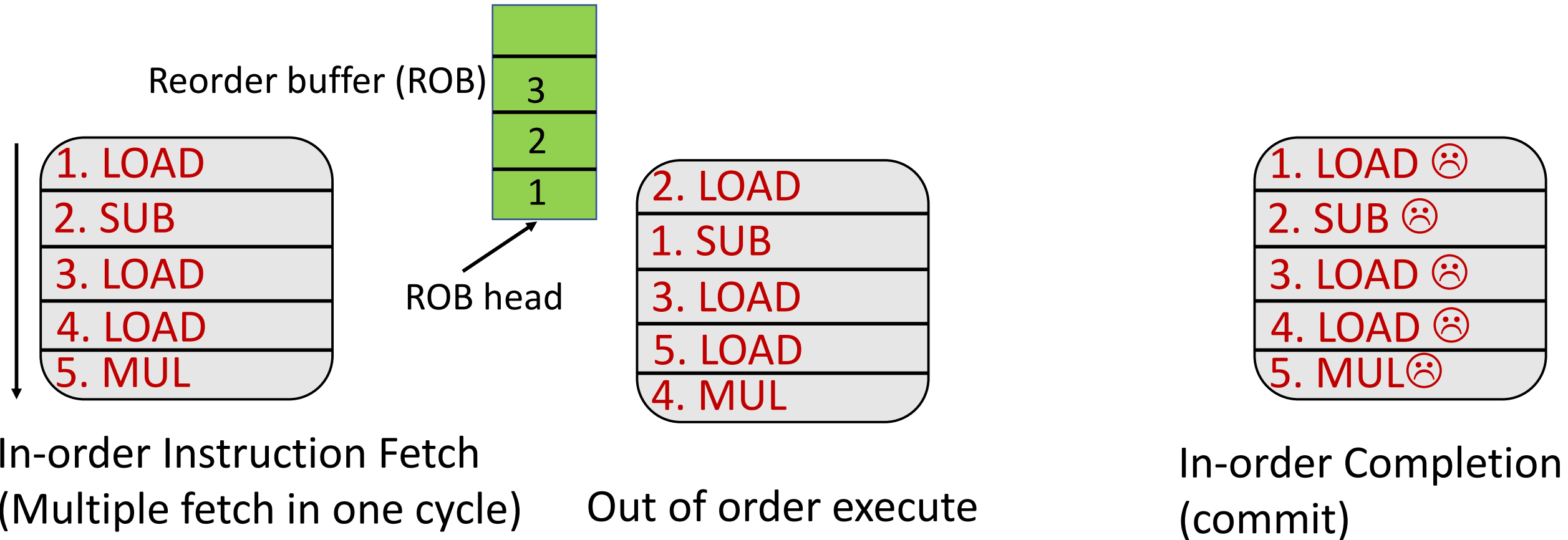


Out of order execute

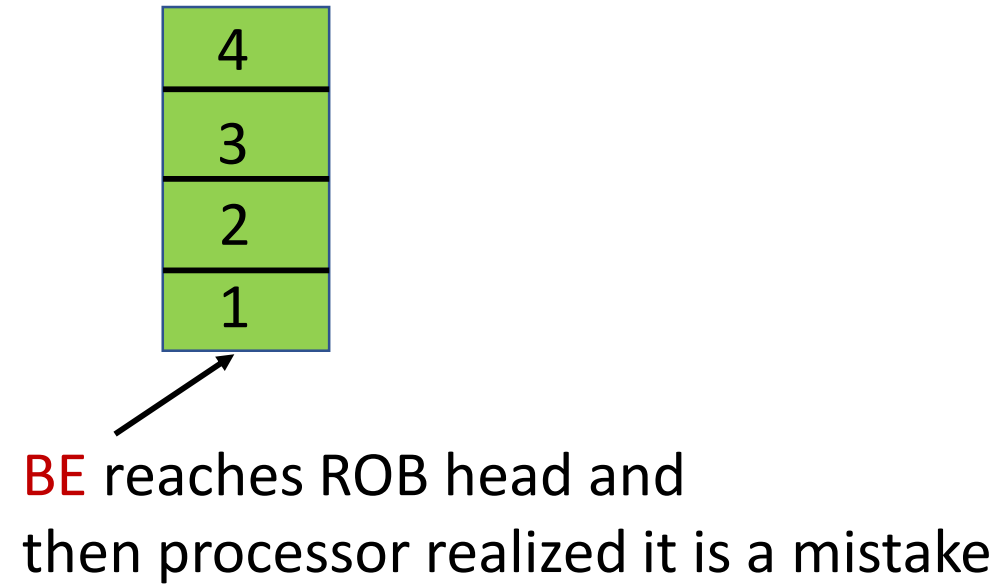
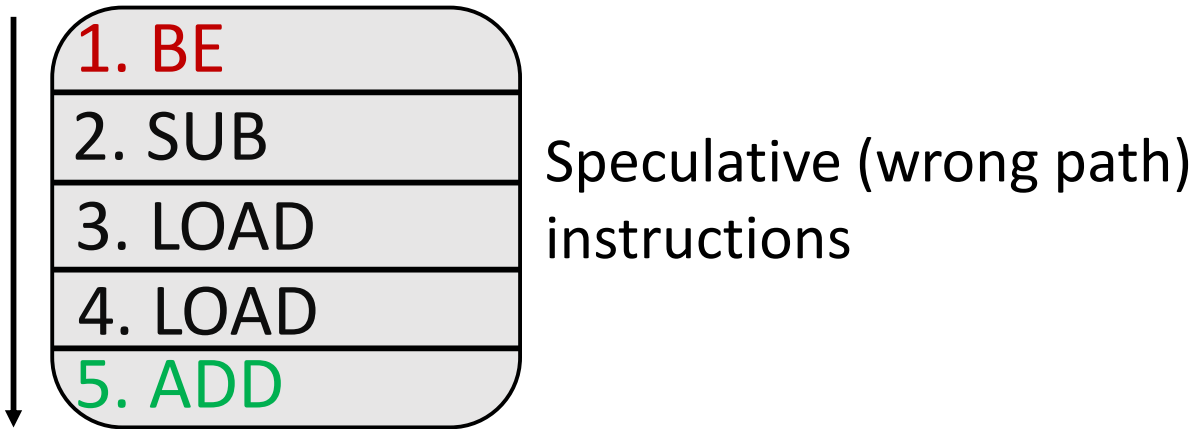


In-order Completion
(commit)

Modern Processors: In-order Commit



Modern Processors: Speculative Execution

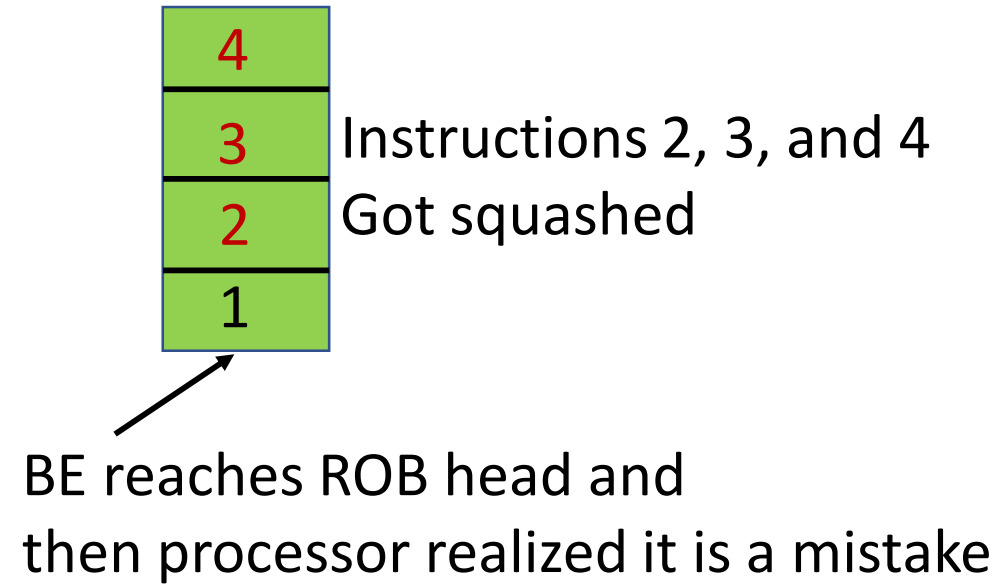
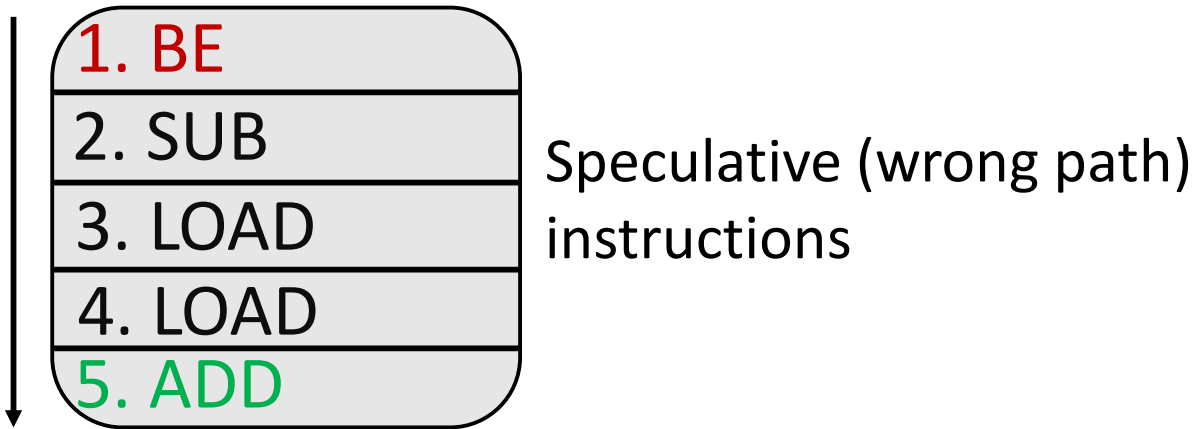


In-order Instruction Fetch
(Multiple fetch in one cycle)

Recent Intel processors have 512-entry ROB

Modern Processors: Speculative Execution

 Branch Predictor: TRUE



In-order Instruction Fetch
(Multiple fetch in one cycle)

Same happens in the case of a page fault, exception etc...

Transient instruction

- A speculative instruction that does not commit is a **transient** instruction

Speculative Execution-II

1. Proceed ahead despite unresolved dependencies using a prediction for an architectural or micro-architectural value



2. Maintain both old and new values on updates to architectural (and often micro-architectural) state

After speculation check

3. After sure that there was no mis-speculation and there will be no more uses of the old values, discard old values and just use new values

OR

3. In event of mis-speculation, dispose of all new values, restore old values, and re-execute from point before mis-speculation

Three Great tastes that tastes great together

- $\text{CPI} < 1$: Go for Superscalar
- Superscalar increases data hazards: Go for out-of-order
- Branch instructions are the problem: Speculative execution

Single-thread to SMT (Yes IBM, no Apple)

- What is a thread in hardware? It is a PC, Thread level parallelism

