# Computer Architecture. Week 10

# Pipeline Hazards and Their Solutions

Alexander Tormasov

Innopolis University

*a.tormasov@innopolis.ru*

October 28, 2021

## Topic of the lecture

- Hazards and its solutions

## Content of the class

# Recap: Pipelining (1/2)

| Instr. No. \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | IF | ID | EX | MEM |
| 5 | | | | | IF | ID | EX |

| | |
|---|---|
| **IF:** | Instruction Fetch |
| **ID:** | Instruction Decode |
| **EX:** | Execution |
| **MEM:** | Memory access |
| **WB:** | register Write Back |

- Pipelining attempts to maximize hardware usage by overlapping the execution stages of several different instructions.
- Pipelining offers amazing speedup.
  - The CPU throughput is dramatically improved, because several instructions can be executing concurrently.
  - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.

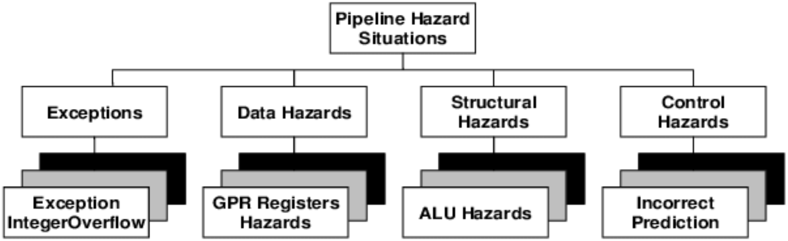
## Recap: Pipelining (2/2)

- The bad news: Instructions can interfere with each other – known as **hazards**
- Different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
- For Example: Instruction may require a result produced by an earlier instruction that is not yet complete

## Hazards

- Hazards prevent next instruction from execution during its designated clock cycle
- Hazards reduce the performance from the ideal speedup gained by pipeline
- Three types of hazards
  - Structural hazards
  - Data hazards
  - Control hazards

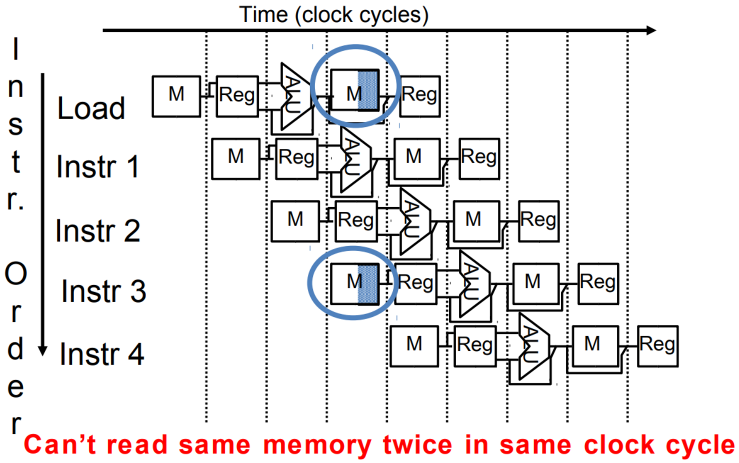
## Types of Hazards and Solutions



- GPR: General Purpose Registers

## Structural Hazards (1/3)

- Consider a Von Neumann architecture (same memory for instructions and data)



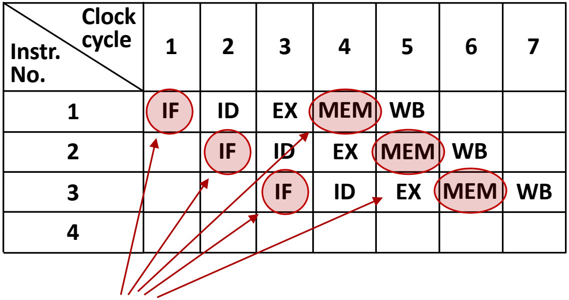
## Structural Hazards (2/3)

- **Structural Hazards:** Attempt to use the same resource by two or more instructions at the same time
- **Example:** Single Memory for instructions and data
  - Accessed by Instruction Fetch (IF) stage
  - Accessed at same time by Memory (MEM) stage
- **Solutions**
  - Delay the second access by one clock cycle
    **OR**
  - Provide separate memories for instructions and data
    - This is called a “Harvard Architecture”
    - Real pipeline processors have separate **caches**

## Structural Hazards (3/3)

- **Explanation**
  - The load instruction wants to access the memory to load data.
  - At the same time instruction 3 wants to fetch an instruction from memory.

## Structural Hazards – Solution (1/4)



| Instr. No. \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | | | | |

These stages require the same hardware resource – system memory

Consider the example above. Assumptions:

- System memory is the only memory unit (no cache, etc.)
- System memory does not support concurrent reads and writes, but only one operation at a time

## Structural Hazards – Solution (2/4)

| Instr. No. \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | ~~IF~~ | | | |

IF requires memory, which is however taken

Consider the example above. Assumptions:

- System memory is the only memory unit (no cache, etc.)
- System memory does not support concurrent reads and writes, but only one operation at a time

# Structural Hazards – Solution (3/4)

| Instr. No. \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | ☁ | ☁ | ☁ | IF |

Solution: Stall cycle for the 4th instruction

☁ - Stall (idle) cycle

## Structural Hazards – Solution (4/4)

| Instr. No. \ Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | ☁ | ☁ | ☁ | IF |

Solution: Stall cycle for the 4th instruction

☁ - Stall (idle) cycle

Note: In practice, some hardware optimizations are applied, such as multiple memory ports for concurrent reads and writes, etc.

## Structural Hazards – Solution (2/2)

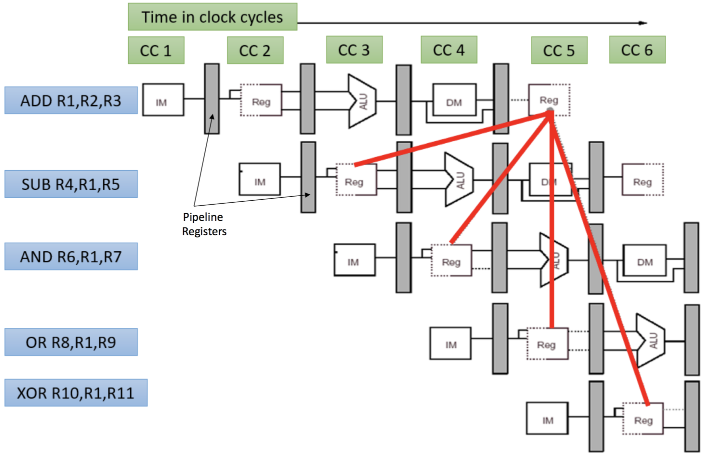
- Why the hardware design allows structural hazards to occur?
- **To reduce cost**
  - Pipelining all the functional units or duplicating them may be too costly
- **To reduce latency**
  - Making functional units pipelined sometimes may add delays (pipeline overhead $\longrightarrow$ registers.)

## Data Hazards (1/2)

- **Data Hazards:** It attempts to use data before it is ready
- Consider the execution of following instructions, on our pipeline example processor:

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```

## Data Hazards (2/2)



- The use of results from ADD instruction causes hazard since the register is not written until after those instructions read it.

## Data Hazards: Example Explanation (1/2)

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```

- **ADD instruction** writes the result in register R1 only at the write back stage, but SUB instruction reads the value during its instruction decode stage. This is called a data hazard.
- **SUB instruction** will read the wrong value and will use it.
- **AND instruction** is also affected by this hazard. The AND instruction that reads the registers in clock cycle 4 will receive the wrong results.

## Data Hazards: Example Explanation (2/2)

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
```

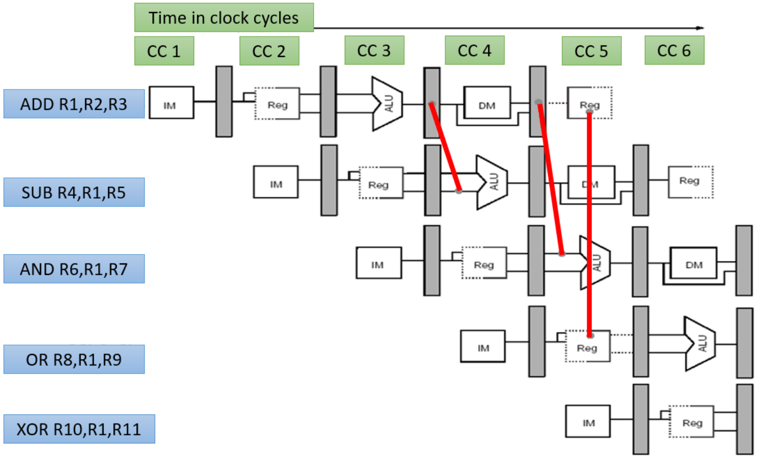
- **OR instruction** can also work fine without incurring a hazard, using a simple implementation technique.
- The technique is to perform the register file reads in the second half of the clock cycle and the writes in the first half.
- **XOR instruction** operates correctly, it reads its inputs (in clock cycle 6) after the ADD has written its result (in clock cycle 5).

## Data Hazards – Solution

- The data hazards can be solved by the following three solutions depending on the situation.
    - Forwarding
    - Stalling
    - Compiler Scheduling

## Data Hazards – Forwarding

- Eliminate the stalls for the hazard involving SUB and AND instructions using a technique called forwarding



## Data Hazards Classification (1/2)

- Depending on the order of read and write access in the instructions, data hazards could be classified as three types.
- Consider two instructions i and j, with i occurring before j.
- Possible data hazards are:
  - **RAW (Read After Write)**
  - **WAW (Write After Write)**
  - **WAR (Write After Read)**

## Data Hazards Classification (2/2)

- **RAW (Read After Write)** j tries to read a source before i writes to it, so j incorrectly gets the old value.
- **WAW (Write After Write)** j tries to write an operand before it is written by i.
  - The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination.
- **WAR (Write After Read)** j tries to write a destination before it is read by i, so the instruction i incorrectly gets the new value.

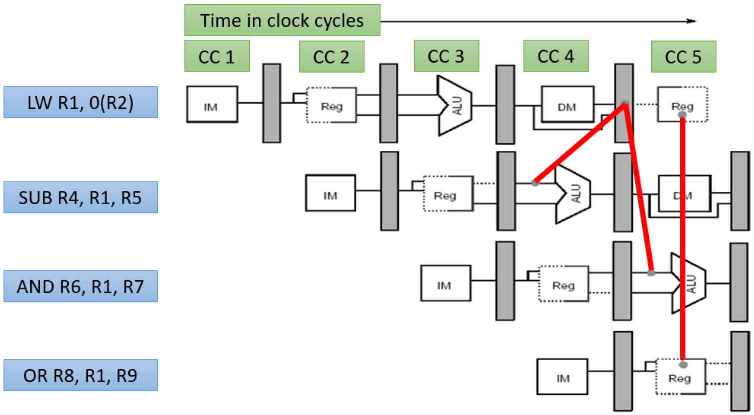
## Data Hazards – Requiring Stalls (1/3)

- Unfortunately not all data hazards can be handled by forwarding.
- Consider the following sequence:

```
LW R1, 0(R2)
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
```

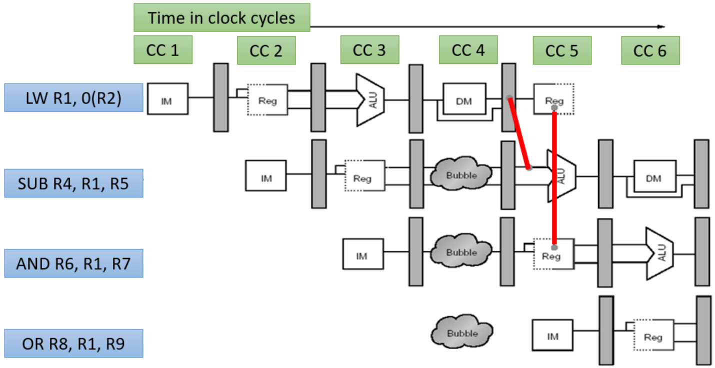
- The problem with this sequence is that the Load operation will not have data until the end of memory stage.

## Data Hazards – Requiring Stalls (2/3)



- The load instruction can forward the results to AND and OR instruction, but not to the SUB instruction since that would mean forwarding results in “negative” time .

## Data Hazards – Requiring Stalls (3/3)



- The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.
- This delay allows the value to be successfully forwarded onto the next clock cycle.

## Data Hazards – Compiler Scheduling

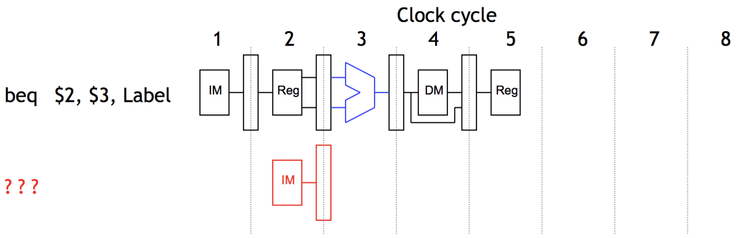
- Rather than just allow the pipeline to stall, the compiler could try to schedule the pipeline to avoid the stalls, by rearranging the code.
    - The compiler could try to avoid generating the code with a load followed by an immediate use of the load destination register.
    - This technique is called **pipeline scheduling** or **instruction scheduling** and it is a very used technique in modern compilers.

## Control Hazards (1/2)

- **Control Hazards:** It attempts to make branching decisions before branch condition is evaluated.
- Most of the work for a branch computation is done in the execution stage.
- The branch target address is computed.
- The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.

## Control Hazards (2/2)

- The branch decision cannot be made until the end of the execution stage.
- But we need to know which instruction to fetch next, in order to keep the pipeline running!
- This leads to what is called a control hazard.



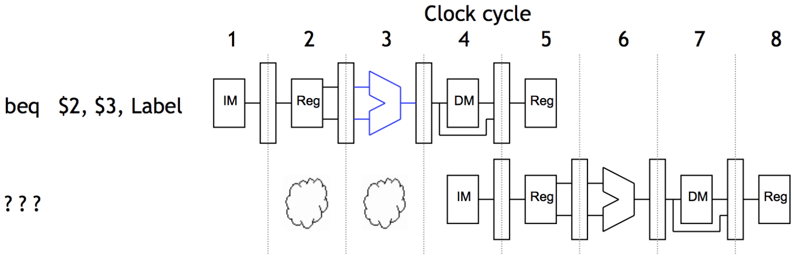
## Control Hazards – Solutions (1/6)

- The control hazards are handled by the following possible ways.
  - Stalls
  - Branch Prediction / Compiler directives (e.g. likely-unlikely)

# Control Hazards – Solution (2/6)

## Stalls

- Stalling is one possible solution.



- Here, we just stall until cycle 4, after we do make the branch decision.

## Control Hazards – Solution (3/6)

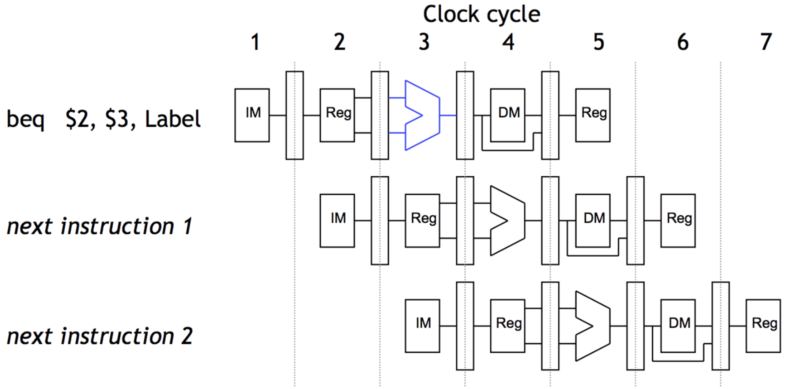
**Branch Prediction**

- Another approach is to guess whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is not taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we are correct, then there is no problem and the pipeline keeps going at full speed.
- To facilitate a correct branch prediction, several compiler directives are available (e.g. "likely/unlikely" directives)

## Control Hazards – Solution (4/6)

**Example:**

- We can get the branch equal results in cycle 3 and possible to make branch decision in cycle 4.



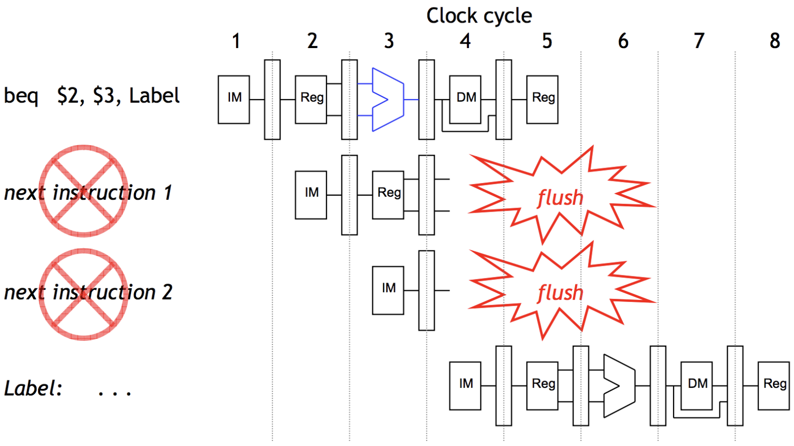
## Control Hazards – Solution (5/6)

- If our guess is wrong, then we would have already started executing two instructions incorrectly.
- We need to discard, or flush, those instructions and begin executing the right ones from the branch target address, `Label`.
- It is also known as **branch misprediction.**

INNOPOLIS UNIVERSITY

## Control Hazards – Solution (6/6)

### Branch Misprediction



## Branch Prediction (1/3)

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- Overall, branch prediction is worth it.
  - Mispredicting a branch means that two clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for every branch.
  - It is known as **Static Branch Prediction**

## Branch Prediction (2/3)

- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically – statistics are kept at runtime to determine the likelihood of a branch being taken.
  - It is known as **Dynamic Branch Prediction**

## Branch Prediction (3/3)

- The pipeline structure has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must be careful that instructions do not modify registers or memory before they get flushed.

## Control Hazards – Alternative Solution (1/2)

**A Possible Solution**

- We can actually decide the branch a little earlier, in instruction decoding stage instead of execution stage.
- We can add a small comparison circuit to the instruction decode stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.

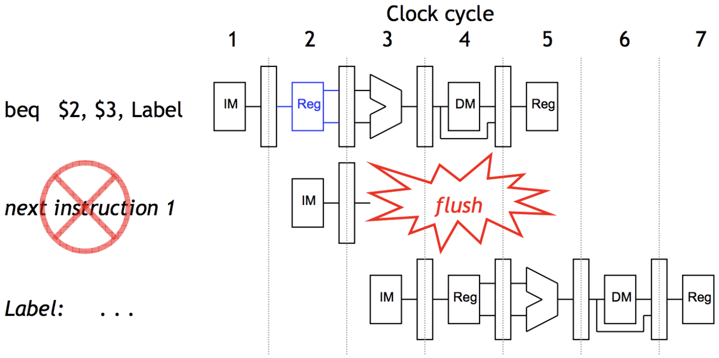
## Compiler Directives for Branch Prediction

**Optimize Code**

- When working with conditional code (if-else statements), we often know which branch is true and which is not. If compiler knows this information in advance, it can generate most optimized code.
- For Example: `likely` and `unlikely` compiler directives help the compiler know whether an if is usually going to be entered or skipped.
- Using it results in some performance improvements.

## Control Hazards – Alternative Solution (2/2)

### A Possible Solution

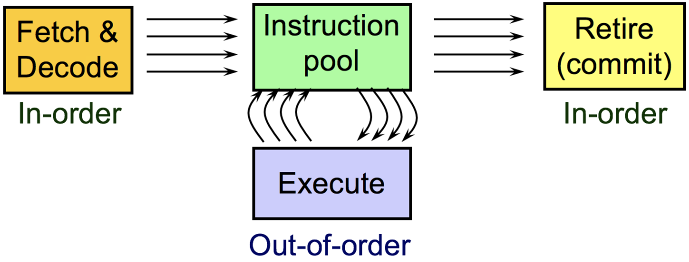


- We must flush one instruction, if the previous instruction is BEQ and its two source registers are equal.
- Flushing introduces a bubble into the pipeline, which represents the one cycle delay in taking the branch.

## A Superscalar CPU

- Duplicating hardware in one pipe stage would not help
  - For Example: have 2 ALUs
  - The bottleneck moves to other stages
- **Is superscalar good enough?**
  - A superscalar processor can fetch, decode, execute and write back 2 instructions in parallel
  - Can execute only independent instructions in parallel
  - But ... adjacent instructions are usually dependent
- **Solution: out-of-order execution**
  - Execute instructions based on "data flow" rather than program order
  - Still need to keep the semantics of the original program

## Out-of-order execution – General Scheme



- It creates new dependencies WAR and WAW.
- These are false dependencies
- There is no missing data
- Still prevent executing instructions out-of-order
- **Solution:** Register Renaming

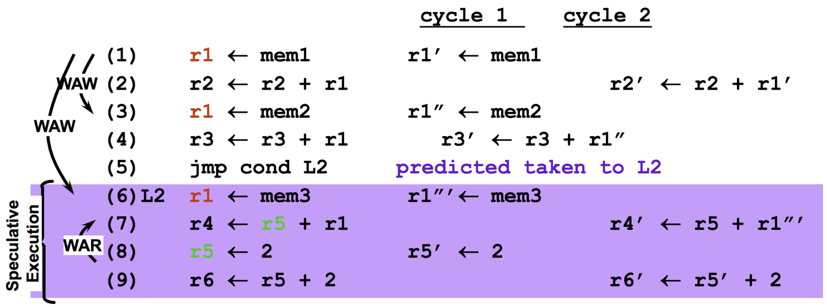
## Executing Beyond Branches

- So far we do not look for instructions ready to execute beyond a branch
  - Limited to the parallelism within a basic-block
- We would like to look beyond branches
  - But what if we execute an instruction beyond a branch and then it turns out that we predicted the wrong path?
- **Solution: Speculative Execution**

## Speculative Execution

- Hold a pool of all not yet executed instructions
- Fetch instructions into the pool from a predicted path
- Instructions for which all operands are ready can be executed
- An instruction may change the processor state (commit) only when it is safe

# Speculative Execution – Example



|  |  | cycle 1 | cycle 2 |
|---|---|---|---|
| (1) | r1 ← mem1 | r1′ ← mem1 |  |
| (2) | r2 ← r2 + r1 |  | r2′ ← r2 + r1′ |
| (3) | r1 ← mem2 | r1″ ← mem2 |  |
| (4) | r3 ← r3 + r1 | r3′ ← r3 + r1″ |  |
| (5) | jmp cond L2 | predicted taken to L2 |  |
| (6) L2 | r1 ← mem3 | r1″′ ← mem3 |  |
| (7) | r4 ← r5 + r1 |  | r4′ ← r5 + r1″′ |
| (8) | r5 ← 2 | r5′ ← 2 |  |
| (9) | r6 ← r5 + 2 |  | r6′ ← r5′ + 2 |

- Instructions 6-9 are speculatively executed
  - If the prediction turns wrong, they will be flushed
- If the branch was predicted taken
  - The instructions from the other path would be have been speculatively executed

## Summary

- Hazards
- Types of Hazards
- Structural Hazards
- Structural Hazards Solutions
- Data Hazards
- Data Hazards Solutions
- Control Hazards
- Control Hazards Solutions
- Superscalar CPU

## Acknowledgements

- This lecture was created and maintained by Muhammad Fahim, Giancarlo Succi, Alexander Tormasov, and Artem Burmyakov