

Pipelining

Wei Wang

Optional Readings from Textbooks

- “Computer Organization and Design,” Chapter 6 “Enhancing Performance with Pipelining.”
- “Computer Architecture: A Quantitative Approach,” Appendix C “Pipelining: Basic and Intermediate Concepts.”

Road Map

- Overview of Pipelining
- Structure Hazard
- Data Hazard
- Control Hazard
- Summary of Hazard Solutions
- Superscalar

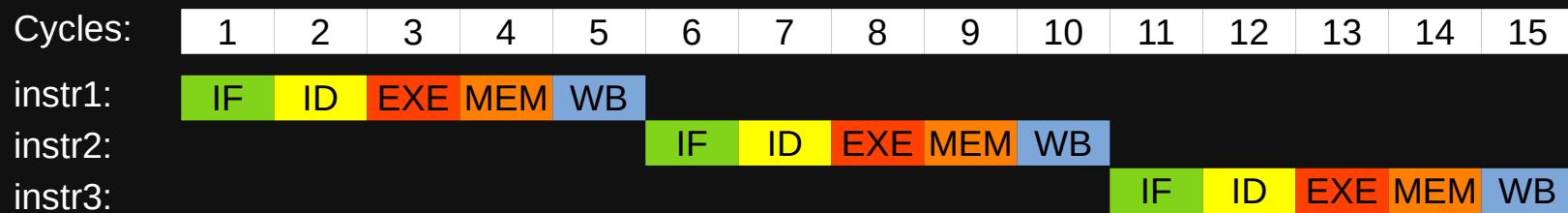
Overview of Pipelining

Overview of Pipelining

- Pipelining is a processor implementation technique in which multiple instructions are overlapped in execution.
 - CPU pipelining is exactly the same like factory pipelines.
- In fact, one of the major reason of breaking instruction execution into stages is to support pipelining.
 - Pipelining essentially overlaps different instructions working on different stages.
- Since 70's, nearly all computers have been pipelined.

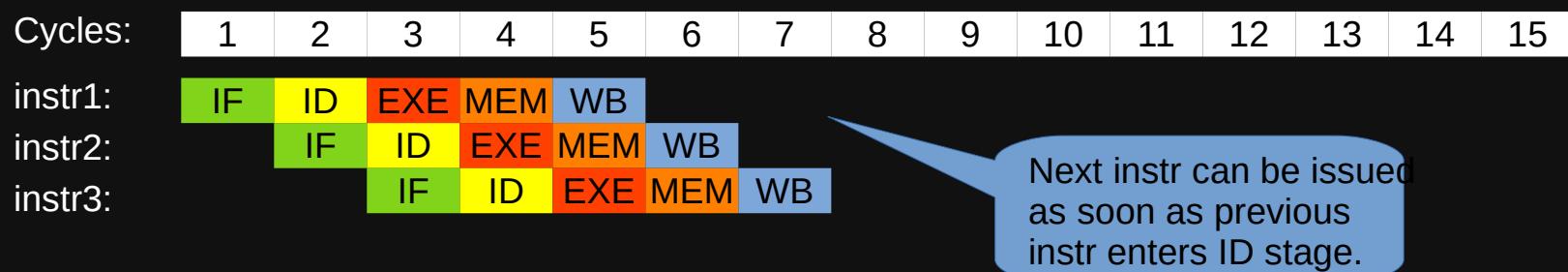
Pipelining vs No-pipelining

- For simplicity, we assume all instructions need 5 stages.
- Without pipelining, 3 instructions takes 15 cycles



– Average CPI is $15/3=5$ cycles.

- With pipelining, 3 instructions take only 7 cycles
 - Average CPI is only $7/3= 2.33$ cycles.

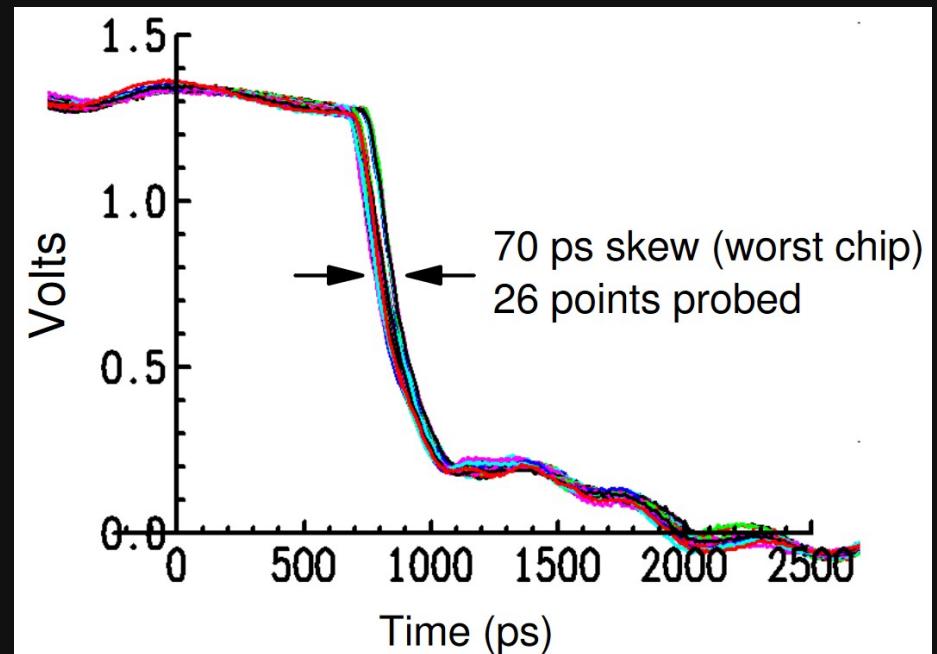


Ideal Pipelining Performance

- Without pipelining, assume instruction execution takes time T ,
 - Single Instruction latency is T
 - Throughput = $1/T$
 - M -Instruction Latency = $M \cdot T$
- If the execution is broken into an N -stage pipeline, ideally, a new instruction finishes each cycle
 - The time for each stage is $t = T/N$
 - Throughput = $1/t$
 - M -instruction Latency $\approx M \cdot t = M \cdot T/N \ll M \cdot T$
- Ideal CPI with pipelining is 1.

Ideal is impossible

- Two reasons why ideal is impossible
 - Pipeline overhead
 - Latches, clock skew, jitters
 - Prolong the time each stage takes to execute
 - Hazards
 - Situations that prevent the next instruction from executing in its designated clock cycle
 - Hardware resource contention, data dependency, branch instructions and exceptions
 - The major hurdle of pipelining



Clock Skew of IBM Power4
Taken from IBM Hot Chips 99
Presentation

Structural Hazard

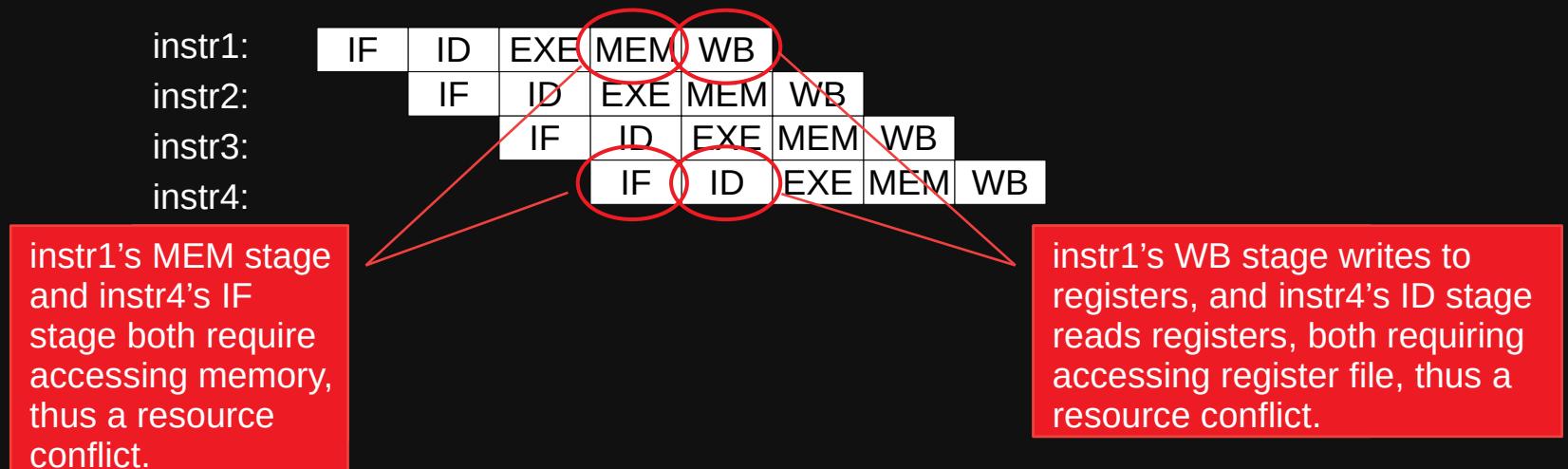
Structural Hazards

- When two different instructions want to use the same hardware resource in the same cycle (resource conflict).
- For example, in the following execution, instr1 and instr4 have two structural hazards at memory and register file.

instr1:	IF	ID	EXE	MEM	WB
instr2:		IF	ID	EXE	MEM WB
instr3:			IF	ID	EXE MEM WB
instr4:				IF	ID EXE MEM WB

Structural Hazards

- When two different instructions want to use the same hardware resource in the same cycle (resource conflict).
- For example, in the following execution, instr1 and instr4 have two structural hazards at memory and register file.



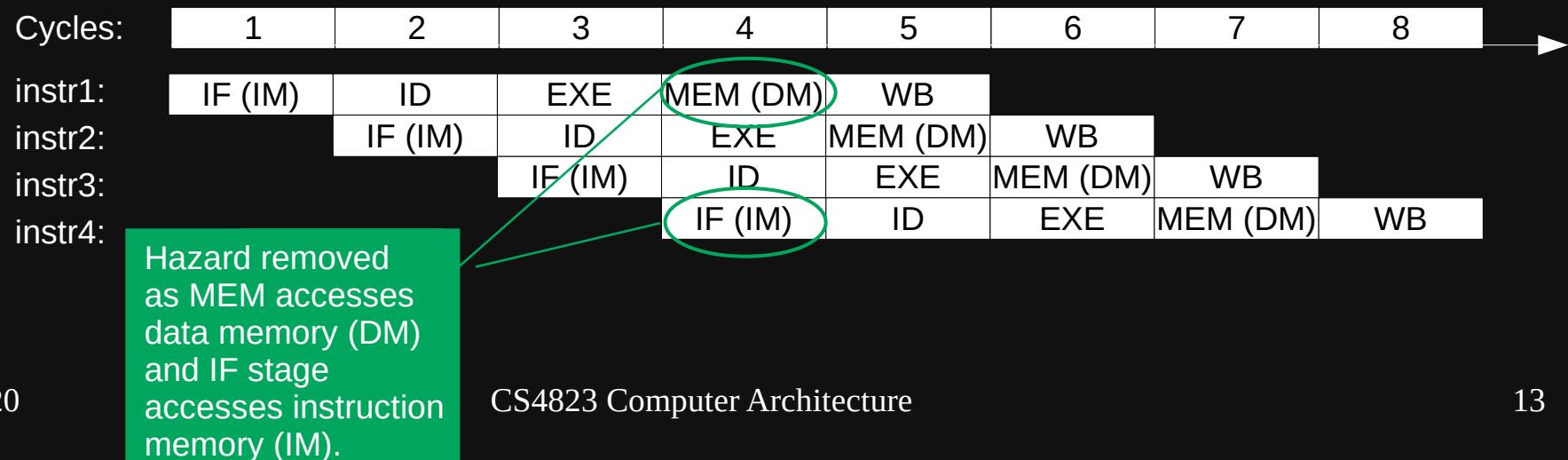
Structural Hazards Solutions

- Solution 1: separate hardware resources
 - For example, for the contention of MEM and IF stages
 - IF stage only reads instructions
 - MEM stage only accesses (read/write) data
 - We can then separate memory for instructions and data
 - This structural hazard is the one of the motivations of using separated instruction and data L1 caches.

Cycles:	1	2	3	4	5	6	7	8	→
instr1:	IF (IM)	ID	EXE	MEM (DM)	WB				
instr2:		IF (IM)	ID	EXE	MEM (DM)	WB			
instr3:			IF (IM)	ID	EXE	MEM (DM)	WB		
instr4:				IF (IM)	ID	EXE	MEM (DM)	WB	

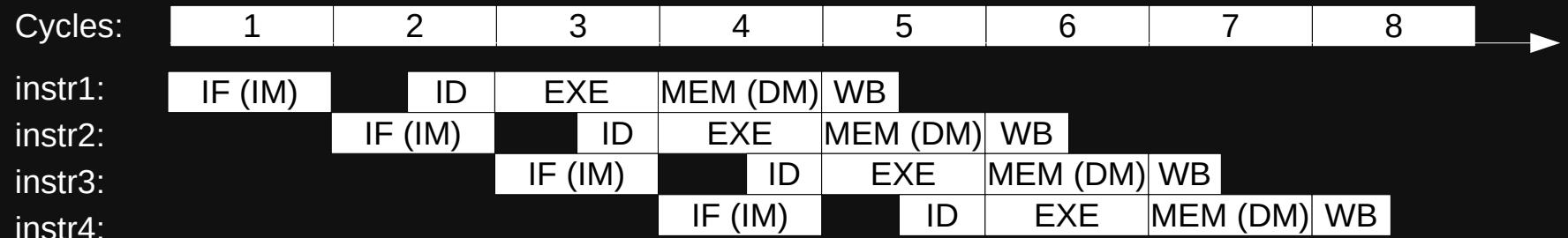
Structural Hazards Solutions

- Solution 1: separate hardware resources
 - For example, for the contention of MEM and IF stages
 - IF stage only reads instructions
 - MEM stage only accesses (read/write) data
 - We can then separate memory for instructions and data
 - This structural hazard is the one of the motivations of using separated instruction and data L1 caches.



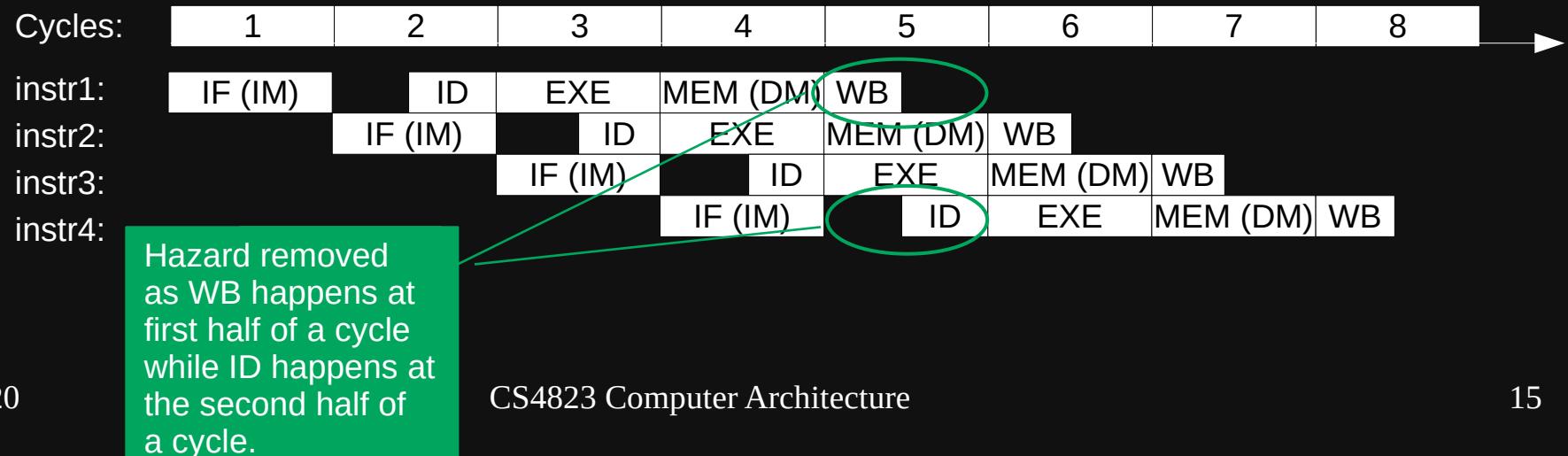
Structural Solutions cont'd

- Solution 2: let the accesses to the same hardware component happens at different part of a cycle
 - For example, for the contention of WB and ID stages
 - Let WB stage accesses registers at the first half of a cycle (e.g., at the falling edge of a cycle).
 - Let ID stage accesses registers at the second half of a cycle (e.g., at the rising edge of a cycle).
 - Alternatively, we can also make a stage multi-cycle and let the conflicted accesses happen at different cycles. This solution can accommodate more than two conflicting stages.



Structural Solutions cont'd

- Solution 2: let the accesses to the same hardware component happens at different part of a cycle
 - For example, for the contention of WB and ID stages
 - Let WB stage accesses registers at the first half of a cycle (e.g., at the falling edge of a cycle).
 - Let ID stage accesses registers at the second half of a cycle (e.g., at the rising edge of a cycle).
 - Alternatively, we can also make a stage multi-cycle and let the conflicted accesses happen at different cycles. This solution can accommodate more than two conflicting stages.



Structural Solutions cont'd

- Solution 3: Duplicate resources
 - Duplicate the contended resources. E.g., add more ALUs, if ALUs are contended.
 - Useful for cheap resources, frequent cases
 - Separate ALU/PC adders, Reg File Ports
 - Advantage: does not increase CPI
 - Disadvantage: increases cost, possibly increases cycle time.

Structural Solutions cont'd

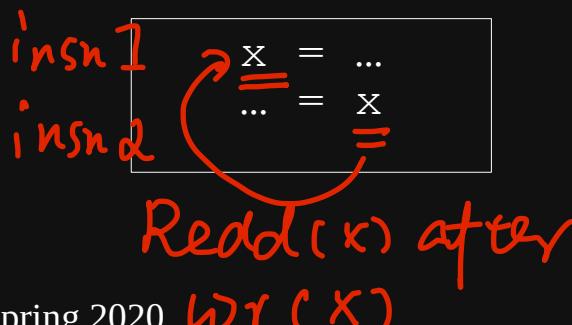
- Good news
 - Structural hazards don't occur as long as each instruction uses a resource
 - At most once
 - Always in the same pipeline stage
 - For one cycle
 - RISC ISAs are designed with this in mind, reduces structural hazards
 - For assignments and exams on simple pipelining (i.e., not OoO), always assume no structural hazards.

Data Hazards

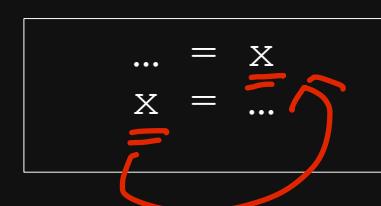
Data Hazards

- Data hazards when an instruction depends on the result of a previous instruction that exposes overlapping of instructions
- Three types of Data Hazards (dependencies)
 - Read-After-Write (RAW)
 - True data-dependence (Most important)
 - Write-After-Read (WAR)
 - Write-After-Write (WAW)

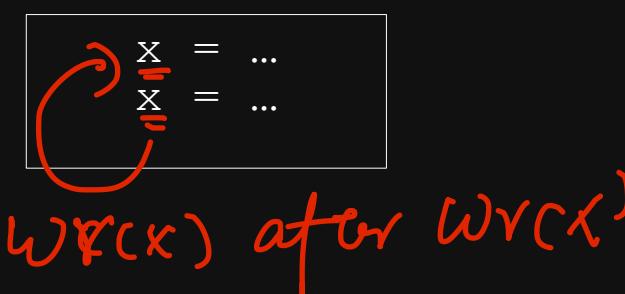
Read-after-Write (RAW)



Write-after-Read (WAR)

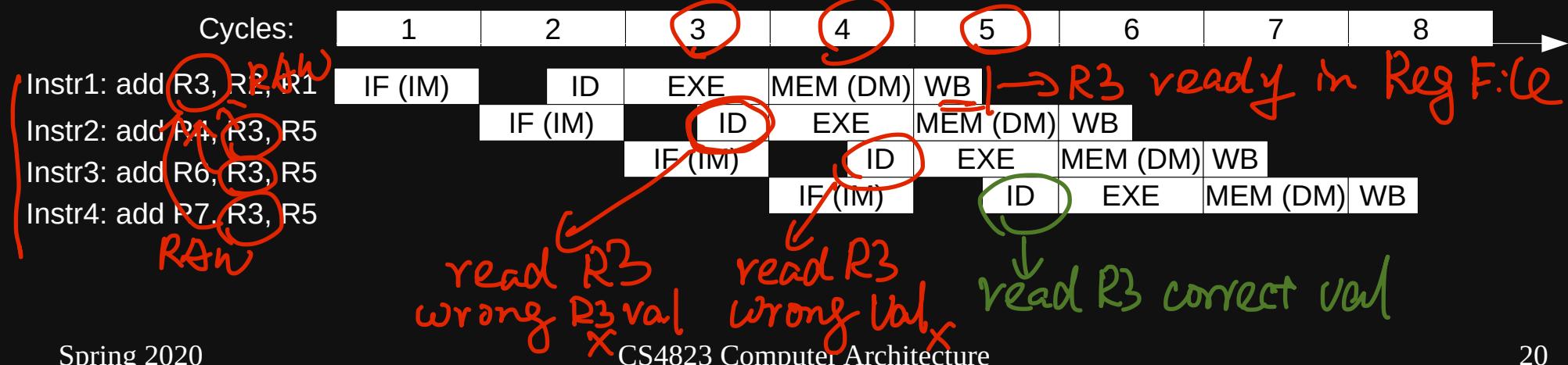


Write-after-Write (WAW)



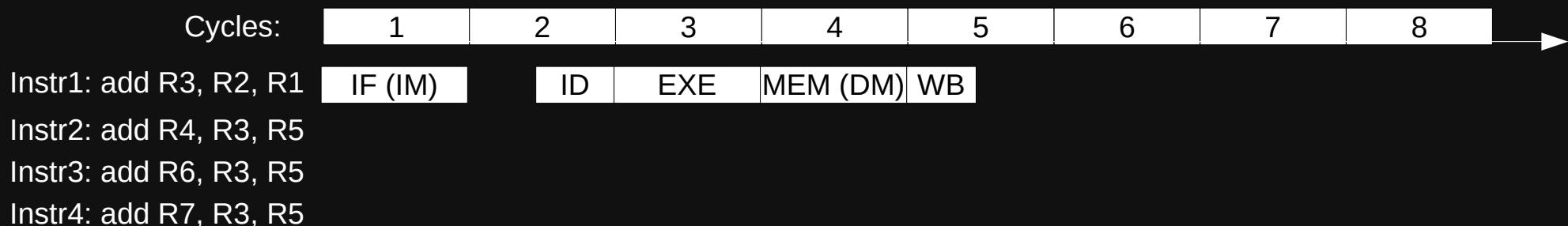
Read-After-Write (RAW) Hazard

- For example, for the following code:
 - Instr1 writes R3 in cycle 5
 - However, instr2 needs R3 in cycle 3 and instr3 needs R3 in cycle 4, before the latest R3 is written.
 - Instr4 reads R3 at the second half of cycle 5. Therefore, there is no data hazard for instr4.



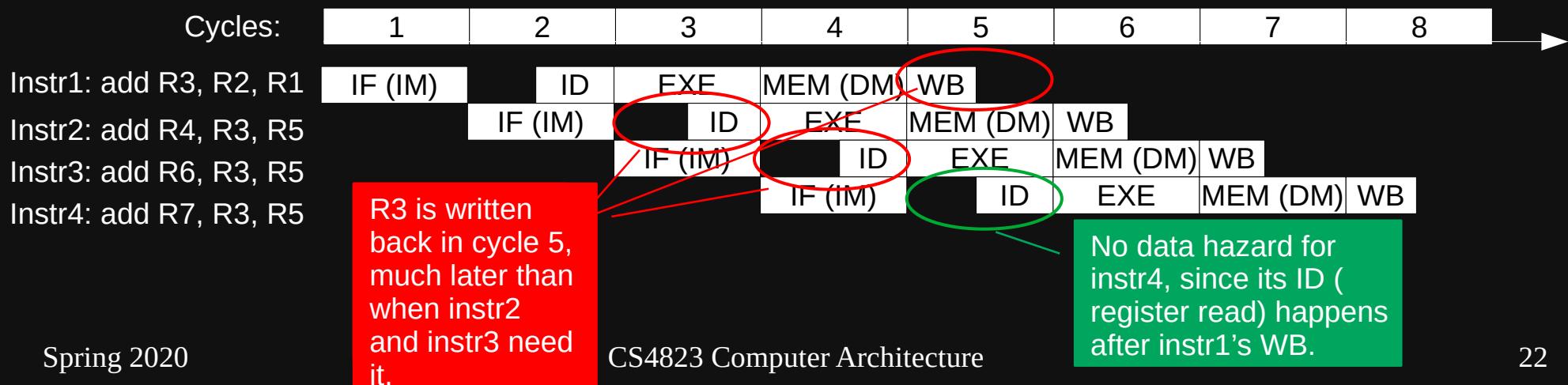
Read-After-Write (RAW) Hazard

- For example, for the following code:
 - Instr1 writes R3 in cycle 5
 - However, instr2 needs R3 in cycle 3 and instr3 needs R3 in cycle 4, before the latest R3 is written.
 - Instr4 reads R3 at the second half of cycle 5. Therefore, there is no data hazard for instr4.



Read-After-Write (RAW) Hazard

- For example, for the following code:
 - Instr1 writes R3 in cycle 5
 - However, instr2 needs R3 in cycle 3 and instr3 needs R3 in cycle 4, before the latest R3 is written.
 - Instr4 reads R3 at the second half of cycle 5. Therefore, there is no data hazard for instr4.



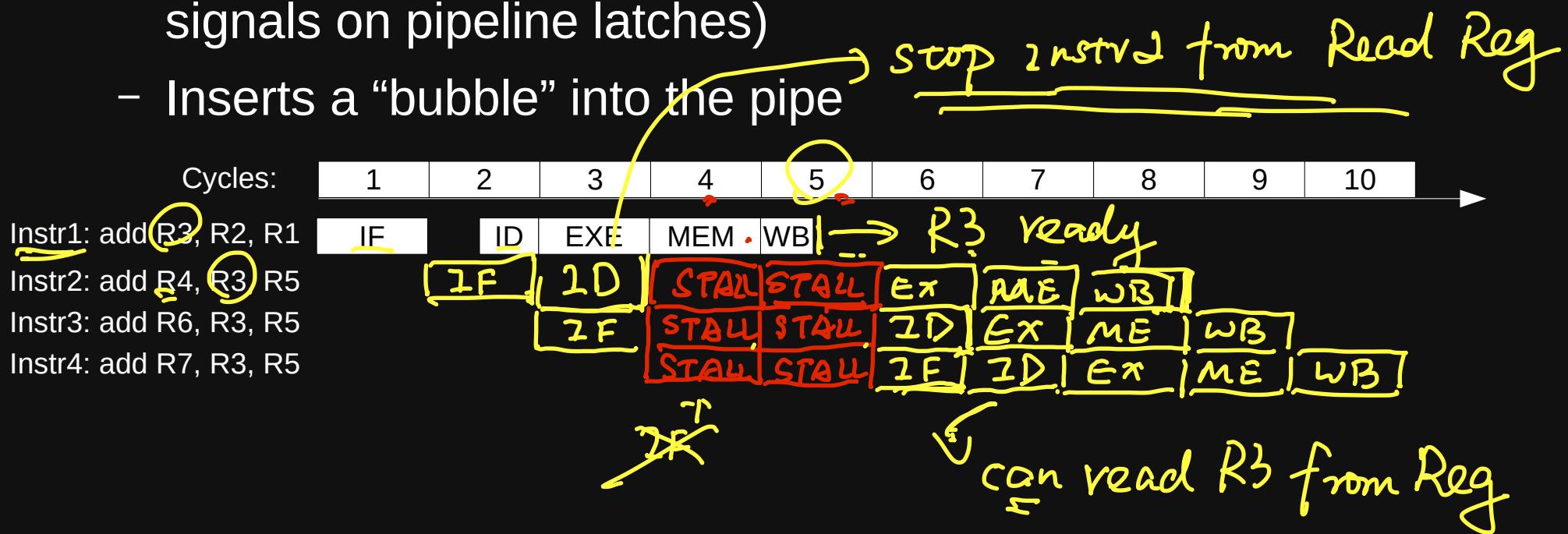
RAW Hazard Solutions

- Before solving RAW hazard, pipeline needs to be able to identify the data dependency that causes RAW hazard.
- The hardware component for detecting data hazard is called pipeline interlock
 - Need to keep register ID's along with pipestages
 - Use comparators to check for hazards



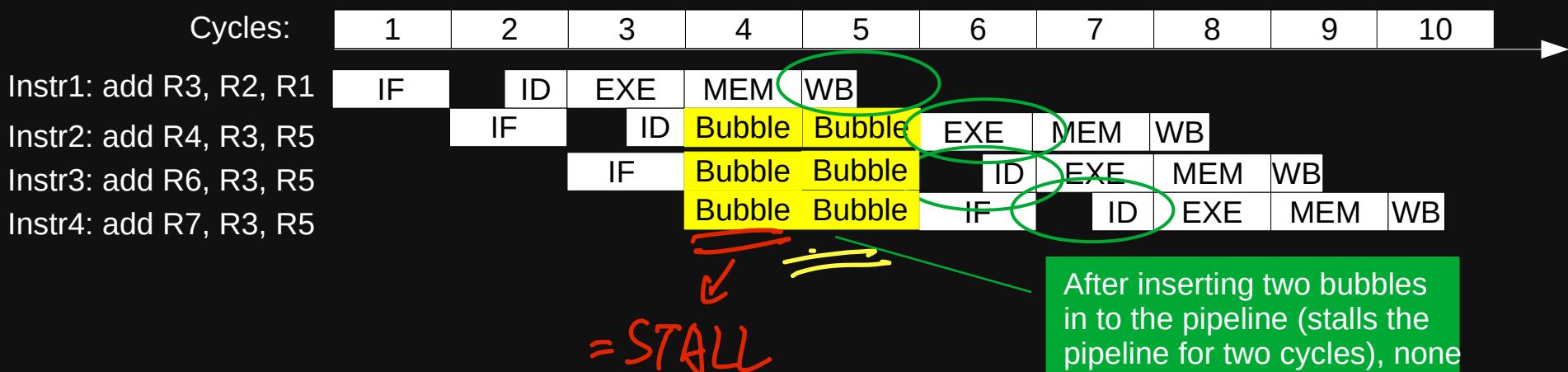
RAW Hazard Solutions cont'd

- Solution 1 (simplest): Stall the pipeline
 - Stops some instructions from executing
 - Make them wait for older instructions to complete
 - Simple implementation to “freeze” (de-assert write-enable signals on pipeline latches)
 - Inserts a “bubble” into the pipe



RAW Hazard Solutions cont'd

- Solution 1 (simplest): Stall the pipeline
 - The downside of stalling is obviously the longer execution time. For example, for the following four instructions, stalling the pipeline prolongs the execution time from 8 cycles to 10 cycles.
 - Given that applications have a lot of internal dependencies, this performance penalty can be quite serious.
 - The advantages of stalling are
 - Easy and low cost to implement
 - Can solve nearly all pipeline hazards, including structural, data and control hazards.
 - Stalling is the last solution to any hazards that cannot be solved by other solutions.



RAW Hazard Solutions cont'd

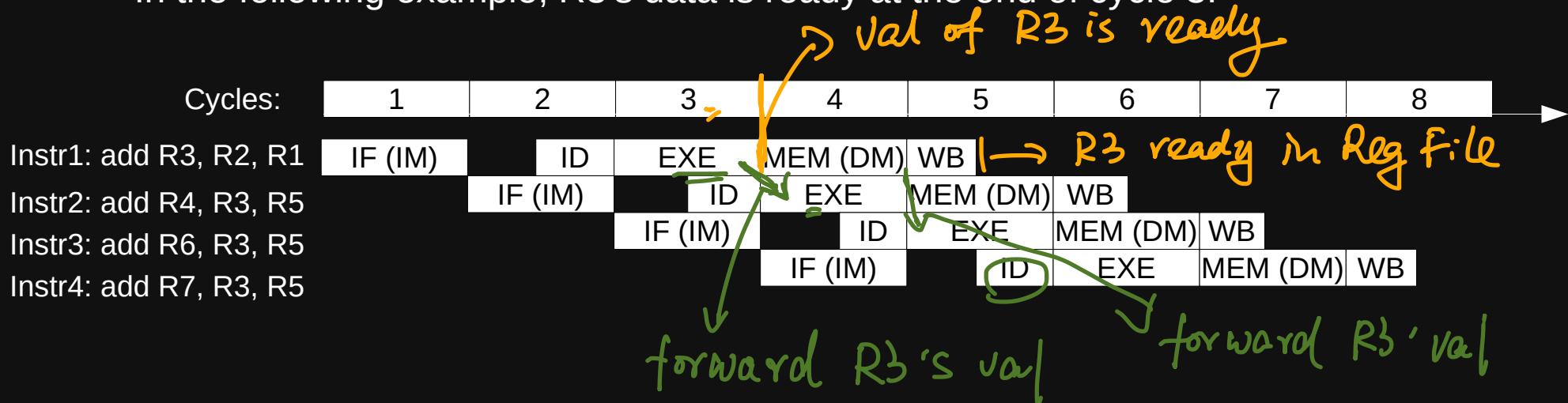
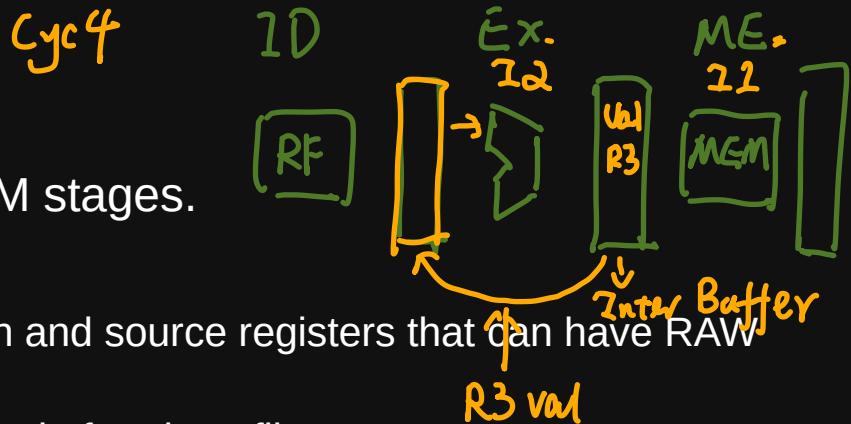
- Solution 2: Bypass/forwarding

- Data is usually ready at the end of EXE or MEM stages.

- Basic idea,

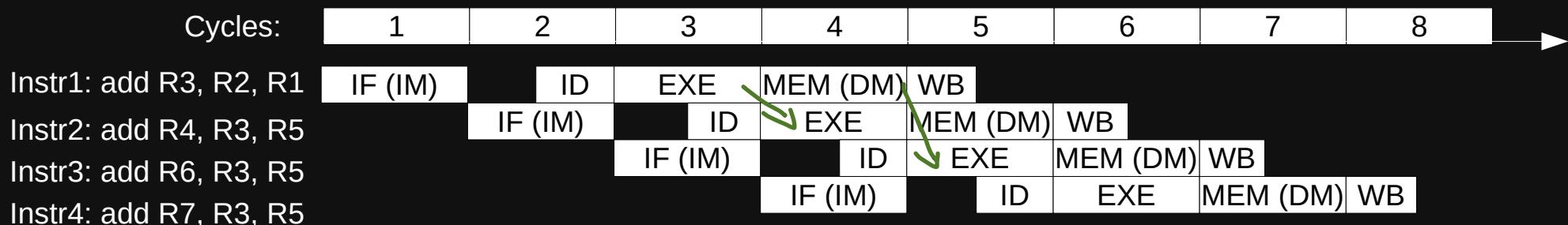
- add comparator for each combination of destination and source registers that can have RAW hazards.
- Add muxes to datapath to select proper value instead of register file.
- The forwarded data are from inter-stage registers.
- Only stall when absolutely necessary

- In the following example, R3's data is ready at the end of cycle 3.



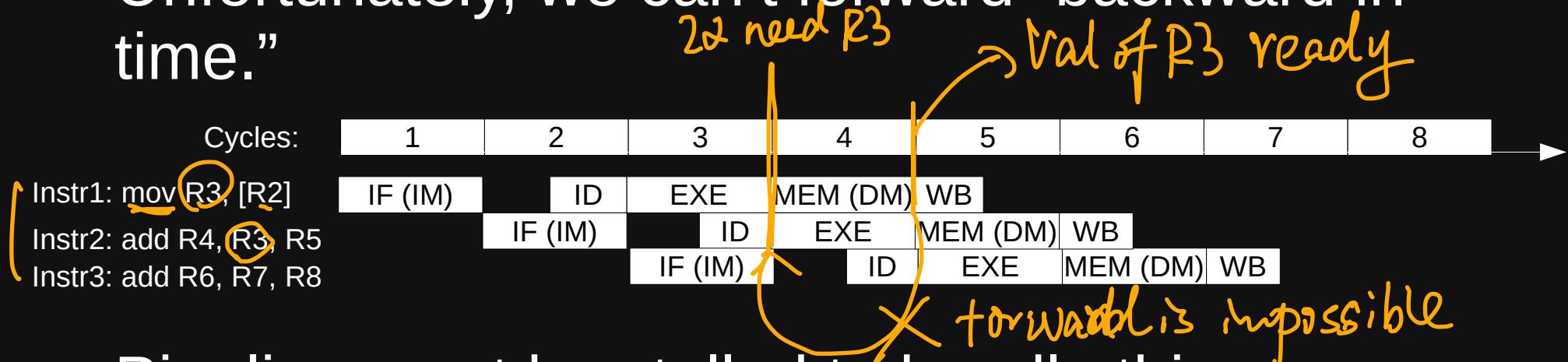
RAW Hazard Solutions cont'd

- Solution 2: Bypass/forwarding
 - In the following example, we can start forwarding the value of R3 at the end of cycle 3 to instr2 and instr3.
 - Advantage: no negative impact on performance. E.g., in the following example, it still takes 8 cycles to execute the four instructions.
 - Disadvantage:
 - Fairly complex change to the control unit and other pipeline resources
 - Cannot solve all RAW hazards (forwarding is only doable when the value is ready).

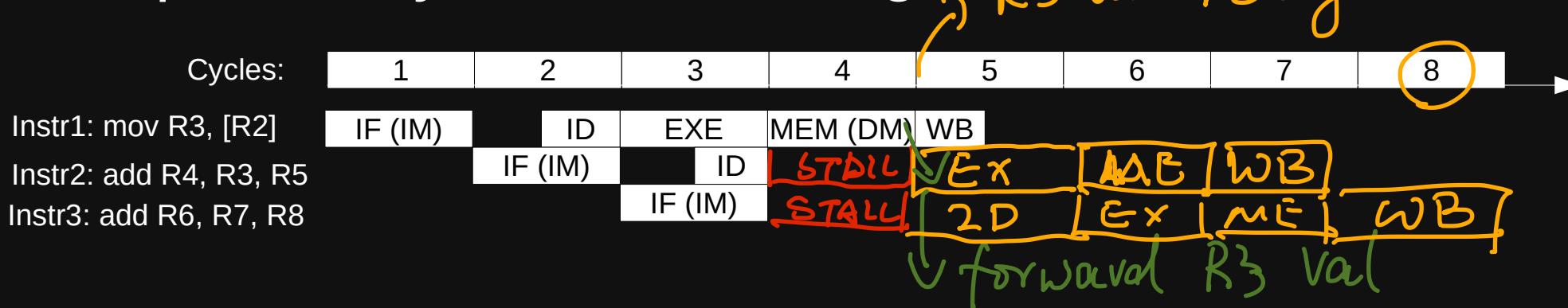


Load-Use Hazards

- Unfortunately, we can't forward "backward in time."

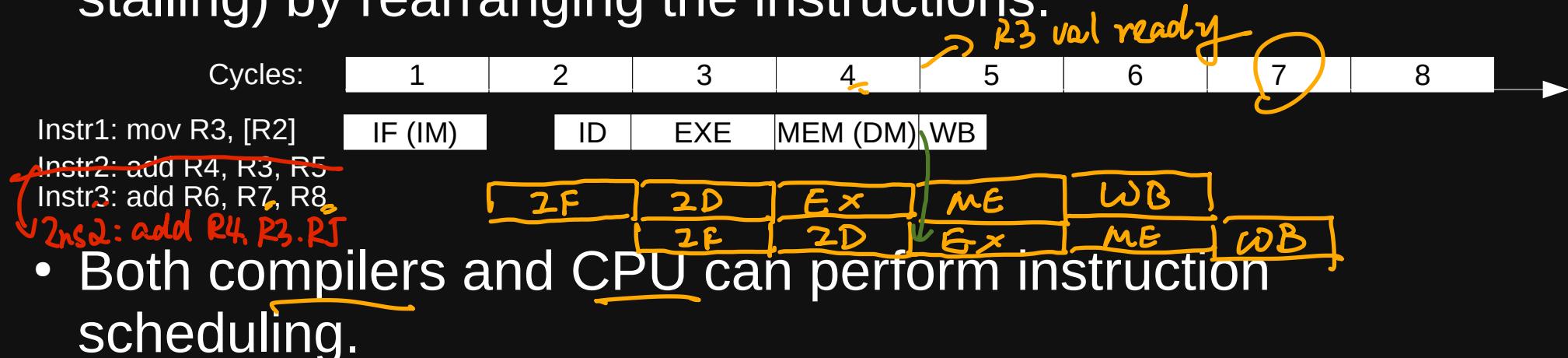


- Pipeline must be stalled to handle this dependency with forwarding.



Instruction Scheduling

- Alternatively, we can remove the bubble (i.e., avoid stalling) by rearranging the instructions.



- Both compilers and CPU can perform instruction scheduling.

- For compiler, it is more challenging to identify the opportunities of instruction scheduling due to unknown memory addresses at compilation time (hence unknown dependences)
- CPU can perform dynamic instruction scheduling with out-of-order (OoO) execution.

Other Data Hazards: WARs and WAWs

(WAW)

- Write-After-Read (WAR) and Write-After-Write Hazards
 - Can't happen in our simple 5-stage pipeline because the order of the instructions is always strictly followed
- But they may happen when the order of instructions are changed. E.g.,
 - in OoO executions,
 - if the stages require different execution time
 - When some instructions have cache misses
- Nonetheless, WAR and WAW can also be handled using stalls and forwarding.

- What about RARs?

Control Hazards

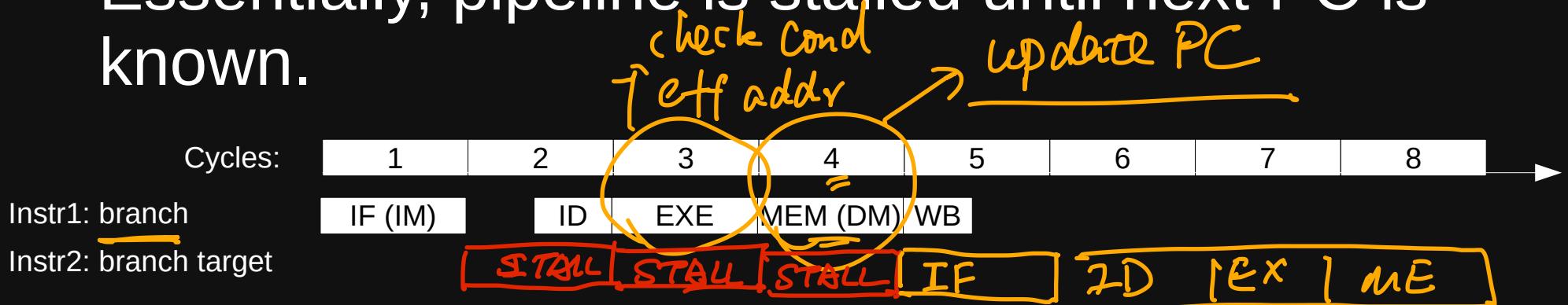
Control Hazards

- Control hazards happens when there are PC-modifying instructions (branch, jump, etc) that prevent knowing which instruction to execute until branch target is computed and conditions are checked.
- In our simple model, the PC for the instruction after a branch is only known at the end of MEM stage.

↓ stage RISC

An Example of Control Hazards

- Branch instruction prevents the issue of next instruction until MEM stage.
- Essentially, pipeline is stalled until next PC is known.



Control Hazard Solutions

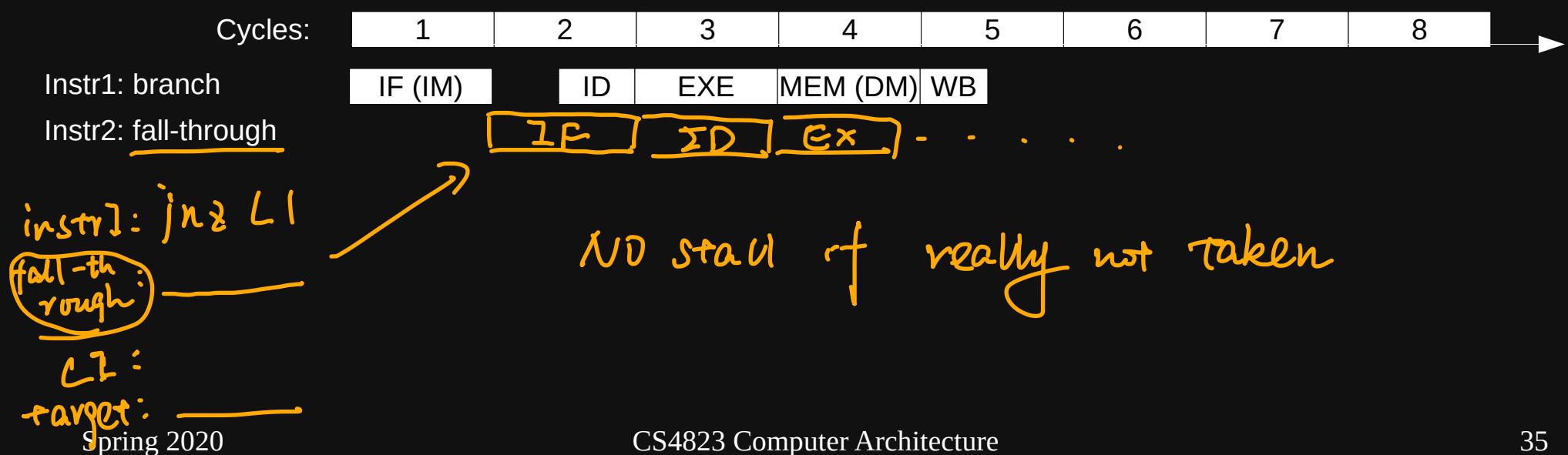
- Solution 1: Fast branch resolution
 - Add extra adder to ID stage to compute branch target
 - Only works for simple conditional jump (compare to 0).
 - For some conditional jumps in CISC ISA that requires comparing two registers, the comparison can be slow.
check cond, effaddr, update PC



Control Hazard Solutions cont'd

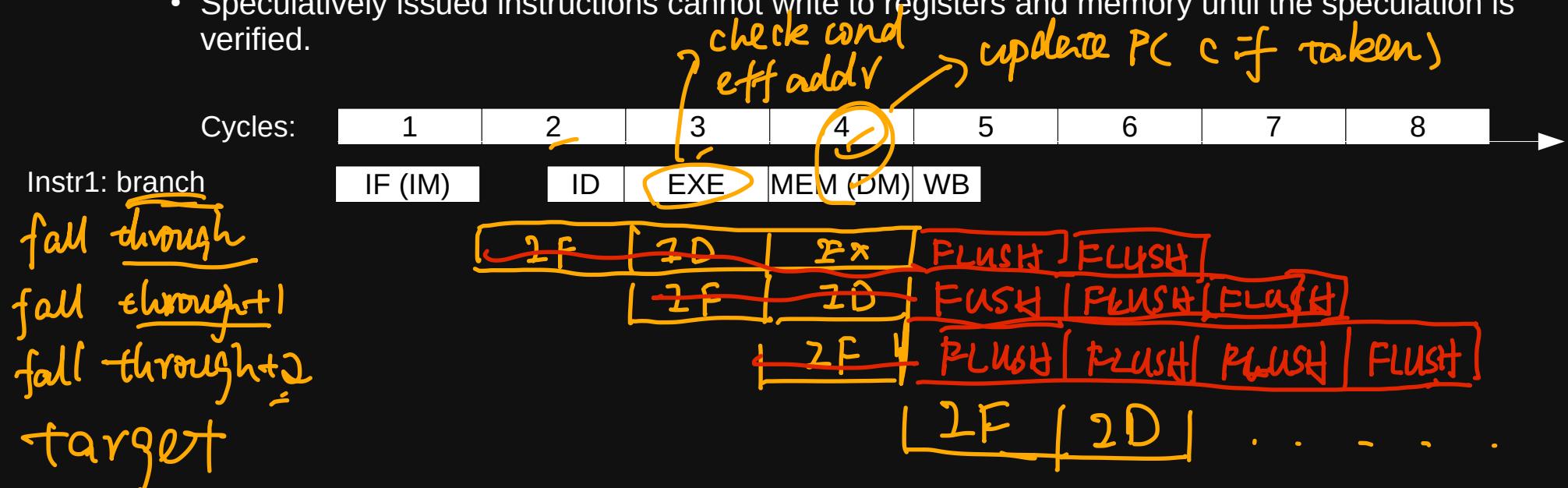
- Solution 2: Assume not taken

- Always assume branch is not taken, i.e., the next sequential instruction should be executed.
- If this assumption is correct, not delay in the pipeline.



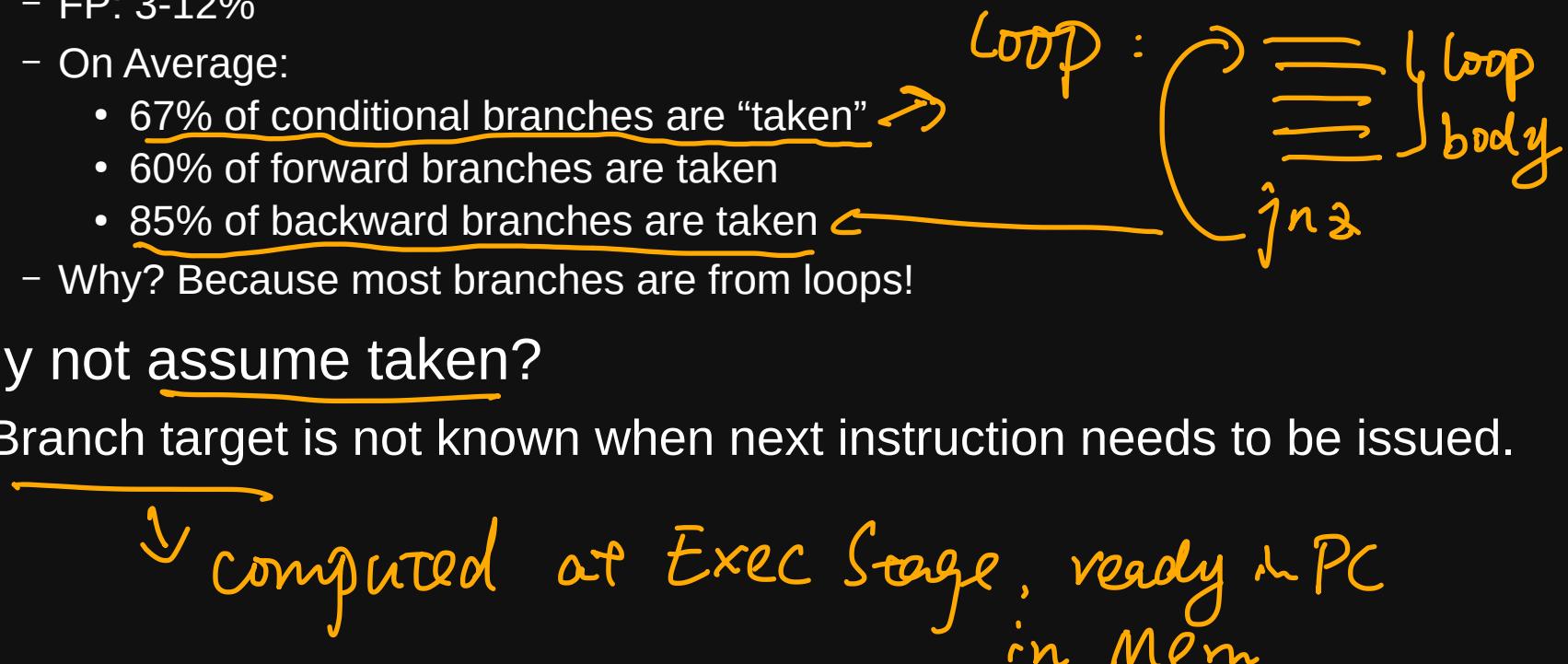
Control Hazard Solutions cont'd

- Solution 2: Assume not taken
 - Always assume branch is not taken, and the next sequential instruction is executed.
 - If the assumption is wrong, need to flush the pipeline to scratch all incorrectly issued instructions.
 - Instructions issued before the actual branch status is known are called **speculatively issued instructions**.
 - Speculatively issued instructions cannot write to registers and memory until the speculation is verified.



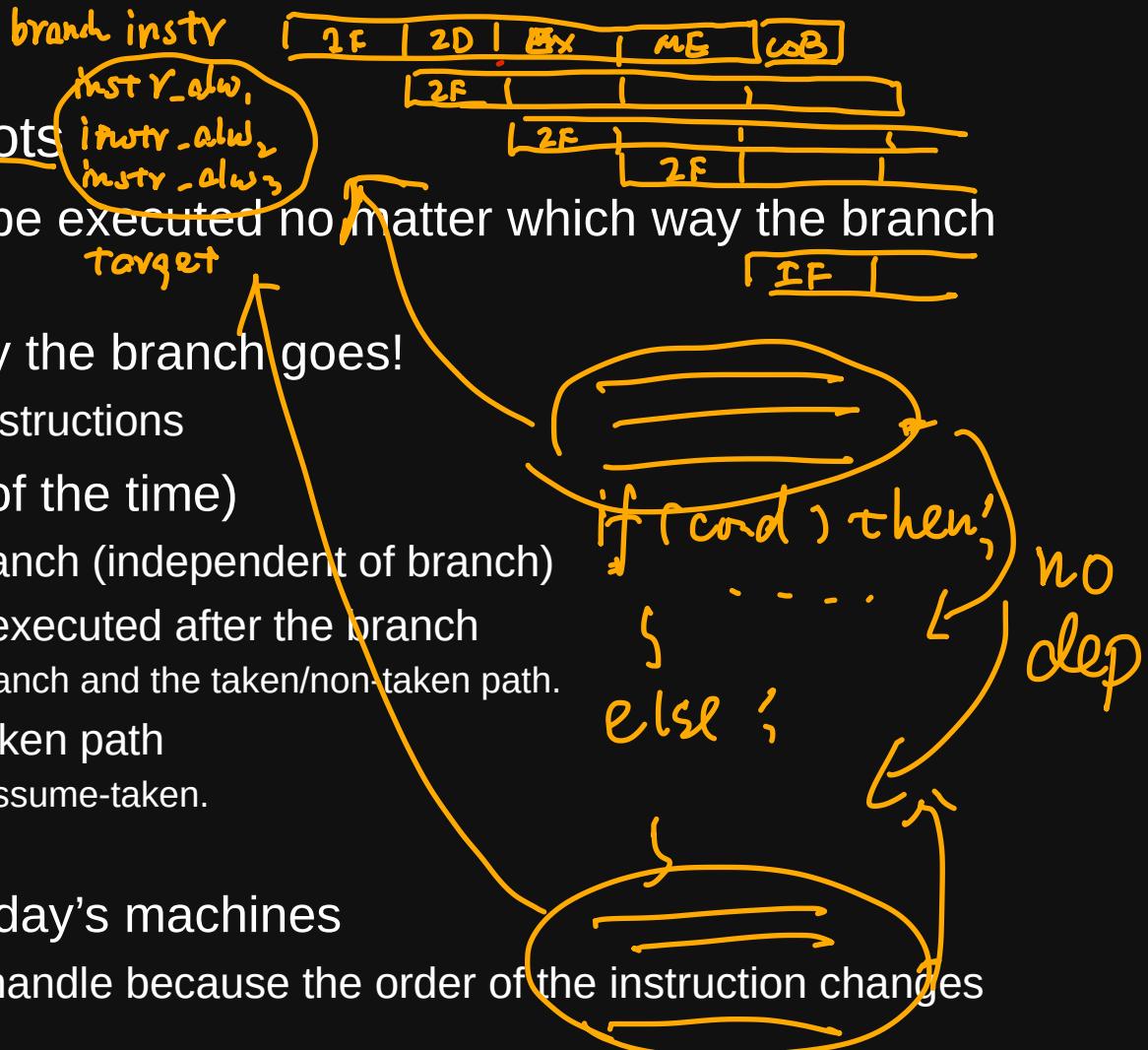
Control Hazard Solutions cont'd

- Solution 2: Assume not taken
 - Assume-not-taken may not help that much
 - Branch characteristics
 - Integer Benchmarks: 14-16% instructions are conditional branches
 - FP: 3-12%
 - On Average:
 - 67% of conditional branches are “taken”
 - 60% of forward branches are taken
 - 85% of backward branches are taken
 - Why? Because most branches are from loops!
 - Why not assume taken?
 - Branch target is not known when next instruction needs to be issued.



Control Hazard Solutions cont'd

- Solution 3: Branch Delay Slots



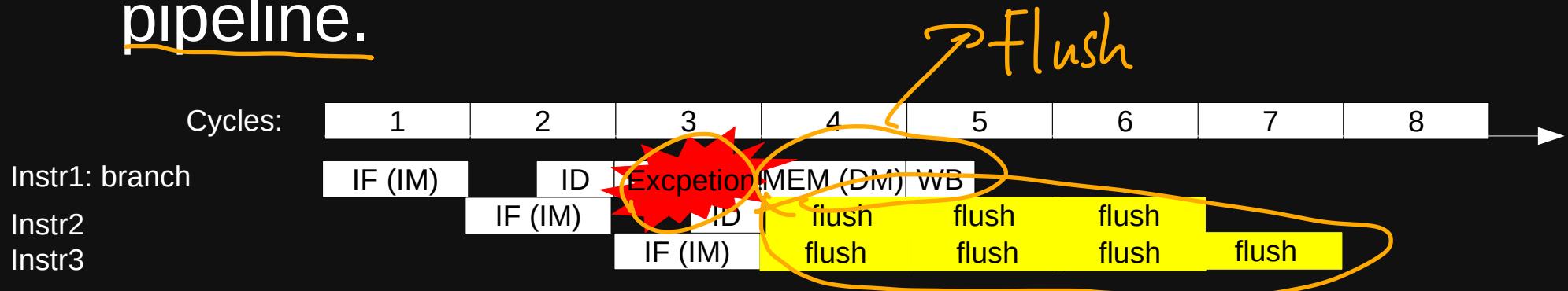
Control Hazard Solutions cont'd

- Solution 4: Branch Prediction
 - The actual solution for branches in modern processors is predicting whether the branch is taken and predicting what is the branch target.
 - Incorrect predictions still need flushing the pipeline.
 - We will discuss branch prediction in details in the lecture for speculative execution.

Control Hazards Due to Exceptions

↳ error interrupt

- Instructions experiencing exceptions during execution also causes control hazards.
- For exceptions, the solution is to flush the pipeline.



Summary of Hazard Solutions

Hazard Solutions

- Generic solution for all hazards:
 - stall and/or flush the pipeline
 - A side note, NOP instructions are issued for stalls, while instructions are switched to NOPs for flushing.
- Solutions only for structural hazard
 - Separate hardware resources → mem < ^{2M}_{DM}
 - Separate hardware access time → Reg File
 - Duplicate resources → ALU, Internal Reg
- Solutions only for data hazard
 - Bypass/forwarding → RAW
 - Instruction scheduling → Do D
- Solutions only for control hazard
 - Move branch logics to the ID stage. → fast branch resolution
 - Assume not taken → wrong most of the time
 - Branch delay slot → no-dep
 - Branch prediction →

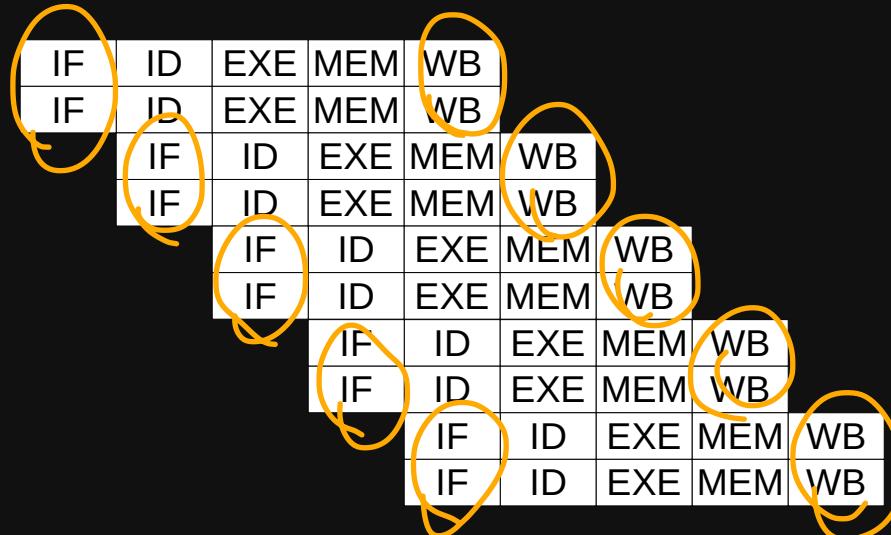
Superscalar

Superscalar

- Modern processors usually have a number of pipeline hardware components.
 - Another benefit of Moore's law.
- Therefore, modern processors usually can issue multiple instructions at each cycle.
 - Usually Superscalar is accompanied with OoO to ensure there are independent instructions that can be issued at the same time.
 - Obviously, multi-issue pipelines are more complex to implement and manage.

Illustration of A Two-issue Superscalar CPU

- Ideally, a two-issue Superscalar CPU has a CPI of 0.5 cycles, and IPC of 2.



Case Study: Intel Skylake Pipeline

- Maximum 8-issue per core.
- Pipeline stages: 14-19

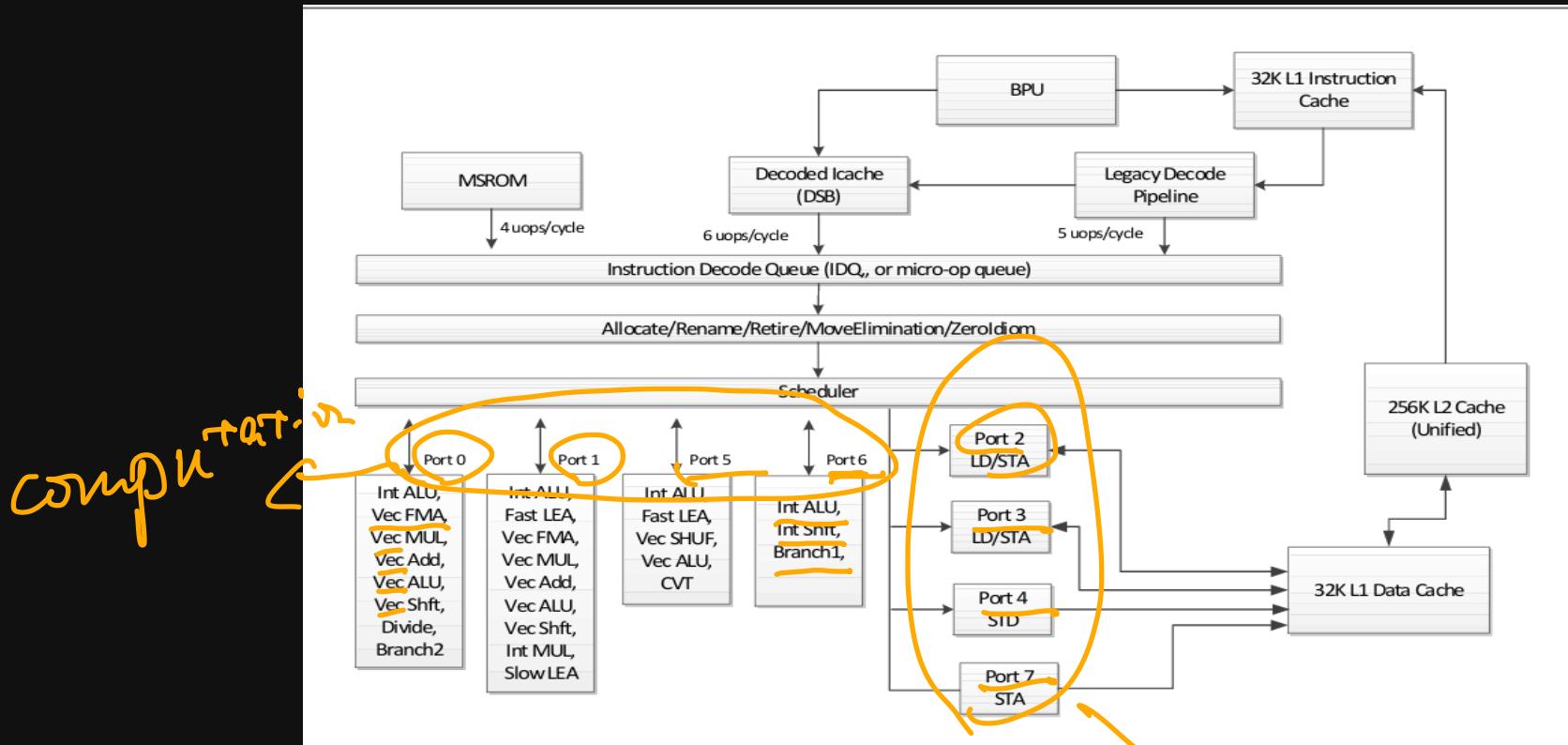


Figure from
Intel
Optimization
Reference
Manual.

Wikichip
has a better
figure.

Case Study: IBM Power9

- 4-8 Issue (maximum 8 fetch 6 decode)
 - Technically more of a SMT design than a Superscalar.
- 12-16 Pipeline stages

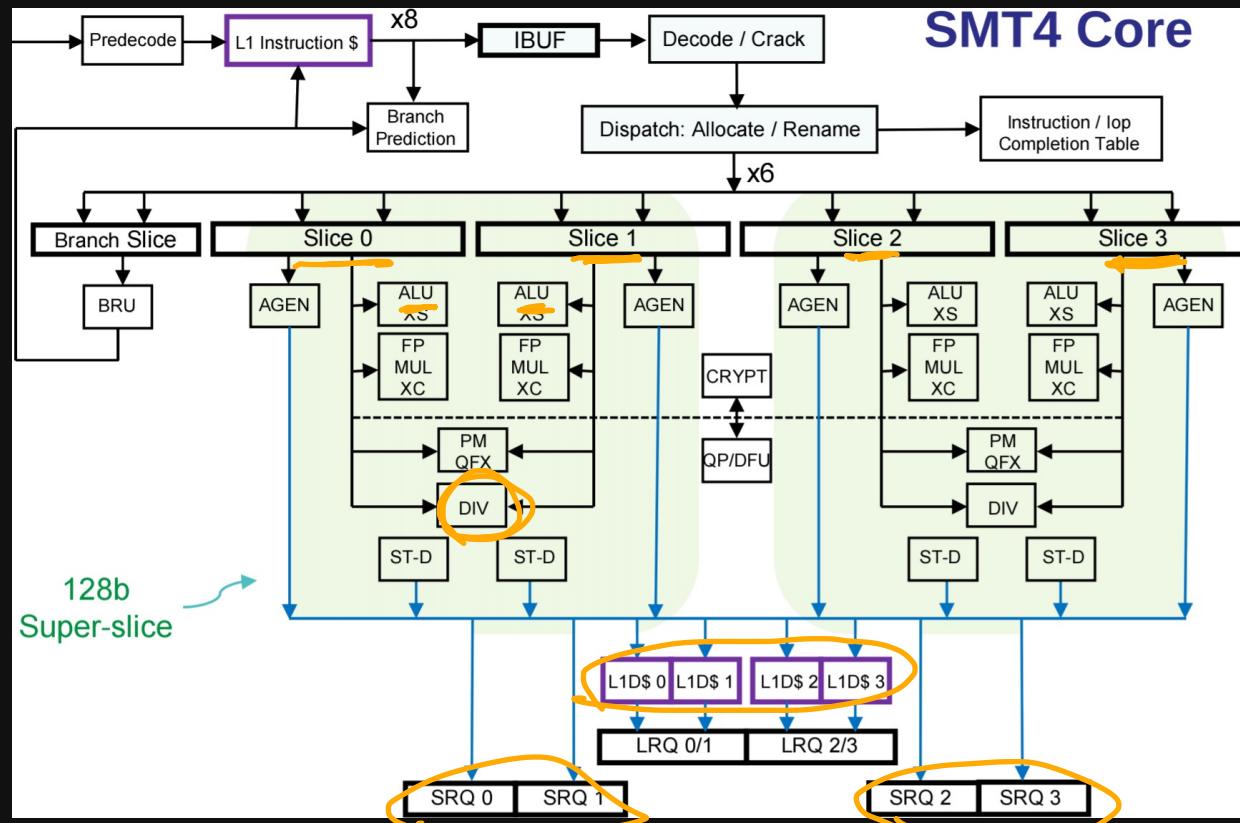


Figure from I
Wikichip.

Acknowledgment

- These slides are partially based on the lecture notes from Dr. David Brooks.