CS60003: High Performance Computer Architecture

Introduction to RISC-V Pipelining

IIT KHARAGPUR

Instructor:
Prof. Rajat Subhra Chakraborty
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India 721302
E-mail: rschakraborty@cse.iitkgp.ac.in

What is Pipelining?

- A fundamental technique to perform overlapped instruction execution
  - Used in most modern processors (even low-end ones)
  - Exploits Instruction-level Parallelism (ILP)
- Results in higher throughput
  - Once pipeline fills-up, (ideally) one instruction is executed per clock cycle
  - *n* pipeline stages => throughput improves by a factor *n* (ideal speedup)
- Enables higher clock frequency
  - Idea: do a small part of the of the work per clock cycle
- Disadvantages:
  - Ideal throughput is hard to achieve because of pipeline stalls
  - Hardware and power overhead (pipeline registers and control circuits)
  - Unbalanced pipeline stage delays => slowest stage determines clock frequency!
  - Makes interrupt/exception handling complicated
  - Total time to perform an operation is actually more than a non-pipelined processor!

Classic Five-stage RISC Pipeline (assuming no stall)

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

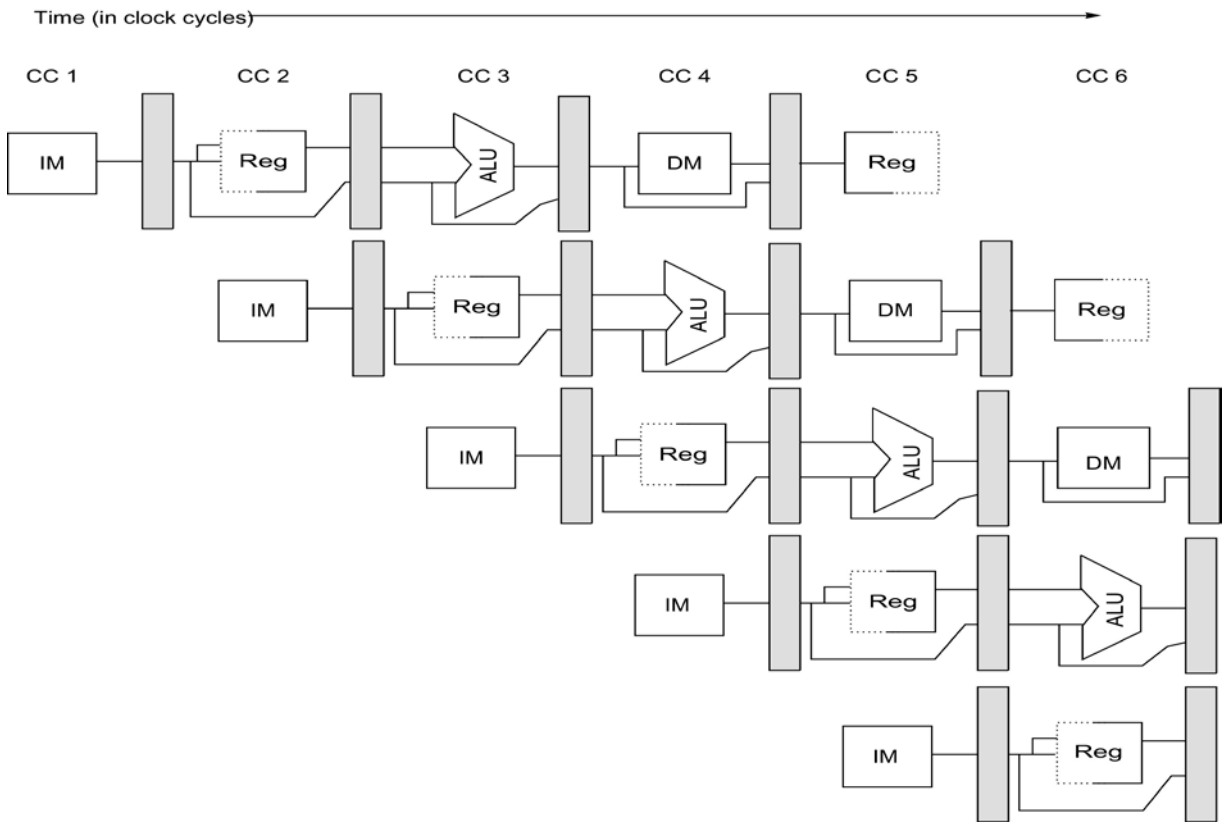
Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

Instruction Fetch (IF)

- **Fetch the instruction whose address in currently in Program Counter (PC)**
  - **Perform V.A. to P.A. translation if needed**
  - **Query the cache**
  - **If not found in cache, get from memory**
    - **In-order execution processor stalls**
    - **Out-of-order execution processor might stall in case of blocking cache**
    - **Out-of-order execution processor continues in case of non-blocking cache**

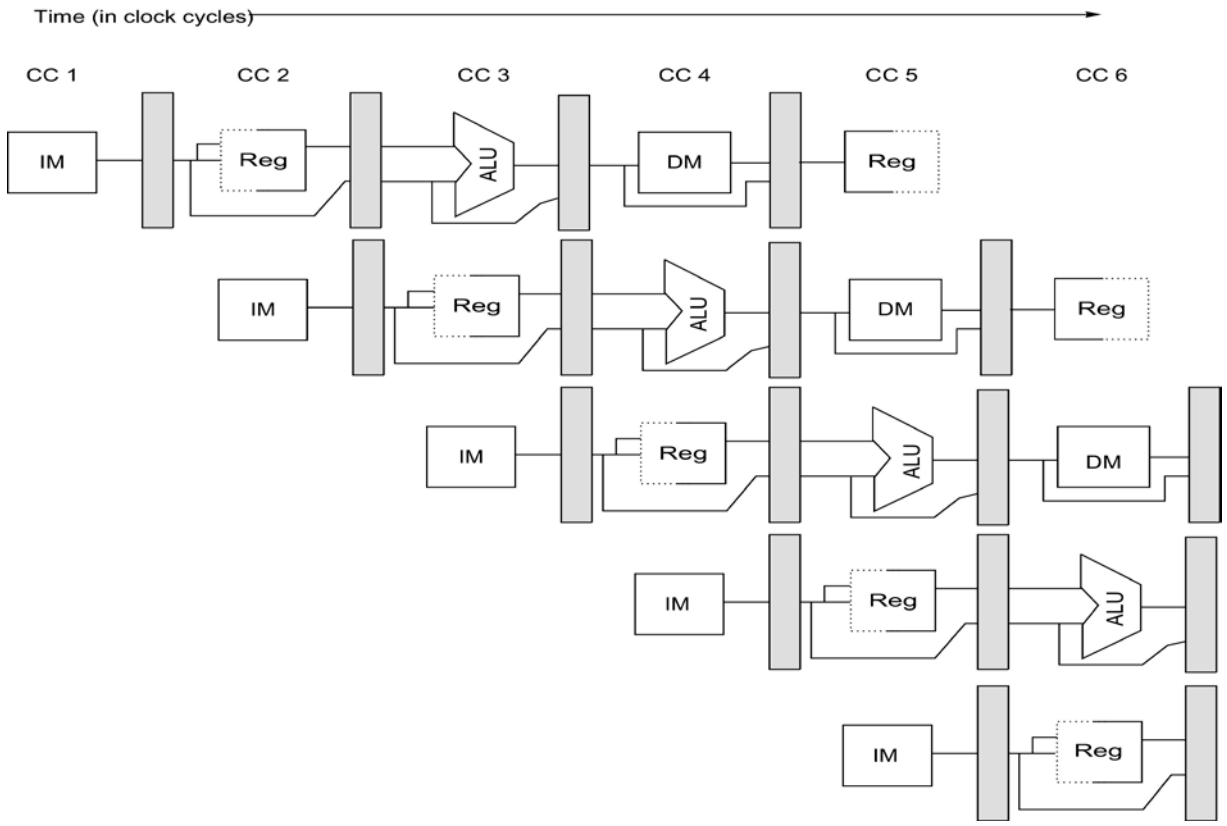
- **Update PC by adding 4 to it**

Pipeline Structure and Operations



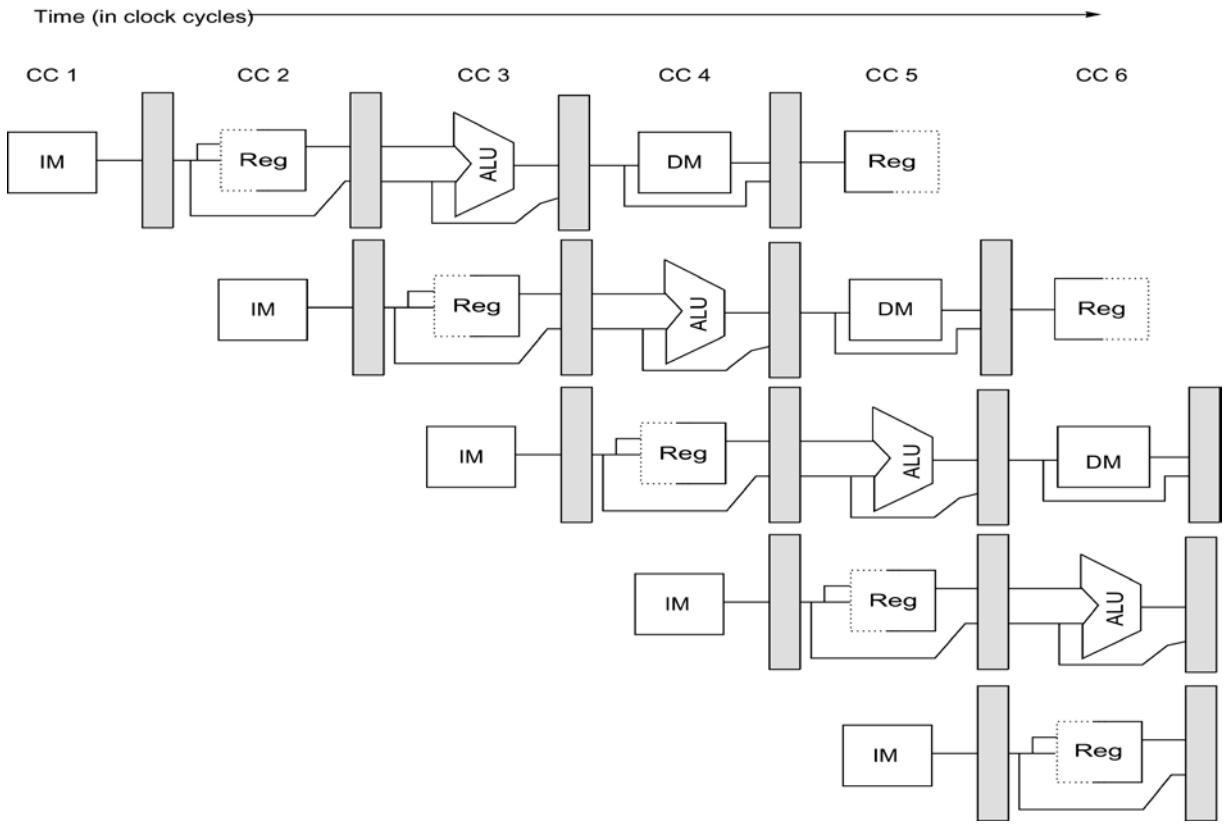
- **Edge-triggered flips-flops used in all registers**
- **ID: probably the most complex step:**
  - **Decode the instruction**
  - **Register specifiers are at fixed locations in an instruction => simplified and efficient decoding ("Fixed-field decoding")**
  - **Sign-extend the offset field (if needed)**
  - **Read the register fields to figure out possible dependences with previous instructions**
  - **Compute possible branch target address by adding the sign-extended offset with PC**

Pipeline Structure and Operations (contd.)



- **Register file is accessed for read in ID and write in WB**
  - **Special circuit design ensures reading and writing completes in half clock cycle!**
  - **ID read: happens in 2nd half of clock cycle**
  - **WB write: happens in 1st half clock cycle**
  - **This optimization is crucial to enable WB to ID data forwarding (described later)**

Pipeline Structure and Operations (contd.)



- **EX stage: might perform different types of operations:**
  - **Register-register ALU operation**
  - **Register-immediate ALU operation**
  - **Calculate effective address for load/store**
  - **Determines condition for conditional branch (e.g. =, >)**
    - **The possible target has already been calculated in the ID stage!**

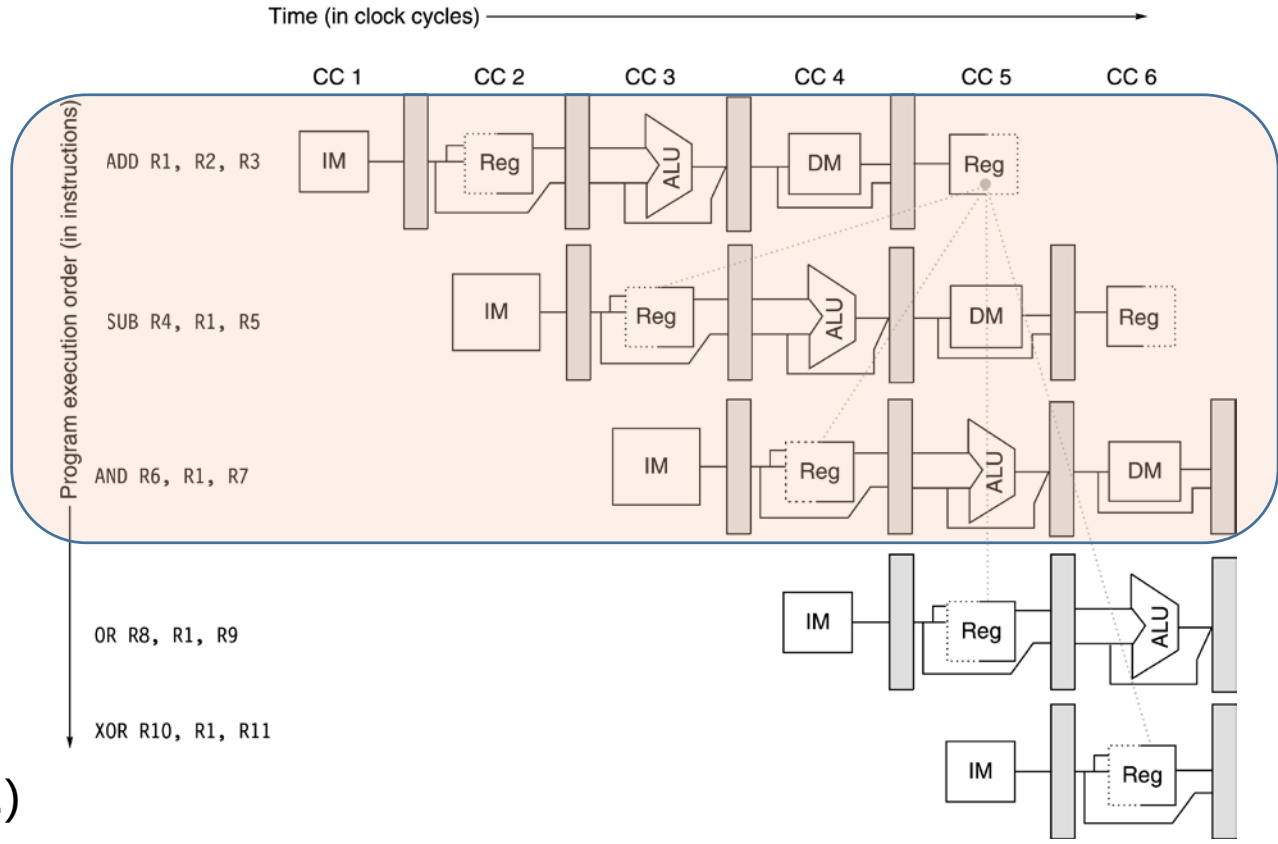
Pipeline Hazards

- **Pipelines in practice can never achieve the ideal speedup of *n***
- **Reason: Hazards**
  - **Potentially results in pipeline stalls**
- **Types of Hazards:**
  - **Structural hazard**
    - **Caused by conflict for specialized hardware units which take large number of clock cycles, e.g. FP units and divider circuits**
    - **Less common issue modern processors**
    - **Avoidable easily: by compiler scheduling**
  - **Data hazard**
    - **Reason: dependency on result of previous instruction**
  - **Control hazard**
    - **Reason: branches and other instructions that modify the PC**

Data Hazards

- **Assume *i*-th instruction occurs before *j*-th instruction, both use register *x***
- **Read After Write (RAW) hazard:**
  - **Instr. *j* reads *x* before write by instr. *i* is complete, thereby using wrong value of *x***
  - **Can happen because WB is last pipeline stage!**
  - **Most common!**

Register *R1* causes RAW hazard between 1st and 2nd and 1st and 3rd instructions



OR: no hazard (why??)
XOR: no hazard, far away
(read happens after WB from ADD has completed)

Data Hazards (contd.)

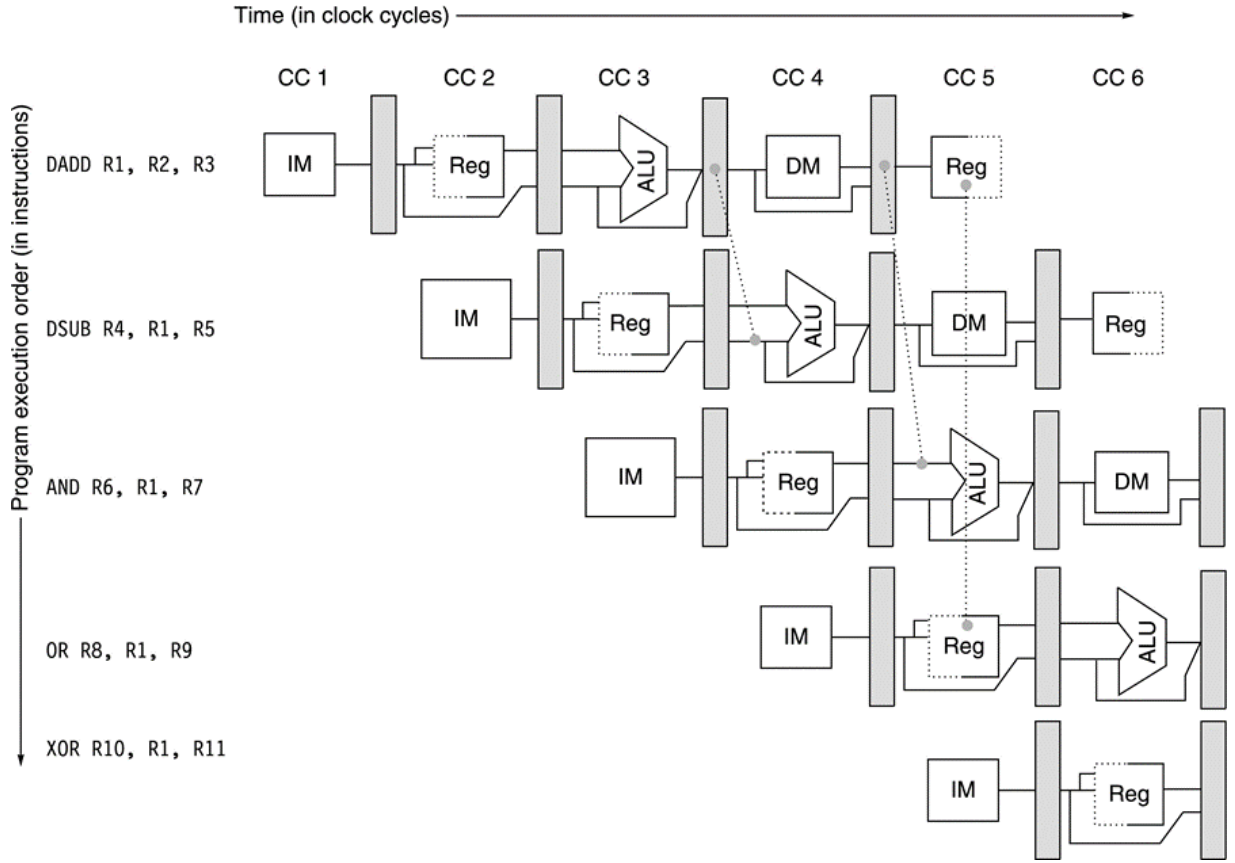
- **Assume *i*-th instruction occurs before *j*-th instruction, both use register *x***

- **Write After Read (WAR) hazard:**
  - **Instr. *i* reads *x after* write by instr. *j*, thereby using wrong value of *x***
  - **Impossible in a simple 5-stage RISC pipeline**
  - **Can happen if instructions are executed out-of-order!**

- **Write After Write (WAW) hazard:**
  - **Instr. *i* writes *x after* write by instr. *j*, thereby setting wrong value of *x* going forward**
  - **Impossible in a simple 5-stage RISC pipeline**
  - **Again, can happen if instructions are executed out-of-order, or for multiple execution units (described later)!**

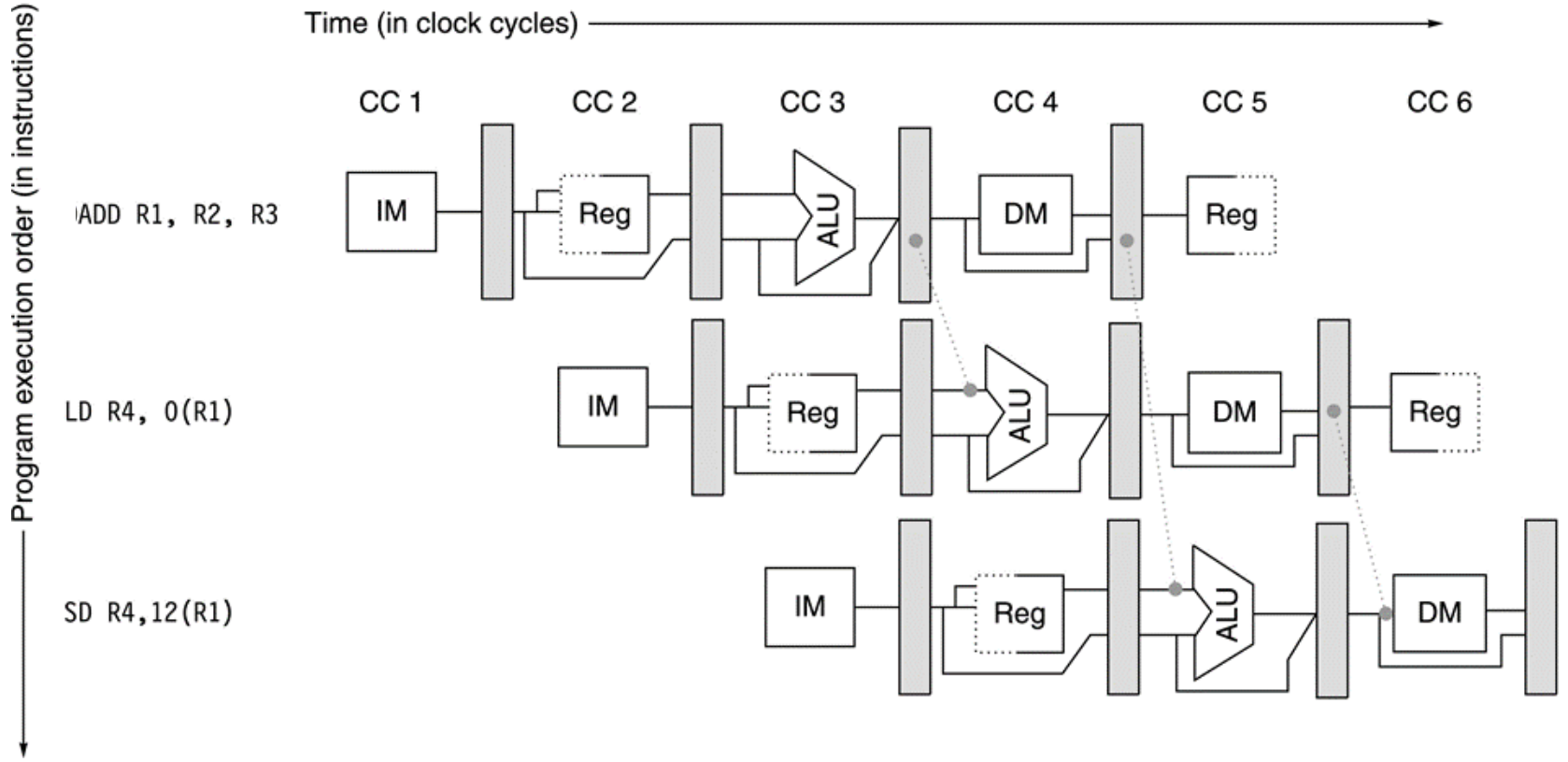
Forwarding to Resolve RAW Hazard

- **Forwarding (aka "bypassing"): hardware technique to resolve RAW hazard**
  - **EX/MEM pipeline register to ALU input feedback**
  - **MEM/WB pipeline register to ALU input feedback**
  - **Multiplexors at ALU inputs select correct operand (normal? feedback?)**



A set of instructions that depends on the add result uses forwarding paths to avoid the data hazard. The inputs for the sub and and instructions forward from the pipeline registers to the first ALU input. The or receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the and instruction was and x6,x1,x4.

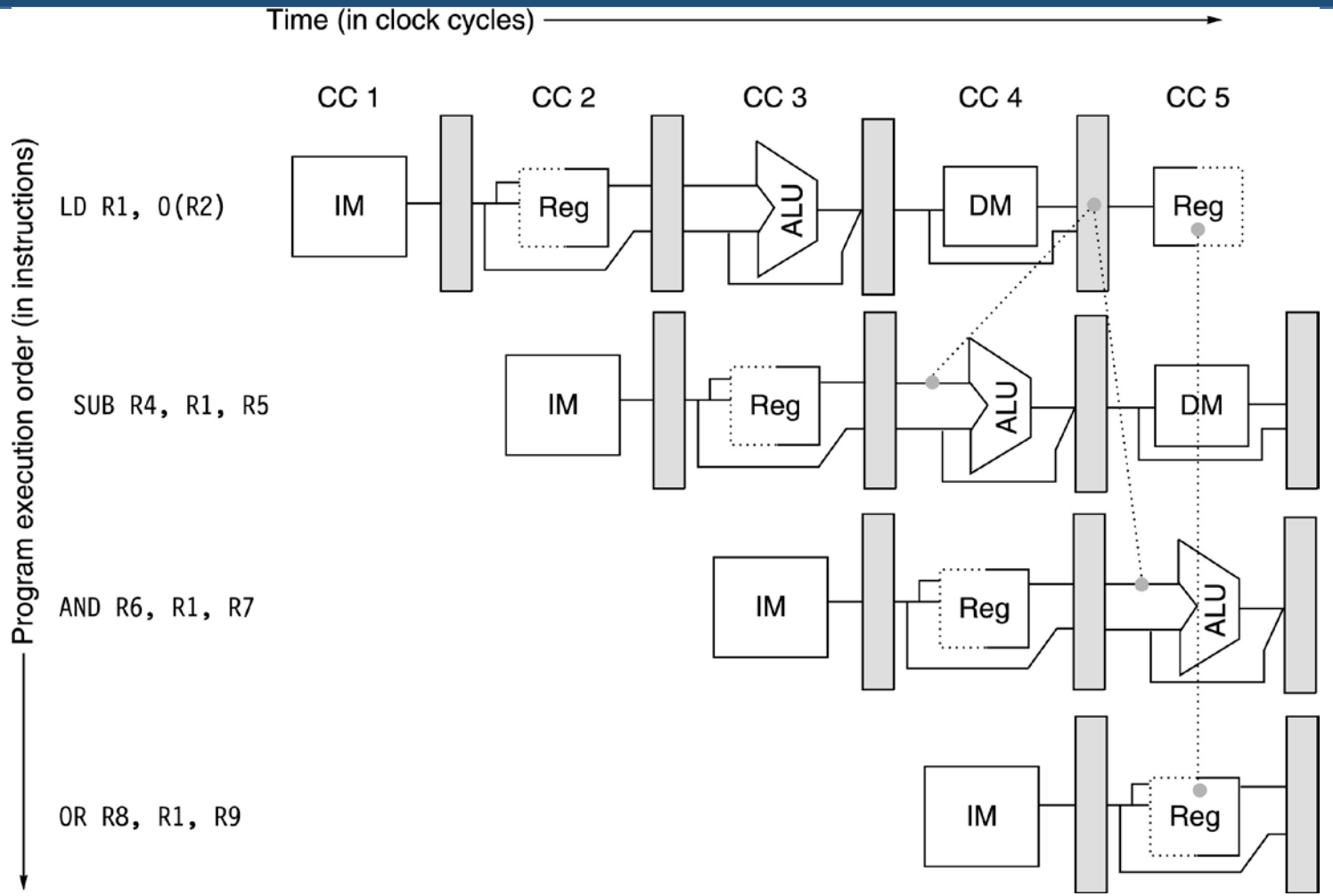
Forwarding to Resolve RAW Hazard (contd.)



Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation).

If the store depended on an immediately preceding ALU operation (not shown herein), the result would need to be forwarded to prevent a stall.

Forwarding is Not Always Sufficient!



The load instruction can bypass its results to the and **and** or **instructions, but not to the** sub **(cannot go back in time) => 1 cycle stall!**

H&P CA:A QA (6th. Ed.)

What Happens When Forwarding is Insufficient?

- **Special control circuit (“pipeline interlock”) inserts stall in the pipeline**
  - **Detects hazard and stalls pipeline until hazard is cleared**

Requires stall

ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	EX	MEM	WB			
and x6,x1,x7			IF	ID	EX	MEM	WB		
or x8,x1,x9				IF	ID	EX	MEM	WB	

After stall insertion:

ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	Stall	EX	MEM	WB		
and x6,x1,x7			IF	Stall	ID	EX	MEM	WB	
or x8,x1,x9				Stall	IF	ID	EX	MEM	WB

and: **uses forwarding**
or: **forwarding not required**

Control Hazard

- **Usually, PC is automatically modified to PC + 4**
- **But:**
  - **Whether branch will be taken/not taken is not known till end of ID stage!**
  - **If branch is taken, then the instruction currently in IF is incorrect!**
  - **Causes 1 cycle stall (pipeline must be *frozen/flushed*) after 1 clock cycle!**
  - **Can we do better?**
- **Two simple solutions: Predicted-branch-not-taken and Predicted-Branch-Taken**
  - **These two strategies are static, i.e. does not change during execution of a branch instruction**
  - **Both of these cause: no stall if correct, 1 cycle stall if is incorrect**

Example: Predicted-not-taken Scheme

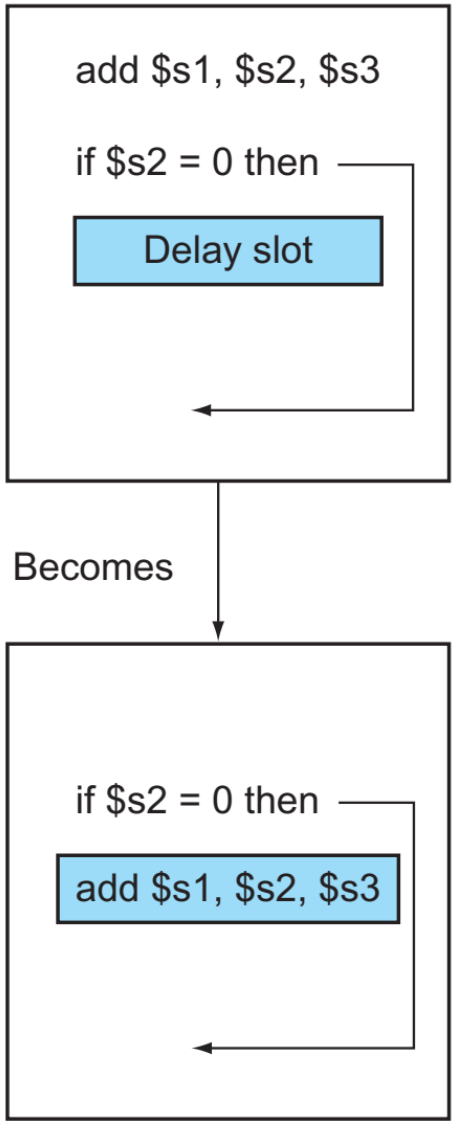
Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Alternative: Delayed Branch

- **A compiler-based technique**
- **Basic idea:**
  - **Find and insert an instruction between a branch instruction and the next instruction (in "branch delay slot")**
  - **Execute the inserted instruction unconditionally**
  - **Inserted instruction must be such that the modified program is correct**
  - **Compiler *must* find an independent instruction to insert**
    - **Situation becomes complex if such an instruction cannot be found**
  - **Not popular nowadays…..RISC-V does not have it**



Better Option: Dynamic Branch Prediction

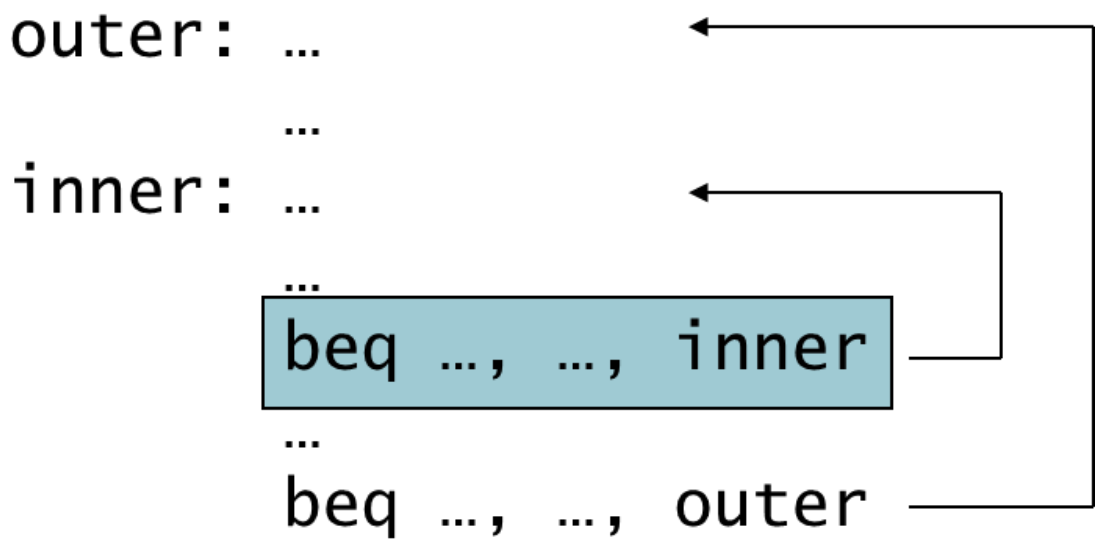
- **In deeper and superscalar pipelines, branch penalty is more significant**

- **In dynamic prediction:**
  - **Branch prediction buffer (aka branch history table) is maintained**
  - **Indexed by lower portion of recent branch instruction addresses**
  - **Stores actual outcome as a single bit: taken/not taken**

- **To execute a branch:**
  - **Check table, expect the same outcome**
  - **Start fetching from fall-through or target**
  - **If wrong, flush pipeline and update prediction**

1-bit Predictor: Shortcoming

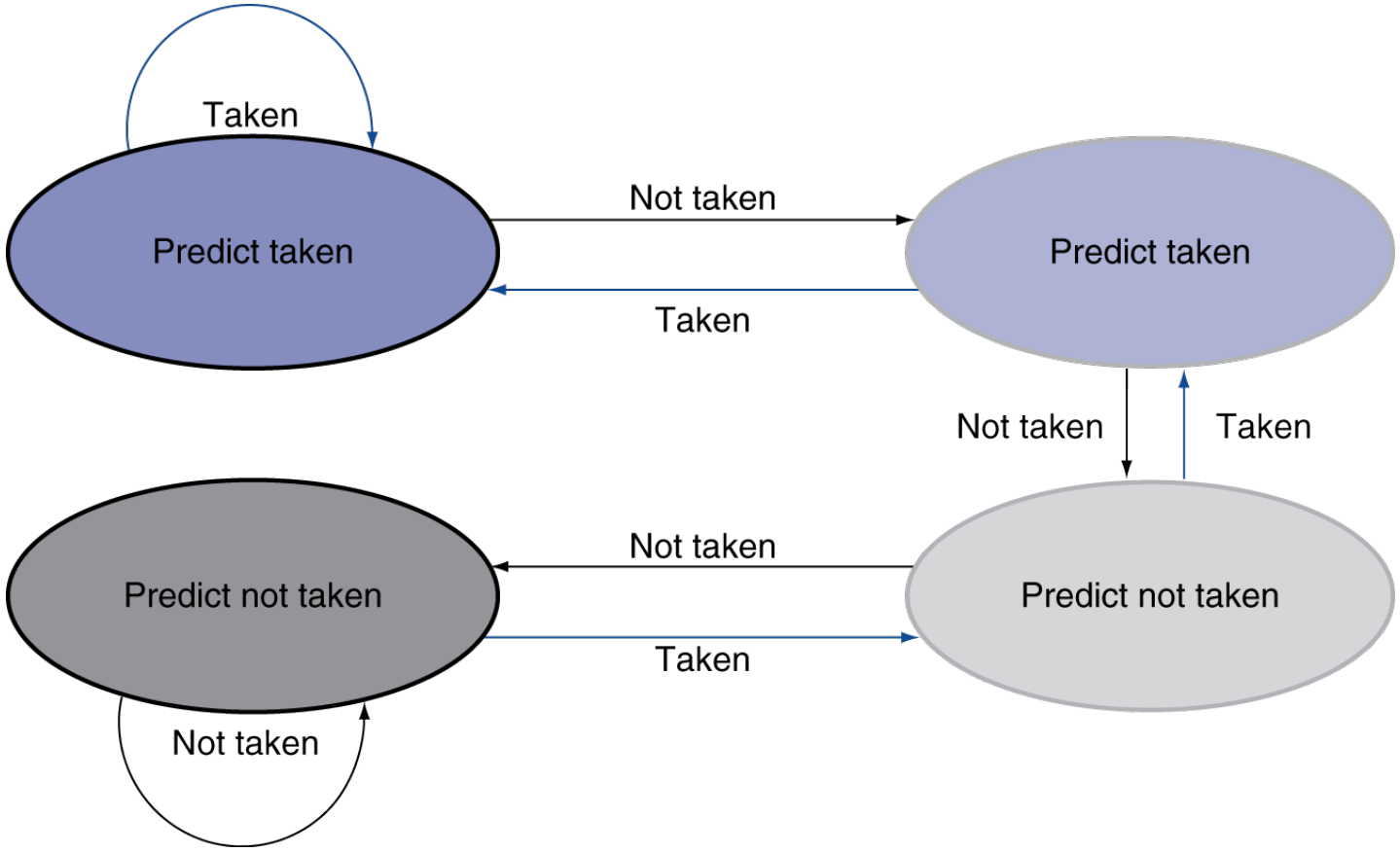
- **Inner loop branches are mis-predicted twice!**



- **Mis-predict as taken on last iteration of inner loop**
- **Then mis-predict as not taken on first iteration of inner loop next time around**

Solution: 2-bit Predictor

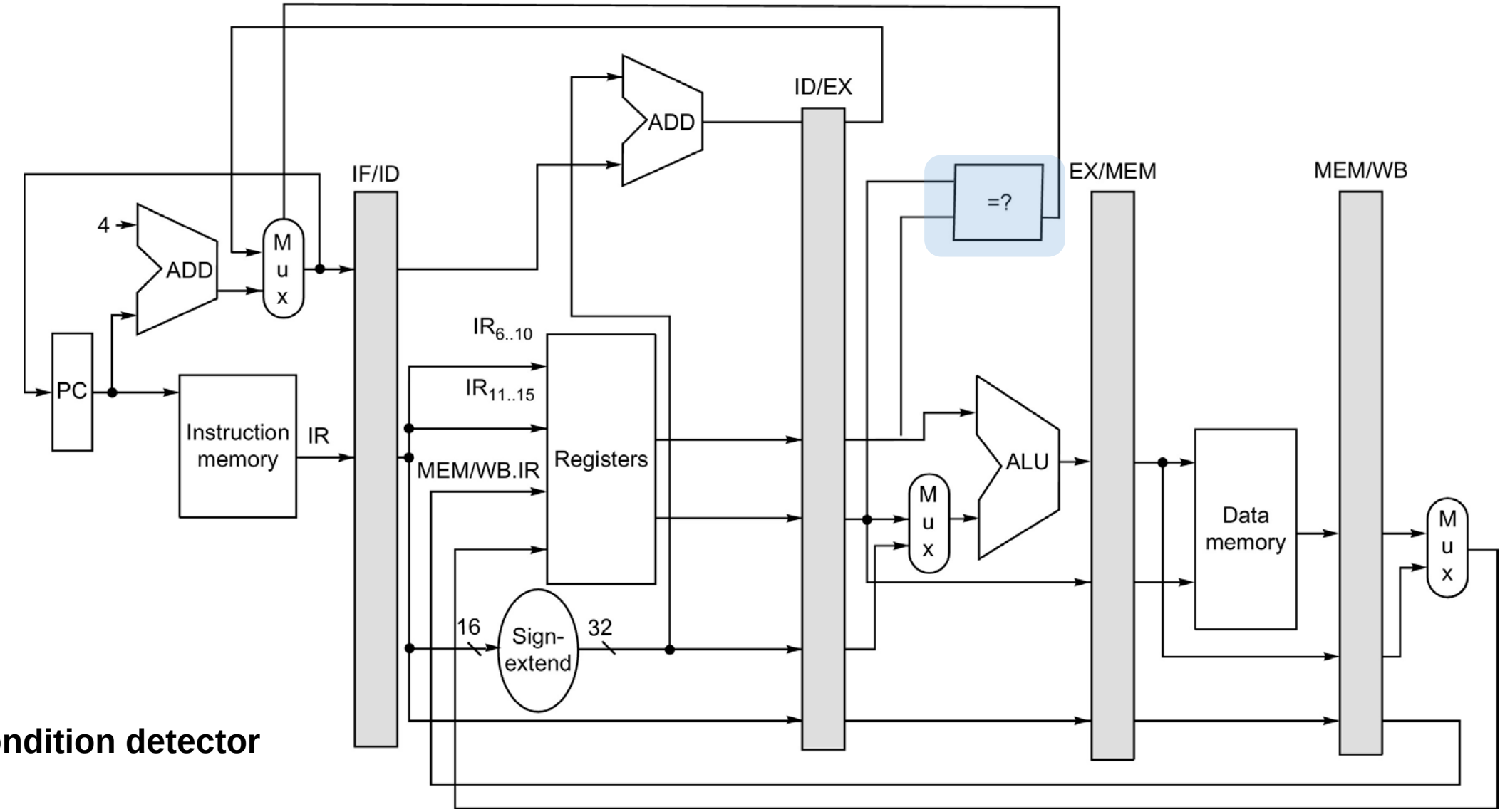
- **A simple Finite State Machine**
- **Idea: change prediction on two successive mis-predictions!**



Where to Detect Branch Condition?

- **Old RISC architectures did branch register comparison and branch address calculation in ID stage (using a dedicated adder in ID stage)**
  - **Adv.: only 1 cycle branch mis-prediction penalty**
  - **Disadv.: if condition checking involves a register whose value would be set in EX stage, a data hazard exists!**
  - **Disadv.: slower ID stage**
- **RISC-V approach: determine branch condition in EX stage, but as before perform branch address calculation in ID stage (using a dedicated adder in ID stage)**
  - **Perform possibly redundant branch address calculation in ID stage**
  - **As if every instruction is a branch instruction (!)**
  - **Adv.: faster EX stage**
  - **Adv.: no data hazard of the type mentioned above is possible**
  - **Disadv.: higher energy consumption**
  - **Disadv.: 2 cycle branch misprediction penalty in RISC-V**

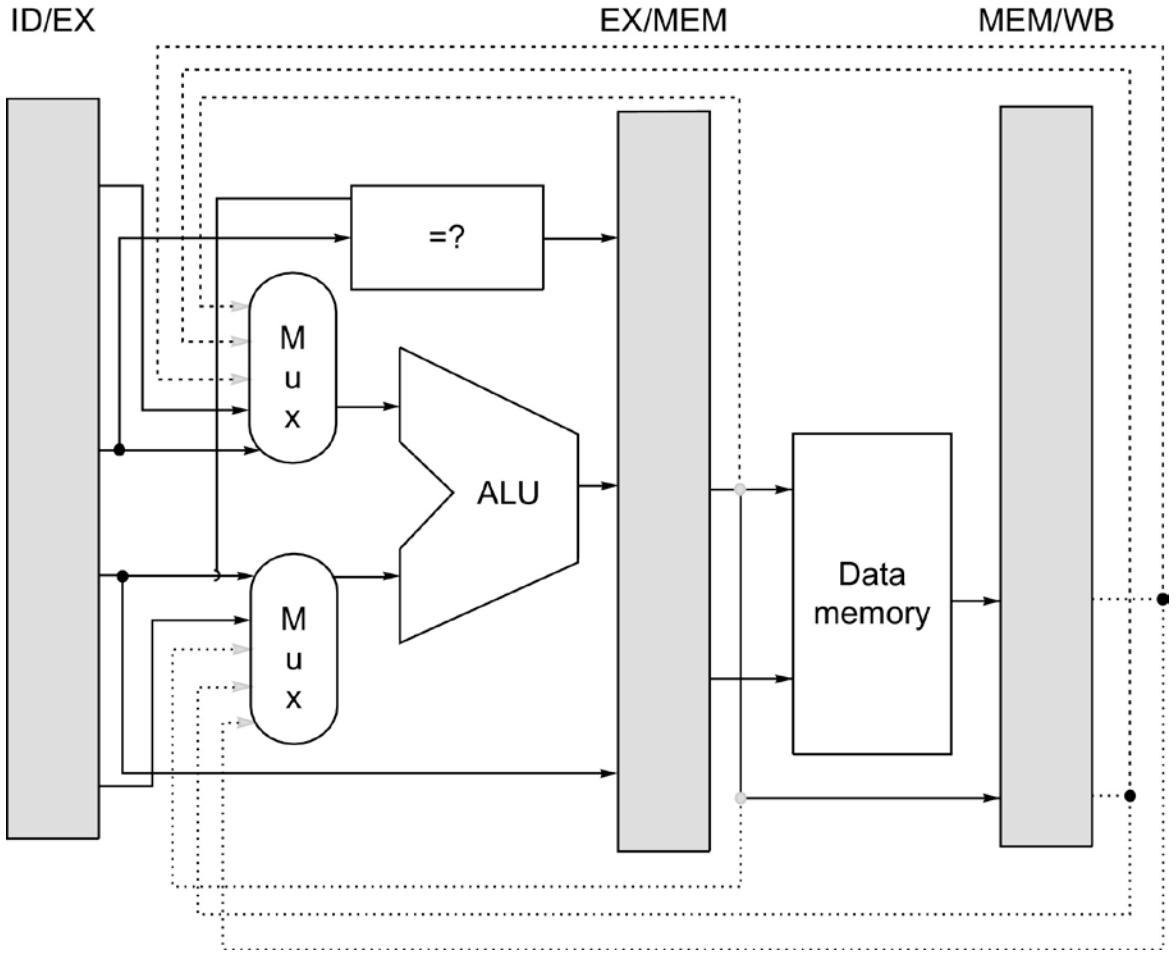
Basic RISC-V Pipeline



Branch condition detector

H&P CA:A QA (6th. Ed.)

Forwarding in Basic RISC-V Pipeline



Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

Examples: Dependences Detected by Pipeline Control

Situation	Example code sequence	Action

No dependence	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		No hazard possible because no dependence exists on x1 in the immediately following three instructions

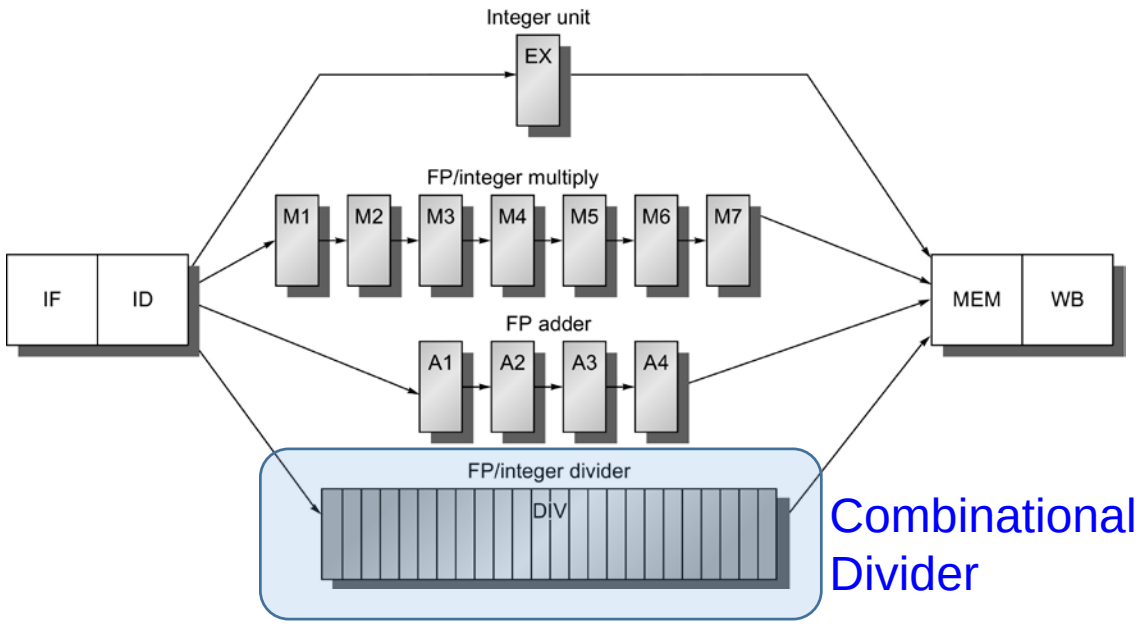
Dependence requiring stall	`ld x1,45(x2)`
	`add x5,x1,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		Comparators detect the use of x1 in the `add` and stall the `add` (and `sub` and `or`) before the `add` begins EX

Dependence overcome by forwarding	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x1,x7`
	`or x9,x6,x7`
		Comparators detect use of x1 in `sub` and forward result of load to ALU in time for `sub` to begin EX

Dependence with accesses in order	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x1,x7`
		No action required because the read of x1 by `or` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only comparison needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case once execution continues (dependence overcome by forwarding). Of course, hazards that involve x0 can be ignored because the register always contains 0, and the preceding test could be extended to do this.

Multicycle Execution Units



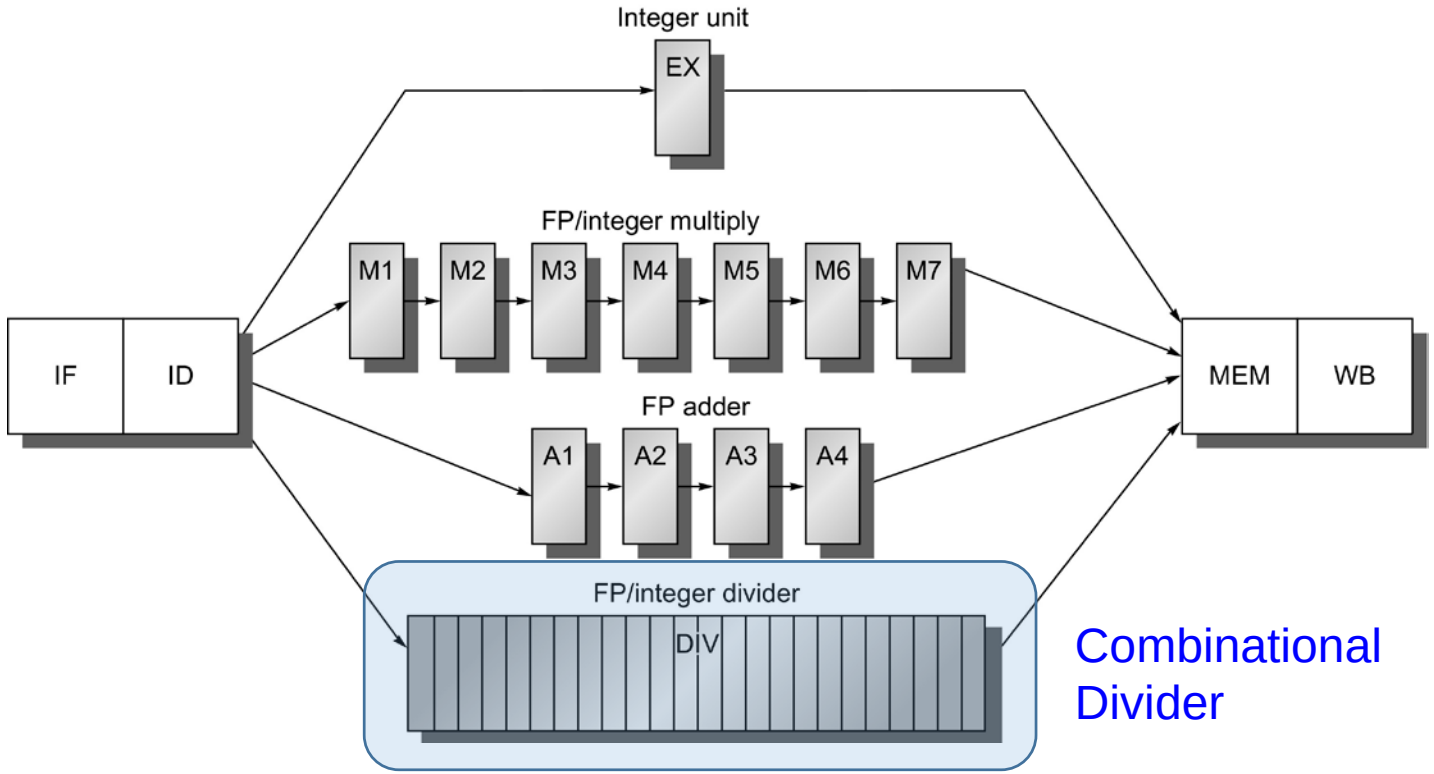
Combinational Divider

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Instruction Issue: an instruction allowed to move from ID to EX

- **Floating-point units are often multicycle**
- **Latency:** # of following instructions which (without stalling) cannot use the result of the current instruction
  - Usually: latency = # of execution unit pipeline stages – 1
- **Initiation Interval:** # of clock cycles between issuing two instructions of the given type

Example: Multicycle Pipeline Timing



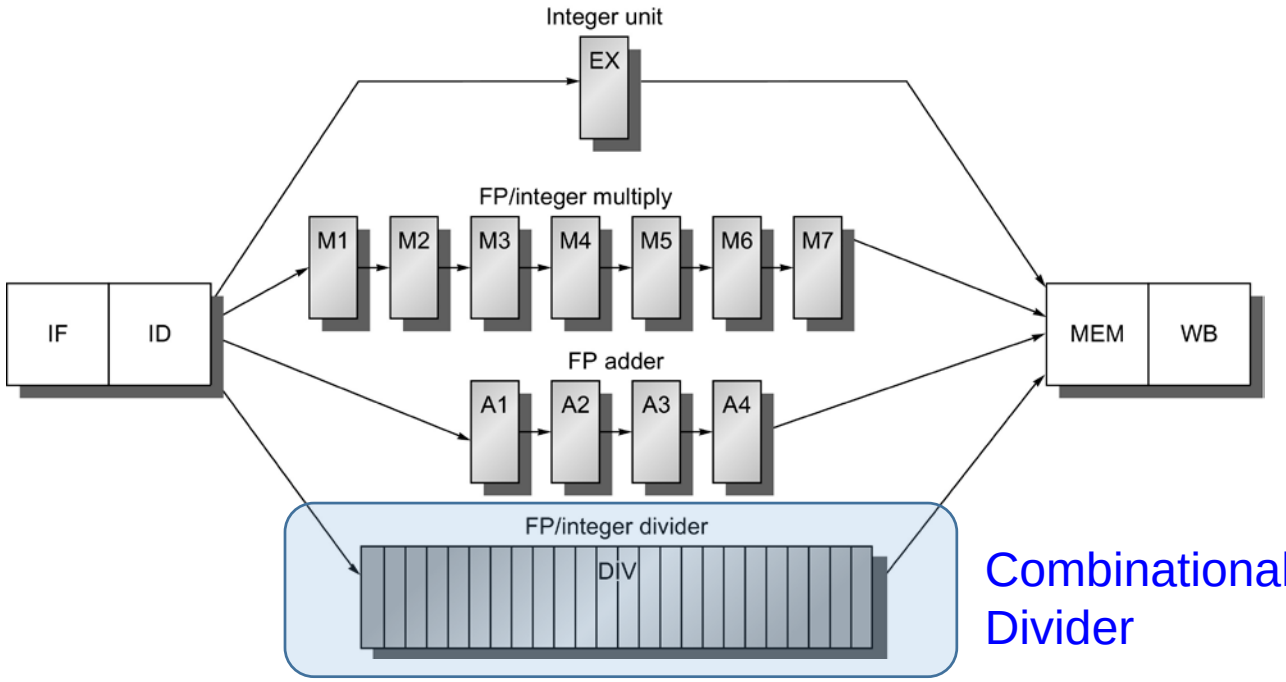
Combinational Divider

Result available *at the end of* this stage

fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
fadd.d		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
add			IF	ID	*EX*	**MEM**	WB				
fsd				IF	ID	*EX*	**MEM**	WB			

The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

Example: Multicycle FP Pipeline Stalls due to RAW Hazards



Combinational Divider

All pipelines have full forwarding and bypassing.
Note: two instructions both trying to use the divider less than 25 clock cycles apart will lead to structural hazard!

Easily avoidable

	Clock cycle number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`fld f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The `fsd` must be stalled an extra cycle so that its MEM does not conflict with the `fadd.d`. But this is easily avoidable, since fadd does not access the memory.

Example: Multicycle FP Pipeline Stalls due to Structural Hazard

	Clock cycle number										
Instruction	1	2	3	4	5	6	7	8	9	10	11
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB

Structural Hazard

No Structural Hazard

Note: fdiv/div **can cause structural hazard, must wait for 25 clock cycles before issuing next** fdiv/div **instruction!**

Note: structural hazards between integer registers and fp registers are minimal

Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11.
Note that although the `fmul.d`, `fadd.d`, and `fld` are in the MEM stage in clock cycle 10, only the `fld` actually uses the memory, so no structural hazard exists for MEM.

Example: Solving Structural Hazard in Multicycle FP Pipeline

	Clock cycle number										
Instruction	1	2	3	4	5	6	7	8	9	10	11
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB

- **In each pipeline, maintain a *ring counter* (with a single 1) to keep track of the clock cycle in which the register file write is to happen**
  - **Shift the 1 every clock cycle**
- **In the ID stage (constant distance from WB for each pipeline), if there seems to be conflict, delay the instruction issue by 1 clock cycle**

Example: WAW Hazard in Multicycle FP Pipeline

	Clock cycle number										
Instruction	1	2	3	4	5	6	7	8	9	10	11
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`						IF	ID	EX	MEM	WB	

WAW Hazard

- fld **has WAW Hazard with** fadd**!**
  - fld **writes to register** f2 ***before*** fadd**, although** fadd **was issued *earlier*!**
- **Soln-(1): (simple) use a shift register, detect the hazard and delay issue of** fld **till** fadd **enters MEM**
- **Soln-(2): since** fadd **result is not used before** fld**, hence do not allow** fadd **to write its result (anyway it will be over-written)!**

Thank you