ECE4100/ECE6100/CS4290/CS6290

Advanced Computer Architecture

Fall 2024

Recitation: Lab 2

Seokjin Go

seokjin.go@gatech.edu

School of Electrical and Computer Engineering

Georgia Institute of Technology

Acknowledgment: Recitation slides largely adapted from prior GT ECE 4100/6100 (Rishov Sarkar)

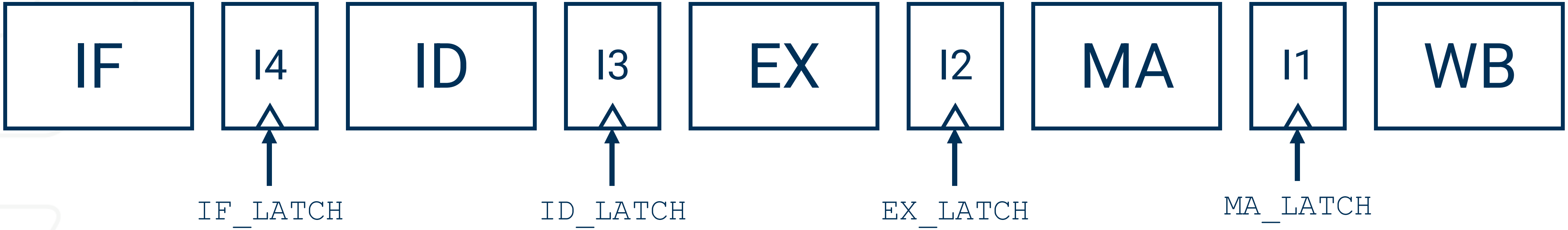
Lab 2A

Implementation and Clarifications

General Advice

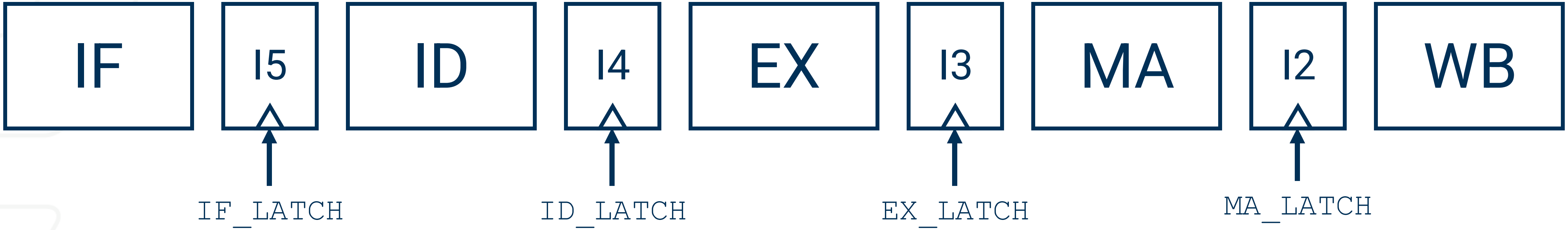
- Always check the `valid` flag! Invalid instructions should be ignored!
- Check stall logic after moving instructions into `ID_LATCH`
- Then, handle stalls in `ID_LATCH` and tell IF to stall too
- For superscalar:
  - During stall logic checks, keep track of the oldest instruction stalled
  - Then, after all checks, go back and stall anything younger than oldest stalled
- For forwarding:
  - Before, you could just detect any possible RAW hazard and stall
  - Now, you need to check if each hazard is overwritten or not
  - Then, you need to check if each hazard is resolvable by forwarding

A More Accurate Depiction of the Pipeline



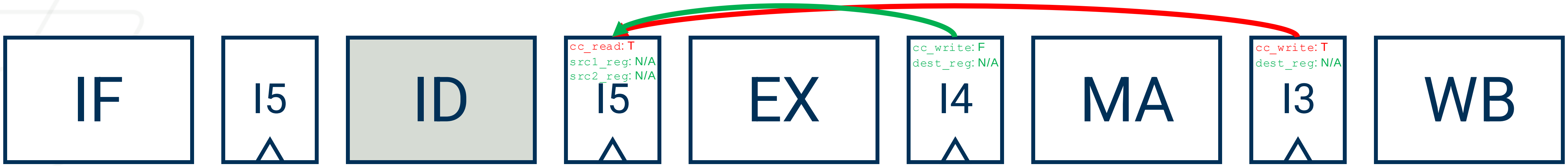
- In our sequential simulator, pipeline stages run backwards
  - In hardware, these would run in parallel
- Instructions are copied (not moved) between stages

A More Accurate Depiction of the Pipeline



- In our sequential simulator, pipeline stages run backwards
  - In hardware, these would run in parallel
- Instructions are copied (not moved) between stages

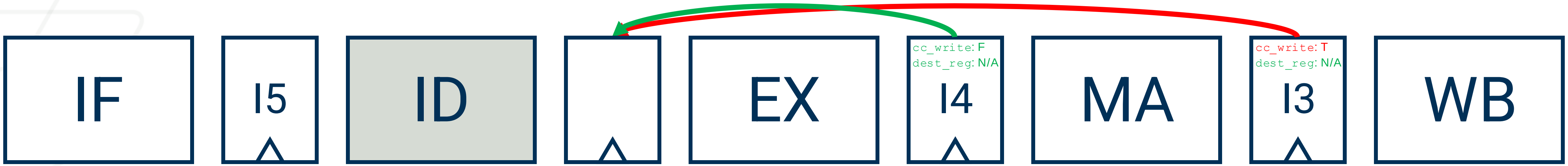
Handling Stalls



- A stall condition is detected!
  - Example: I5 has `cc_read` and I3 has `cc_write`
    - (Sidenote: `cc_write` and `dest_needed` are completely independent!)
- What to do?

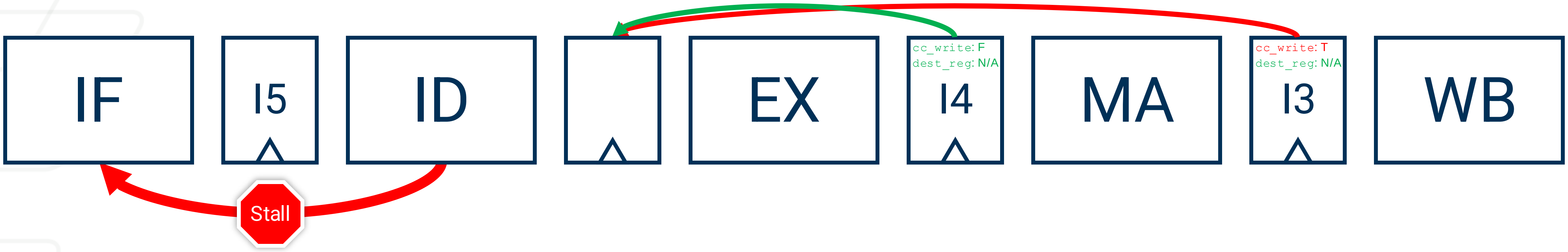
Handling Stalls

6



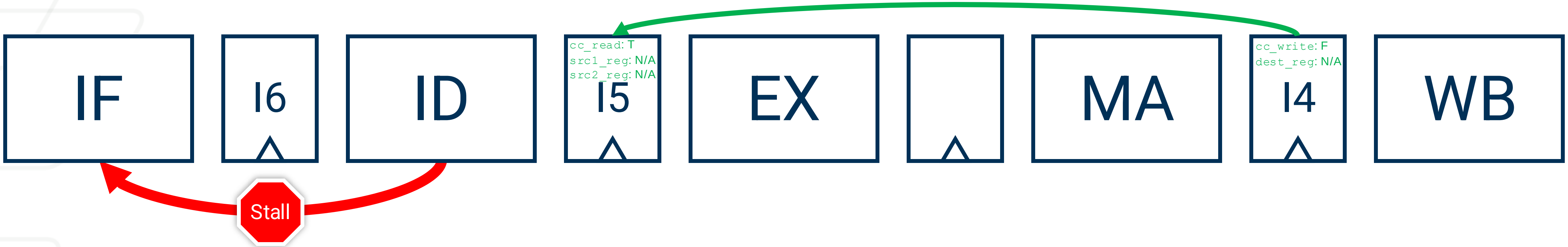
- A stall condition is detected!
  - Example: I5 has `cc_read` and I3 has `cc_write`
    - (Note: cc_write and dest_needed are completely independent!)
- What to do?
  - Insert a bubble into the pipeline (set `valid` to `false` to clear the instruction)

Handling Stalls



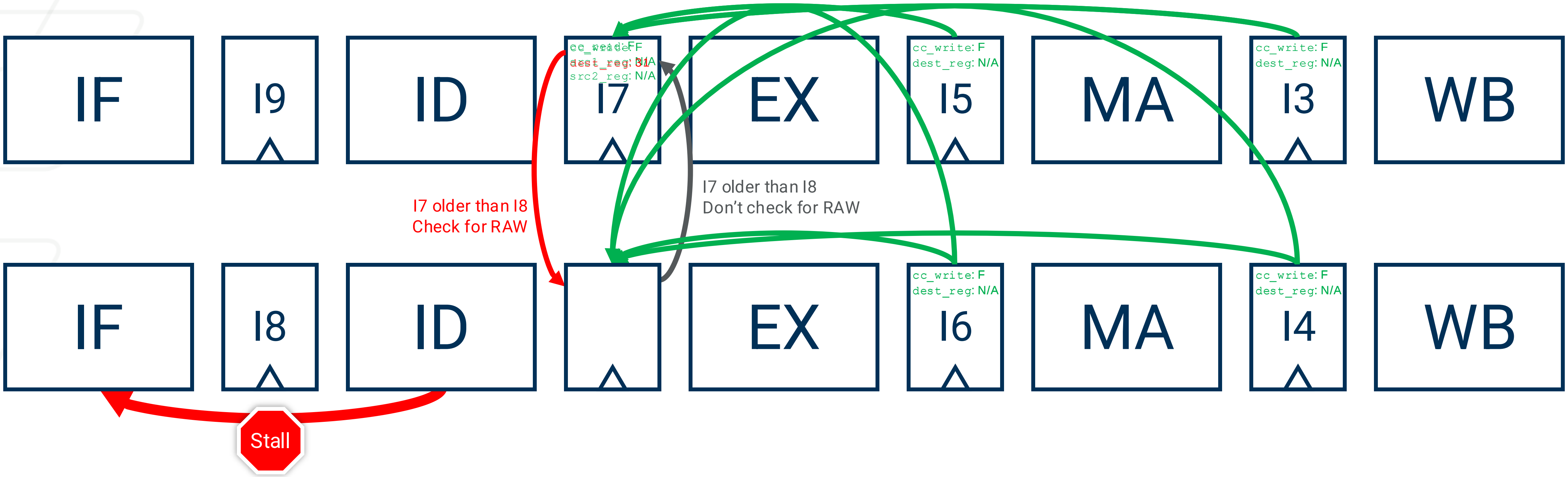
- A stall condition is detected!
  - Example: I5 has `cc_read` and I3 has `cc_write`
    - (Note: cc_write and dest_needed are completely independent!)
- What to do?
  - Insert a bubble into the pipeline (set `valid` to `false` to clear the instruction)
  - Assert a stall signal so that IF does not overwrite the stalled instruction

Handling Stalls



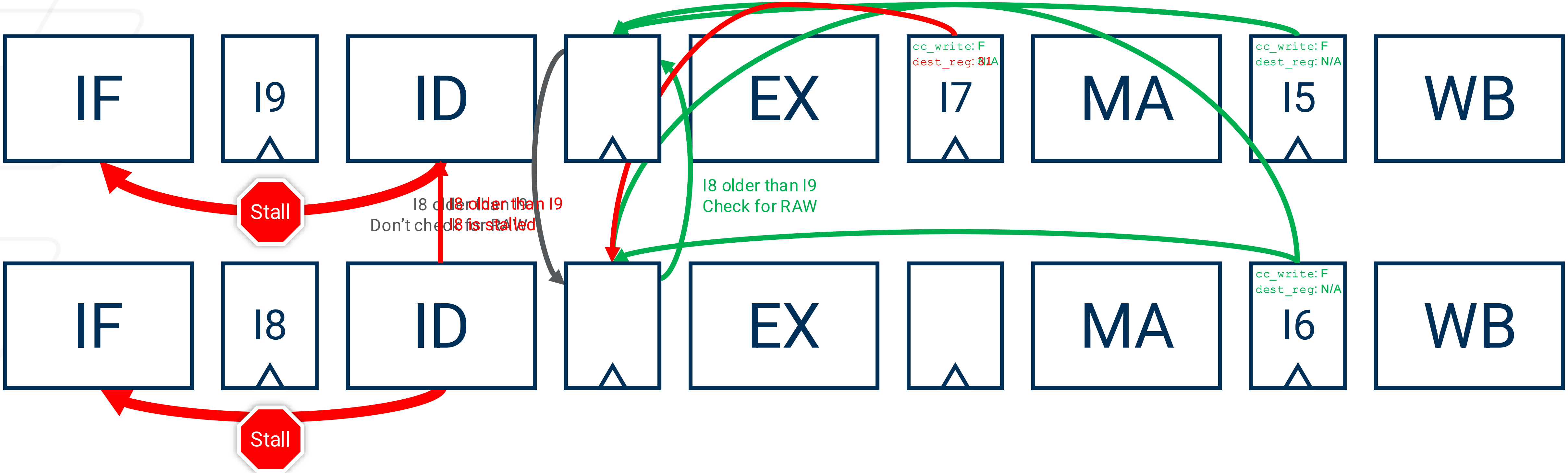
- On each cycle, reevaluate stall condition
  - If stall still present, insert another bubble and keep stall signal asserted
  - Otherwise, remove stall signal

Superscalar Pipeline



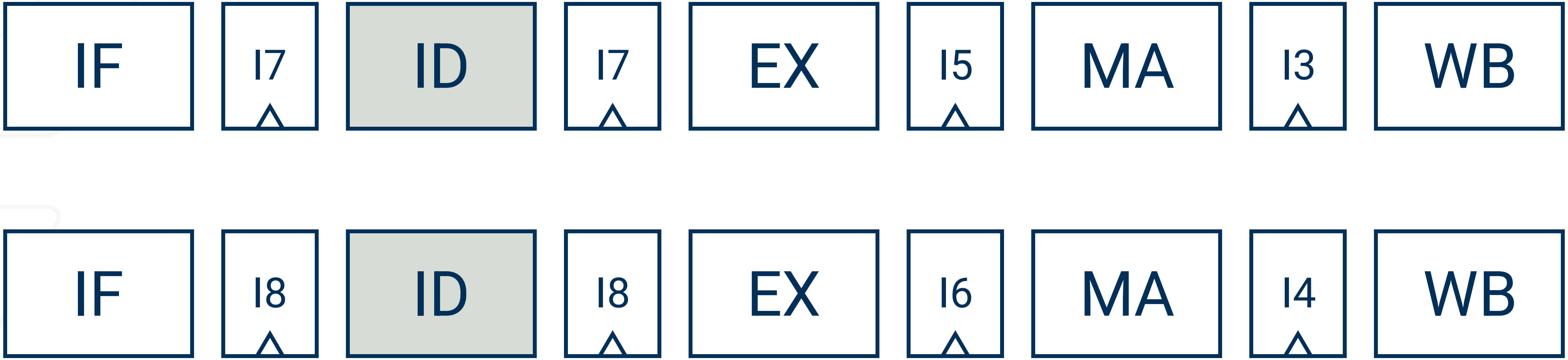
- For superscalar pipelines, add extra stall checks
  - In addition to MA/ID and EX/ID detection of RAW hazards, add ID/ID detection
    - Only consider when other instruction in ID is older

Superscalar Pipeline



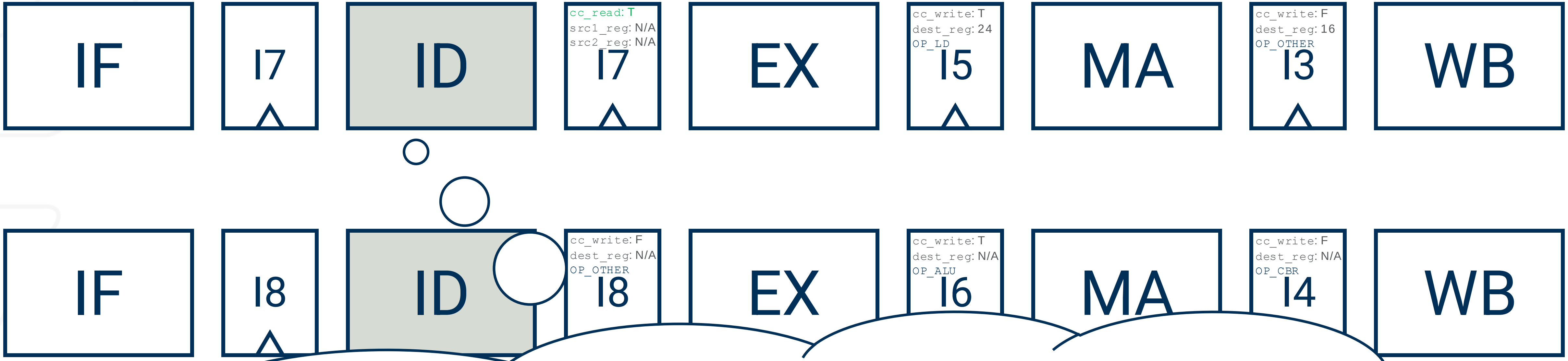
- For superscalar pipelines, add extra stall checks
    - In addition to MA/ID and EX/ID detection of RAW hazards, add ID/ID detection
        - Only consider when other instruction in ID is older
    - Whenever you stall, check for any unstalled instructions that are younger than the stalled one – if any, stall to keep instructions in-order

Forwarding



- `ENABLE_MEM_FWD` → can forward data from MA
- `ENABLE_EXE_FWD` → can forward data from EX *except from loads*
- Must track youngest instruction that writes to each source operand (`src1_reg`, `src2_reg`, `cc`) to see if that operand can be forwarded

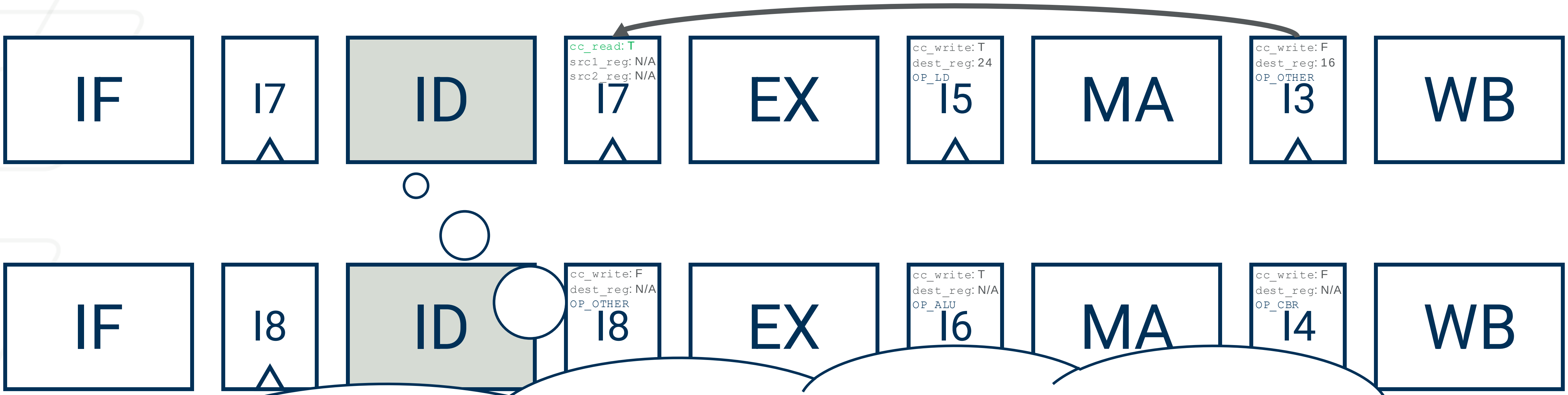
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	Register file			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

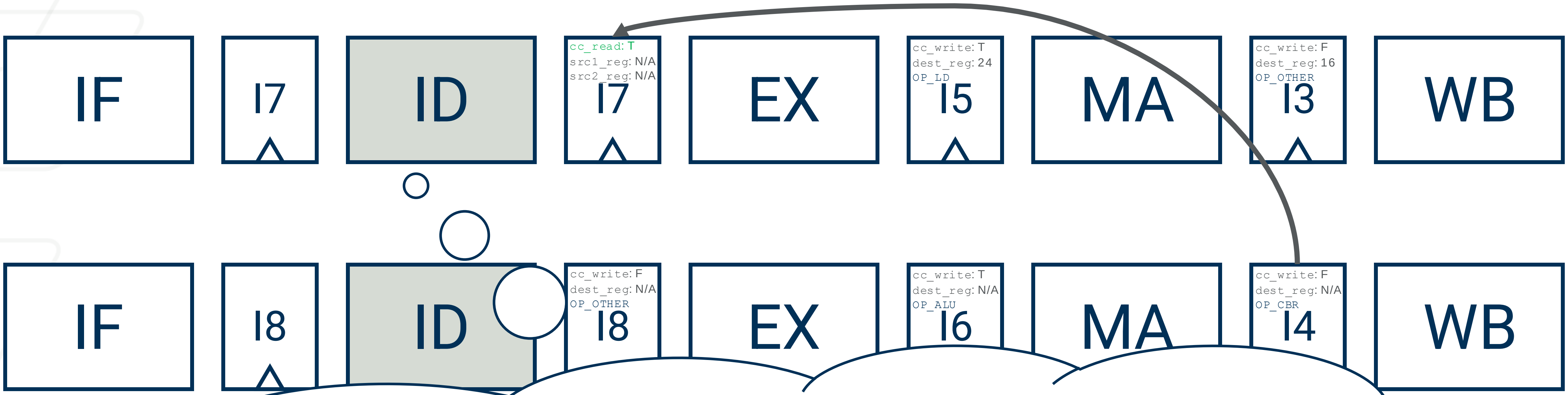
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	Register file			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

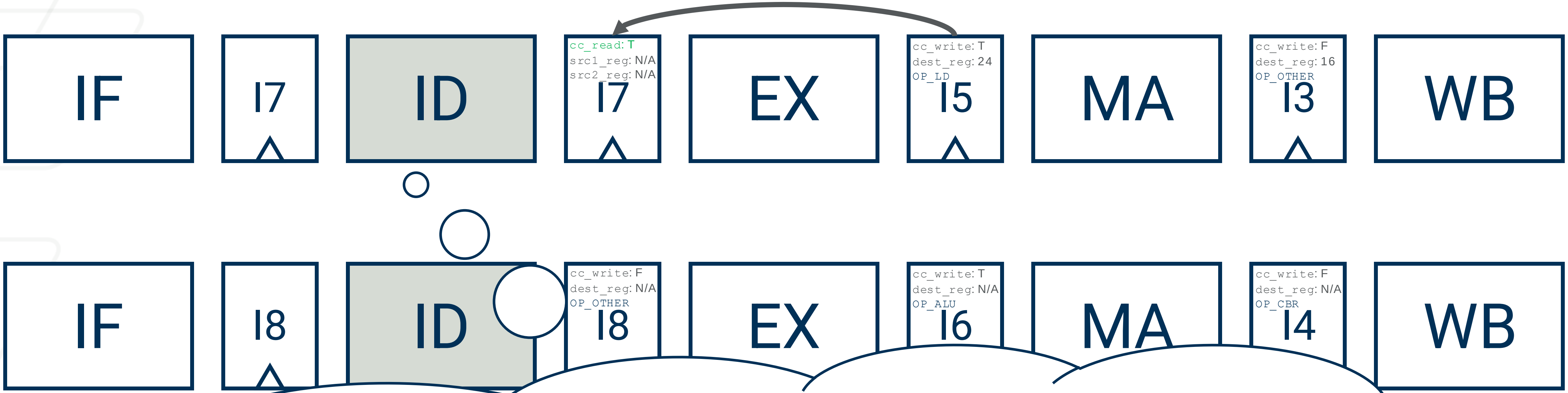
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	Register file			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

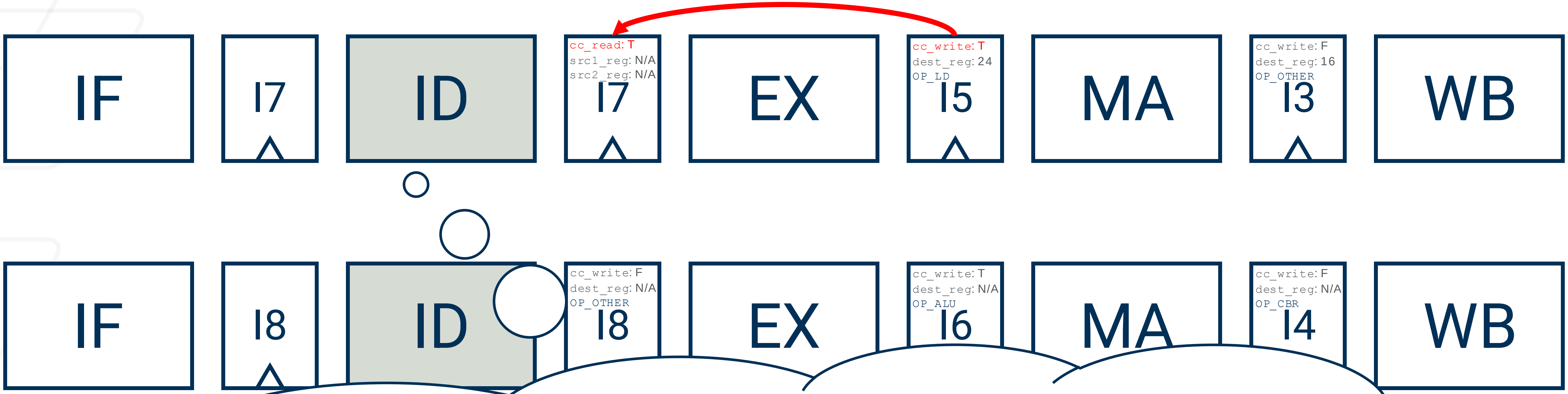
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	Register file			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

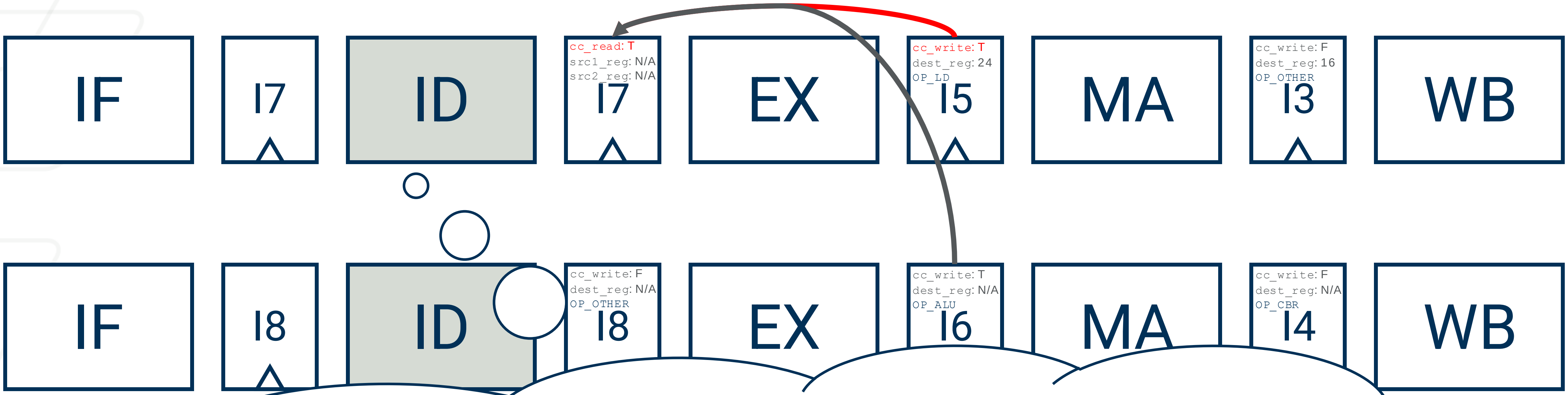
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	I5	EX_LATCH	OP_LD	

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

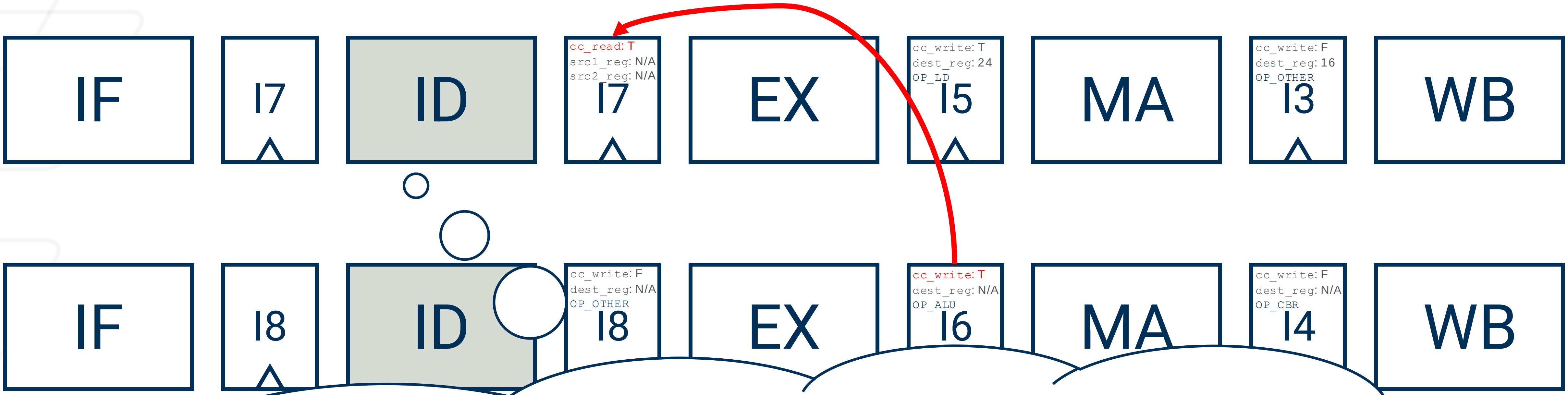
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	I5	EX_LATCH	OP_LD	

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

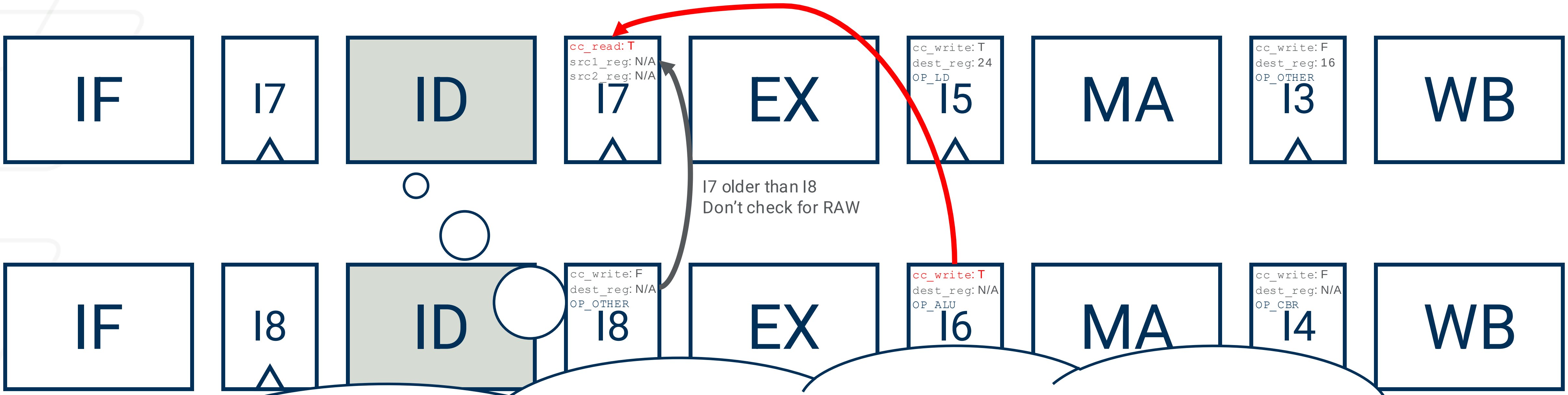
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
`src1_reg`	N/A			
`src2_reg`	N/A			
`cc`	I6	`EX_LATCH`	`OP_ALU`	

I6 is younger than I5 → it will overwrite I5’s `cc`. Track I6 instead.

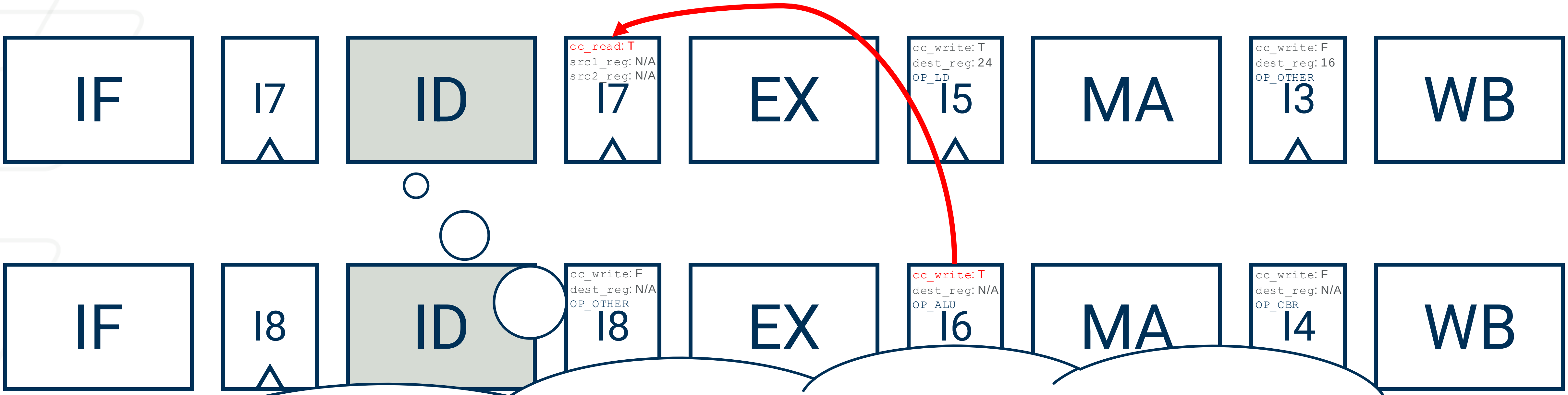
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	I6	EX_LATCH	OP_ALU	

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

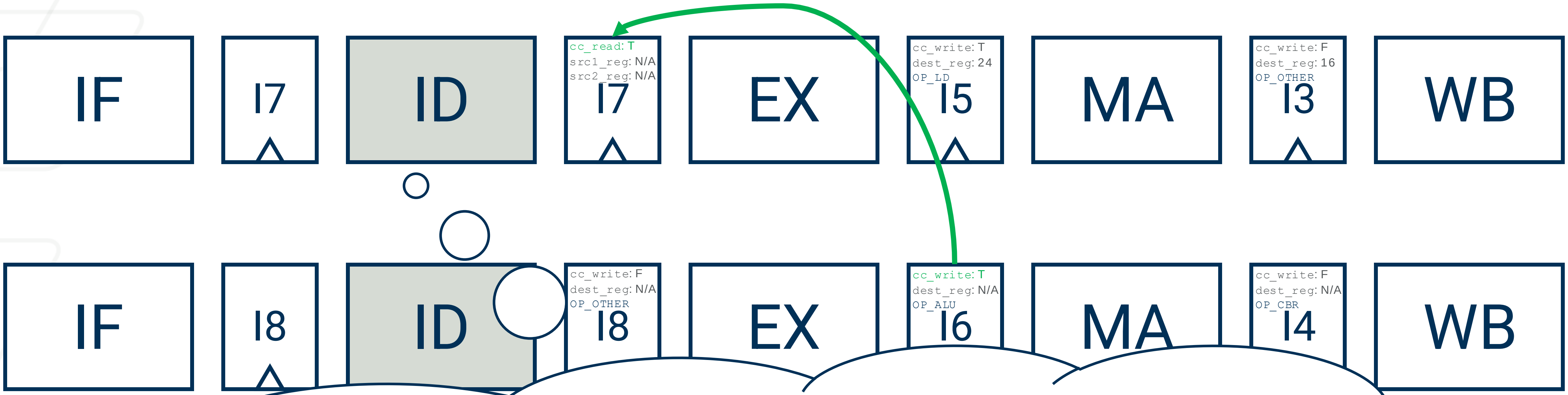
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	I6	EX_LATCH	OP_ALU	

Check if any RAW hazards are resolvable by forwarding.

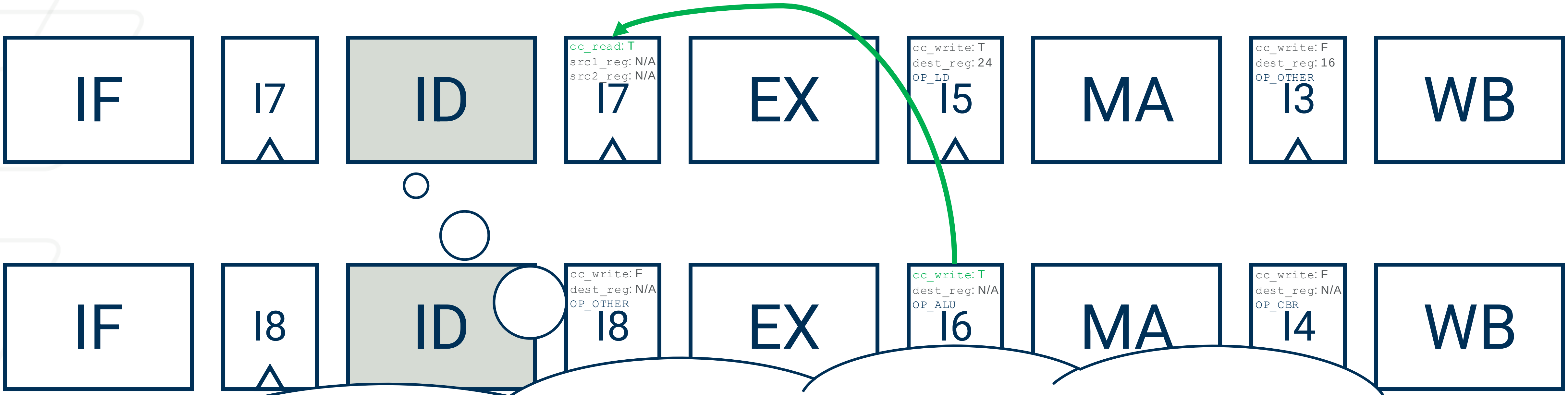
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	I6	EX_LATCH	OP_ALU	Yes

Check if any RAW hazards are resolvable by forwarding.
(Assuming ENABLE_EXE_FWD enabled)

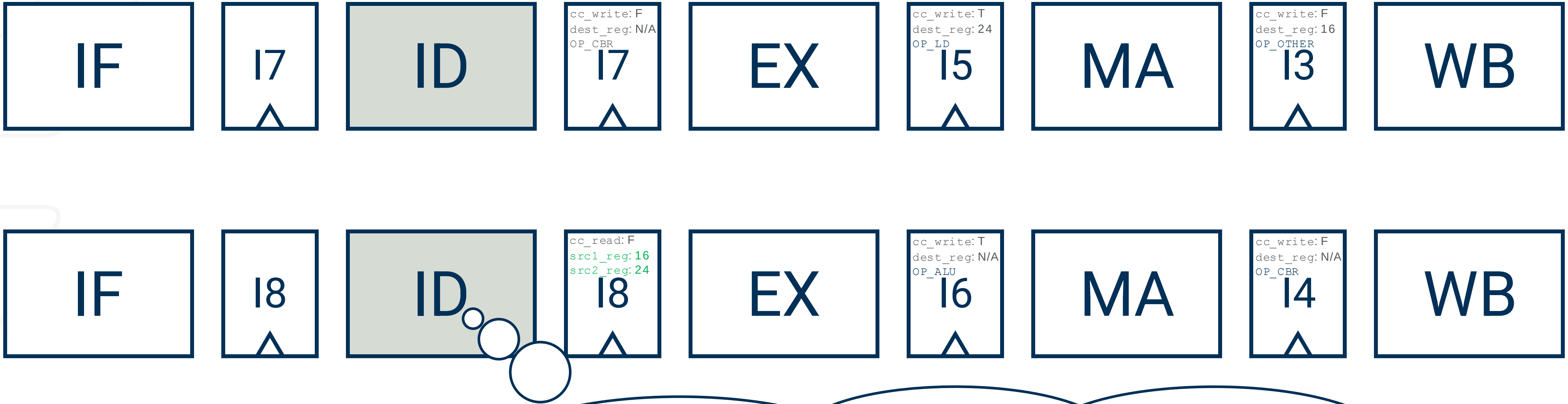
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg	N/A			
src2_reg	N/A			
cc	I6	EX_LATCH	OP_ALU	Yes

All RAW hazards resolved by forwarding → I7 should not be stalled.

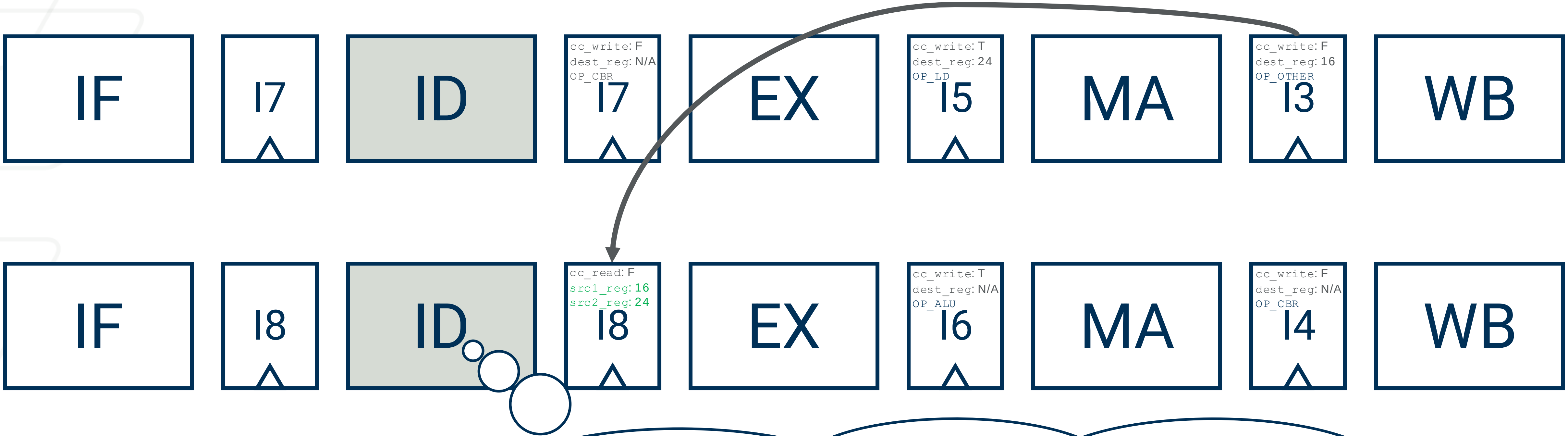
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	Register file			
src2_reg (r24)	Register file			
cc	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

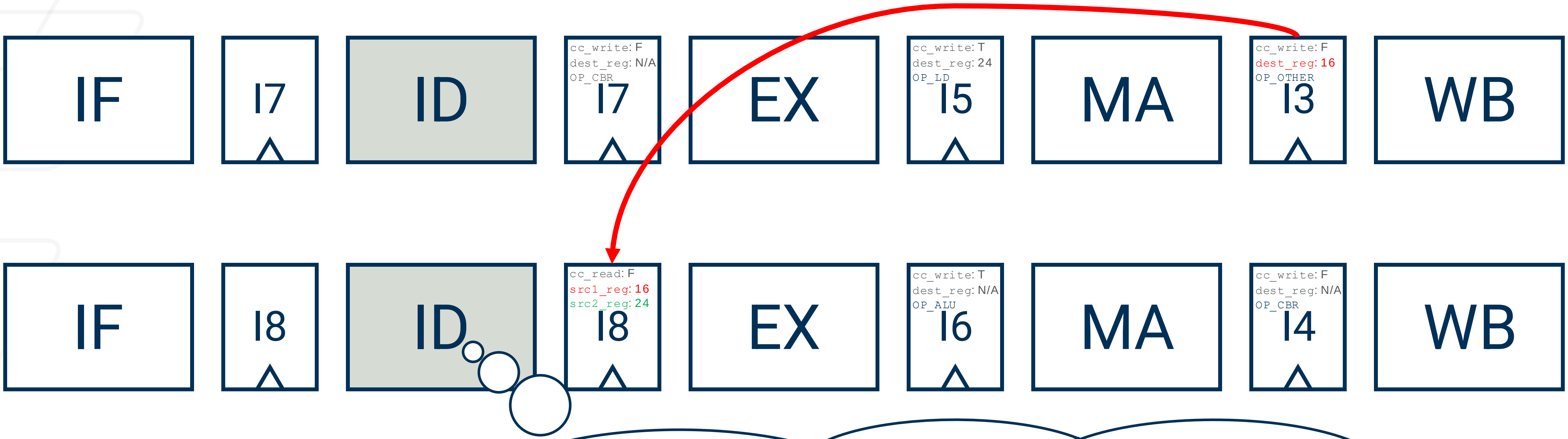
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
`src1_reg` (r16)	Register file			
`src2_reg` (r24)	Register file			
`cc`	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

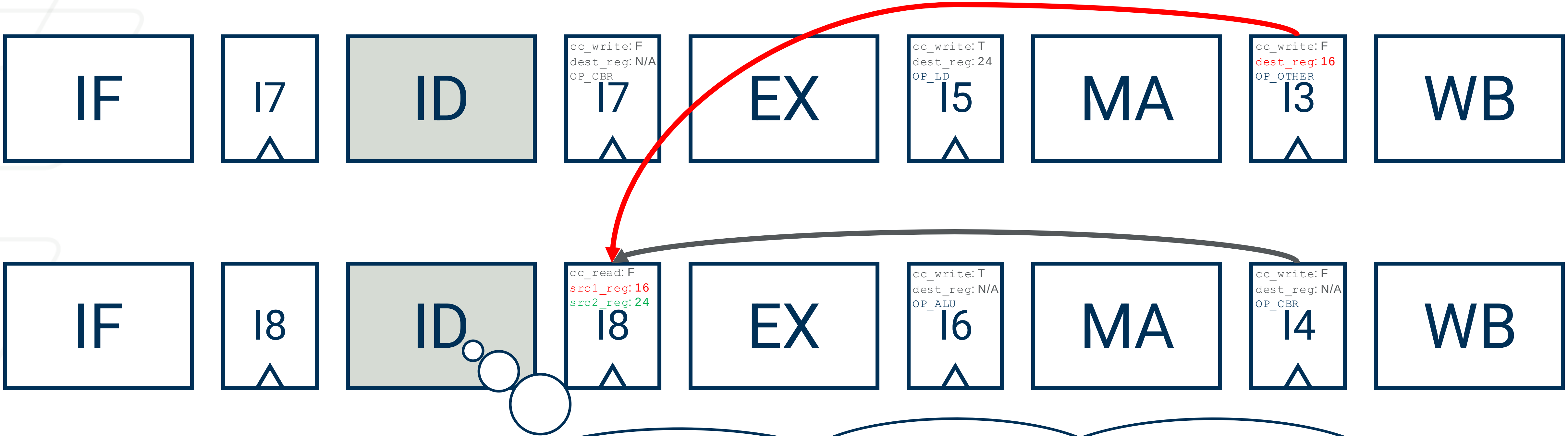
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	
src2_reg (r24)	Register file			
cc	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

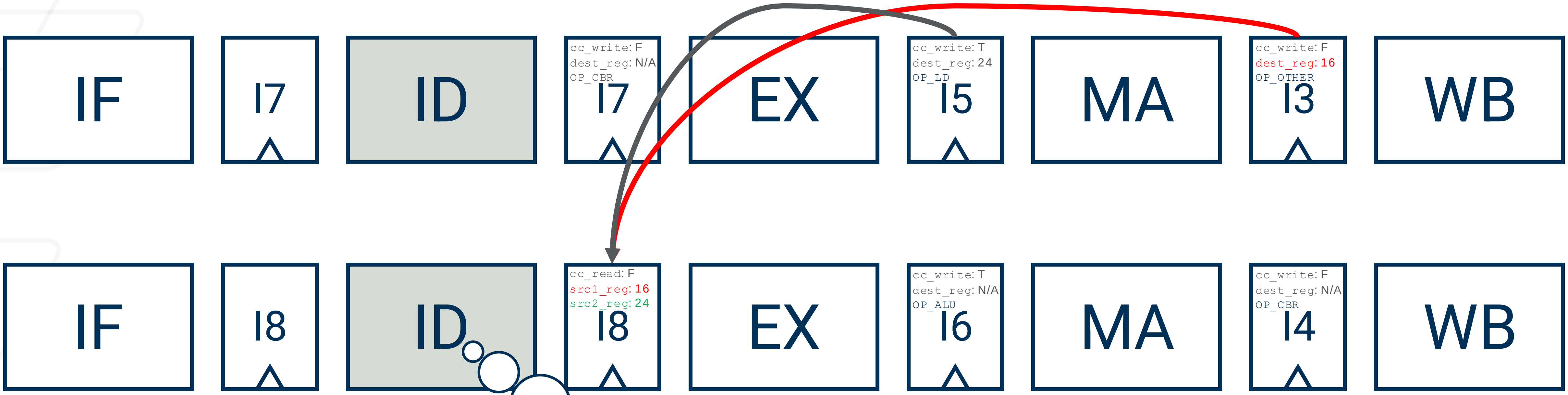
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
`src1_reg` (r16)	I3	`MA_LATCH`	`OP_OTHER`	
`src2_reg` (r24)	Register file			
`cc`	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

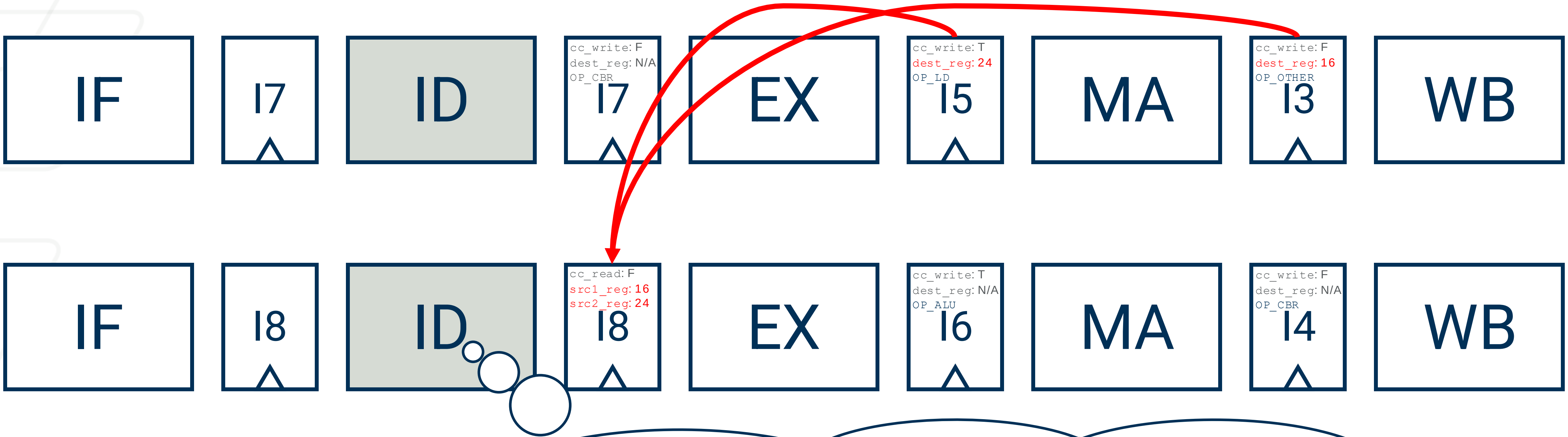
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	
src2_reg (r24)	Register file			
cc	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

Forwarding

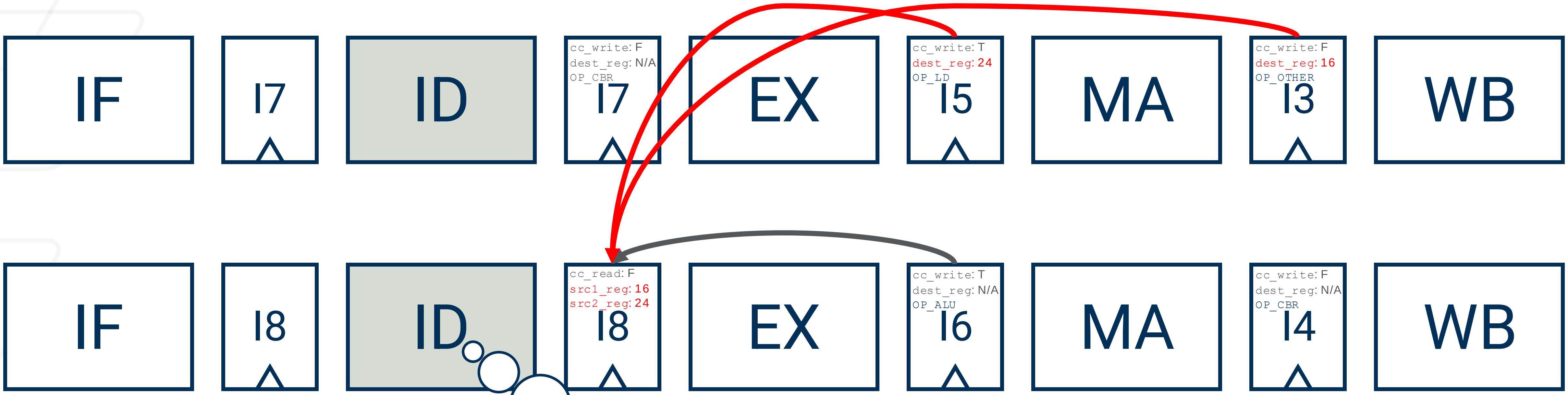


Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	
src2_reg (r24)	I5	EX_LATCH	OP_LD	
cc	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

Forwarding

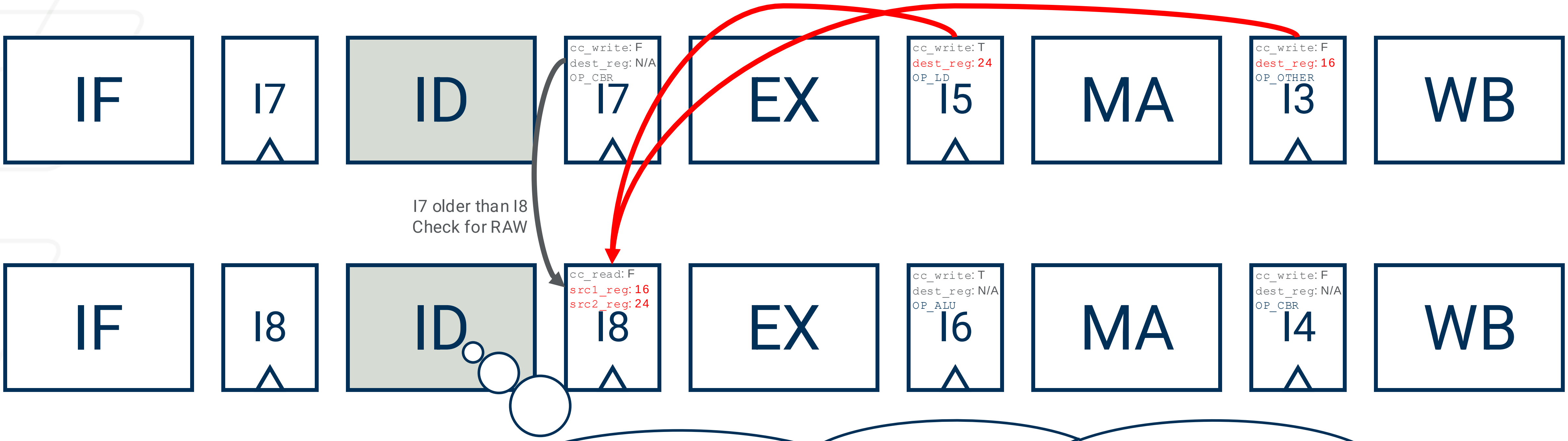
5



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	
src2_reg (r24)	I5	EX_LATCH	OP_LD	
cc	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

Forwarding

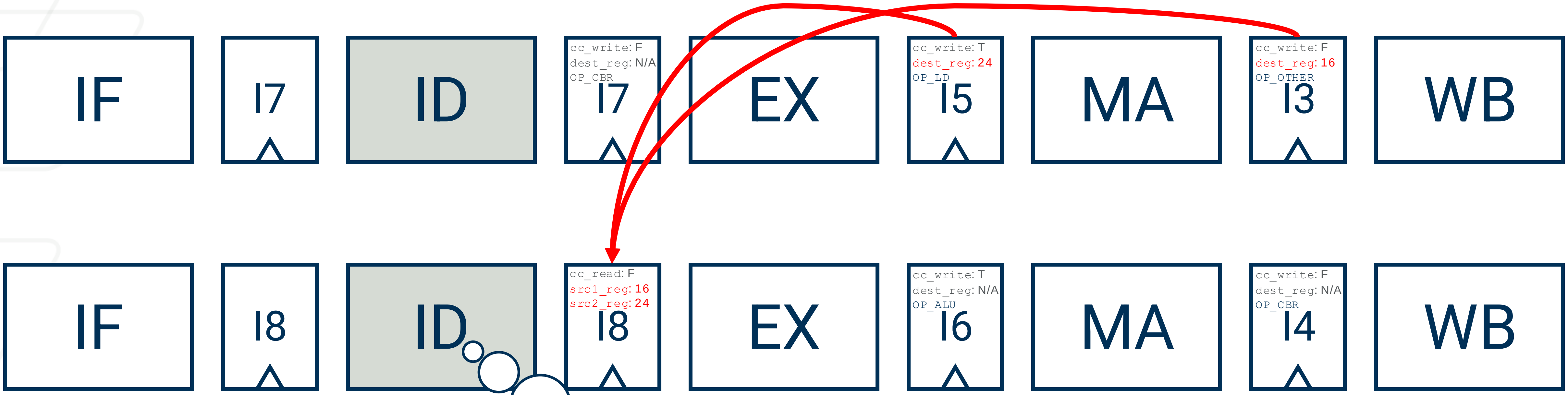


I7 older than I8
Check for RAW

Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	
src2_reg (r24)	I5	EX_LATCH	OP_LD	
cc	N/A			

Iterate through instructions and find any dependencies (potential RAW hazards). Just keep track of them for now.

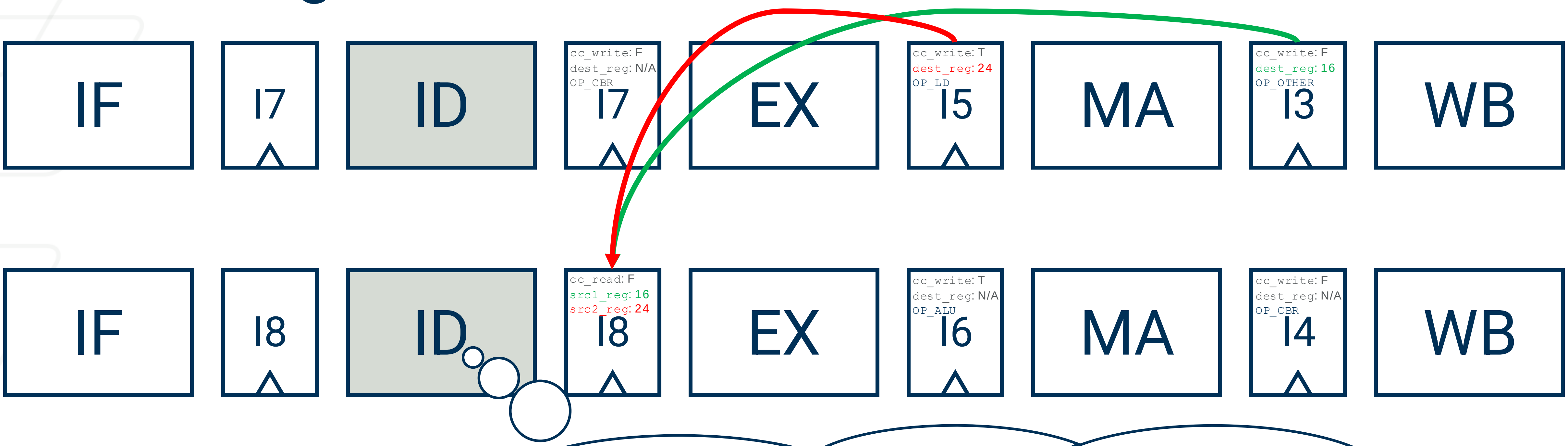
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	Yes
src2_reg (r24)	I5	EX_LATCH	OP_LD	No
cc	N/A			

Check if any RAW hazards are resolvable by forwarding.

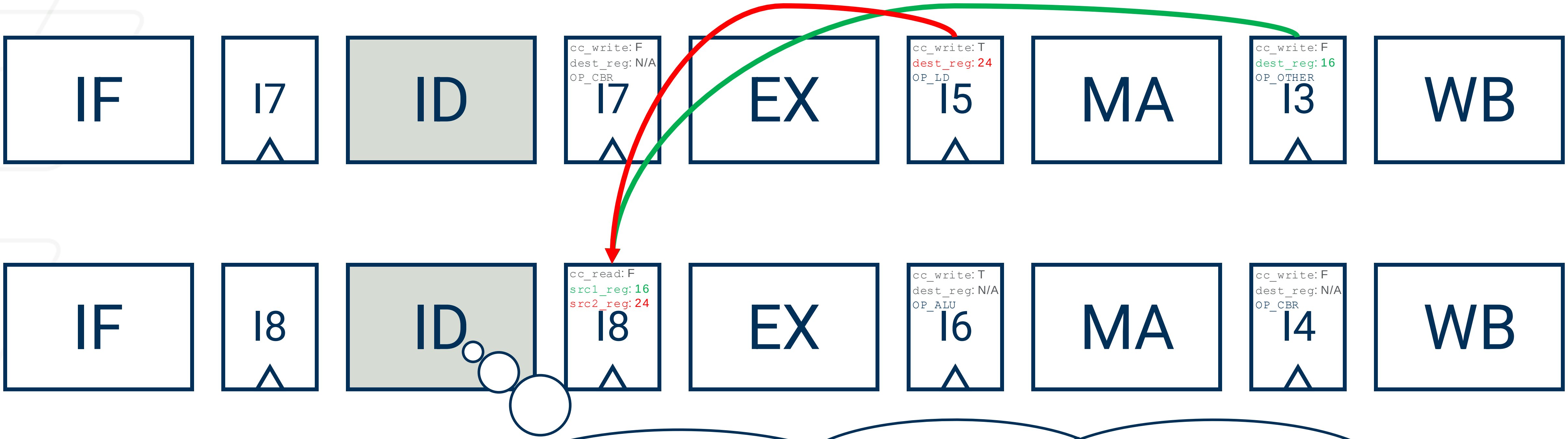
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	Yes
src2_reg (r24)	I5	EX_LATCH	OP_LD	No
cc	N/A			

Check if any RAW hazards are resolvable by forwarding.
(Assuming at least ENABLE_MEM_FWD enabled; OP_LD cannot be forwarded from EX stage even if ENABLE_EXE_FWD enabled)

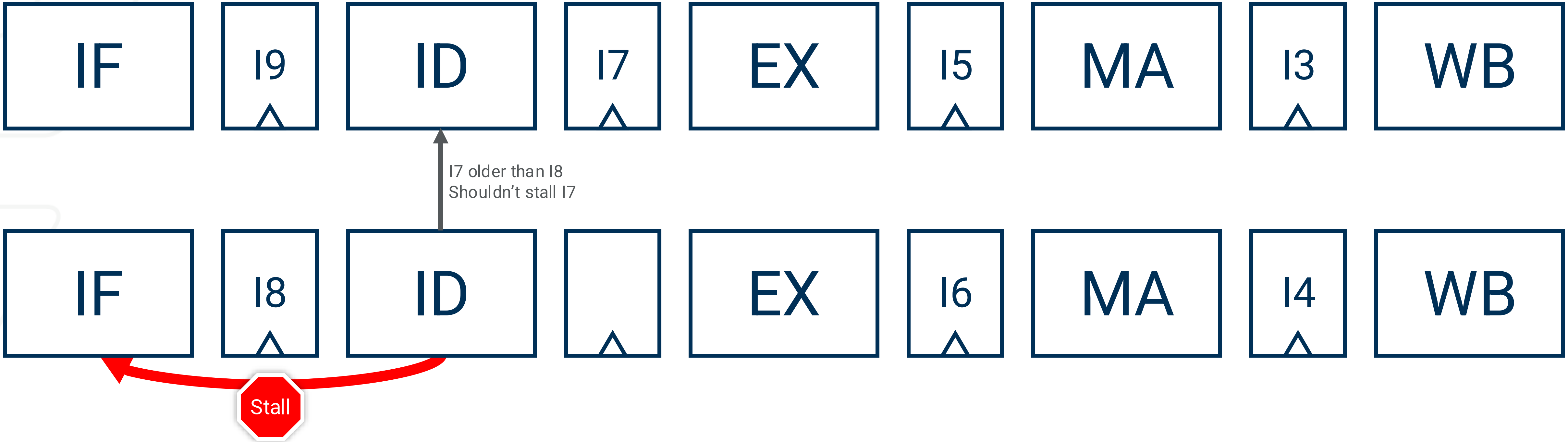
Forwarding



Operand	Read from	Pipe stage	Op type	Can forward?
src1_reg (r16)	I3	MA_LATCH	OP_OTHER	Yes
src2_reg (r24)	I5	EX_LATCH	OP_LD	No
cc	N/A			

At least one RAW hazard not resolved by forwarding → I8 must be stalled.

Forwarding



- `ENABLE_MEM_FWD` → can forward data from MA
- `ENABLE_EXE_FWD` → can forward data from EX *except from loads*
- Must track youngest instruction that writes to each source operand (`src1_reg`, `src2_reg`, `cc`) to see if that operand can be forwarded

Lab 2B

Implementation and Clarifications

Overview

The goal is to implement a simulator for an **in-order 5-stage pipelined Harvard-style CPU** in several steps:

A. Accounting for data hazards — last week and today

  1. …by implementing stalls for a **scalar** machine
  2. …by extending that for **superscalar** machines
  3. …and implementing **forwarding/bypass paths**

B. Accounting for control hazards — today and maybe next week

  A. …by implementing “stalls” for a simple “Always Taken” branch predictor
  B. …by implementing “stalls” for a Gshare branch predictor

Why did I put “stalls” in quotation marks?

- When a real processor sees a branch, it will make a prediction and fetch instructions down that predicted path
  - If the prediction is wrong, the processor must flush the pipeline and start fetching the correct instructions
  - The mispredicted instructions are replaced with NOPs/pipeline bubbles
  - This is effectively the same as any other stall
- In the trace files, there’s no record of the instructions on the mispredicted branch
  - Any work done between misprediction and resolution is killed anyway
  - So, we can simulate the same number of cycles by injecting bubbles directly after a branch misprediction

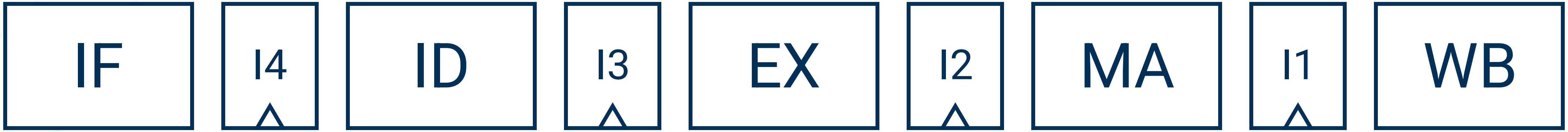
Relevant Points from Assignment Document

- “We will assume that the machine has an idealized Branch Target Buffer (BTB), which identifies the conditional branches (CBR) as soon as the instruction is fetched, and also provides the correct target address.”
  - You can assume that branch targets are known immediately at IF
  - Therefore, correct predictions incur no penalty
- “Your job is to consult direction prediction on instruction fetch. If the prediction is correct, the fetch unit continues to fetch subsequent instructions; otherwise, the fetch unit stalls until the branch resolves.”
  - Besides implementing the predictors, the only other code necessary for part B is calling them from pipeline.cpp and stalling on mispredictions

Relevant Points from Assignment Document

- “You will also need to implement the stall of fetch on branch mispredictions and release the stall when the branch resolves (when the branch is in the MEM stage, however you can fetch only in the next cycle)”
    - Assume the branch outcome is known at the end of MA
        - In our sequential simulator, signaling the release of the stall in MA would cause IF to start fetching the same cycle
    - To have IF start fetching instructions the cycle after the branch resolves in MA, you should release the stall when the branch is past MA, i.e., in WB
        - Since our simulator runs each stage backwards, IF will stop stalling that same cycle

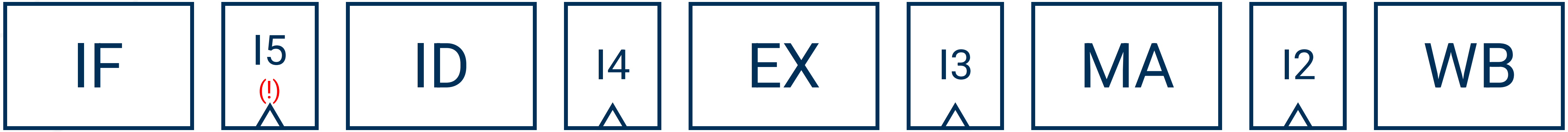
Example



- Assume I5 is a mispredicted branch
- IF is stalled until I5 resolves after MA (once it is in WB)
  - In reality, IF would keep fetching the wrong instructions and kill them all once the branch is resolved
  - We simulate just using a simple stall

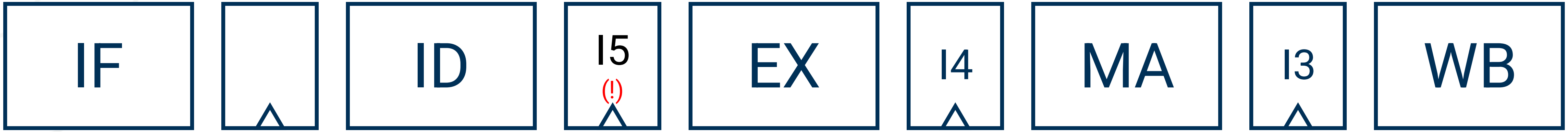
Example

5



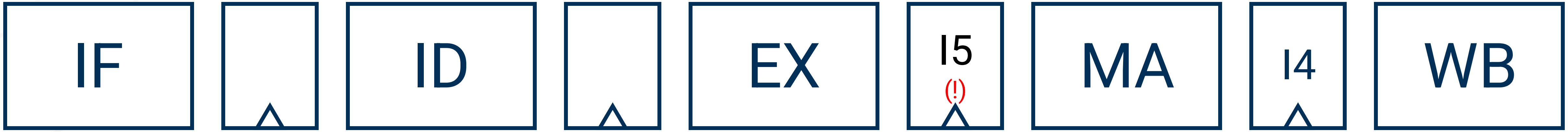
- Assume I5 is a mispredicted branch
- IF is stalled until I5 resolves after MA (once it is in WB)
    - In reality, IF would keep fetching the wrong instructions and kill them all once the branch is resolved
    - We simulate just using a simple stall

Example



- Assume I5 is a mispredicted branch
- IF is stalled until I5 resolves after MA (once it is in WB)
    - In reality, IF would keep fetching the wrong instructions and kill them all once the branch is resolved
    - We simulate just using a simple stall

Example

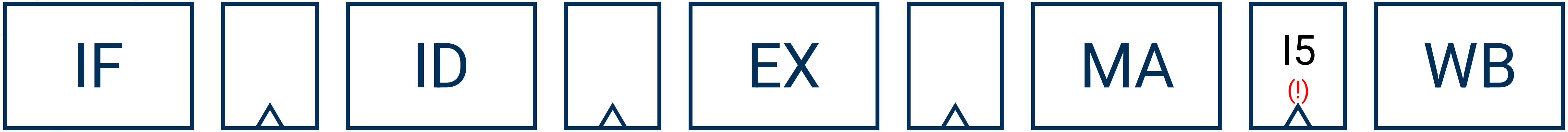


IF ID EX I5 (!) MA I4 WB

- Assume I5 is a mispredicted branch
- IF is stalled until I5 resolves after MA (once it is in WB)
  - In reality, IF would keep fetching the wrong instructions and kill them all once the branch is resolved
  - We simulate just using a simple stall

Example

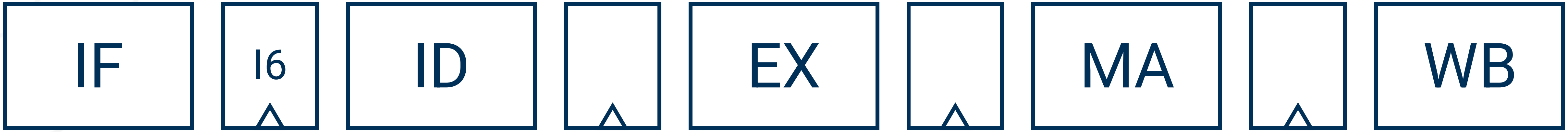
8



Now, the branch has resolved, and IF can start fetching new instructions again.

- Assume I5 is a mispredicted branch
- IF is stalled until I5 resolves after MA (once it is in WB)
  - In reality, IF would keep fetching the wrong instructions and kill them all once the branch is resolved
  - We simulate just using a simple stall

Example



- Assume I5 is a mispredicted branch
- IF is stalled until I5 resolves after MA (once it is in WB)
  - In reality, IF would keep fetching the wrong instructions and kill them all once the branch is resolved
  - We simulate just using a simple stall

Predictors to Implement

- Always taken
  - A stateless predictor that always predicts a branch taken
- Gshare
  - Uses a Pattern History Table (PHT) of 2-bit saturating counters indexed by the XOR of 12-bit Global History Register (GHR) with the lower 12-bits of the instruction PC
  - Required for students in the graduate sections of the course
    - Extra credit for those in the undergraduate sections

Code Architecture

- `class BPred` defined in bpred.h
  - You will be implementing the public methods on this class in bpred.cpp
    - `BPred::BPred(BPredPolicy policy)`
      - Use this to initialize member variables, including the policy this predictor should implement (e.g., `BPRED_ALWAYS_TAKEN` or `BPRED_GSHARE`)
    - `BranchDirection BPred::predict(uint64_t pc)`
      - Should return a prediction for the branch with the given address according to the policy
    - `void BPred::update(uint64_t pc, BranchDirection prediction, BranchDirection resolution)`
      - Used to update branch predictor state
  - You can modify the header file to add other methods or member variables
    - e.g., to hold any data needed for Gshare
- Other enums and utility functions defined in bpred.h

Global History Register (GHR)

- The global history register (GHR) stores the directions (taken or not taken) of the most recent branches in the program.
- For this assignment, the GHR is 12 bits wide.

Example:

- I4: `beq addr_x` (taken)
- I13: `bne addr_y` (not taken)
- I20: `beq addr_z` (not taken)

Global History Register

0	0	0	0	0	0	0	0	0	0	0	0

Global History Register (GHR)

- The global history register (GHR) stores the directions (taken or not taken) of the most recent branches in the program.
- For this assignment, the GHR is 12 bits wide.

Example:

- **I4: `beq addr_x` (taken)**
- I13: `bne addr_y` (not taken)
- I20: `beq addr_z` (not taken)

Global History Register

0	0	0	0	0	0	0	0	0	0	0	1

Global History Register (GHR)

- The global history register (GHR) stores the directions (taken or not taken) of the most recent branches in the program.
- For this assignment, the GHR is 12 bits wide.

Example:

- I4: `beq addr_x` (taken)
- **I13: `bne addr_y` (not taken)**
- I20: `beq addr_z` (not taken)

Global History Register

0	0	0	0	0	0	0	0	0	0	1	0

Global History Register (GHR)

- The global history register (GHR) stores the directions (taken or not taken) of the most recent branches in the program.
- For this assignment, the GHR is 12 bits wide.

Example:

- I4: `beq addr_x` (taken)
- I13: `bne addr_y` (not taken)
- **I20: `beq addr_z` (not taken)**

Global History Register

0	0	0	0	0	0	0	0	0	1	0	0

Pattern History Table (PHT)

- In Gshare, the pattern history table (PHT) records the outcomes of branches that were seen when a certain value of (GHR XOR lower 12 bits of PC) was matched.
  - In other words, it's a table indexed on (GHR XOR lower 12 bits of PC).
- Each entry is a 2-bit saturating counter: incremented when a branch is taken, up to a maximum value of 3 (binary 11), and decremented when a branch is not taken, down to a minimum value of 0.
  - Utility functions for saturating increment and decrement are provided for convenience.
- You should predict a branch taken when the matching entry of the PHT is ≥2, or not taken otherwise.

Questions?

- Ask away!
- Also feel free to ask on Piazza or by email:
  - seokjin.go@gatech.edu
  - seonho.lee@gatech.edu