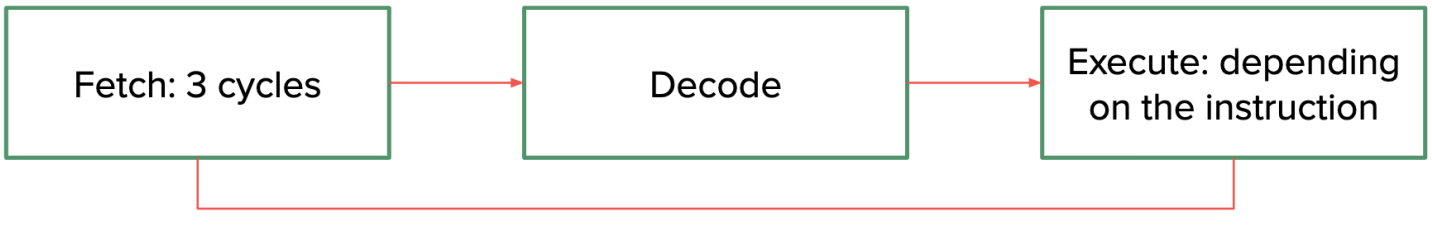
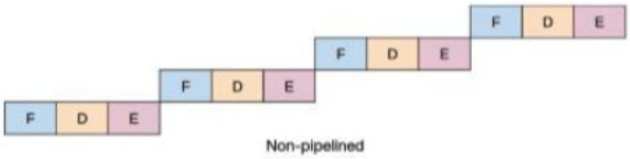
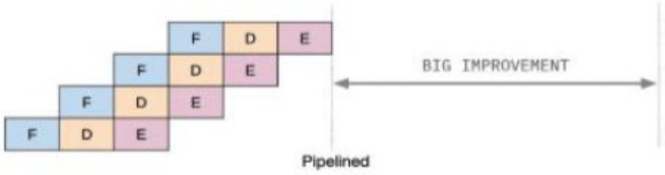
Why Pipelining?

- Currently, we can only execute one thing per cycle



Let's try to make this better!





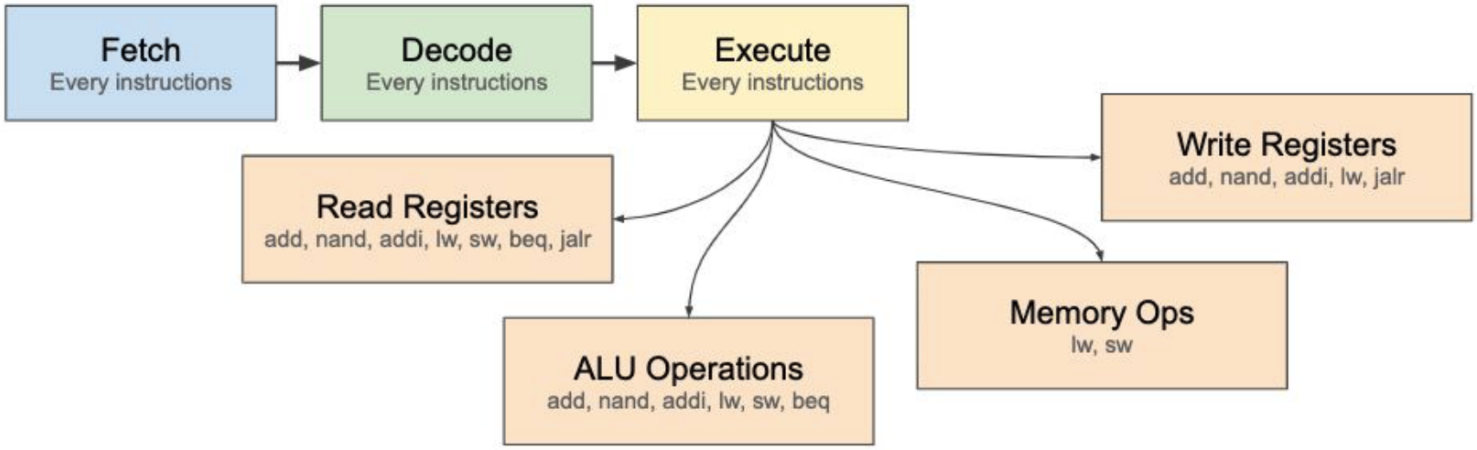
Pipeline Conscious Architectural Design

- Need equal amount of work in each cycle
  - Say you have one person working at each station to add different ingredients to the sub
  - Each person needs to have an equal amount of work to make sure that no one person is holding up the line
- The clock cycle is equal to the length of the longest task, so if each cycle contains just about the same amount of work, you can minimize the clock cycle
- Need a symmetric instruction format:
  - All stages need to have the same registers and hardware components for the pipeline to work (well)
  - Not providing all stages with the same register specifications or size and position offsets is like giving a Subway station worker the tools used by Arby's to make sandwiches
- Need all stages to have access to all resources:
  - Say a customer wants cheese toasted with their bread, and another customer wants the cheese added to the sandwich directly with the vegetables
  - What happens if only the bread station, or only the veggies station, has access to the cheese?
  - In terms of our instruction set, for instance, both the fetch and execute macrostates need the ALU, so the ALU, along with other resources, needs to be accessible to all stages

Summary of Instructions

- Read instructions
- ALU ops
- Memory ops

- Write registers



Latency and Throughput

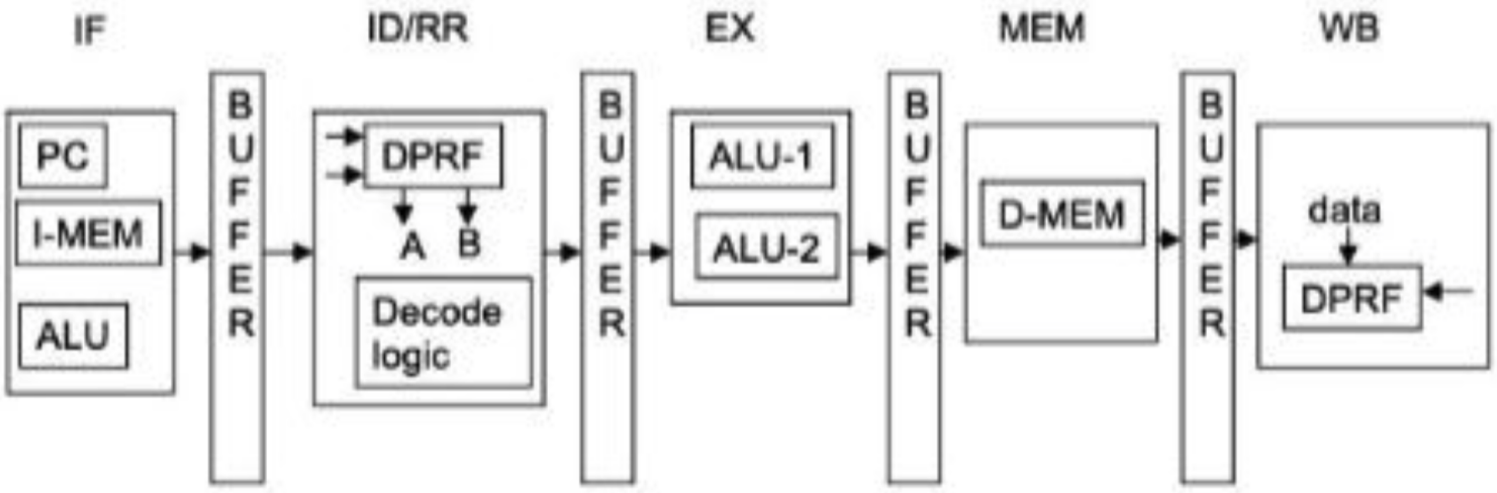
Latency: the amount of time it takes for a single instruction to execute start to finish

- More steps = more latency
- Smaller latency = better

Throughput: the number of instructions you can execute concurrently in a pipeline

- Increases if we divide instructions into more steps

The Pipeline



IF

- fetch instruction from memory
- increment PC
- all instructions need fetch, so this needs to be its own stage

ID/RR

- decode and read if necessary
- all instructions need to be decoded
- some instructions need to be read from registers

EX

- arithmetic and logic operations
- ALU
- offsets
- branch requires two instructions

MEM

- data memory read/write
- load word or read word actions

WB

- stores results back into registers

Pipelining Buffers

Pipeline buffers store related instruction information and partial results b/w stages

- buffer size = maximum required for the passage of any instruction
- buffer content = depend on the stage and instruction (opcode)

FBUF

- contains IF output
- fetched instruction buffer

DBUF

- contains ID/RR output
- decoded instruction and associated register values

EBUF

- contains EX output
- instruction and results from ALU operations

MBUF

- contains MEM output
- EBUF contents + results from memory op

Pipelining Tracing: ADD

```
ADD $Rx, $Ry, $Rz
→ $Rx = $Ry + $Rz
```

General format:

- IF: Fetch instruction
- ID/RR: Decode, read $Ry, $Rz
- EX: Add $Ry and $Rz
- MEM: nothing
- WB: Write ALU result to $Rx

IF: Fetch

- Put I-MEM[PC] in FBUF
- `PC = PC + 1`

ID/RR: Decode, read `$Ry/$z`

- Opcode, $Rx, $Ry, $Rz
- Put DPRF[$Ry] into DBUF as A
- Put DPRF[$Rz] into DBUF
- Put opcode into DBUF
- Put $Rx into DBUF

EX: Add $Ry and $Rz

- Calculate $A + B$
- Put result of $A + B$ into EBUF
- Copy opcode from DBUF to EBUF
- Copy $Rx from DBUF to EBUF

MEM: Nothing

WB: Write ALU result to $Rx

- DPRF[$Rx] = ALU result from MBUF

General Comments

- ID/RR decodes the instruction and provides all parts (register numbers, opcode, offsets in other instructions) needed in future stages
- We passed along the opcode in each stage so that we knew what to do in each stage

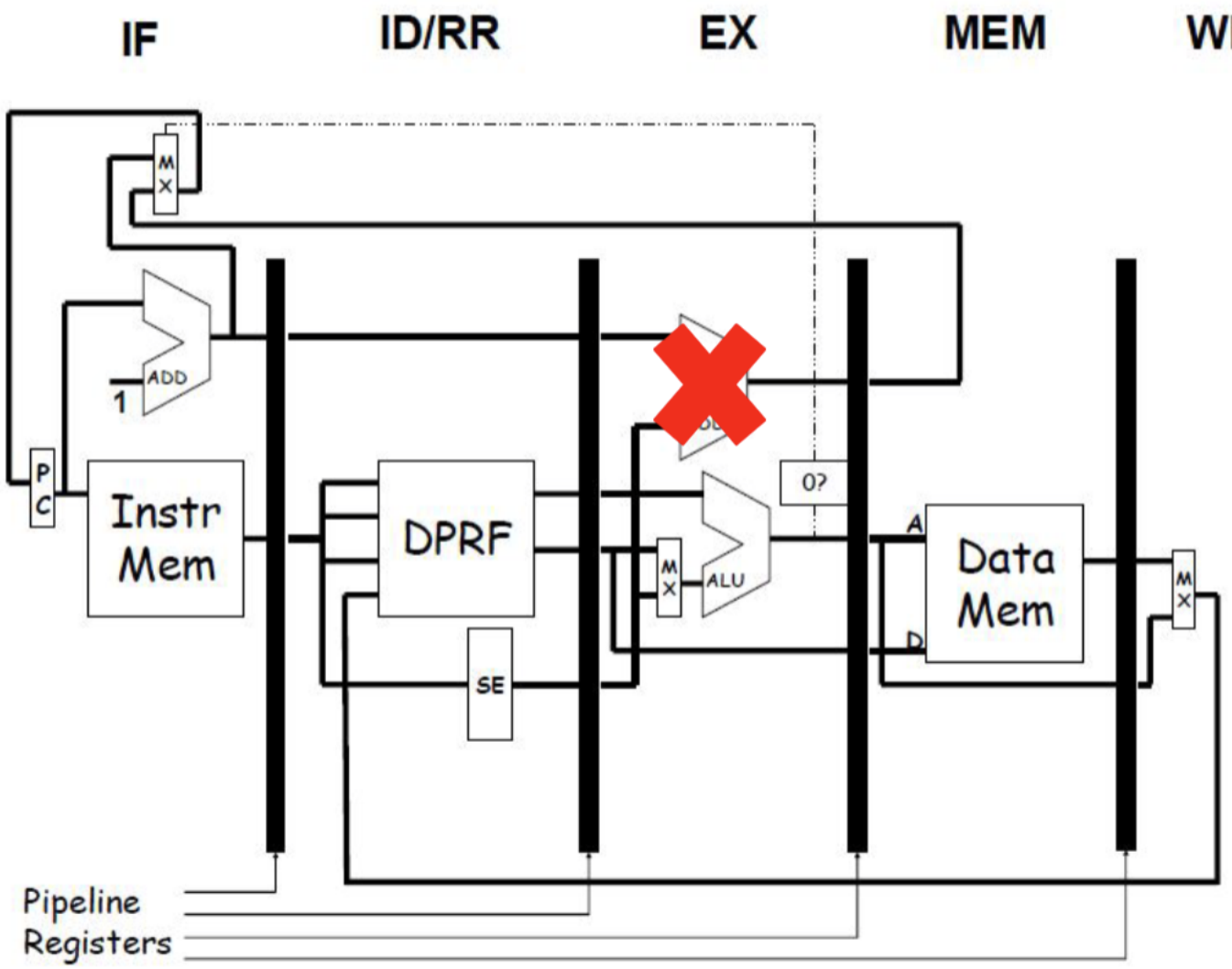
- $Rx was passed all the way to WB so that we knew which register to write to

Hazards

- Pipelines are great at optimization, however we often run into **Hazards**
  - Concurrency causes problems! E.g. you can read and write to a register at the same time
- Solution 1: Halt, Flush the program and start over
  - Problem: slow and inefficient, defeats the purpose of pipelining
- Sollution 2: Introduce bubbles to stall operation
- NOP - no operation
- Types of Hazards
  - Structural
  - Data
  - Control

Structural Hazards

- occur due to a lack of or conflict between hardware components
- BEQ needs to calculate `A - B` and `PC+offset` at the same time
- If we only have one ALU, we can only calculate one
  - Need to delay an extra cycle
- Solution: Add another ALU



Data Hazards

- Three types of Data hazards (RAW, WAR, WAW)
  - WAR and WAW aren't problems in LC-2200
- Read After Write (RAW) Hazard
  - When is the Register Update?
  - What happens when we access R1 in I2?
  - How many NOPs do we need?

I_1: R1 <- R2 + R3

I_2: R4 <- R1 + R5

RAW hazard - read after write

I_1: R4 <- R1 + R5

I_2: R1 <- R2 + R3

WAR hazard - write after read

I_1: R1 <- R4 + R5

I_2: R1 <- R2 + R3

WAW hazard - write after write

I_1: R1 <- R2 + R3

I_2: R4 <- R1 + R5

R1 was update at T5 but was accessed at T3!!

	IF	ID/RR	EX	MEM	WB
T1	I_1				
T2	I_2	I_1			
T3		I_2	I_1		
T4			I_2	I_1	
T5				I_2	I_1
T6					I_2

	IF	ID/RR	EX	MEM	WB
T1	I_1				
T2	I_2	I_1			
T3		I_2	I_1		
T4		I_2	NOP	I_1	
T5		I_2	NOP	NOP	I_1
T6		I_2	NOP	NOP	NOP
T7			I_2	NOP	NOP
T8				I_2	NOP
T9					I_2

How many bubbles?

I_1: add R1, R2, R3

I_2: lea R2, LABEL

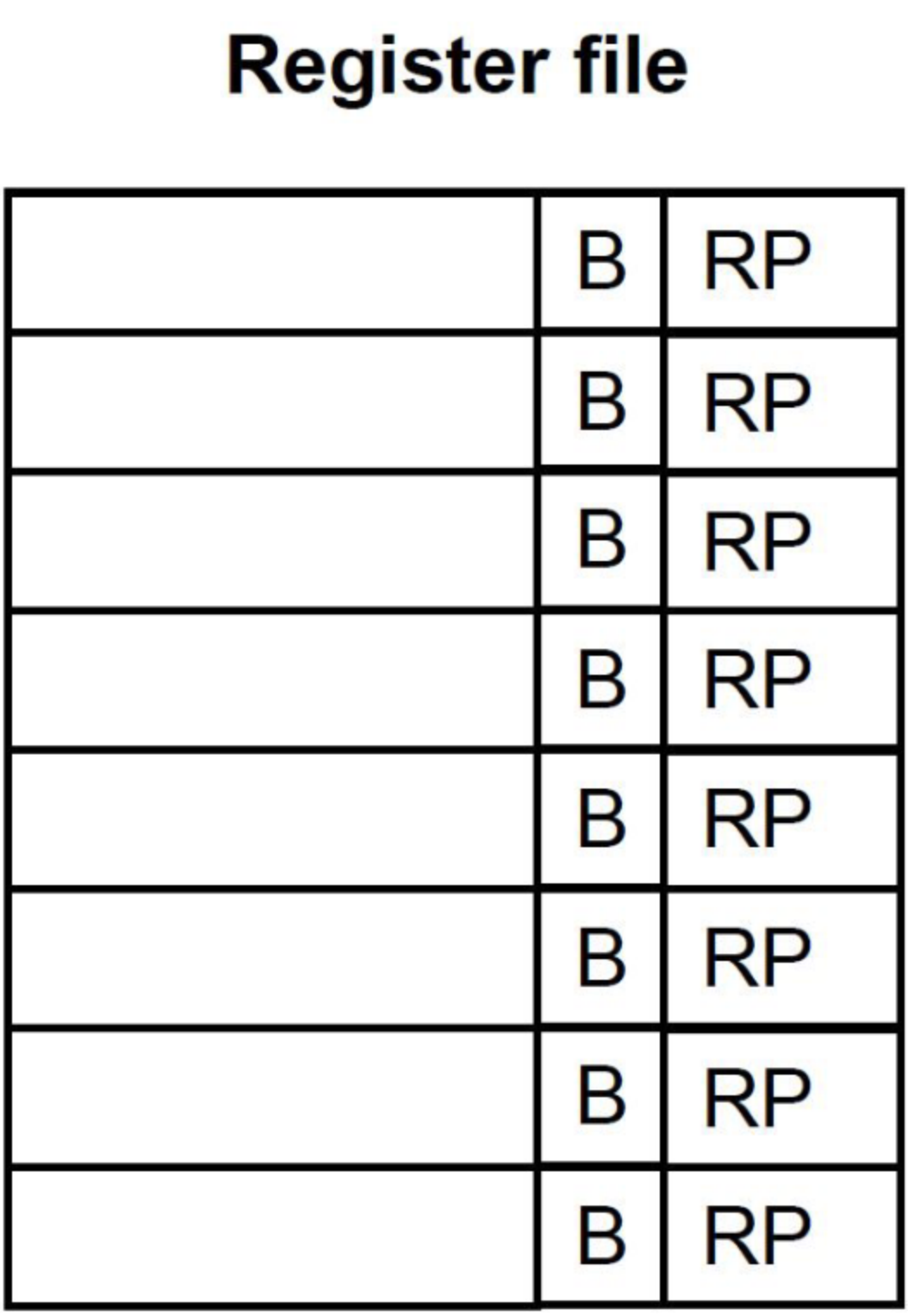
I_3: add R4, R1, R2

	IF	ID/RR	EX	MEM	WB
T1	I_1				
T2	I_2	I_1			
T3	I_3	I_2	I_1		
T4		I_3	I_2	I_1	
T5		I_3	NOP	I_2	I_1
T6		I_3	NOP	NOP	I_2
T7		I_3	NOP	NOP	NOP
T8			I_3	NOP	NOP
T9				I_3	NOP
T10					I_3

Data Forwarding

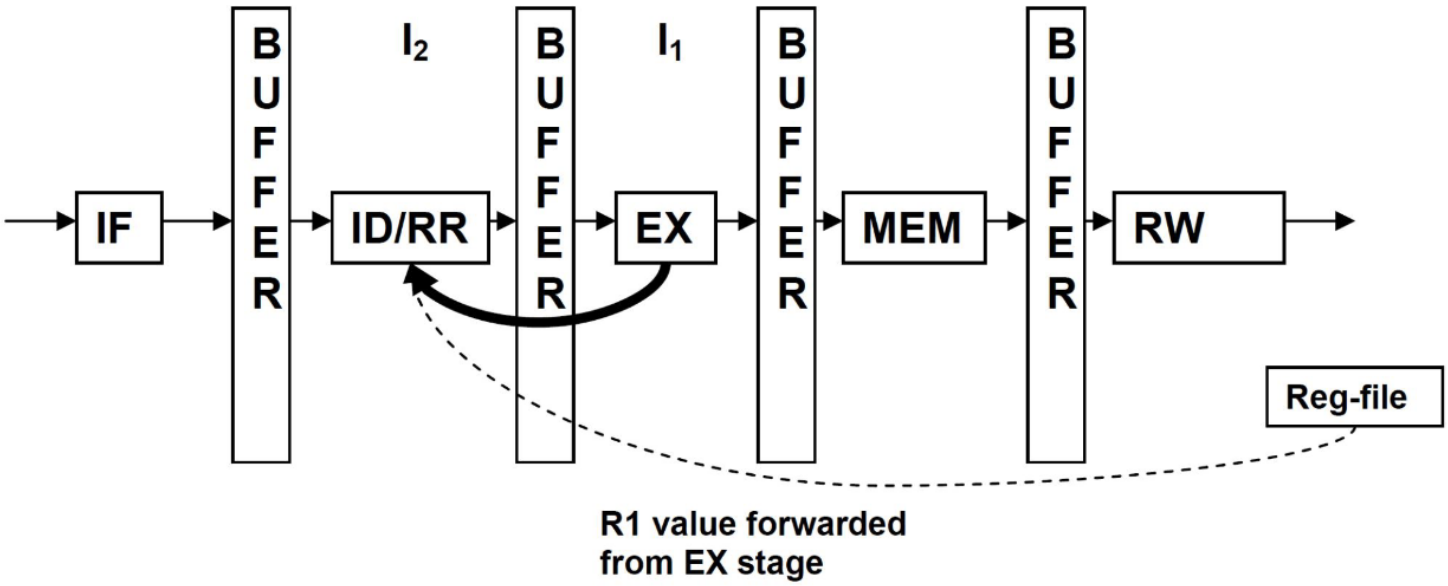
In a RAW hazard: we calculate the value we need in EX... why do we need to wait until WB to use it?

- Need a few changes to the register file
  - **Busy bit** - the register will be written to by an instruction currently being executed
  - Cleared in WB
  - **Read pending** - the register is being read from this cycle
  - Both ID/RR



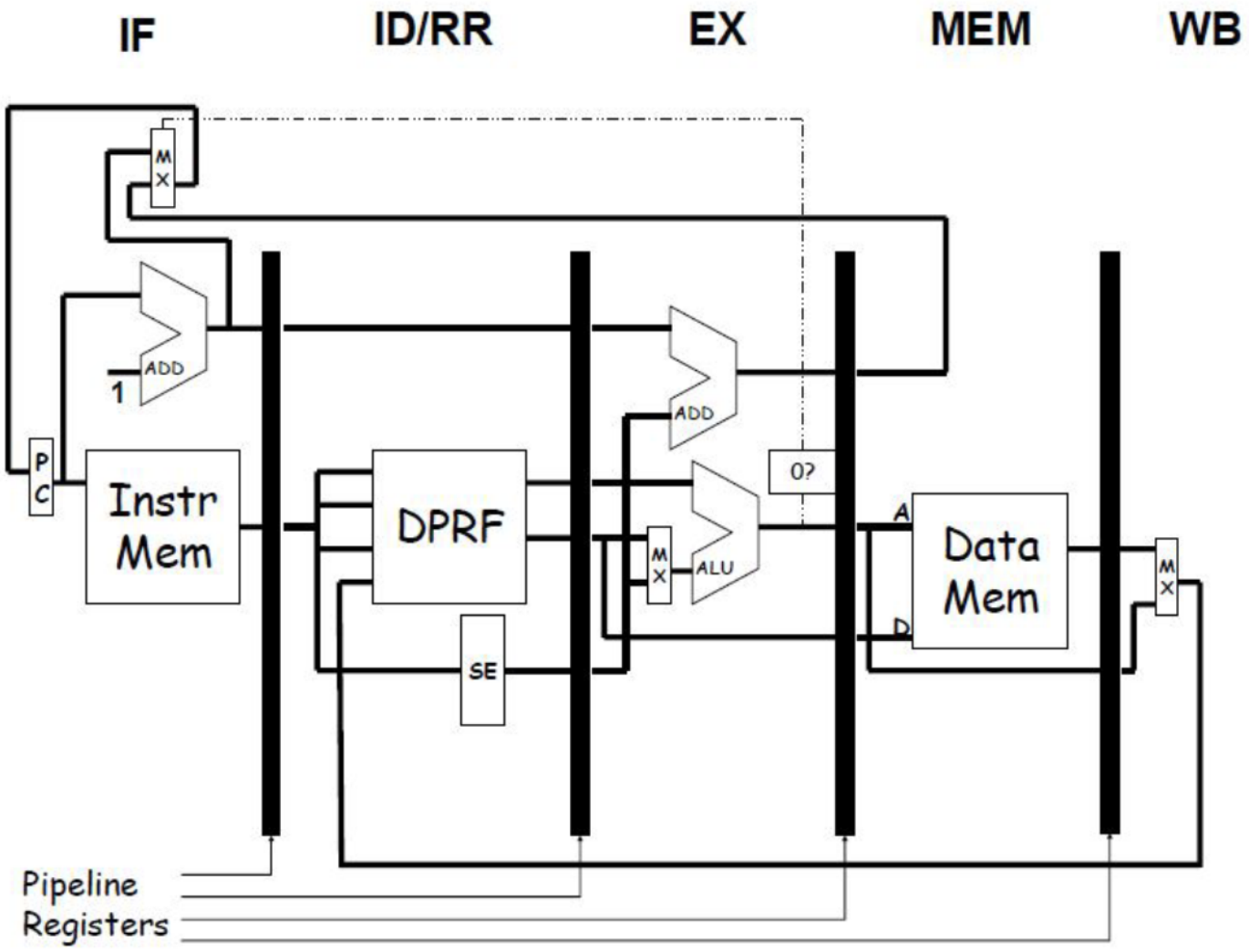
If an instruction in EX, MEM, or WB sees the RP bit set for a register it has a new value for, it can forward the data to the ID/RR stage for use there.

- Needs lots of additional wiring; connecting RP bits to stages, connecting values in later stages to ID/RR
- But... no bubbles needed! (We can use the values right away)
- Remember -- if we're writing a value to a register, we must be passing the register number along, so we can identify the RP bit for the right register in later stages



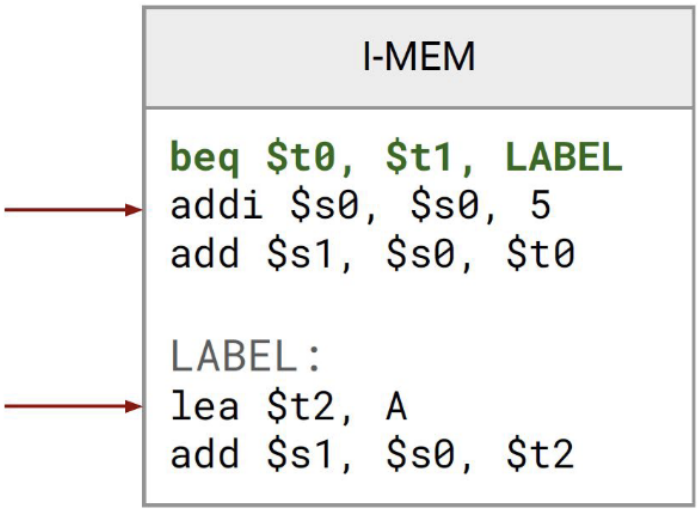
Handling Hazards: Pipeline Branching

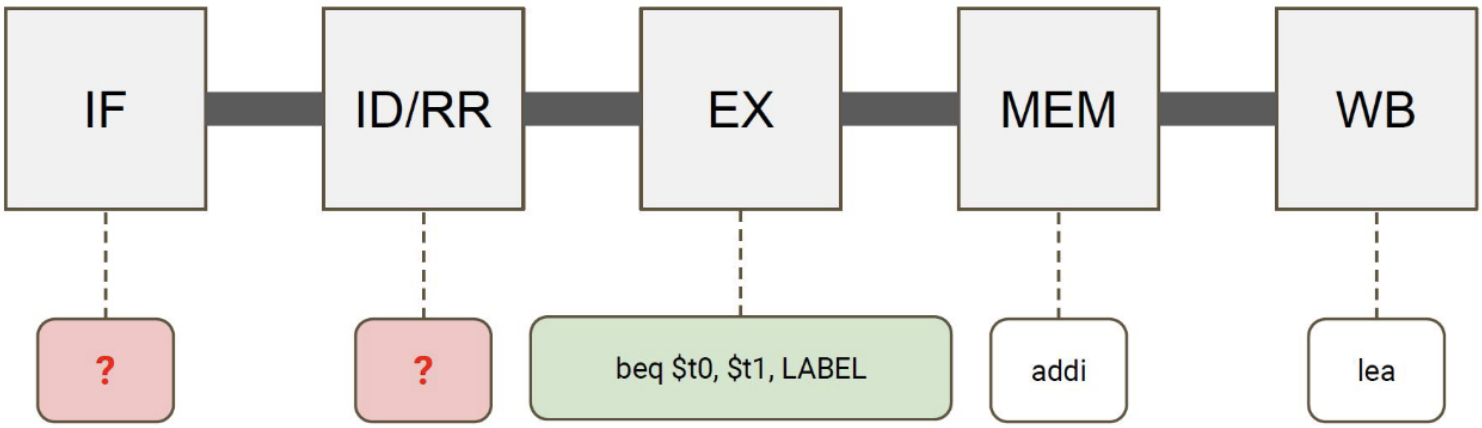
- How do we perform branches?
  - Conditional branch result will only be available after the EX stage
  - EX stage updates the PC after the comparison result becomes available
  - IF fetch the next instruction based on the updated PC



Control Hazards

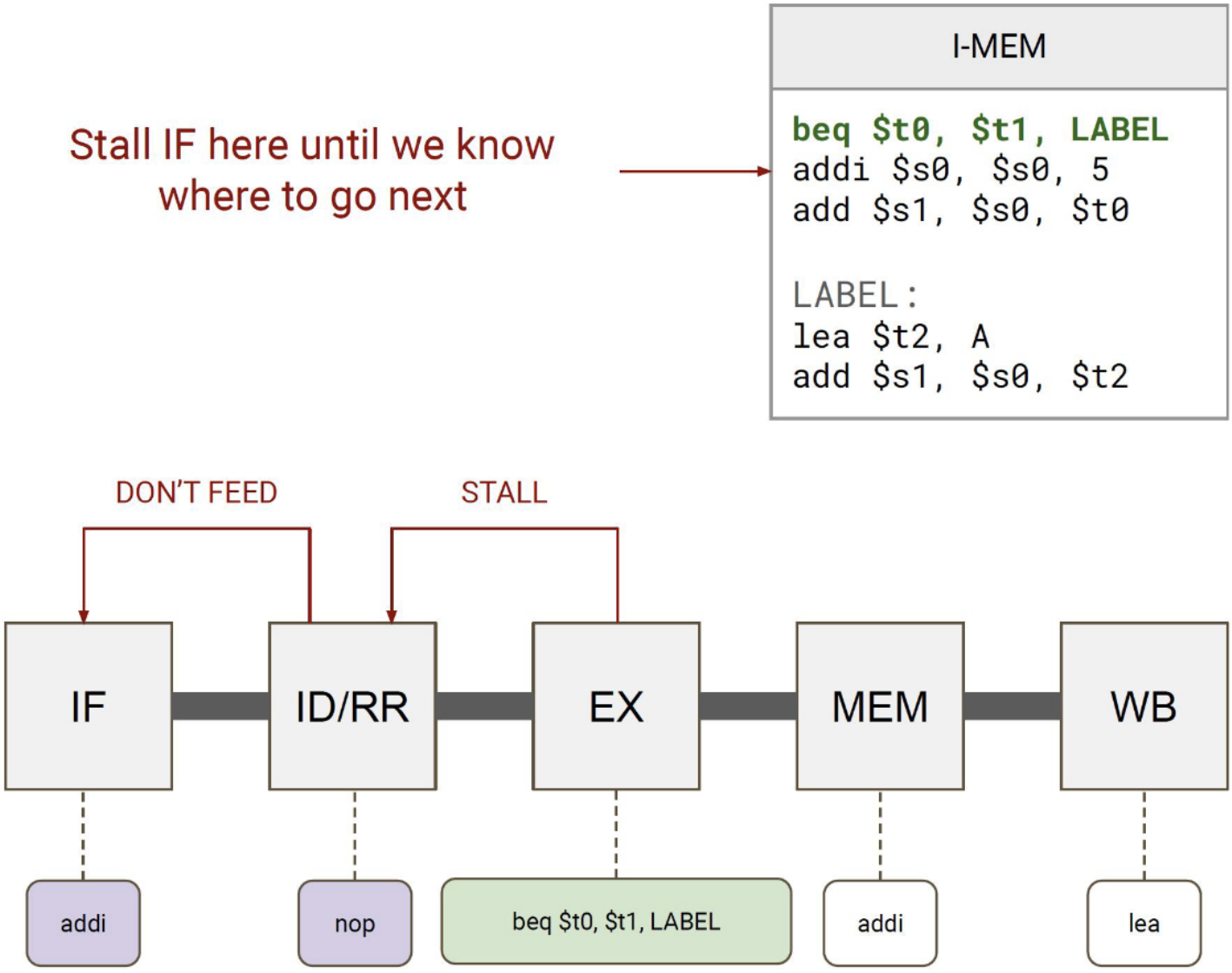
From where should we continue?





Conservative Branch Handling

- Stall the pipeline until the results become available
- Use feedback lines to tell the IF stage to stop feeding instructions to the pipeline
- Pipeline Bubble Count
  - Not taken = 1
  - Branch Taken = 2
  - Need to update PC



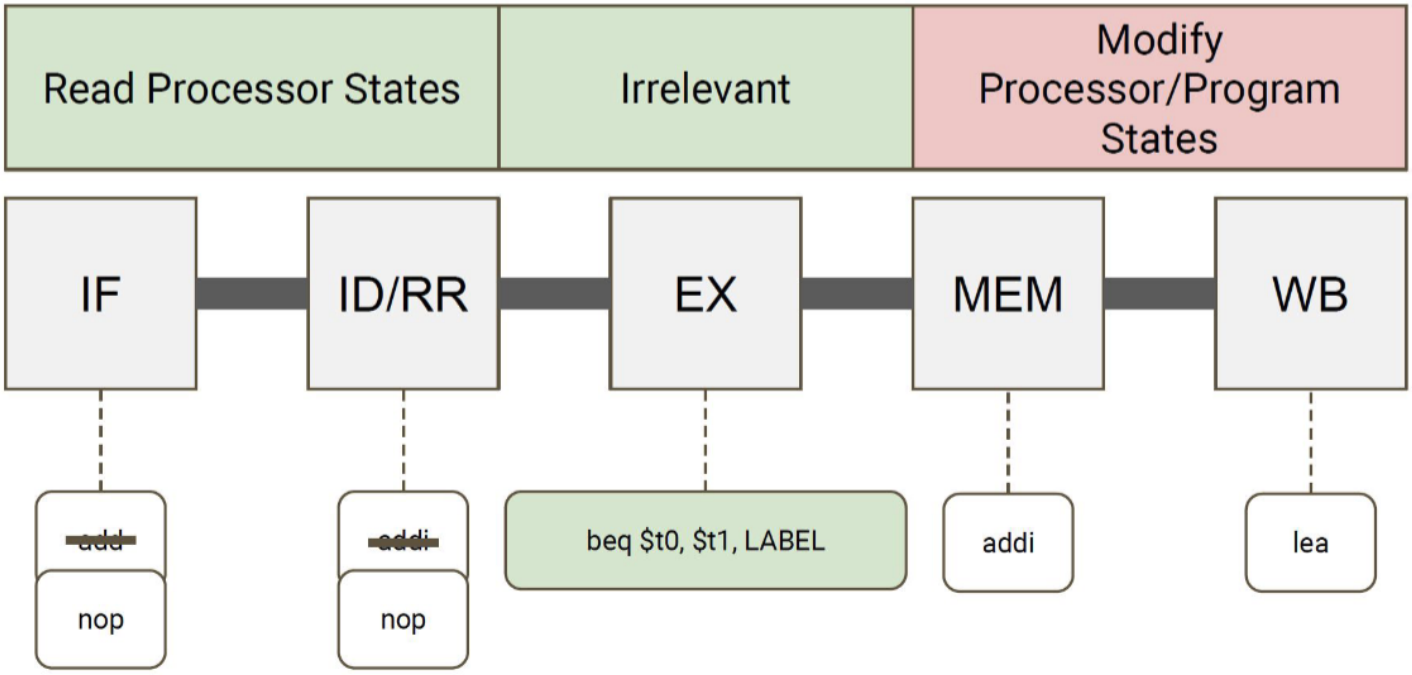
Branch Prediction

- Instructions don't affect the processor/program states until after the EX stage

We can **flush** (replace with nop) the instructions in the **green stages**, as if they never entered the pipeline!

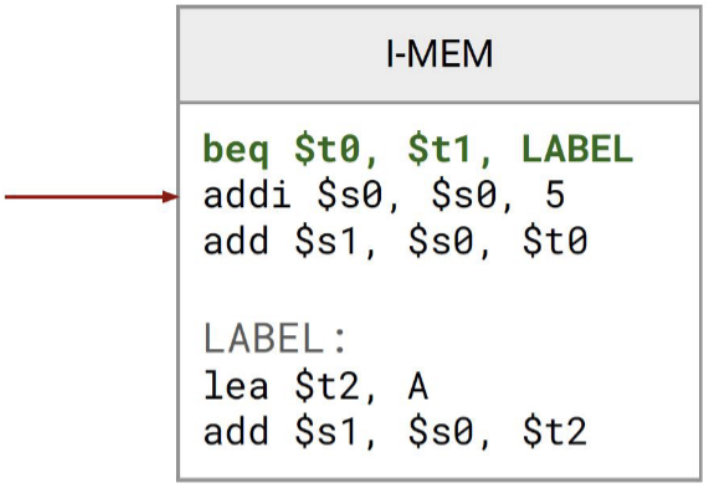
I-MEM

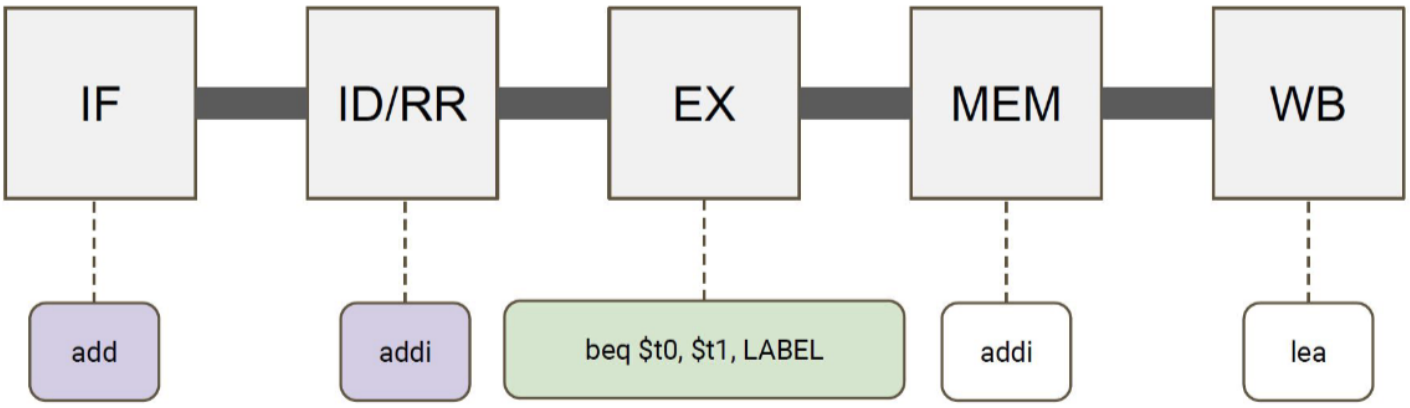
`beq $t0, $t1, LABEL`
`addi $s0, $s0, 5`
`add $s1, $s0, $t0`
 	
`LABEL:`
`lea $t2, A`
`add $s1, $s0, $t2`



- If the prediction was correct, then there would be no pipeline bubble

Example: fetch the next instructions as if the branch was not taken





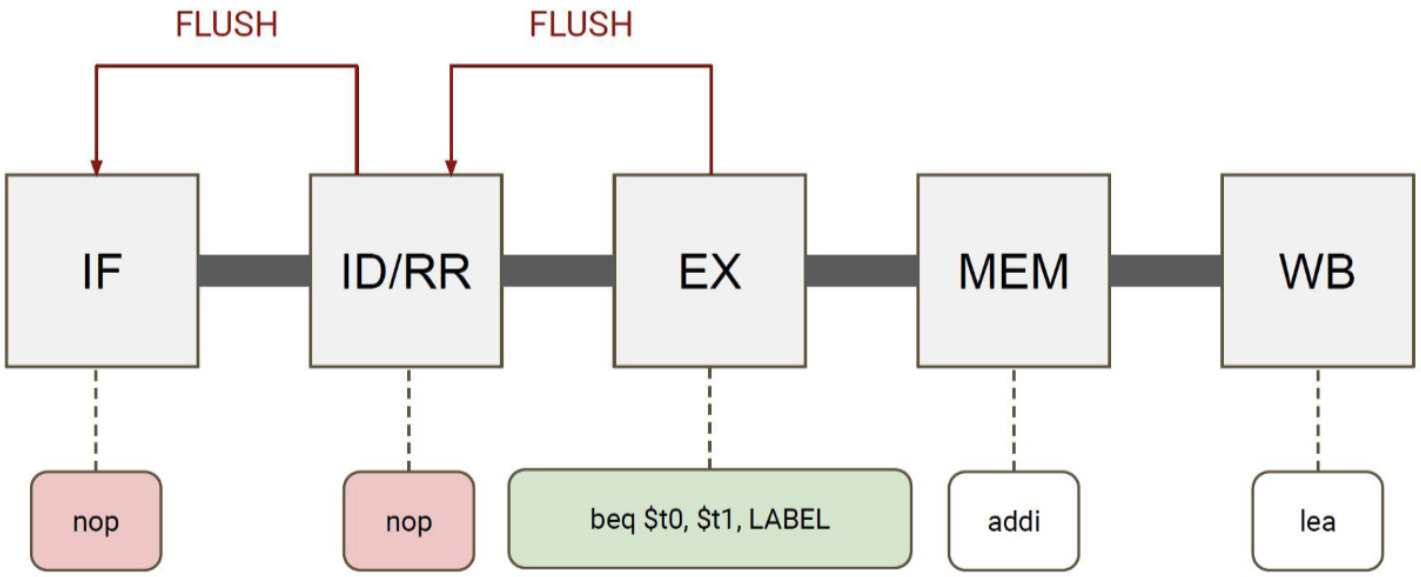
- Mispredict introduces 2 pipeline bubbles (2 nop)
- Naturally, we want to avoid branch misprediction

Pretend the two instructions never entered the pipeline (flush). Update PC to the branching location.

I-MEM

```
beq $t0, $t1, LABEL
addi $s0, $s0, 5
add $s1, $s0, $t0

LABEL:
lea $t2, A
add $s1, $s0, $t2
```



Summary

Name	Pros	Cons	Use cases
Stall the pipeline	Simple strategy, no hardware needed for flushing instructions	Loss of performance	Early pipelined machines such as IBM 360 series
Branch Prediction (branch not taken)	Results in good performance with small additional hardware since the instruction is anyhow being fetched from the sequential path already in IF stage	Needs ability to flush instructions in partial execution in the pipeline	Most modern processors such as Intel Pentium, AMD Athlon, and PowerPC use this technique; typically they also employ sophisticated branch target buffers; MIPS R4000 uses a combination 1-delay slot plus a 2-cycle branch-not-taken prediction
Branch Prediction (branch taken)	Results in good performance but requires slightly more elaborate hardware design	Since the new PC value that points to the target of the branch is not available until the branch instruction is in EX stage, this technique requires more elaborate hardware assist to be practical	-
Delayed Branch	No need for any additional hardware for either stalling or flushing instructions; It involves the compiler by exposing the pipeline delay slots and takes its help to achieve good performance	With increase in depth of pipelines of modern processors, it becomes increasingly difficult to fill the delay slots by the compiler; limits microarchitecture evolution due to backward compatibility restrictions; it makes the compiler not just ISA-specific but implementation specific	Older RISC architectures such as MIPS, PA-RISC, SPARC

Instruction	Type of Hazard	Potential Stalls	With Data forwarding	With branch prediction (branch not taken)
ADD, NAND	Data	0, 1, 2, or 3	0	Not Applicable
LW	Data	0, 1, 2, or 3	0 or 1	Not Applicable
BEQ	Control	1 or 2	Not Applicable	0 (success) or 2 (mispredict)