

CS2200

Systems and Networks

Spring 2022

Lecture 11: Pipeline Hazards

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

STAY CALM & TUTOR ON!

LET US HELP YOU FACE YOUR MIDTERMS

FIND YOUR SUCCESS BY ASKING FOR HELP AND
TAKING ADVANTAGE OF THE AVAILABLE
RESOURCES AT TECH!




**KEEP
CALM**
AND
**TUTOR
ON**

KeepCalmAndPosters.com

TUTORING

SCHEDULE A FREE 1-ON-1 TUTORING SESSION! CHECK
OUT OUR SUPPORTED COURSES AND SCHEDULE AN
APPOINTMENT HERE:

[TUTORING.GATECH.EDU/TUTORING](https://tutoring.gatech.edu/tutoring)

PLUS

JOIN A PLUS SESSION! MANY INTRODUCTORY
COURSES HAVE FREE WEEKLY STUDY SESSIONS LED
BY PEERS WHO HAVE SUCCESSFULLY TAKEN THE
COURSE. INFO MAY BE ON YOUR COUSE CANVAS
SITE, AND YOU CAN CHECK IT OUT HERE:

[TUTORING.GATECH.EDU/PLUS-SESSIONS](https://tutoring.gatech.edu/plus-sessions)

Test I Logistics

- Syllabus

- Chapters 1-4
 - ISA, Processor design, Interrupts
- Chapter 5 (up to and including 5.6)
 - Processor performance

- Flow

- 80% of the questions will be released at 11:59pm on Friday 2/18/2022
 - You will see the remaining 20% when you start the timed exam
- 11:59pm Friday to noon Saturday
 - Clarification of questions on Piazza (NOT EMAIL)
- Test will be available as a Canvas quiz
 - Timed test (60 minutes) using Honorlock + Gradescope
 - CLOSED everything (except your mind and intellect)
 - Test-taking window: 11:59pm Friday (2/18) to 11:59pm Monday (2/21)
 - **AVOID TAKING THE TEST until NOON on Saturday for your own benefit to ensure you don't miss out on any clarifications**

Pipeline Hazards

- Structural hazards
 - Datapath/resource limitations
- Data hazards
 - Program limitations
- Control hazards
 - Program limitations

Bill's Mega-Sandwich Shop



Station 1
(place order)
New (5th order)
"What will you have?"

station II
(select bread)
4th order

station III
(cheese)
3rd order

station IV
(meat)
2nd order

station V
(veggies)
1st order

**And
sometimes
fries on the
side**

**Stay on the
same order!**

**Stay on the
same order!**

**No work for
one cycle**

Customers behind

Pipeline Stall

Customers ahead

Bill's Mega-Sandwich Shop



Station 1
(place order)
New (5th order)
"What will you have?"

station II
(select bread)
4th order

station III
(cheese)
3rd order

station IV
(meat)
2nd order

station V
(veggies)
1st order



**And
sometimes
fries on the
side**



**No work for
one cycle**



Pipeline resumed



Fries with that?

If the cheese person has to **add fries** to some orders, then the pipeline...

- 0% A. Continues to produce 1 sandwich per clock cycle
- 100% B. Produces more than 1 sandwich per clock cycle
- 0% C. Produces less than 1 sandwich per clock cycle

Bill's Mega-Sandwich Shop



Station 1
(place order)
New (5th order)

"What will you have?"



station II
(select bread)
4th order



station III
(cheese)
3rd order

**And
sometimes
fries on the
side**



station IV
(meat)
2nd order



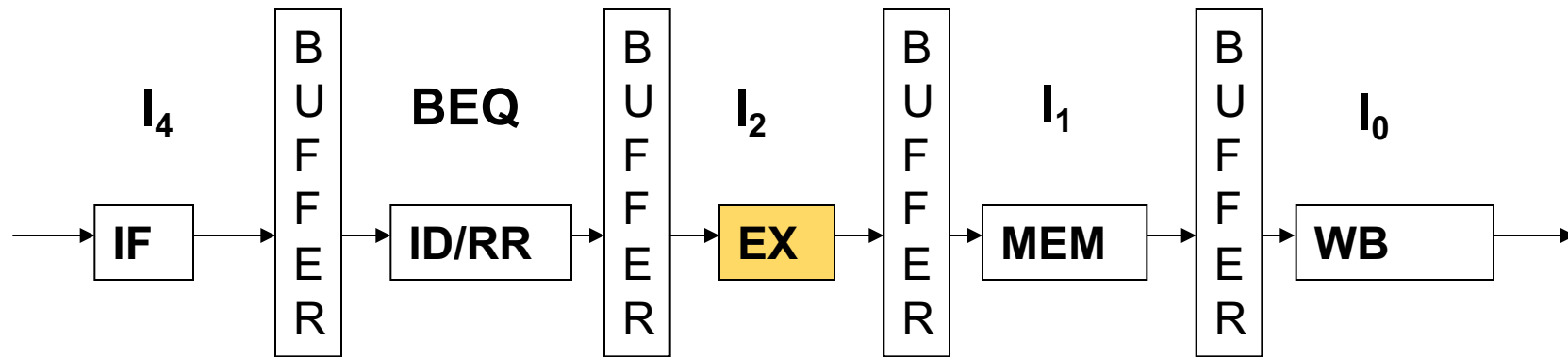
station V
(veggies)
1st order



This is a structural hazard –
not enough resources

The effect:
→ Introduces "bubbles" into the pipeline
→ Reduces efficiency

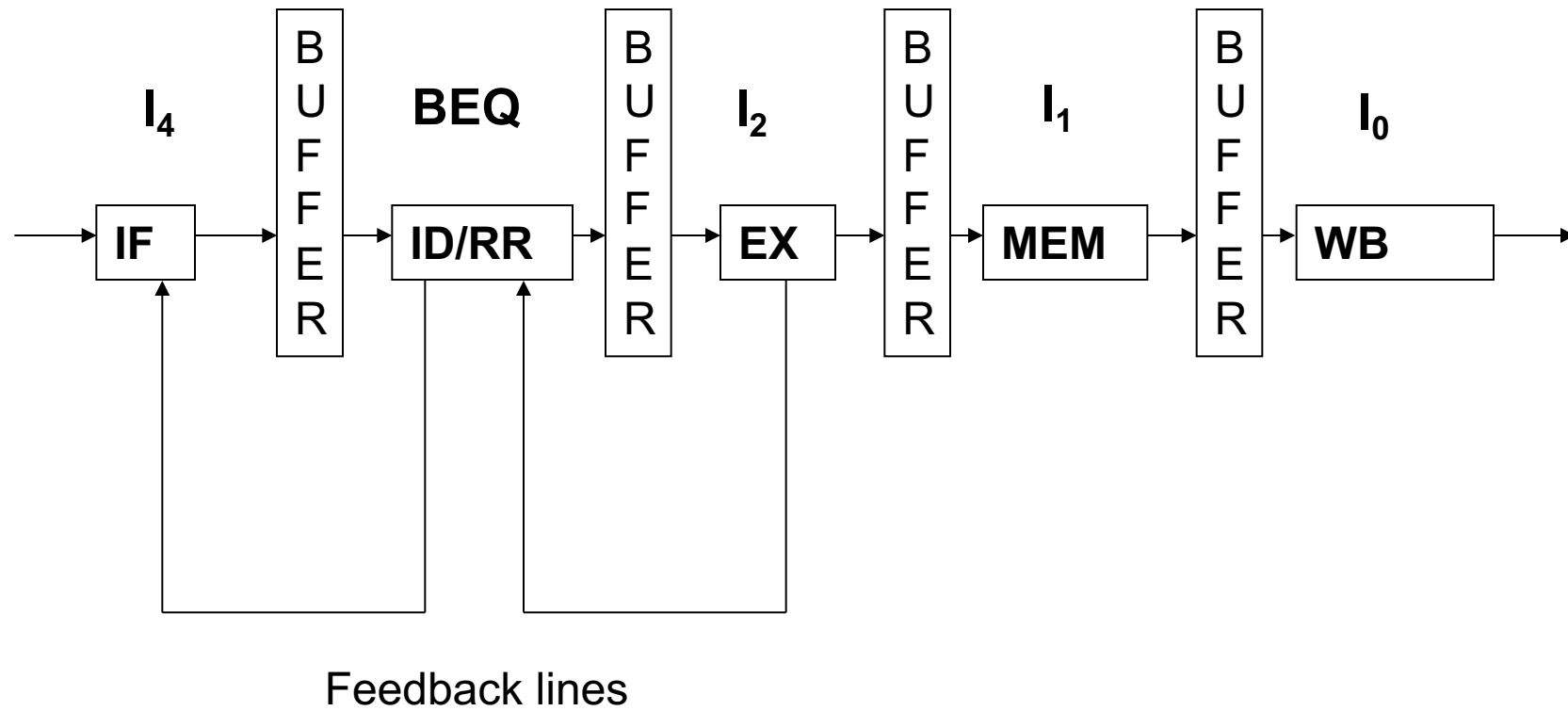
Structural Hazard



- Potentially need two arithmetic ops
 - $A - B$ (always needed)
 - $PC + \text{offset}$ (sometimes needed)
- Where is this going to mean trouble?

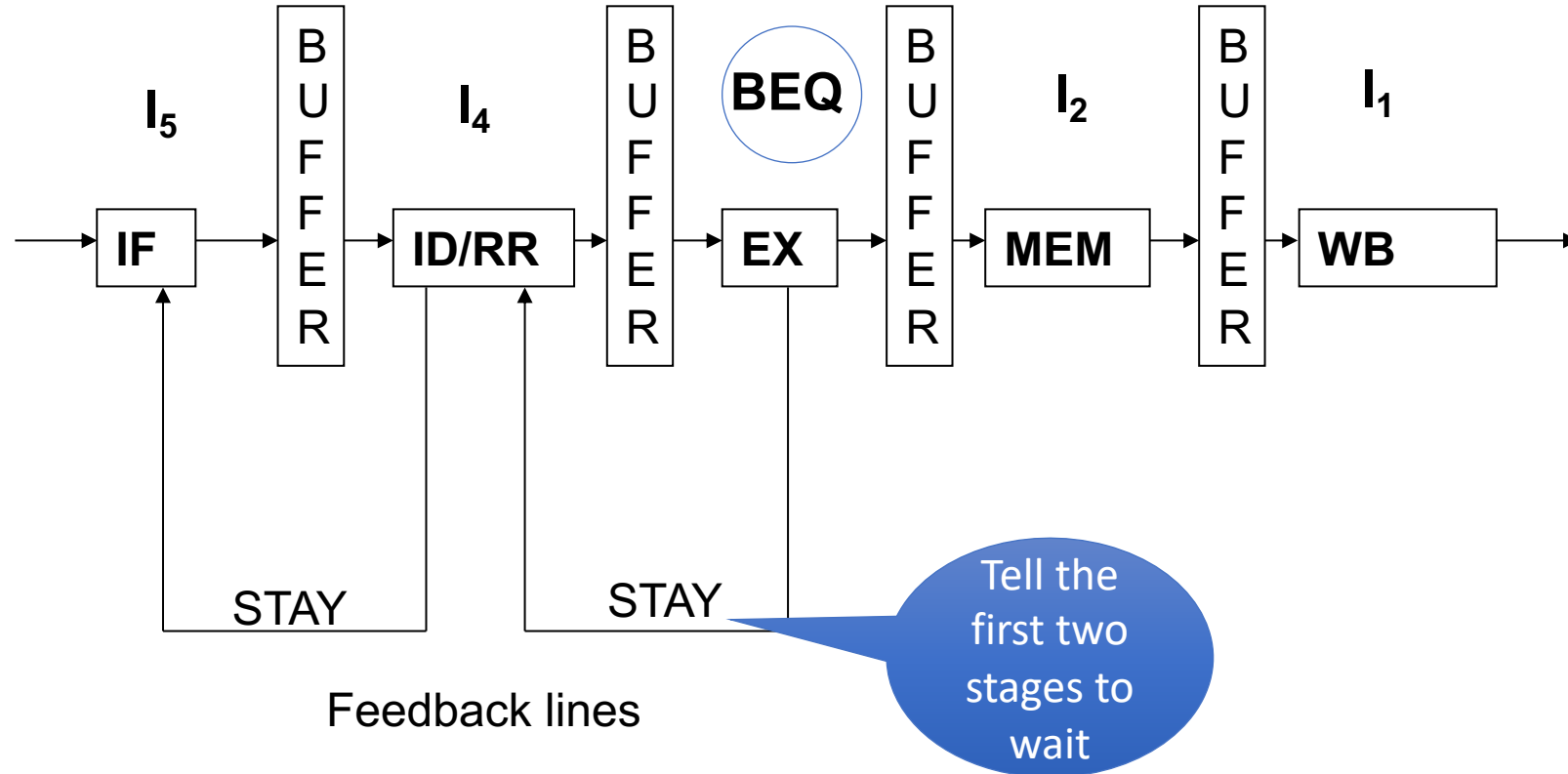
Structural Hazard

Cycle 1

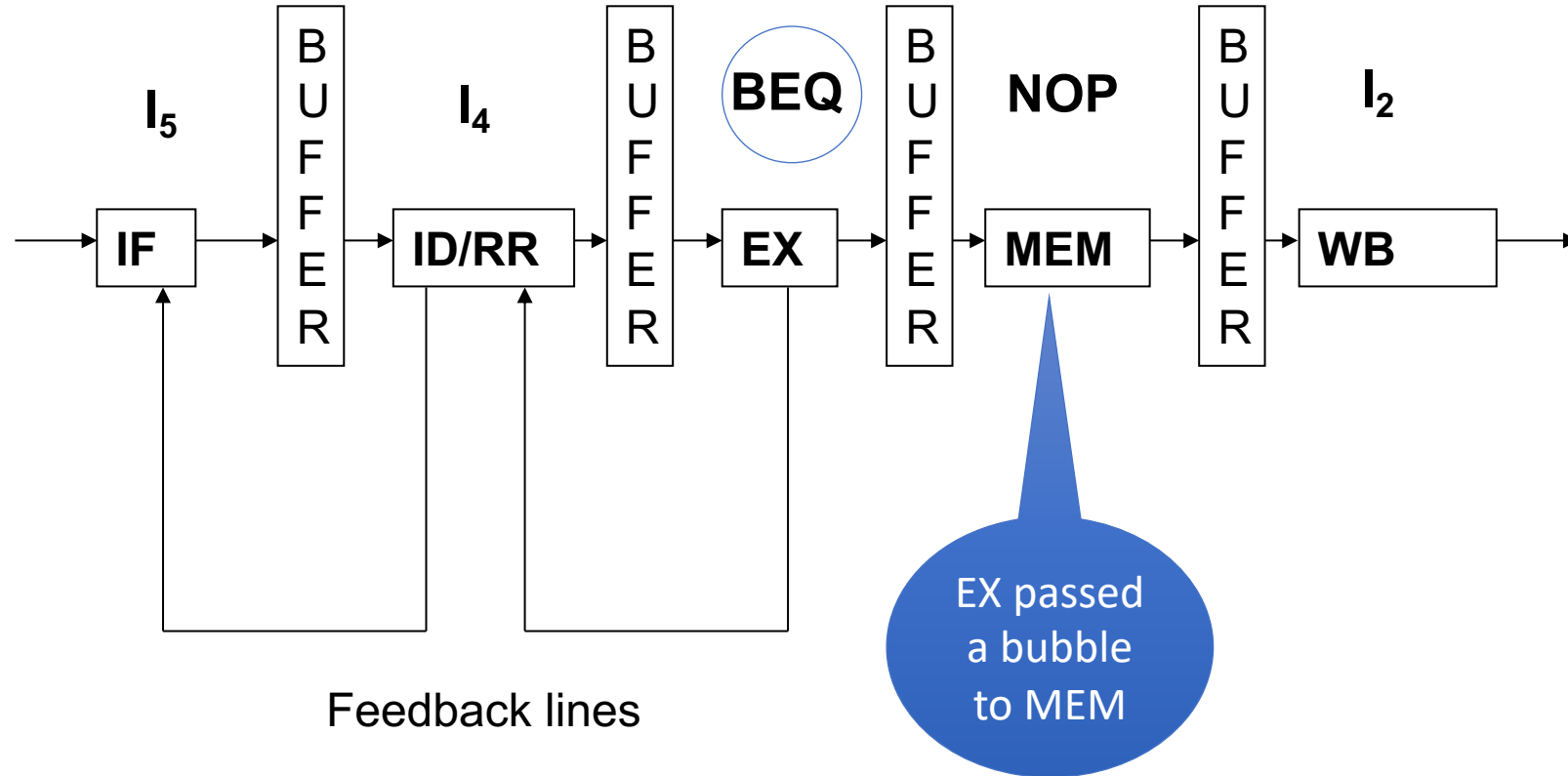


If BEQ branches, we will need to use the ALU twice!

Cycle 2

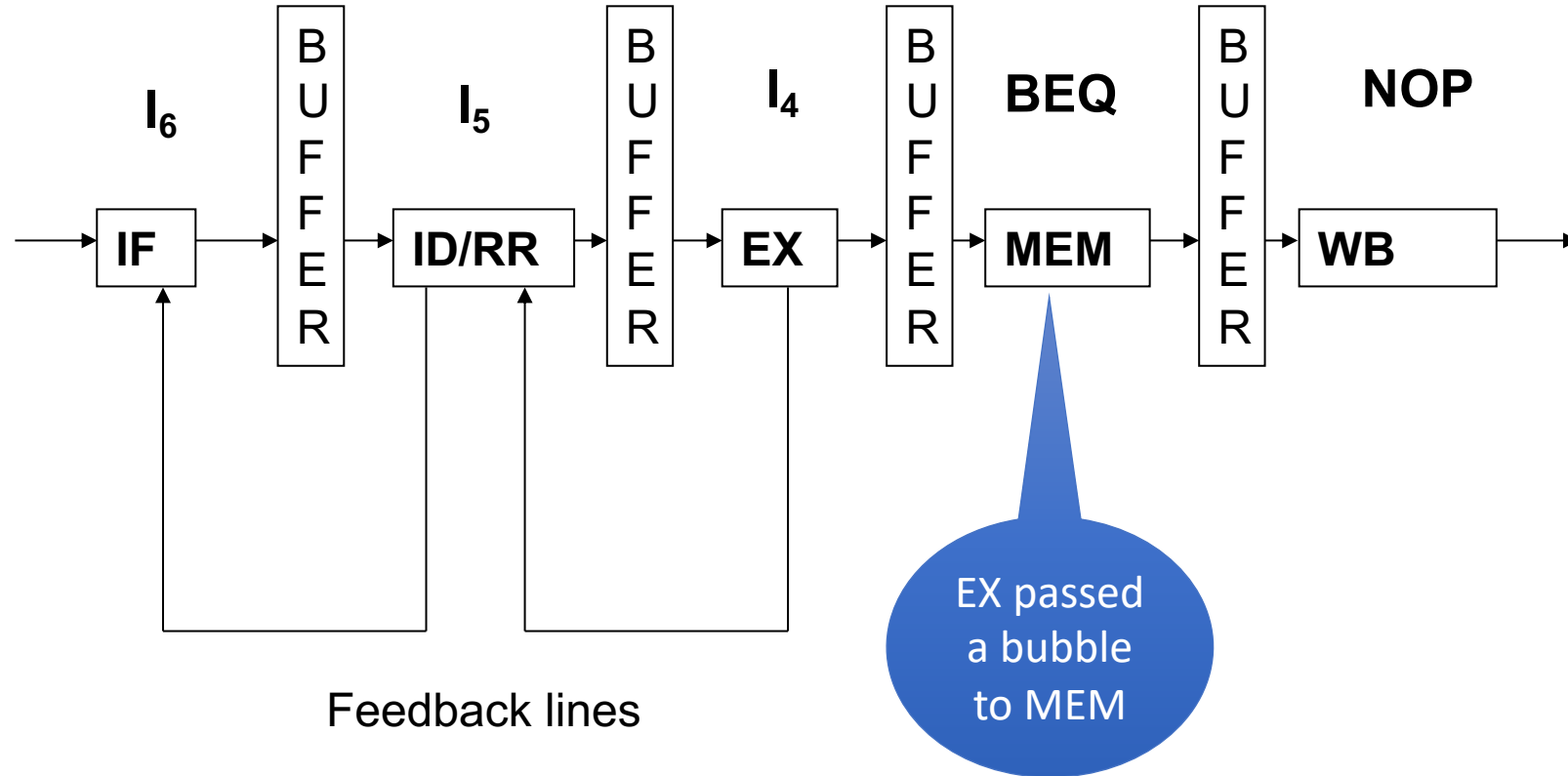


Cycle 3



But life goes on

Cycle 4



Bill's Mega-Sandwich Shop



Station 1
(place order)
New (5th order)

"What will you have?"



station II
(select bread)
4th order



station III
(cheese)
3rd order

**And
sometimes
fries on the
side**



Cheese
guy



Fries guy

station IV
(meat)
2nd order



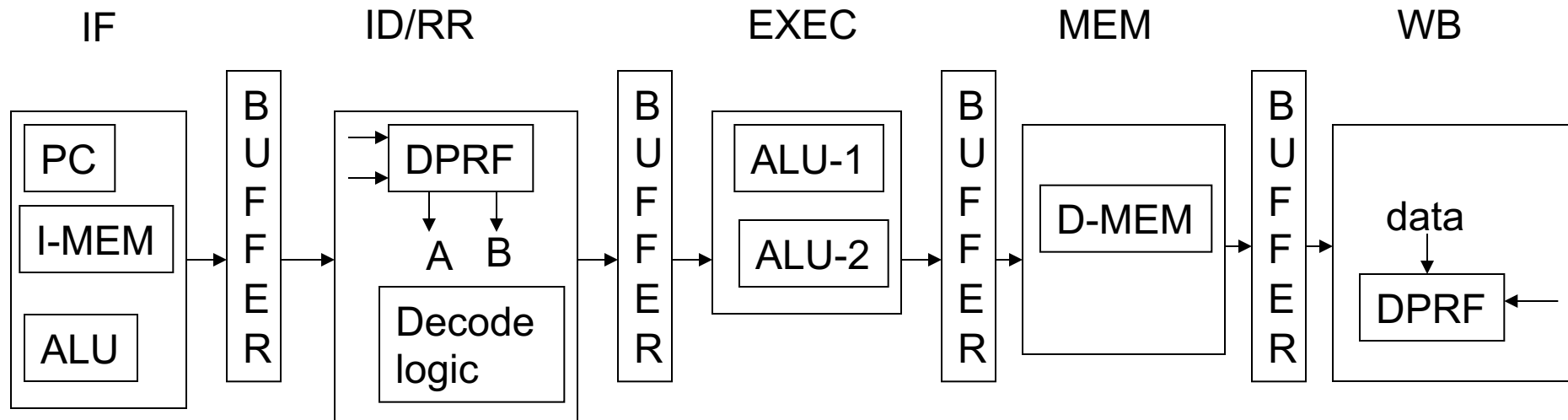
station V
(veggies)
1st order



But we can also add
more "hardware"!

Throw hardware at it!

- So we add another ALU to the EX stage





Pipeline structural hazards are caused by...

- A. Branch instructions in the program
- B. Load instructions in the program
- C. Data dependencies in the program
- D. Hardware limitations in the datapath

Data Hazard

RAW

I₁: R1 <- R2 + R3

I₂: R4 <- R1 + R5

True dependency

WAR

I₁: R4 <- R1 + R5

I₂: R1 <- R2 + R3

Anti dependency

WAW

I₁: R1 <- R4 + R5

I₂: R1 <- R2 + R3

Output dependency

Dealing with a RAW dependency

$I_1: R1 \leftarrow R2 + R3$

$I_2: R4 \leftarrow R1 + R5$



Cycle 1



Cycle 2

At this point, R1 hasn't even been computed

I_2

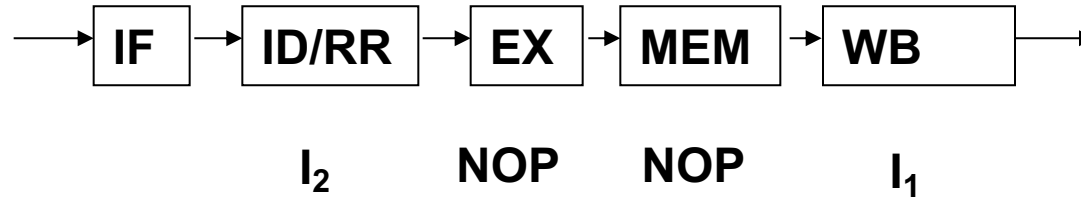
I_1

If we don't stall the pipeline here, the execution of I_2 will result in a semantic inconsistency

I_2

I_1

Fix – introduce bubbles



- We need some hardware help to know when to introduce bubbles
- So we're going to add a “busy” bit to the register file
- ID/RR sets it and WB clears it

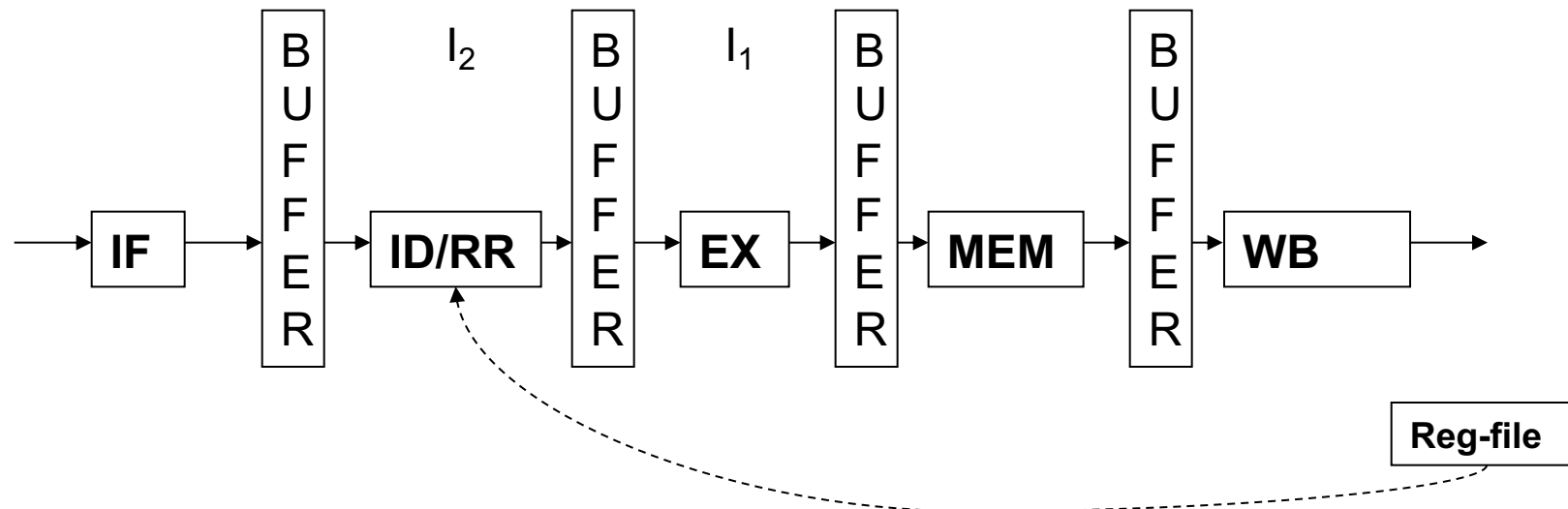
Register file

[illegible]

RAW Hazard – Cycle 1

I_1 : $R1 \leftarrow R2 + R3$

I_2 : $R4 \leftarrow R1 + R5$

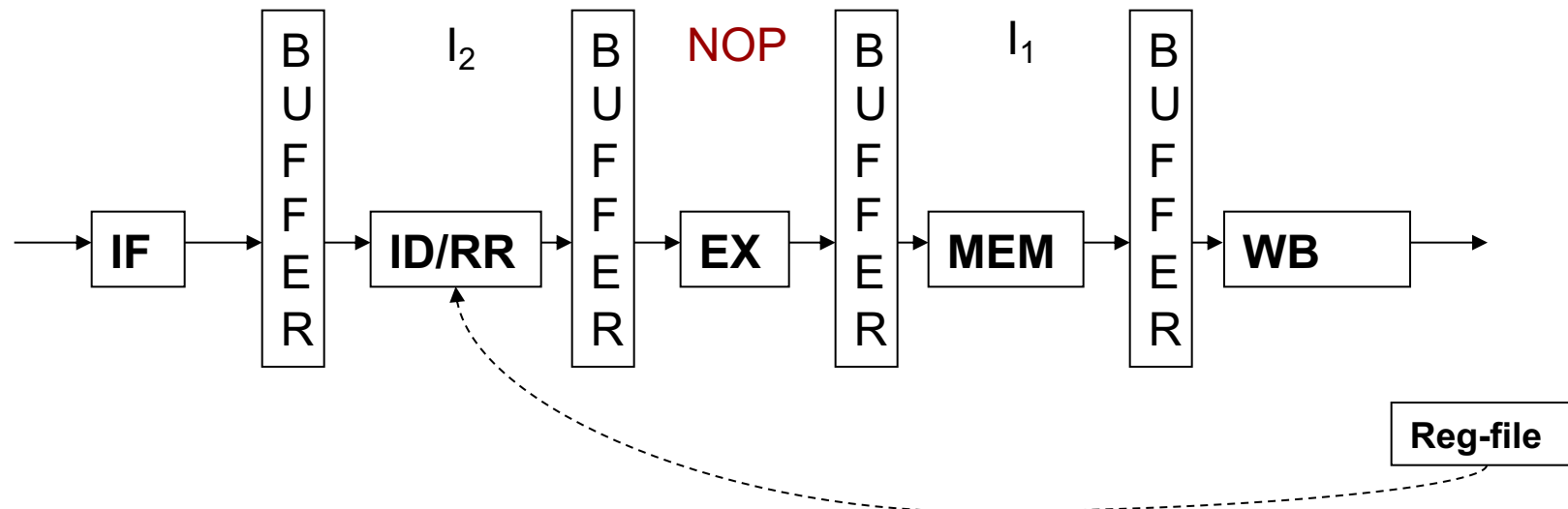


R1 not ready to be read

RAW Hazard – Cycle 2

I_1 : $R1 \leftarrow R2 + R3$

I_2 : $R4 \leftarrow R1 + R5$

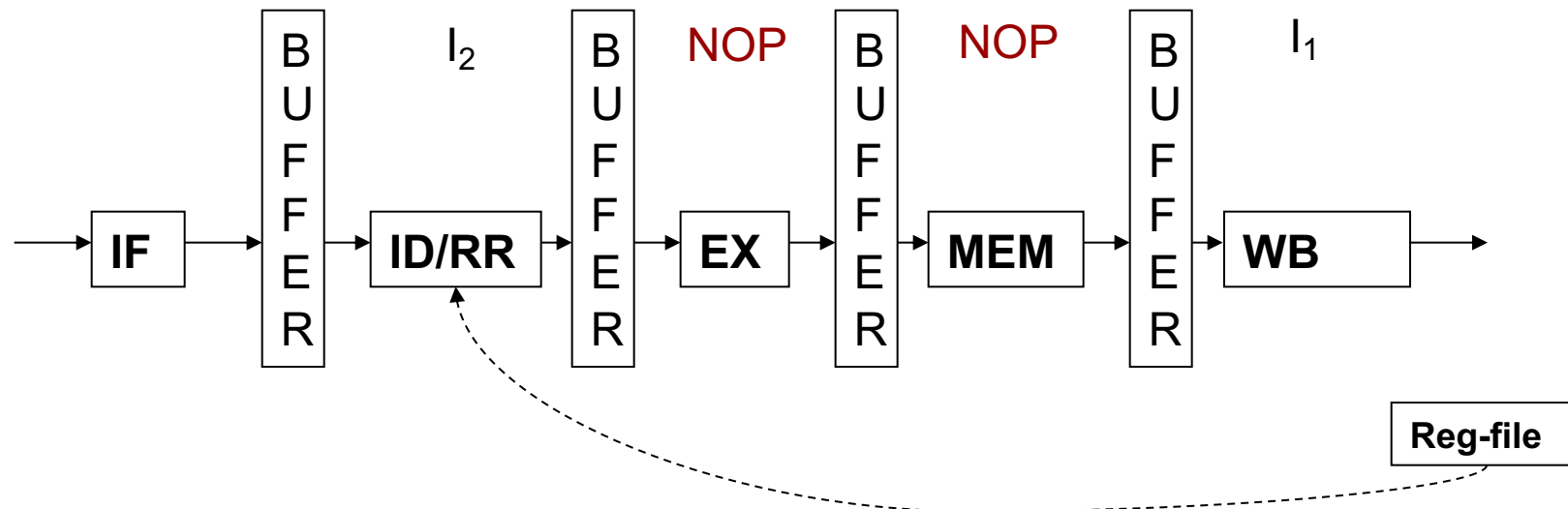


R1 not ready to be read

RAW Hazard – Cycle 3

I_1 : $R1 \leftarrow R2 + R3$

I_2 : $R4 \leftarrow R1 + R5$

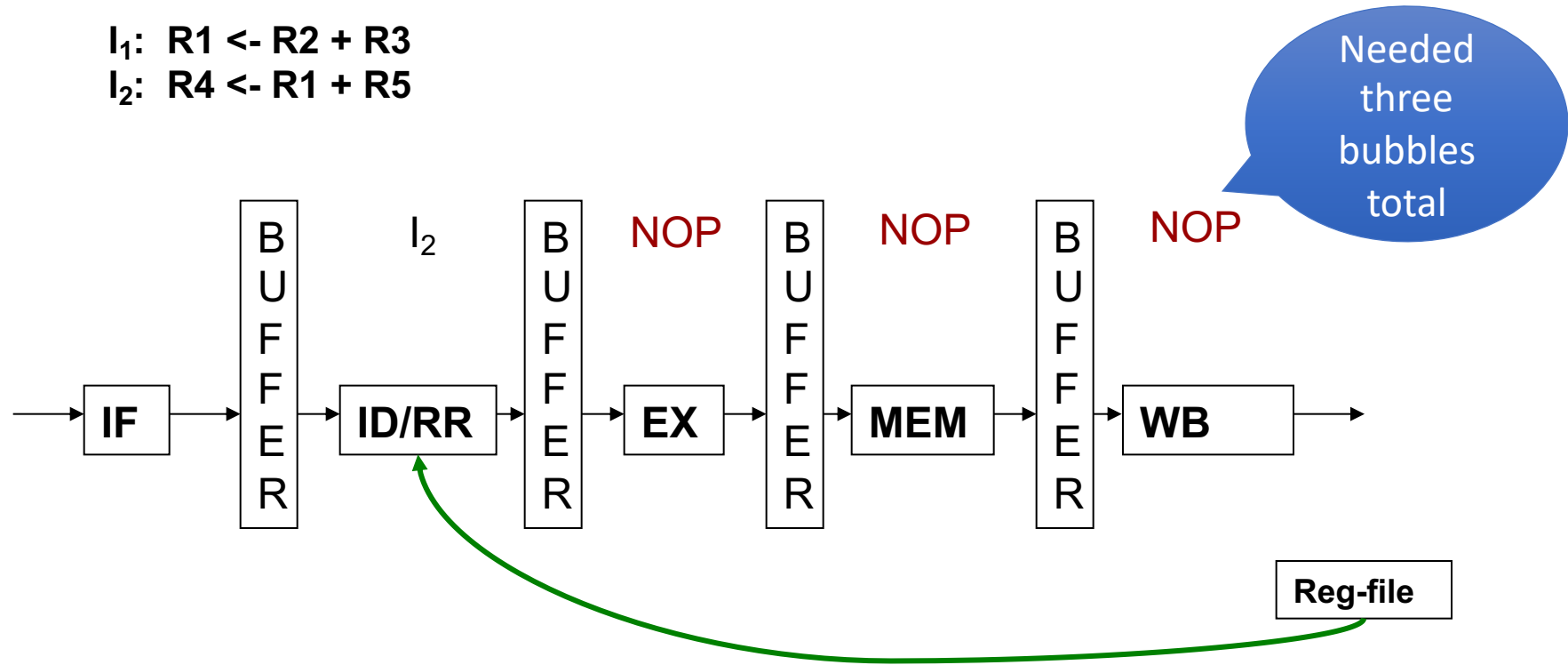


R1 not ready to be read

RAW Hazard – Cycle 4

I_1 : $R1 \leftarrow R2 + R3$

I_2 : $R4 \leftarrow R1 + R5$



R1 is ready to be read

A question...

$I_1: R1 \leftarrow R2 + R3$

$I_2: R4 \leftarrow R4 + R3$

$I_3: R5 \leftarrow R5 + R3$

$I_4: R6 \leftarrow R1 + R6$



- Say we insert 2 instructions between “definition” and “use” of a register
- Does this help to reduce “bubbles”?



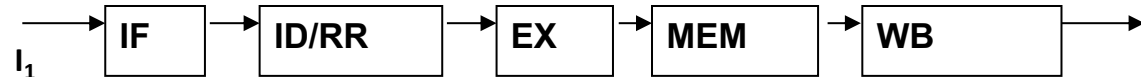
With 2 instruction separation...

$I_1: R1 \leftarrow R2 + R3$

$I_2: R4 \leftarrow R4 + R3$

$I_3: R5 \leftarrow R5 + R3$

$I_4: R6 \leftarrow R1 + R6$



- A. Continue to have 3 bubbles in pipeline
- B. Reduce the number of bubbles to 0
- C. Reduce the number of bubbles to 1
- D. Reduce the number of bubbles to 2

Two instruction separation

I₁: R1 <- R2 + R3

I₂: R4 <- R4 + R3

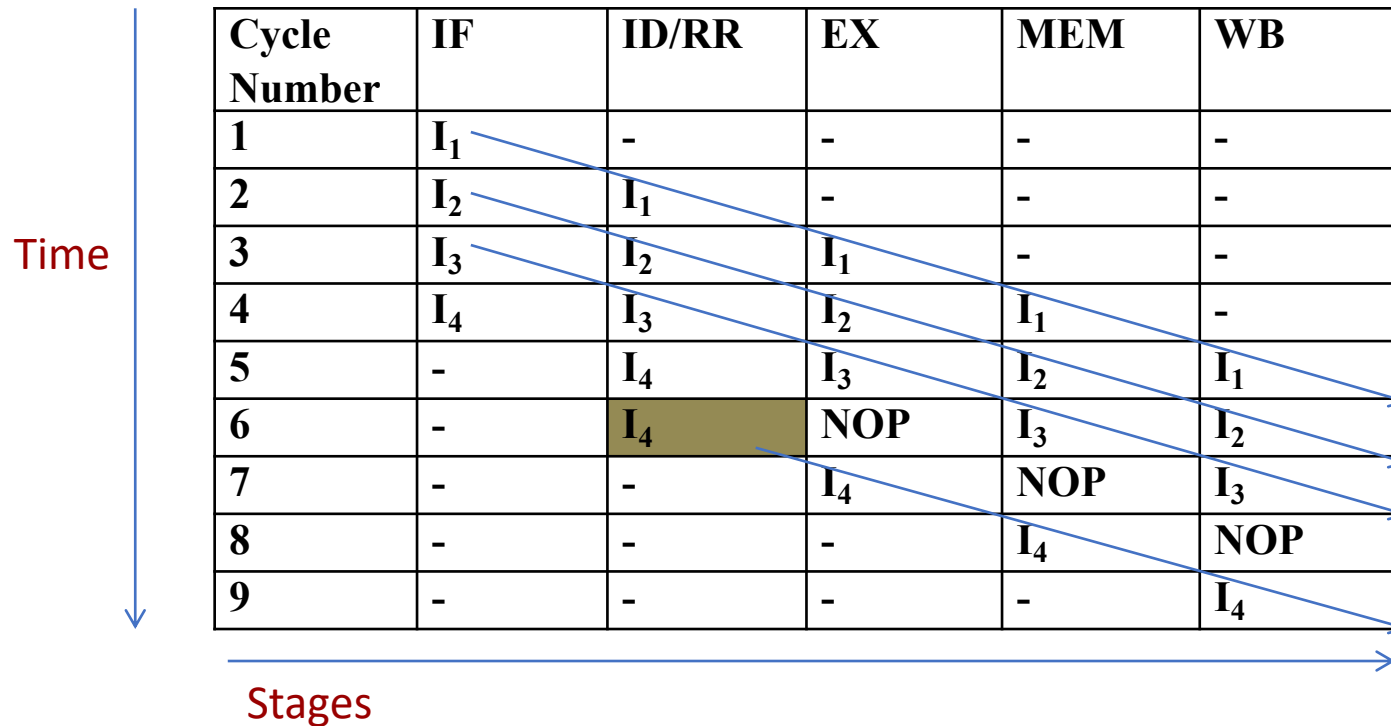
I₃: R5 <- R5 + R3

I₄: R6 <- R1 + R6



- We can restructure the program to reduce “bubbles”
- This is something we expect modern compilers to be able to do for us.

Waterfall Diagram



I₁: R1 <- R2 + R3

I₂: R4 <- R4 + R3

I₃: R5 <- R5 + R3

I₄: R6 <- R1 + R6

If we can't restructure our program?

- Can we throw hardware at the problem and eliminate bubbles?

-

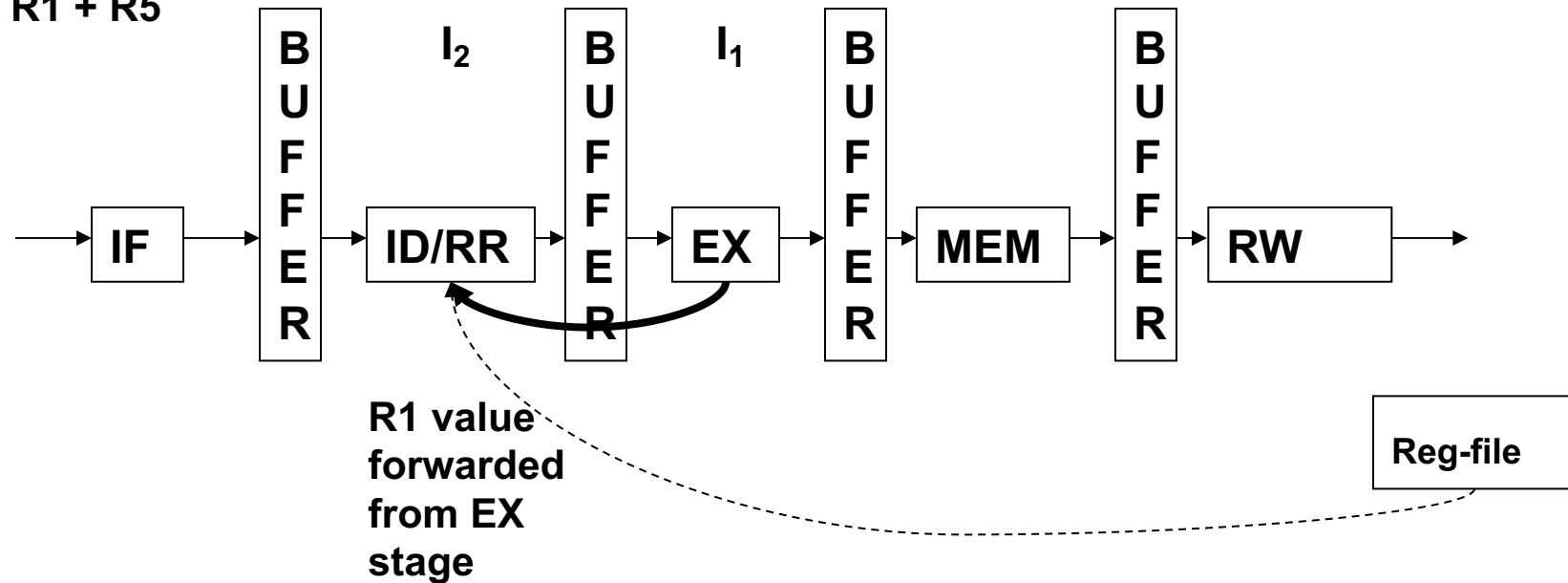
Register file

[illegible]

Forwarding example

I_1 : $R1 \leftarrow R2 + R3$

I_2 : $R4 \leftarrow R1 + R5$



Forwarding logic in ID/RR:

If $RP == 1$ then

 use forwarded value

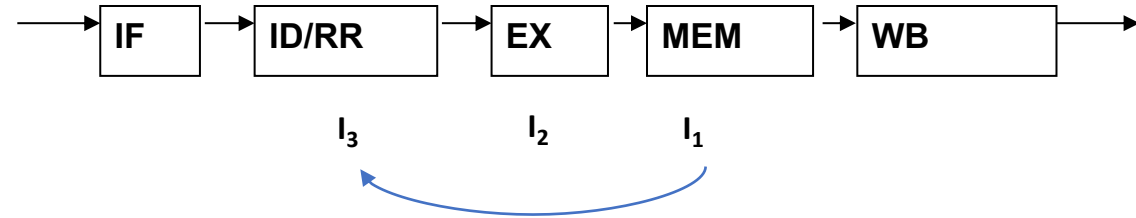
else

 read entry from DPRF

R1 not ready to be read

What if an instruction intervenes?

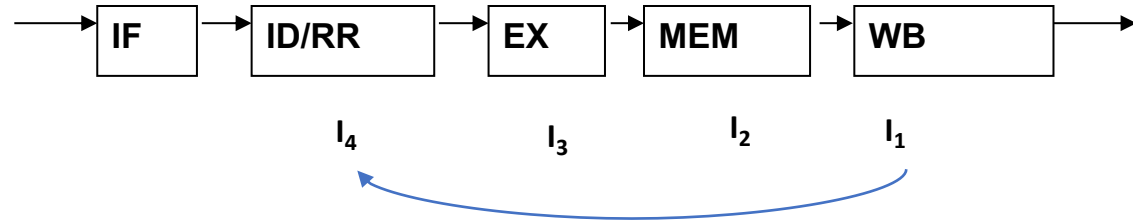
$I_1: R1 \leftarrow R2 + R3$
 $I_2: R4 \leftarrow R4 + R3$
 $I_3: R6 \leftarrow R1 + R6$



Now we need to forward from MEM to ID/RR

What if two instructions intervene?

$I_1: R1 \leftarrow R2 + R3$
 $I_2: R4 \leftarrow R4 + R3$
 $I_3: R5 \leftarrow R5 + R3$
 $I_4: R6 \leftarrow R1 + R6$



Now we need to forward from WB to ID/RR

Bubbles introduced by RAW

I_1 : $R1 \leftarrow R2 + R3$

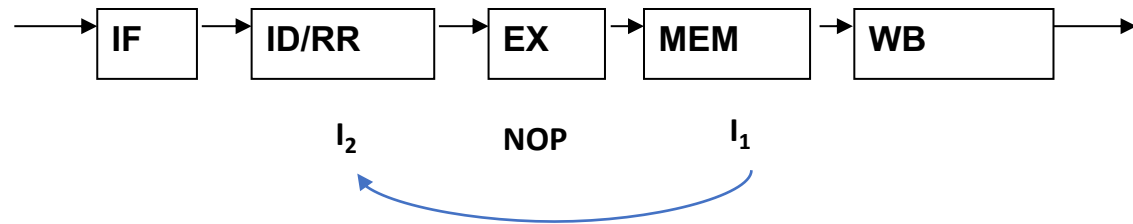
...

I_n : $R4 \leftarrow R1 + R5$

| Number of unrelated instructions Between I_1 and I_n | Number of bubbles Without forwarding | Number of bubbles With forwarding |
|---|---|--------------------------------------|
| 0 | 3 | 0 |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 or more | 0 | 0 |

LW instructions are special

I_1 : LW R1, 0(R2)
 I_2 : R4 \leftarrow R1 + R5



Even with data forwarding, we have to insert a bubble!

Why? We don't get the data until I_1 gets to MEM!

So LW changes the table

RAW Hazard

I₁: R1 <- R2 + R3

...

I₂: R4 <- R1 + R5

| Number of unrelated instructions Between I ₁ and I ₂ | Number of bubbles Without forwarding | Number of bubbles With forwarding |
|---|---|--------------------------------------|
| 0 | 3 | 0 |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 or more | 0 | 0 |

RAW Hazard

I₁: LW R1, 0(R2)

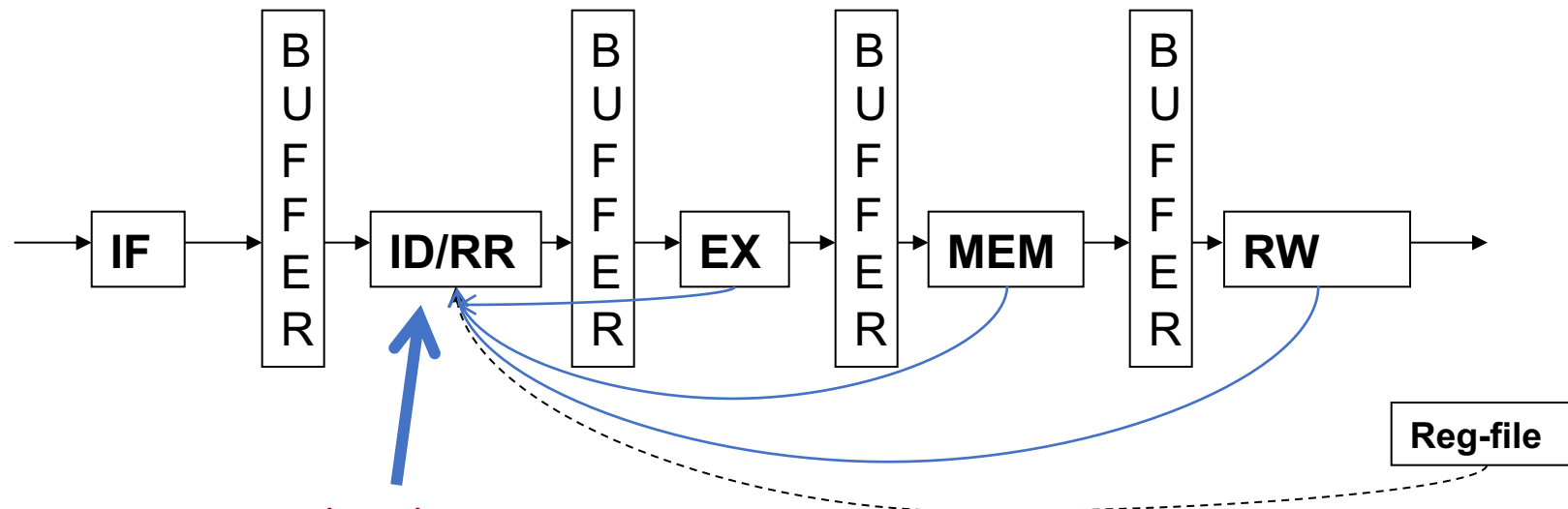
...

I₂: R4 <- R1 + R5

| Number of unrelated instructions Between I ₁ and I ₂ | Number of bubbles Without forwarding | Number of bubbles With forwarding |
|---|---|--------------------------------------|
| 0 | 3 | 1 |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 or more | 0 | 0 |

Generalized register forwarding

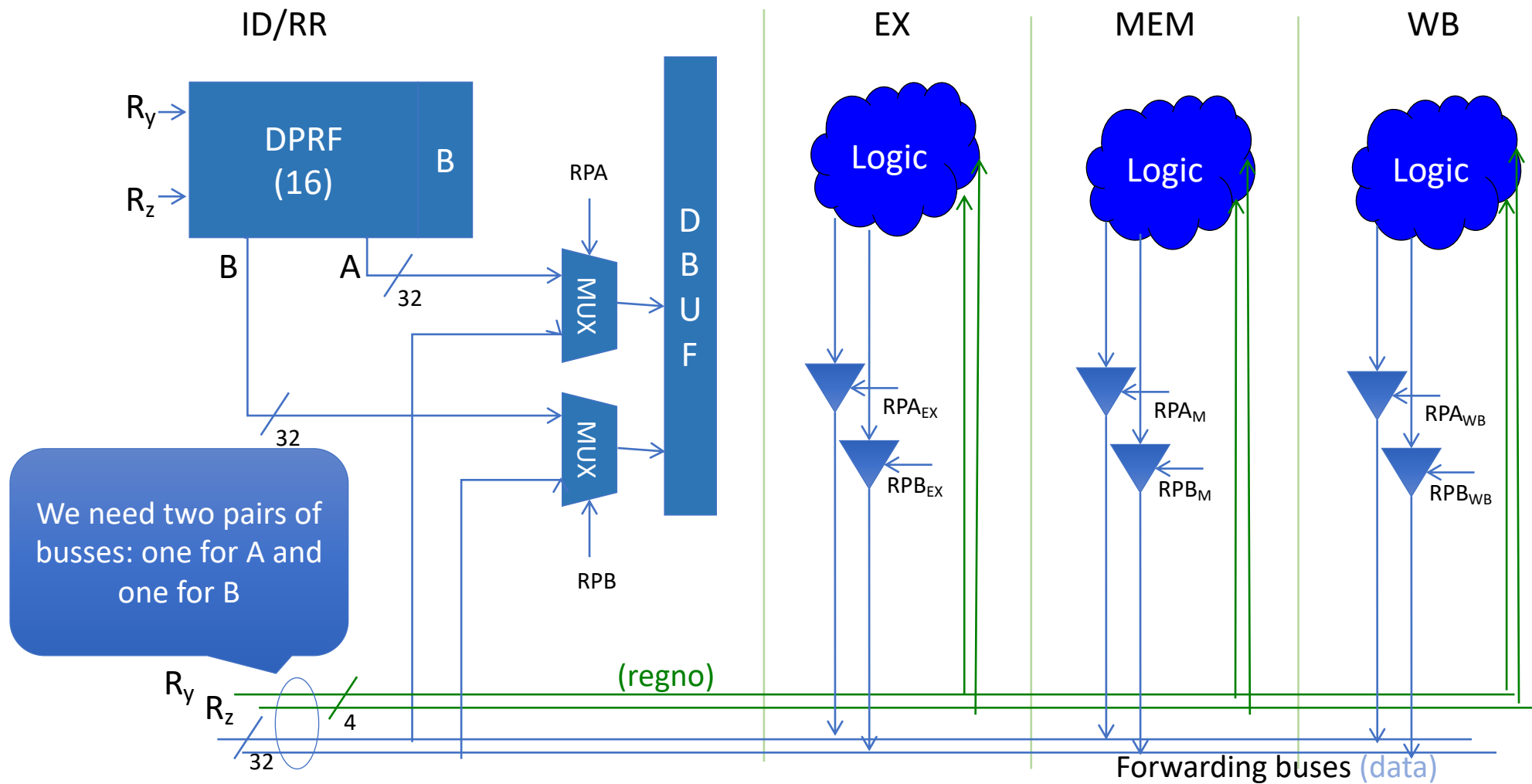
We need forwarding data lines from ALL stages



**And we have two
registers to be fed**

R1 not ready to be read

An example of implementation



What's going on?

This is a rough outline of what happens in the previous diagram:

1. ID/RR puts the registers numbers used for A and B on the two forwarding register busses
2. The other stages compare any results for their destination register with the register numbers on the two register busses
3. If the destination register matches a register on a register forwarding bus, the stage opens the driver to allow the destination value it has onto the appropriate A and/or B forwarding data bus and signals ID/RR that a match has been found on the A or B operand via RPA/RPB.
4. There is circuitry (sort of like interrupt priority) that makes sure the first stage that matches on the forwarding register bus is the one that gets to use the forwarding data bus (Why?)
5. If the RPA/RPB signals are sent, multiplexers in the ID/RR stage select the the data from the corresponding A or B forwarding data bus instead of the register file

Other data hazards

WAR $I_1: \quad R2 \leftarrow R1 + R3$
 $I_2: \quad R1 \leftarrow R4 + R5$

Fortunately, this isn't a problem since we've already read the registers for I_1 into DBUF before I_2 starts.

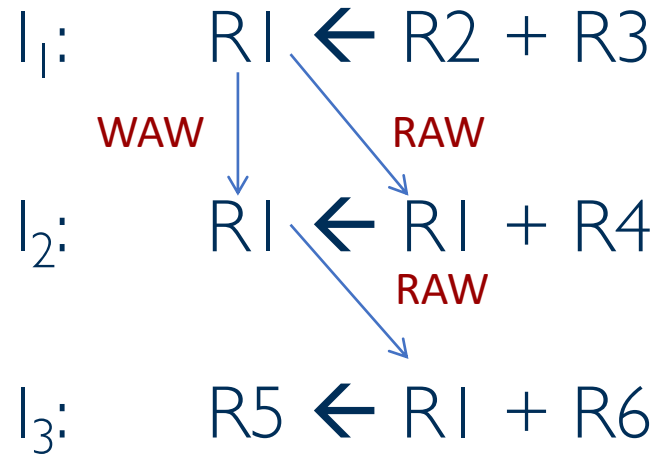
WAW $I_1: \quad R1 \leftarrow R2 + R3$
 $I_2: \quad R1 \leftarrow R4 + R5$

Currently, the B bit set by I_1 will stall I_2

But why do we see WAW anyway?

- Shouldn't the compiler get rid of useless writes?
- There are unexpected code sequences
 - Exception handling code
- Check out Sec 5.13.2 page 200 and Sec 5.15.4 if you're curious

What hazards?



Can't let I₂ move forward until I₁ is retired from pipeline

Otherwise I₃ could get the wrong value (from I₁ instead of I₂)

Fortunately, the existing behavior of WAW hazards will cause I₂ to stall until I₁ completes its write in WB.

Control hazard

Is there any analog in the sandwich pipeline?

| Cycle | IF* | ID/RR | EX | MEM | WB |
|-------|------|-------|----|-----|----|
| 1 | BEQ | | | | |
| 2 | ADD | | | | |
| 3 | NAND | | | | |
| 4 | LW | | | | |

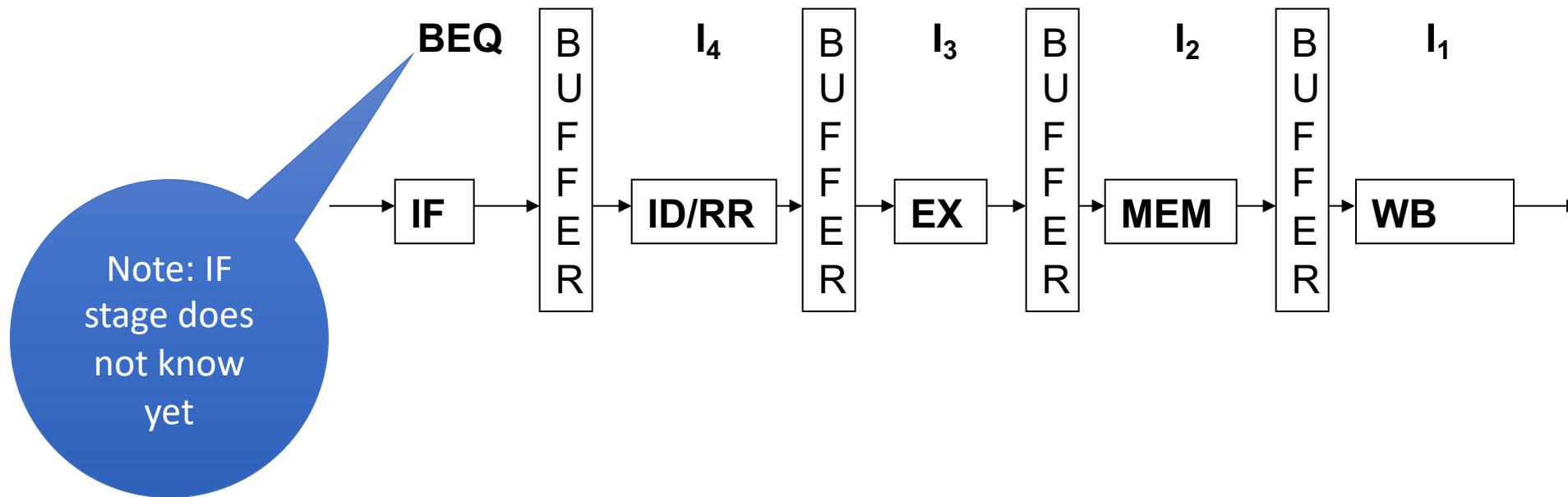
* We do not actually know what the instruction is in the Fetch Stage before we decode it

Conservative handling of branches

- Wait until branch **outcome** is known
 - **Stall** the pipeline → send **NOPs** from IF state
- Resume normal execution when branch outcome is known
 - Result of **comparison** (A-B) is known
 - If there is need to branch, **PC** gets the **target address** of the branch
- Two cases
 - Branch **TAKEN** (i.e. $A==B$)
 - Branch **NOT TAKEN** (i.e. $A!=B$)

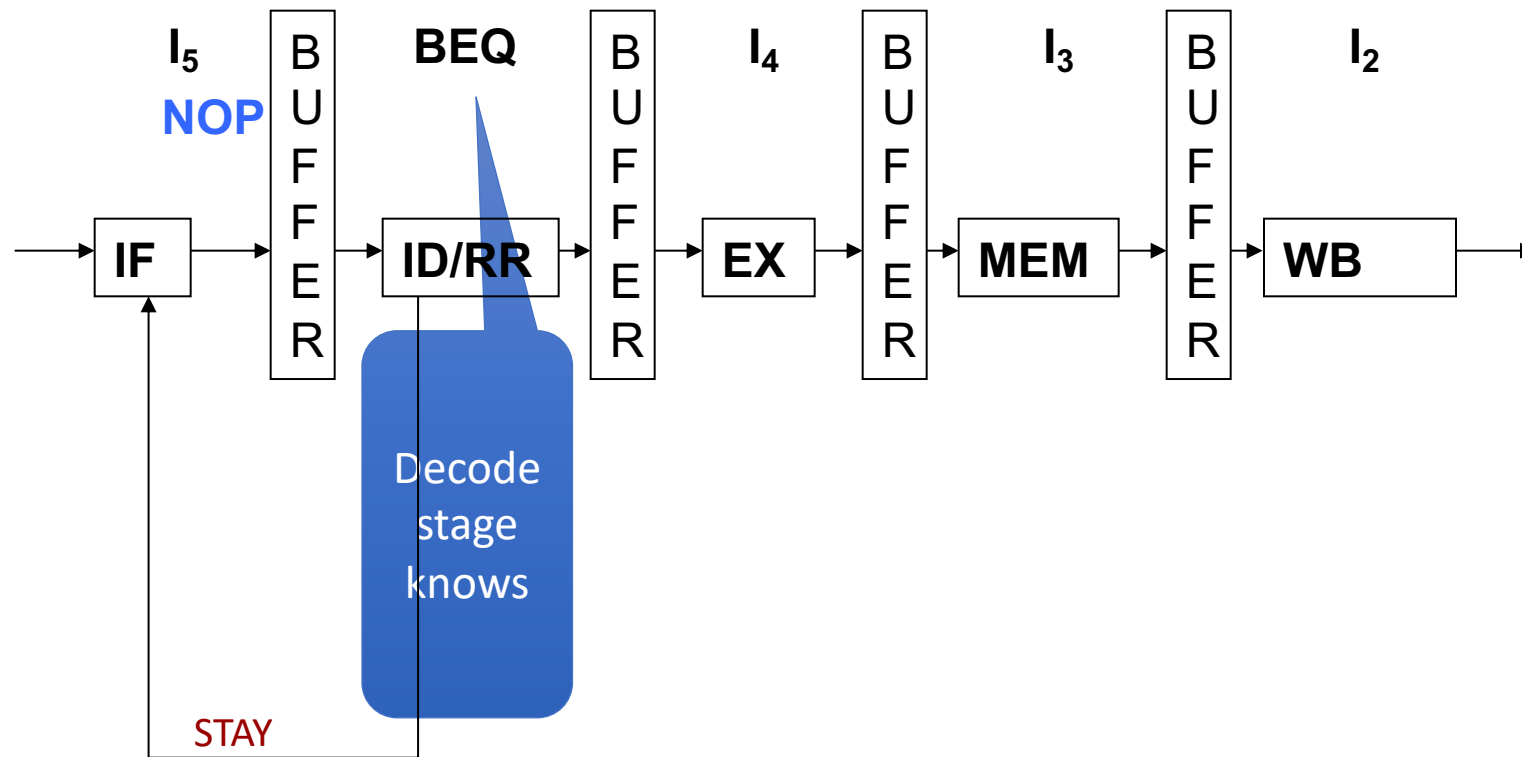
Conservative branch handling

Cycle 1



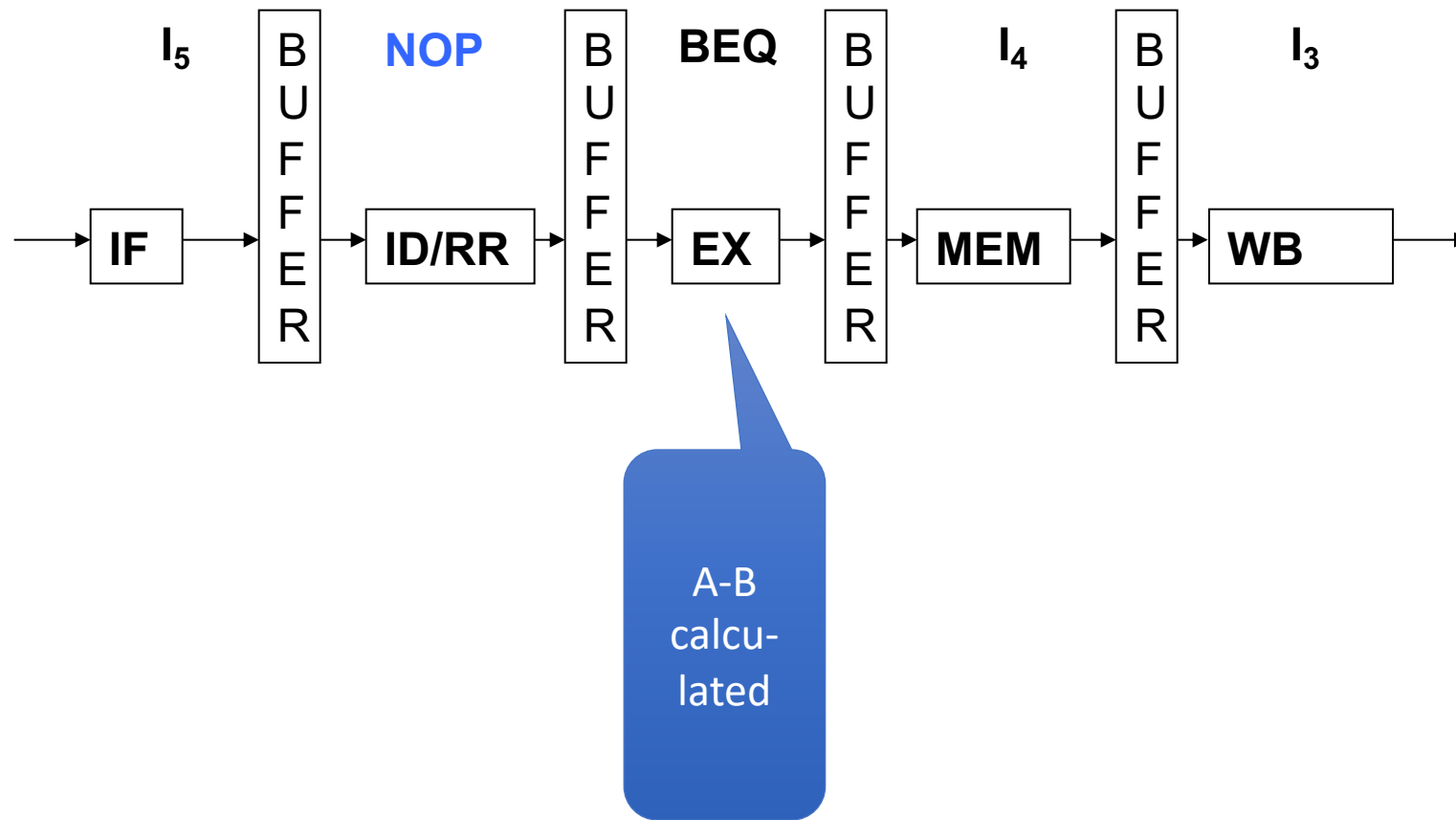
Branch handling

Cycle 2



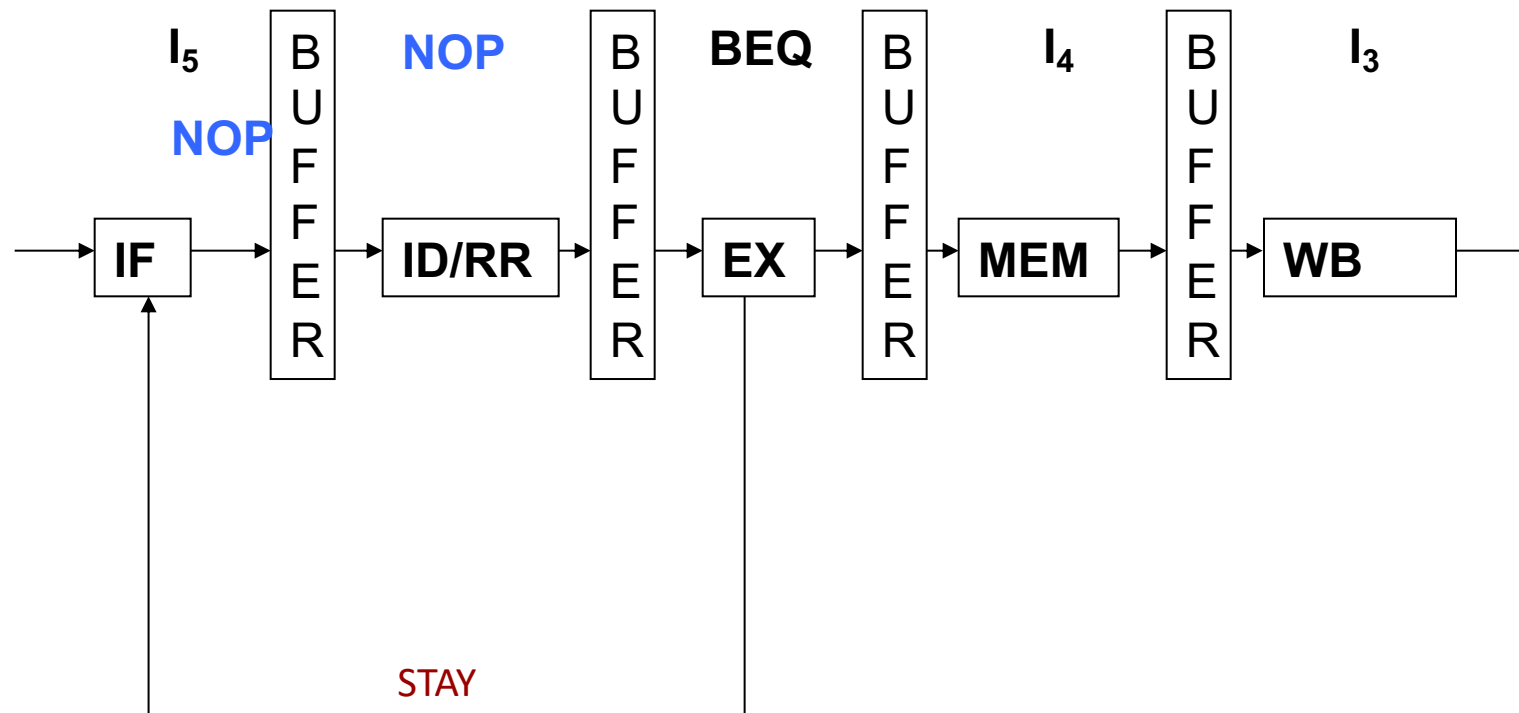
Branch handling

Cycle 3



Branch handling (taken)

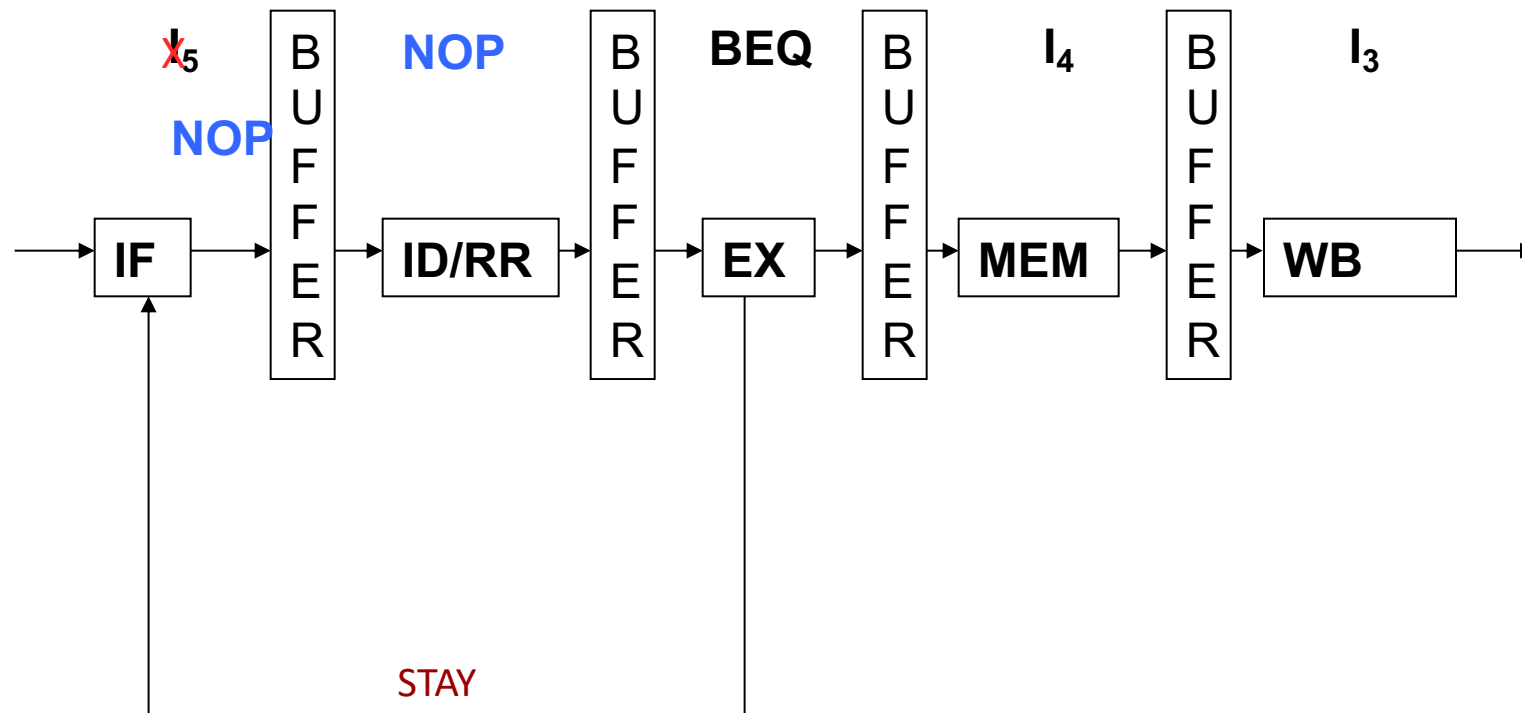
Cycle 3



A-B is calculated as zero and branch is going to be taken
PC+offset is being calculated

Branch handling (taken)

Cycle 3

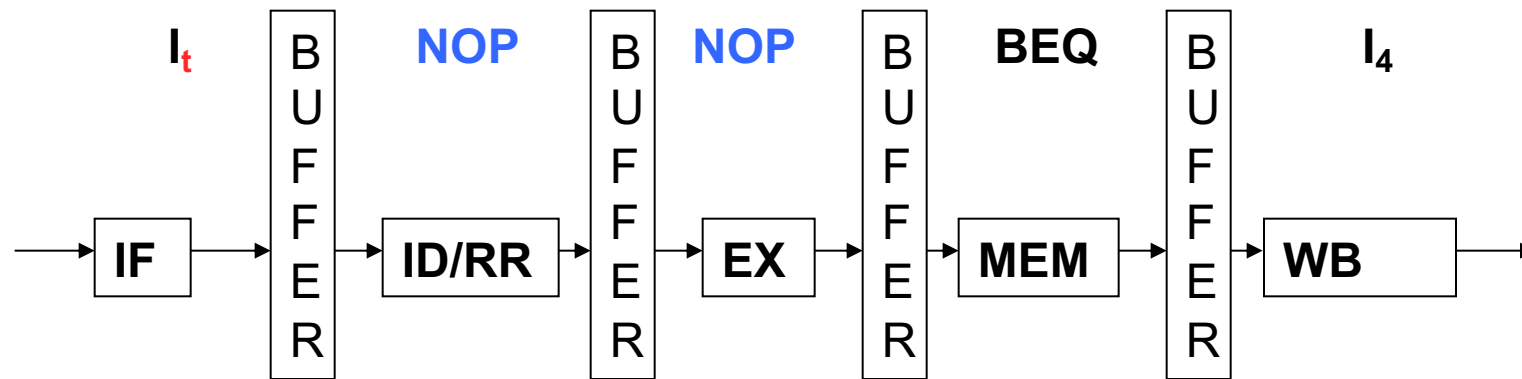


A-B is calculated as zero and branch is going to be taken

PC+offset is being calculated → value is clocked into PC at end of EX

Branch handling (taken)

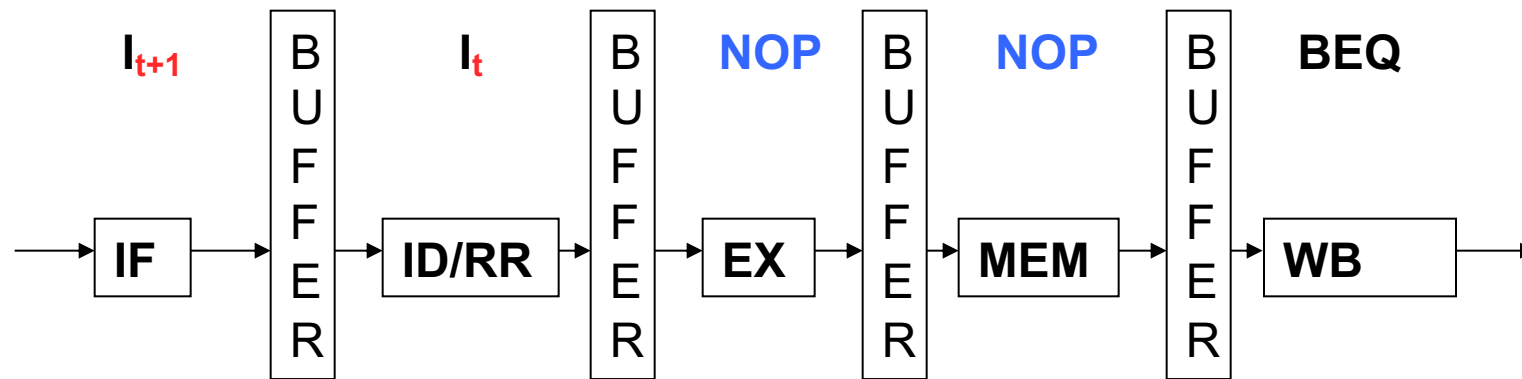
Cycle 4



We used updated PC to fetch the target of branch

Branch handling (taken)

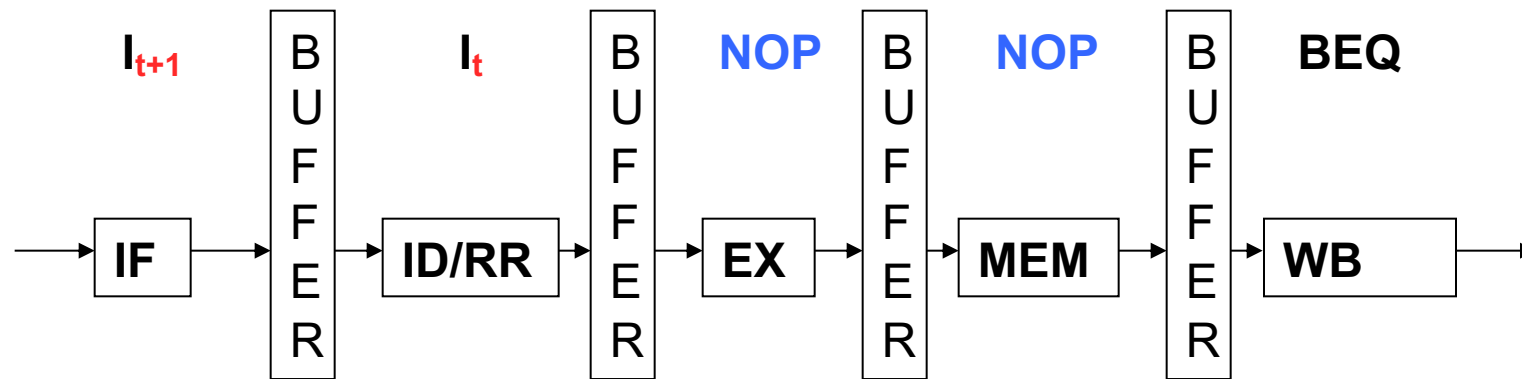
Cycle 5



Normal pipeline operation resumed

Branch handling (taken)

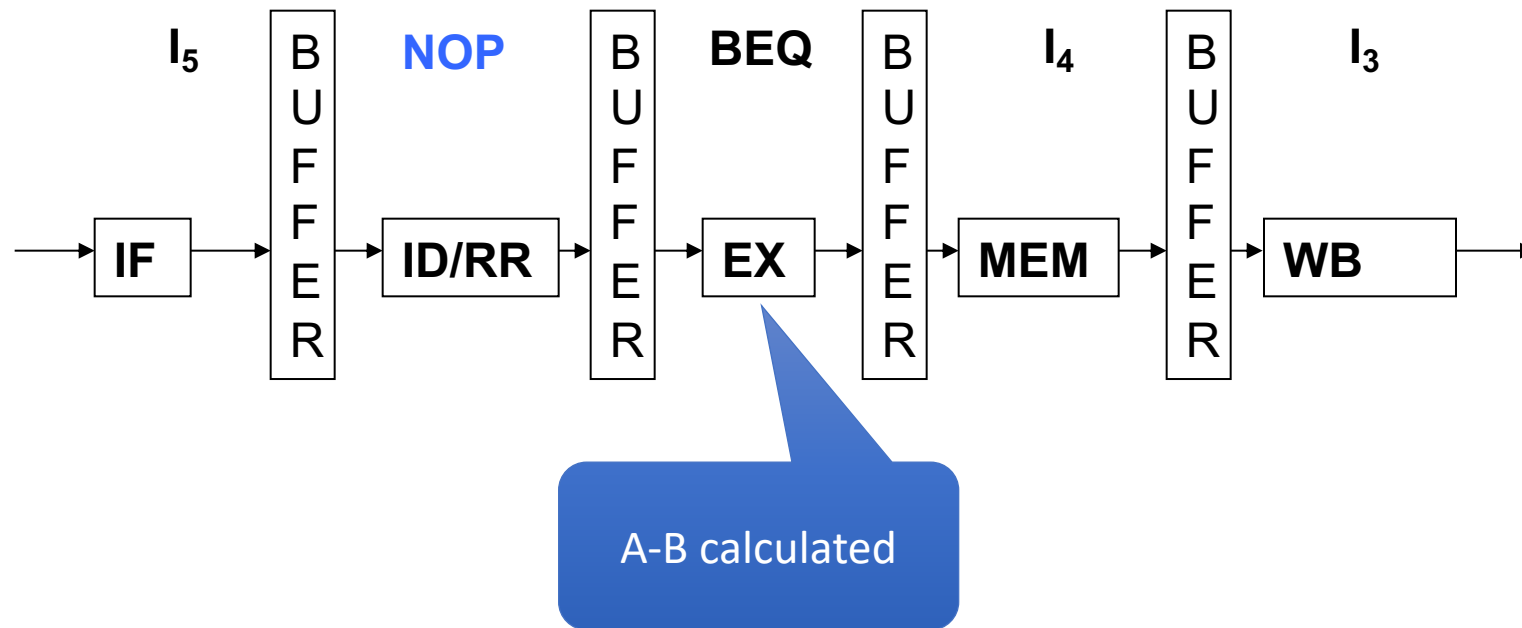
Cycle 5



Outcome: Two pipeline bubbles

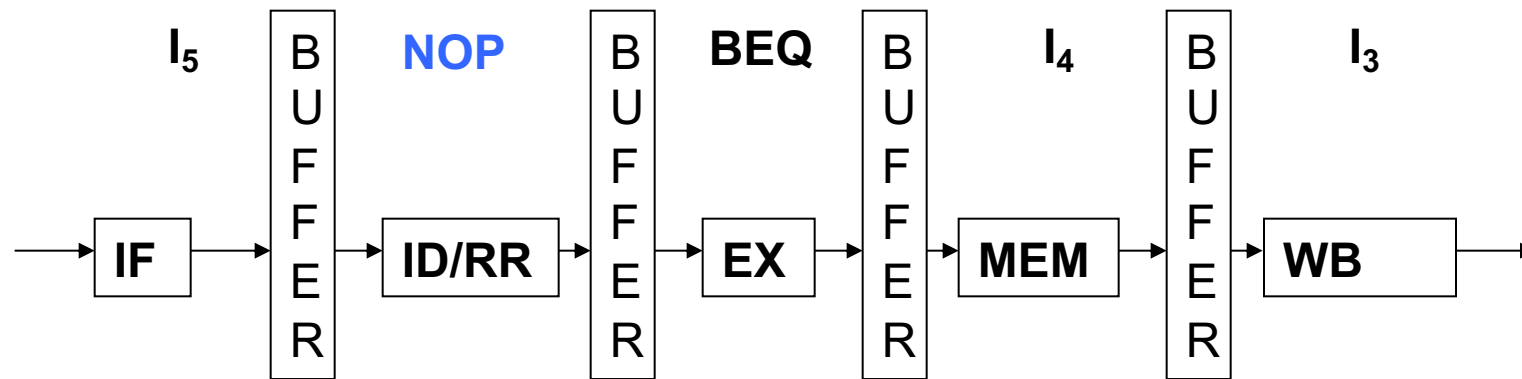
Branch handling

Cycle 3



Branch handling (NOT taken)

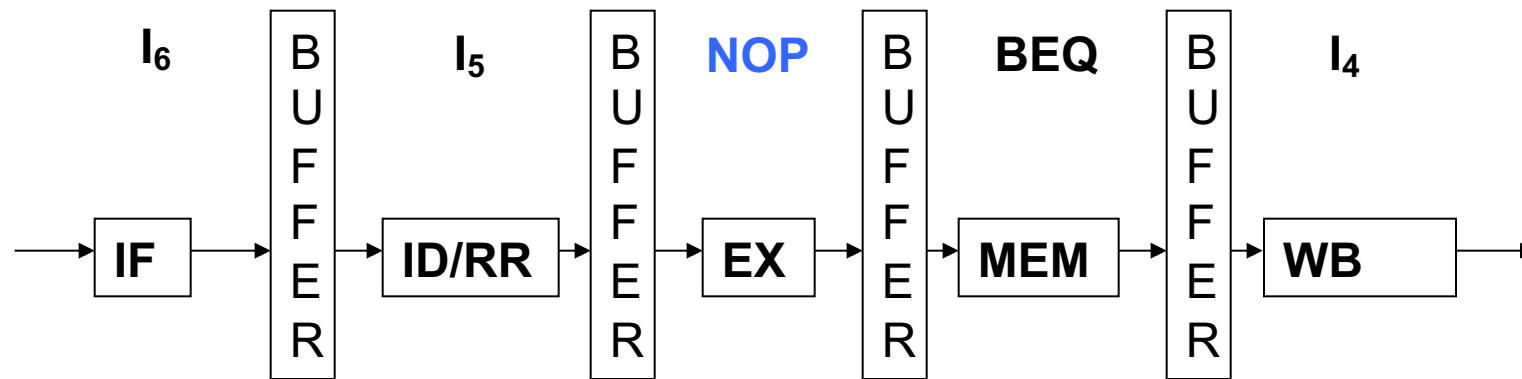
Cycle 3



A-B calculated as non-zero; branch not taken
PC is already pointing to the sequential path

Branch handling (NOT taken)

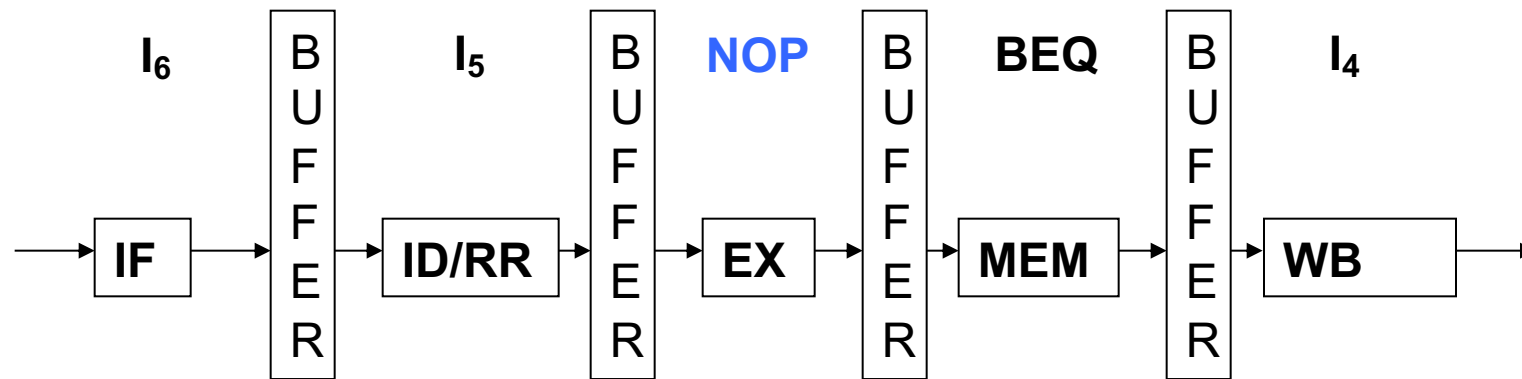
Cycle 4



A-B calculated as non-zero; branch not taken
Normal pipeline operation resumed

Branch handling (NOT taken)

Cycle 4



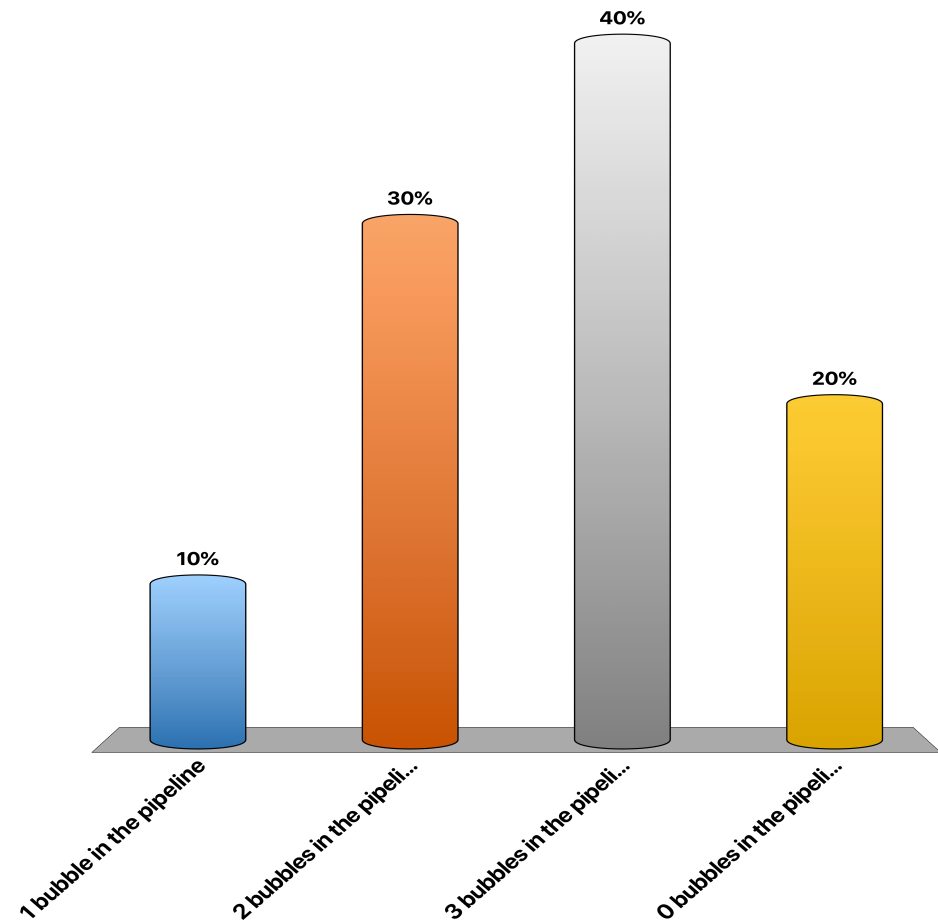
Outcome: one pipeline bubble



Question

With conservative handling of branches there will always be at least...

- A. 1 bubble in the pipeline
- B. 2 bubbles in the pipeline
- C. 3 bubbles in the pipeline
- D. 0 bubbles in the pipeline

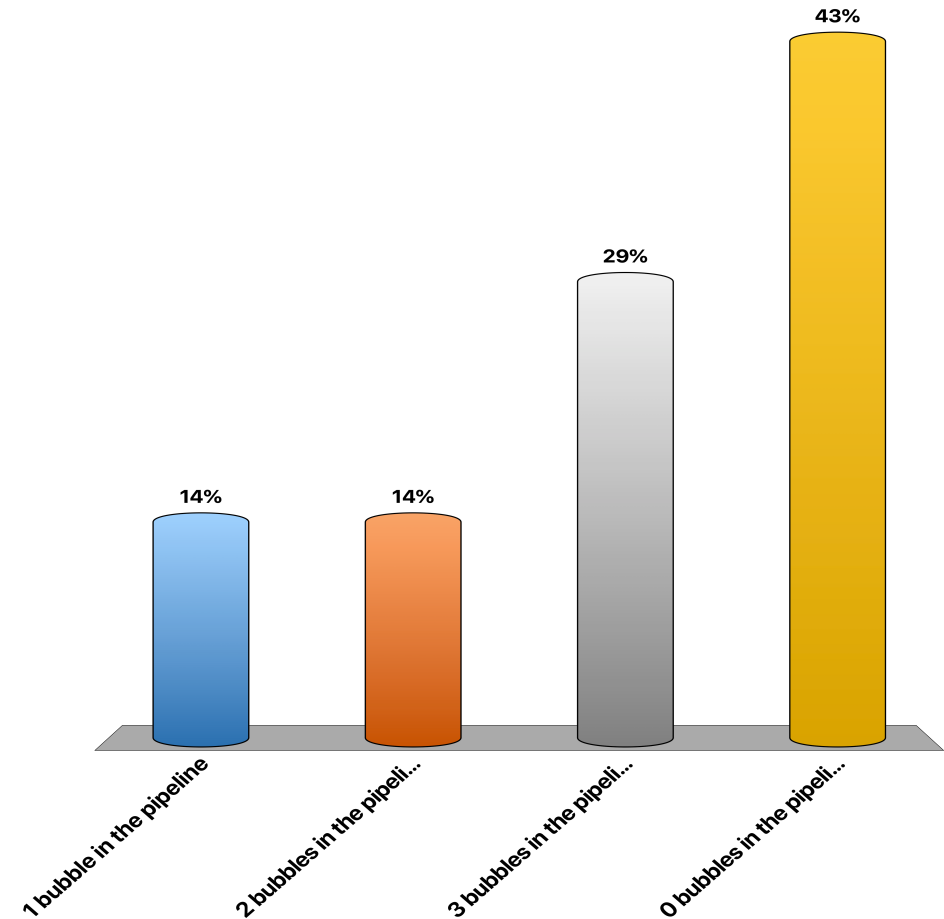




Question

With conservative handling of branches there will always be **at most...**

- A. 1 bubble in the pipeline
- B. 2 bubbles in the pipeline
- C. 3 bubbles in the pipeline
- D. 0 bubbles in the pipeline



Code for our examples

Assume the following sequence of instructions:

| | | |
|----|------|----|
| | BEQ | LI |
| | ADD | |
| | LW | |
| | | |
| LI | NAND | |
| | SW | |

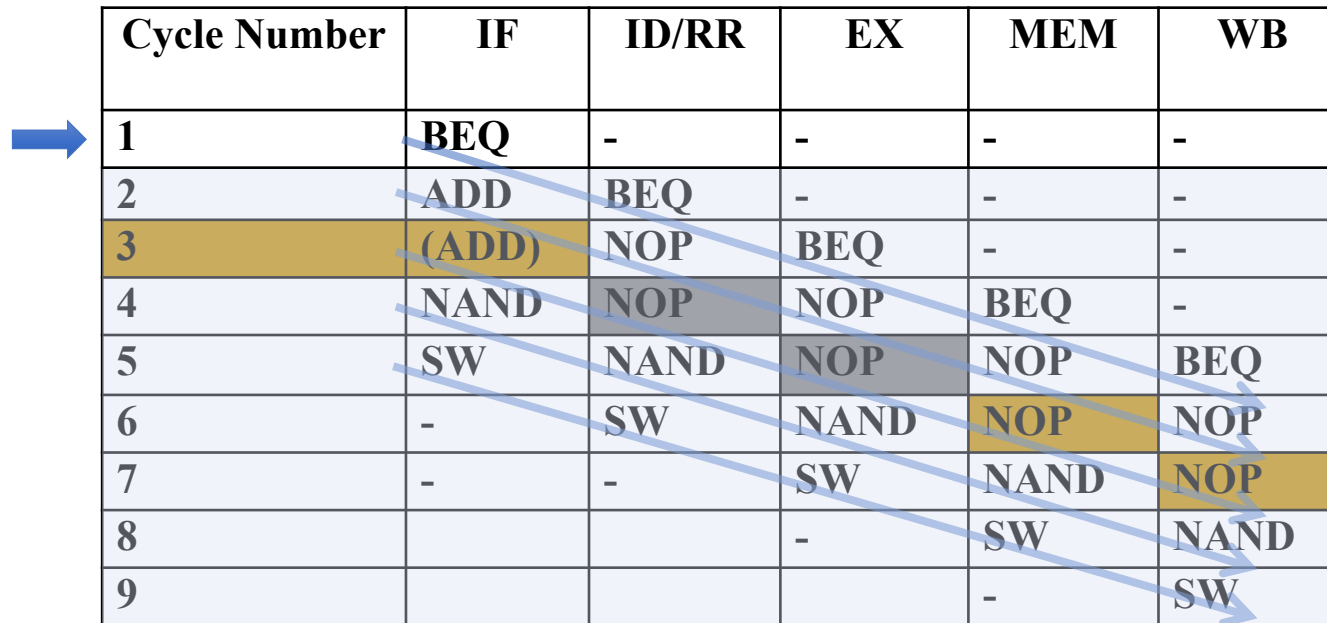
The instruction executed following the BEQ will either be ADD or it will be NAND depending on BEQ

Perfect pipeline

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | | - | - |
| 3 | NAND | ADD | BEQ | - | - |
| 4 | SW | NAND | ADD | BEQ | - |
| 5 | - | SW | NAND | ADD | BEQ |
| 6 | - | - | SW | NAND | ADD |
| 7 | | | - | SW | NAND |
| 8 | | | | - | SW |
| 9 | | | | | |

Unfortunately stalls ruin a perfect pipeline

Conservative Branch Handling



| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |

Cycle 2: Still Don't Know If We Branch

Only at the end of EX stage is the target address of BEQ known

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |

What should cycle 2 do?

- ➔ ID/RR tells IF to stall after fetching
- ➔ IF sends NOP to ID/RR at end of cycle

Cycle 3: Branch taken...

IF sends another NOP into ID/RR at cycle 3 end

EX sets PC to PC saved in FBUF + offset so NAND is fetched

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |

A - B

PC \leftarrow PC + offset

Assume BR
is taken

Cycle 4: Branch Taken

Only at the end of the EX stage was the target addr of BEQ known

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |



Assume BR
is taken

We have these two NOPs introduced by BEQ stalling the pipeline

Cycle 9: Finished

Cycles 5-9, the pipeline runs normally (i.e. no more stalls)


| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |



Finish here with stalls

Cycle 4: Branch Not Taken

EX on cycle 3 allows IF and ID/RR to proceed normally on cycle 4



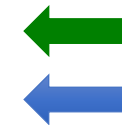
| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|-----|-----|-----|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | LW | ADD | NOP | BEQ | - |
| 5 | | LW | ADD | NOP | BEQ |
| 6 | - | | LW | ADD | NOP |
| 7 | - | - | | LW | ADD |
| 8 | | | - | | LW |
| 9 | | | | - | |



Finish here with stalls

What's the CPI difference?

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | (ADD) | NOP | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |



Branch taken: 9 total cycles / 3 instructions = 3 CPI

Not taken: 8 total cycles / 3 instructions = 2.66 CPI

Strategies for improving branches

- Branch prediction
- Delayed branch

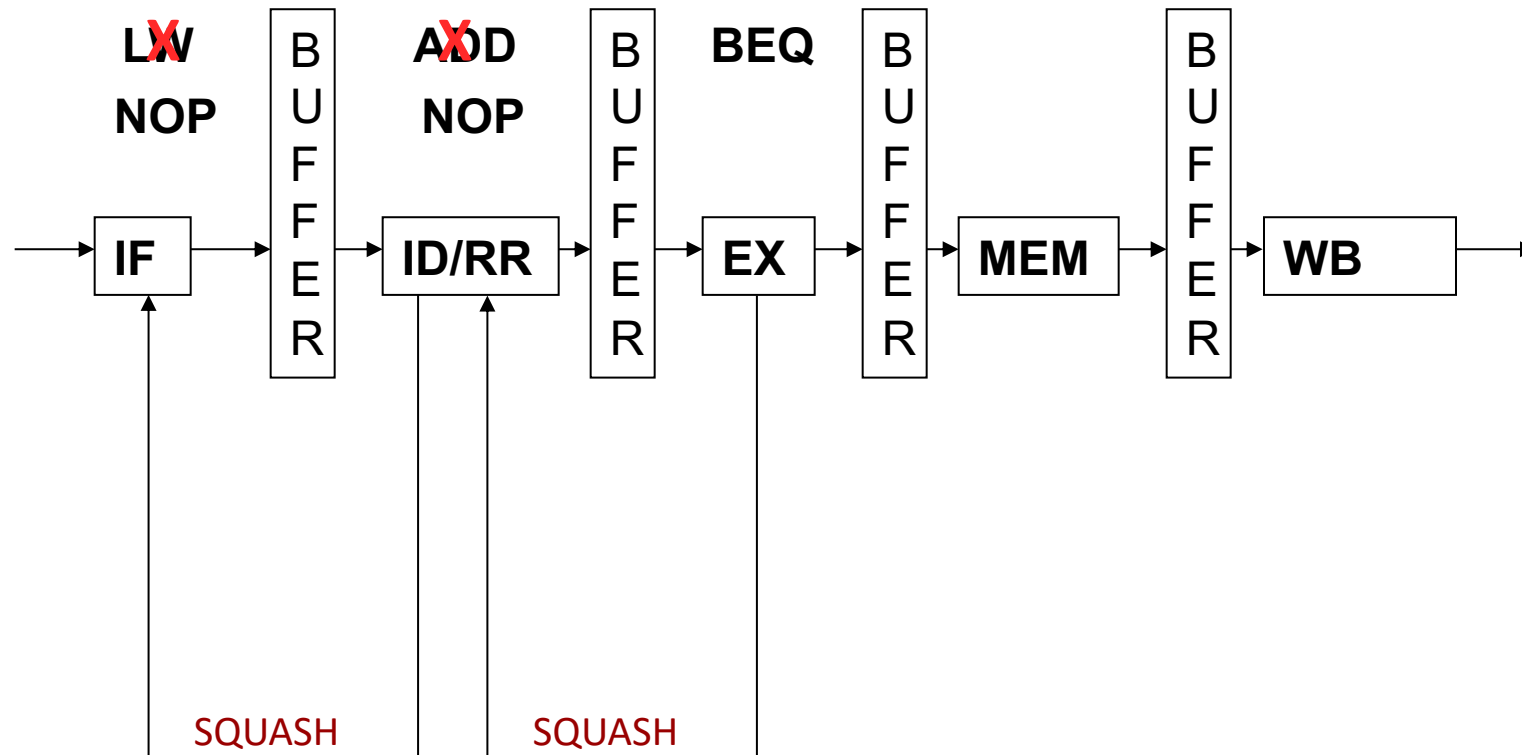
Branch prediction

Assume branch is not taken

| Cycle | IF* | ID/RR | EX | MEM | WB |
|-------|-----|-------|-----|-----|----|
| 1 | BEQ | | | | |
| 2 | ADD | BEQ | | | |
| 3 | LW | ADD | BEQ | | |

* We do not actually know what the instruction is in the Fetch Stage

Branch Prediction



Our prediction was wrong! Branch will be taken
Squash the instructions at IF and ID/RR

Branch not taken

Given the following sequence of instructions:

| | | |
|----|------|----|
| | BEQ | L1 |
| | ADD | |
| | LW | |
| | | |
| L1 | NAND | |
| | SW | |

Using the branch prediction approach, what is the observed CPI for the 3 instructions (BEQ, ADD, LW) or (BEQ, NAND, SW) in each case?

Happy path – prediction true!

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|-----|-------|-----|-----|-----|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | LW | ADD | BEQ | - | - |
| 4 | - | LW | ADD | BEQ | - |
| 5 | - | - | LW | ADD | BEQ |
| 6 | - | - | - | LW | ADD |
| 7 | - | - | - | - | LW |

7 cycles / 3 instructions – 2.33 CPI

What if prediction turns out false?

Branch prediction failed

Squash ADD and LW and replace with NOPs;
basically we degrade to the conservative approach

| Cycle Number | IF | ID/RR | EX | MEM | WB |
|--------------|------|-------|------|------|------|
| 1 | BEQ | - | - | - | - |
| 2 | ADD | BEQ | - | - | - |
| 3 | LW | ADD | BEQ | - | - |
| 4 | NAND | NOP | NOP | BEQ | - |
| 5 | SW | NAND | NOP | NOP | BEQ |
| 6 | - | SW | NAND | NOP | NOP |
| 7 | - | - | SW | NAND | NOP |
| 8 | | | - | SW | NAND |
| 9 | | | | - | SW |

9 cycles / 3 instructions – 3 CPI

Branch prediction heuristics

- How do we (quickly) guess which way a branch will go?
- Can often tell by comparing target with current PC
 - Loops usually branch backwards (target < PC)
 - Guess the branch will be taken
 - Can construct conditionals to usually branch forwards (target > PC)
 - Guess the branch won't be taken
- Some ISAs have 2 different families of branch instructions so that the compiler can signal whether it thinks the branch will be taken or not taken
- Or we could keep a history...

Branch History Table and Branch Target Buffer

PC Instr
1000 BEQ
...
1300 BEQ
...
1500 BEQ
...
1600 BEQ
...
1800 BEQ
...

| PC of branch instruction | Taken/Not Taken | PC of branch target |
|--------------------------|-----------------|---------------------|
| 1000 | T | 1200 |
| 1300 | N | xxxx |
| 1500 | T | 1100 |
| 1600 | N | xxxx |
| 1800 | N | xxxx |



Delayed branching

- Ignore the problem in hardware (i.e. no automatic generation of NOPs in fetch stage)
- Give the NOP instructions to the compiler
 - E.g. the instruction after a branch is *always* executed
- Let the compiler worry about it
 - Compile in a NOP instruction after every branch instruction
- Why could this be a good idea?
- The compiler may be able to optimize...

Example using MIPS-style branch delays

```
; Add 7 to each element of a ten-element array
; whose address is in a0
    addi    $t1, $a0, 40; When a0=t1 we are done
loop:
    beq     $a0, $t1, done
    nop                      ; branch delay slot
    lw      $t0, 0($a0)
    addi    $t0, $t0, 7
    sw      $t0, 0($a0)
    addi    $a0, $a0, 4
    beq     $zero, $zero, loop
    nop                      ; branch delay slot
done: halt
```


Delayed branch – before optimization

Let's
assume
one delay
slot

```
; Add 7 to each element of a ten-element array
; whose address is in a0
    addi    $t1, $a0, 40; When a0=t1 we are done
loop:
    beq     $a0, $t1, done
    nop                      ; branch delay slot
    lw      $t0, 0($a0)
    addi    $t0, $t0, 7
    sw      $t0, 0($a0)
    addi    $a0, $a0, 4
    beq     $zero, $zero, loop
    nop                      ; branch delay slot
done: halt
```

Delayed branch – during optimization


```
; Add 7 to each element of a ten-element array
; whose address is in a0
    addi    $t1, $a0, 40; When a0=t1 we are done
loop:
    beq     $a0, $t1, done
    nop                                ; branch delay slot
    lw      $t0, 0($a0)
    addi    $t0, $t0, 7
    sw      $t0, 0($a0)
    addi    $a0, $a0, 4
    beq     $zero, $zero, loop
    nop                                ; branch delay slot
done: halt
```



OK if LW is
redundantly
executed on loop exit

Delayed branch – during optimization

```
; Add 7 to each element of a ten-element array
; whose address is in a0
    addi    $t1, $a0, 40; When a0=t1 we are done
loop:
    beq     $a0, $t1, done
    nop                                ; branch delay slot
    lw      $t0, 0($a0)
    addi    $t0, $t0, 7
    sw      $t0, 0($a0)
    addi    $a0, $a0, 4
    beq     $zero, $zero, loop
    nop                                ; branch delay slot
done: halt
```



The diagram illustrates the execution of a branch instruction in a processor with a branch delay slot. It shows two instances of a branch instruction: `beq $a0, $t1, done` and `beq $zero, $zero, loop`. For each branch, a green arrow points from the branch instruction to the instruction in the following slot, which is a `nop` (no operation). This `nop` is labeled as the "branch delay slot". The first `nop` is followed by the instructions `lw $t0, 0($a0)`, `addi $t0, $t0, 7`, `sw $t0, 0($a0)`, and `addi $a0, $a0, 4`. The second `nop` is followed by the instruction `beq $zero, $zero, loop`. The code ends with `done: halt`.

Delayed branch – after opt

```
; Add 7 to each element of a ten-element array
; whose address is in a0
    addi    t1, a0, 40    ; When a0=t1 we are done
loop:
    beq     $a0, $t1, done
    lw      $t0, 0($a0) ; branch delay slot

    addi    $t0, $t0, 7
    sw      $t0, 0($a0)

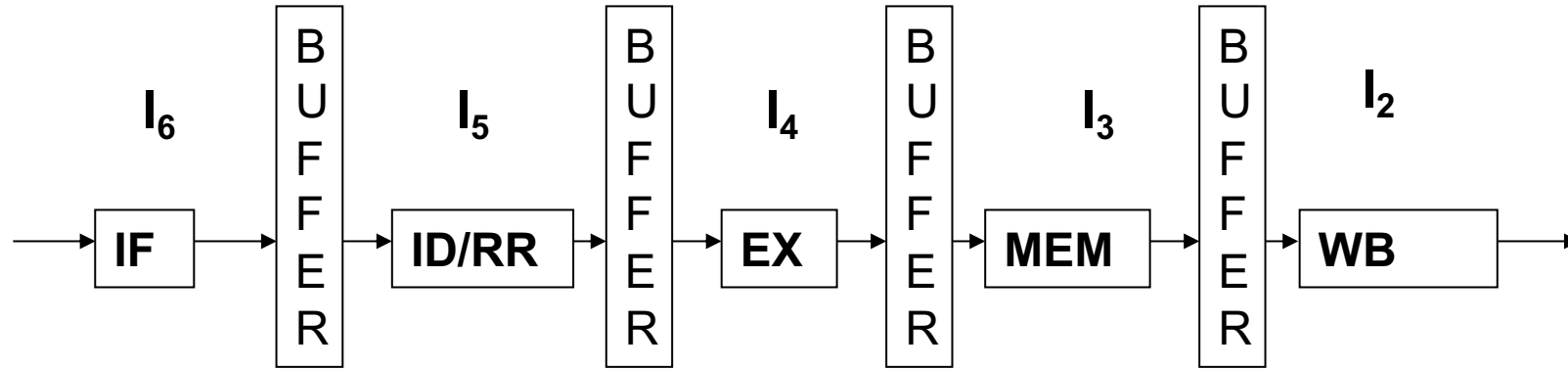
    beq     $zero, $zero, loop
    addi    $a0, $a0, 4 ; branch delay slot
done: halt
```


| Name | Pros | Cons | Use cases |
|---|---|---|---|
| Stall the pipeline | Simple strategy, no hardware needed for squashing 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 squash instructions in partial execution in the pipeline | 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 squashing 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 Exposes microarchitectural details to ISA level Assembly harder to understand | Older RISC architectures such as MIPS, PA-RISC, SPARC |

Summary of hazards

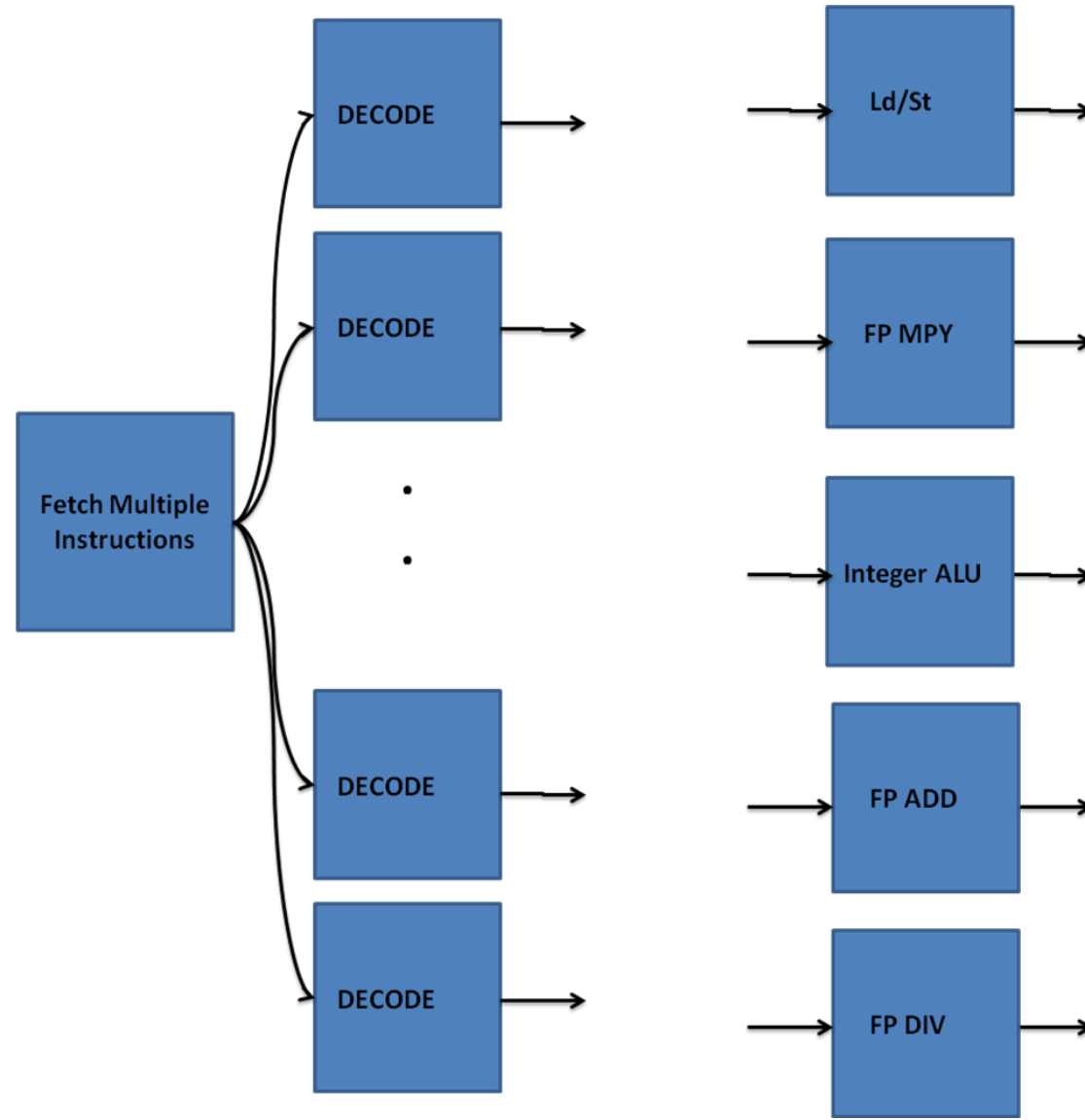
| Instruction | Type of Hazard | Potential Stalls | With Data forwarding | With branch prediction (branch not taken) |
|------------------|----------------|------------------|----------------------|---|
| ADD, NAND | Data | 0, 1, 2, or 3 | 0 | Not Applicable |
| LW | Data | 0, 1, 2, or 3 | 0 or 1 | Not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

Program discontinuities

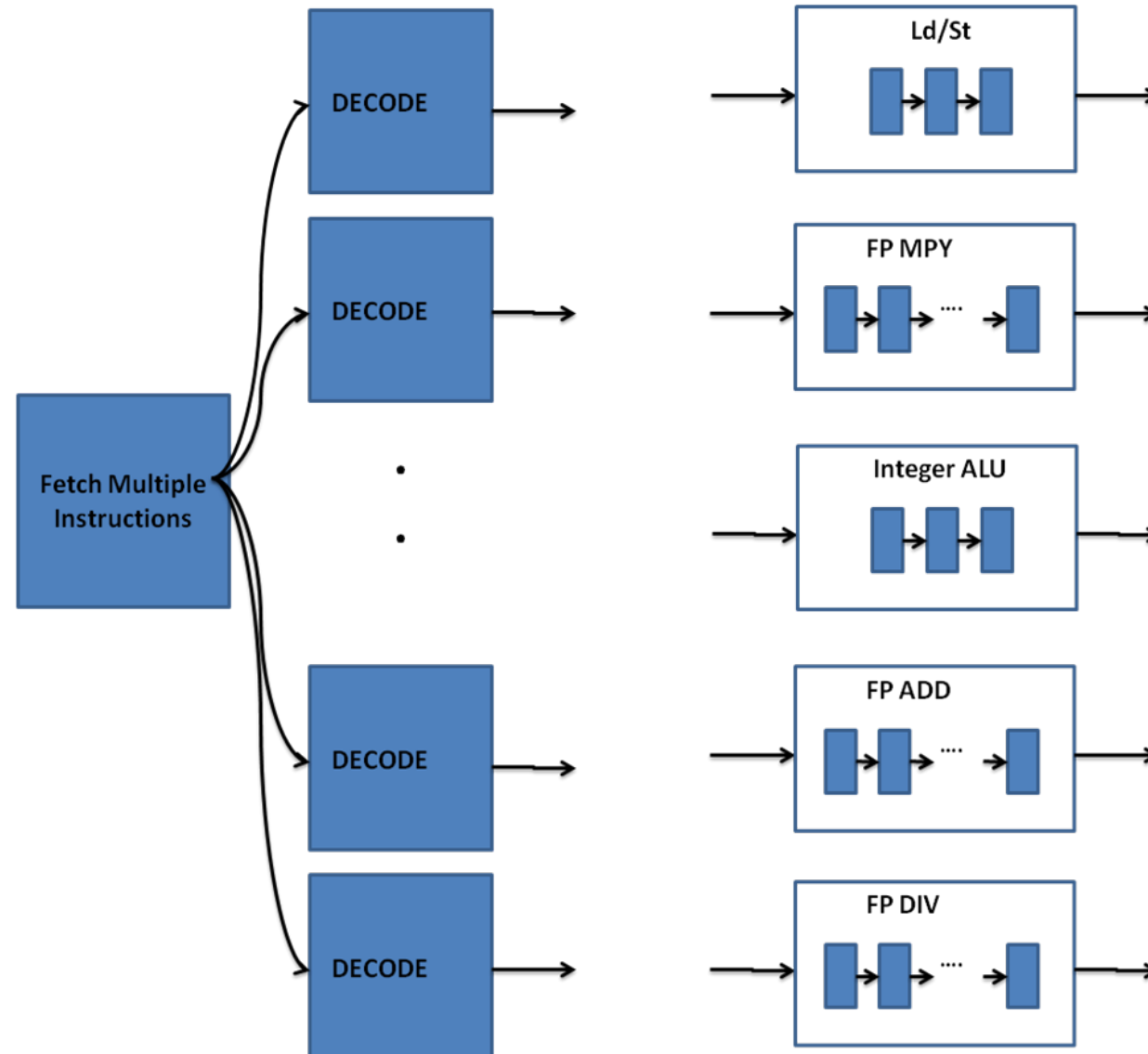


- External interrupt can occur at any time
- Any one of the 5 instructions may cause an exception
- Any one of the the 5 instructions may be a TRAP instruction
- When do we take the interrupt?
- STOP, DRAIN, go to INT state
 - Slow to respond to external events

Superscalar pipelines



Different Pipeline Depths



Intel Core Microarchitecture: An example pipeline

