

# **CSC 252/452: Computer Organization**

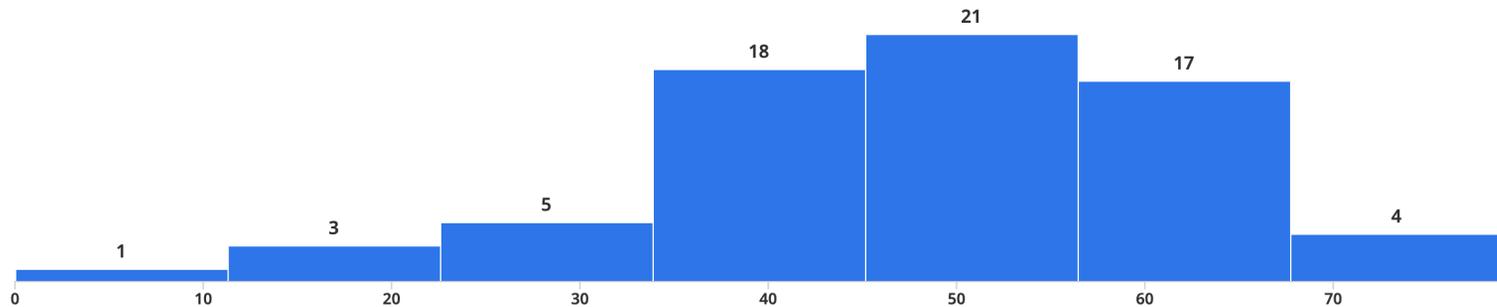
## **Fall 2025: Lecture 14**

Instructor: Yanan Guo

Department of Computer Science  
University of Rochester

# Announcements

- Midterm grades are out
  - You can see who graded your exam on Gradescope.
  - You can submit re-grade request on Gradescope or going to office hours.



Minimum

**6.45**

Median

**50.1**

Maximum

**71.45**

Mean

**47.77**

Std Dev [?](#)

**13.36**

# Pipeline Stages

## Fetch

- Use PC to read instruction
- **Compute new PC for non-jump instructions**

## Decode

- Read program registers

## Execute

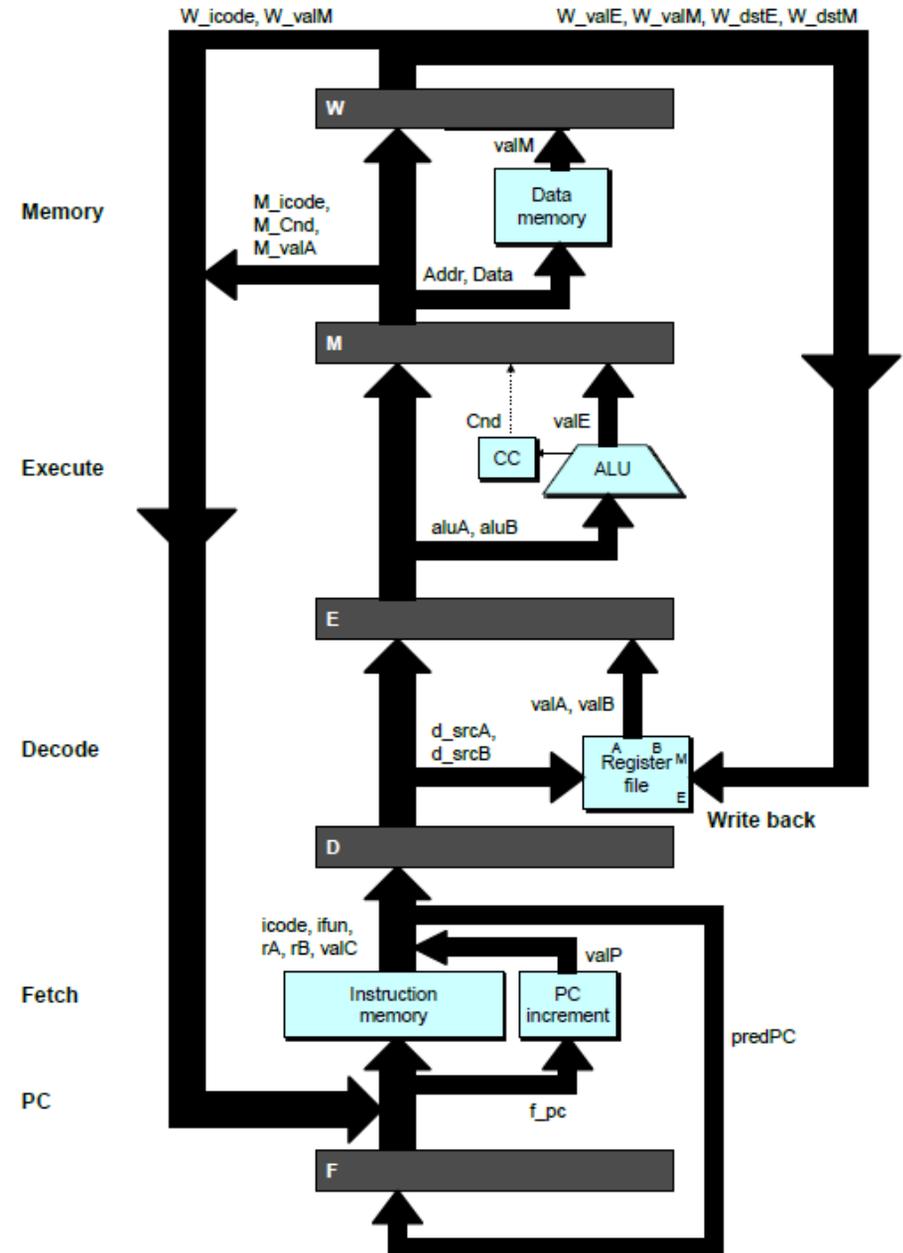
- Operate ALU
- **Compute new PC for jump instructions**

## Memory

- Read or write data memory

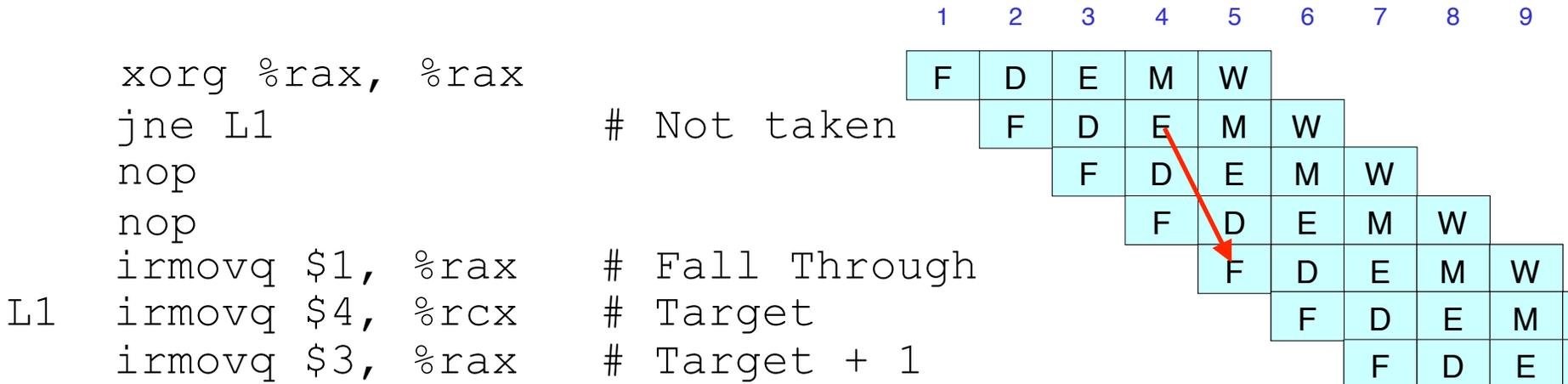
## Write Back

- Update register file



# Control Dependency

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



# Better Pipelining

## Fetch

- Use PC to read instruction
- **Compute new PC for non-jump instructions**

## Decode

- Read program registers
- **Compute new PC for jump instructions**

## Execute

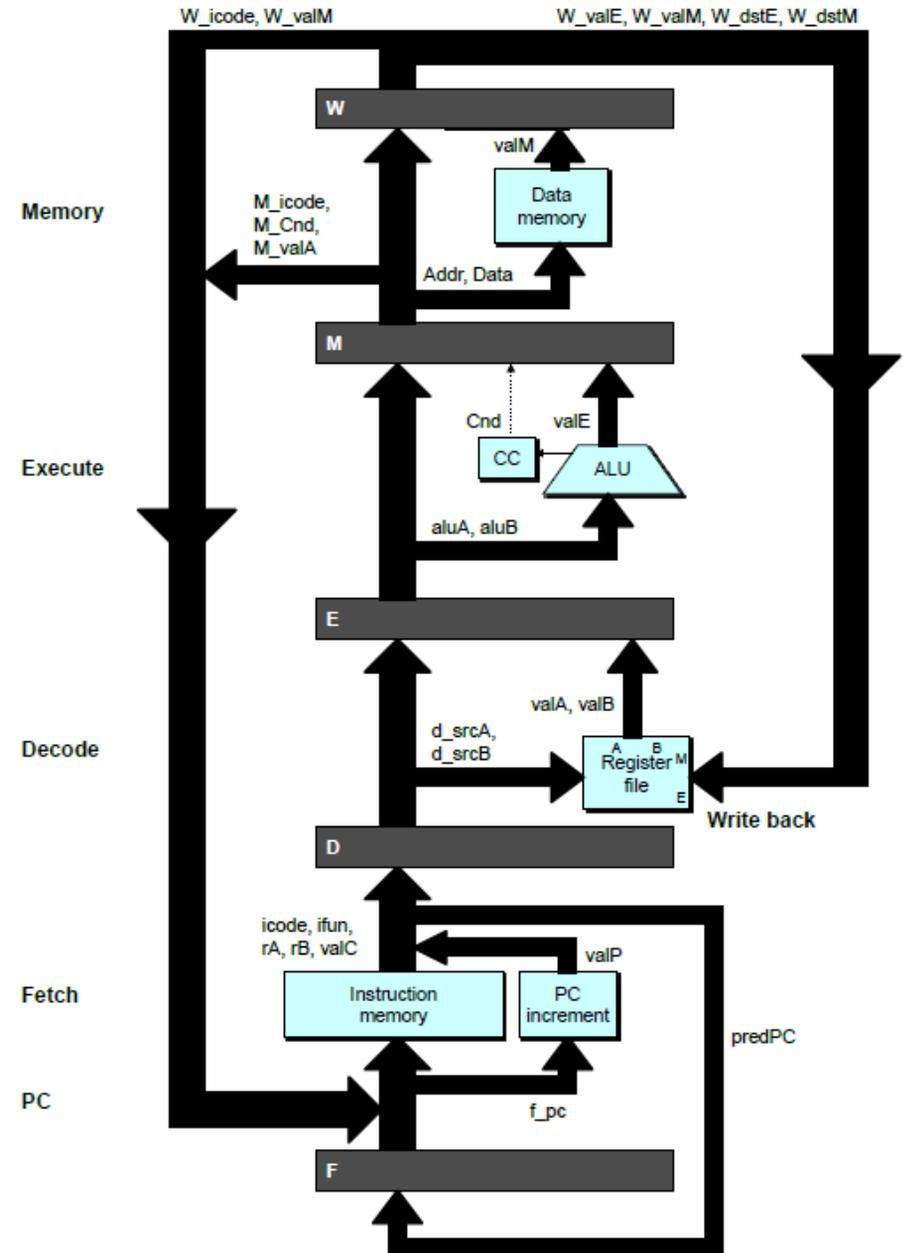
- Operate ALU

## Memory

- Read or write data memory

## Write Back

- Update register file



# Better Pipelining

## Fetch

- Use PC to read instruction
- **Compute new PC for non-jump instructions**

## Decode

- Read program registers
- **Compute new PC for jump instructions**

## Execute

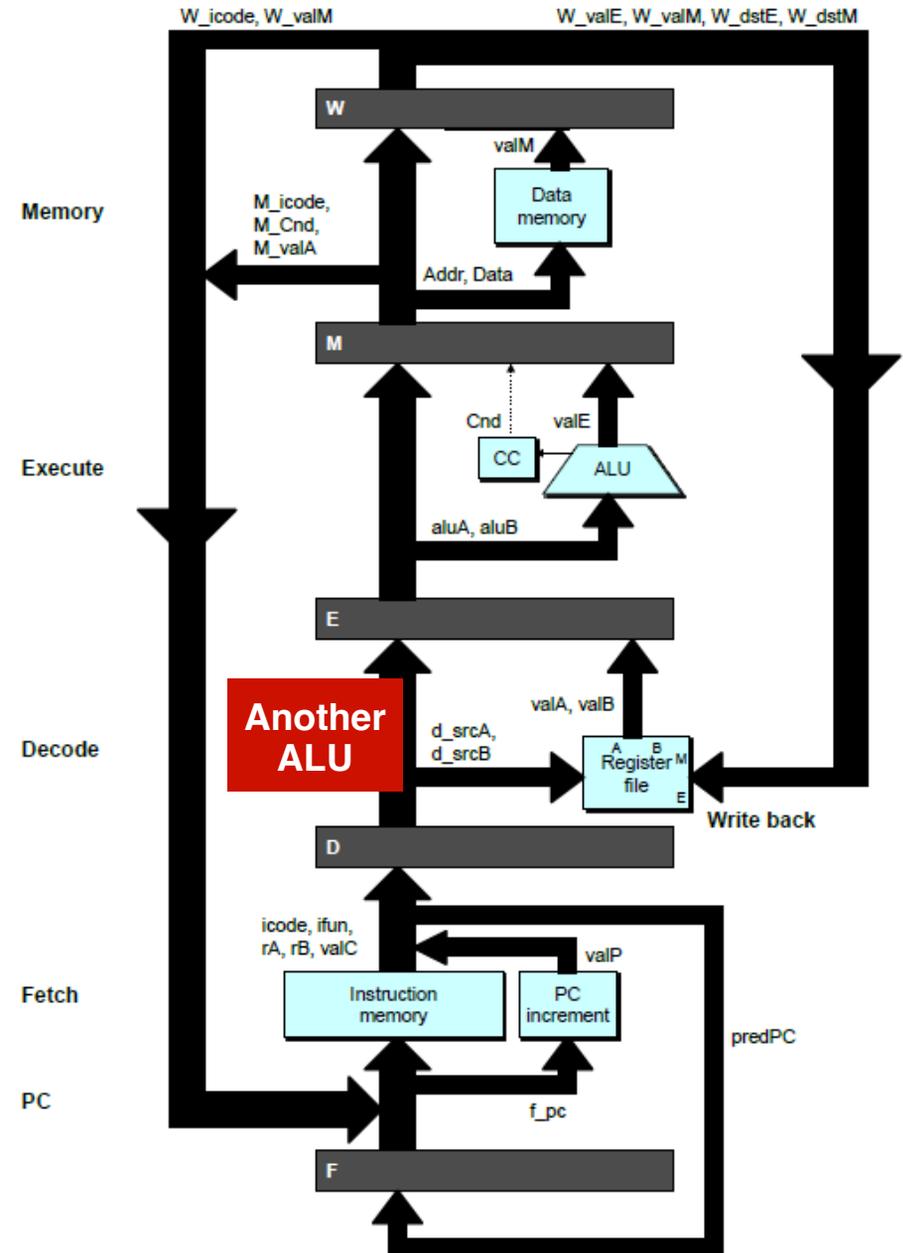
- Operate ALU

## Memory

- Read or write data memory

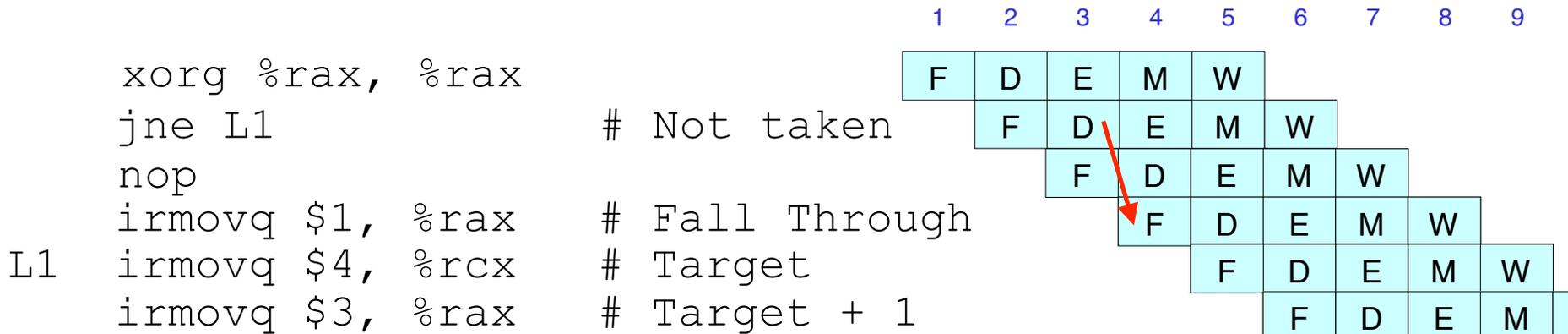
## Write Back

- Update register file



# Saving One Cycle

- **Definition:** Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



# Resolving Control Dependencies

- **Software Mechanisms**

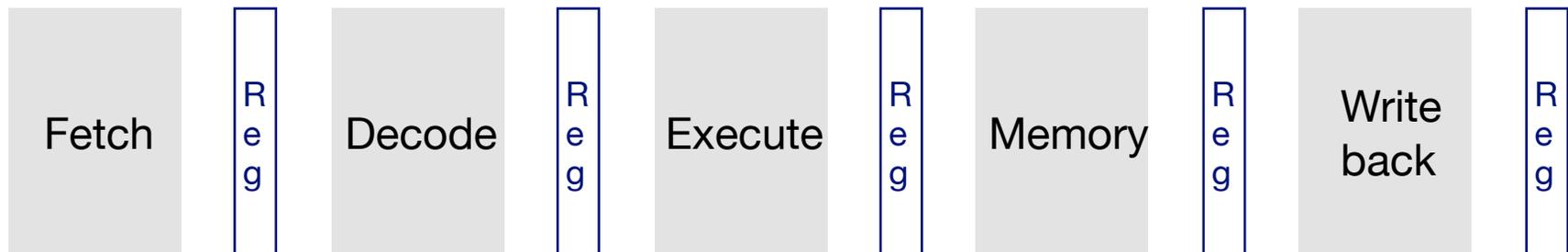
- Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach

- **Hardware mechanisms**

- Stalling (Think of it as hardware automatically inserting nops)
- Branch Prediction
- Return Address Stack

# Hardware Generated Nops (Bubble and Stalling)

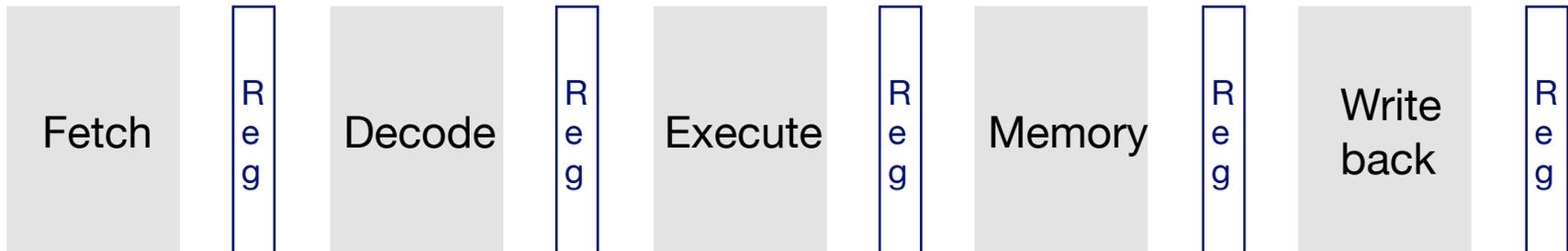
- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

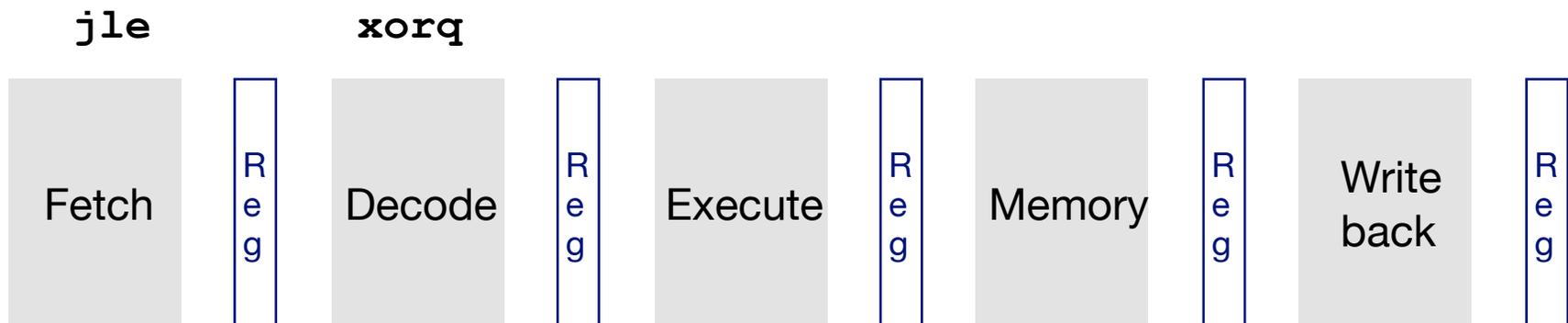
- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?

`xorq`



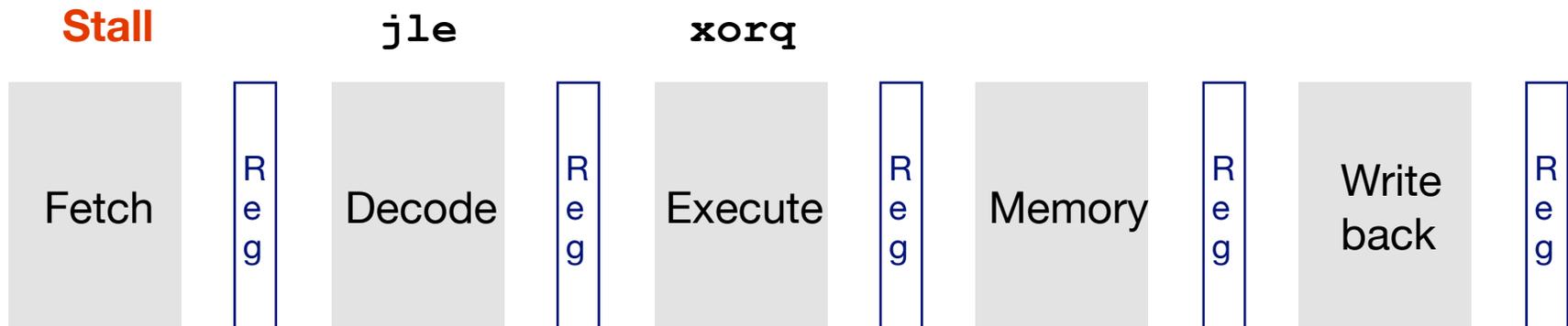
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



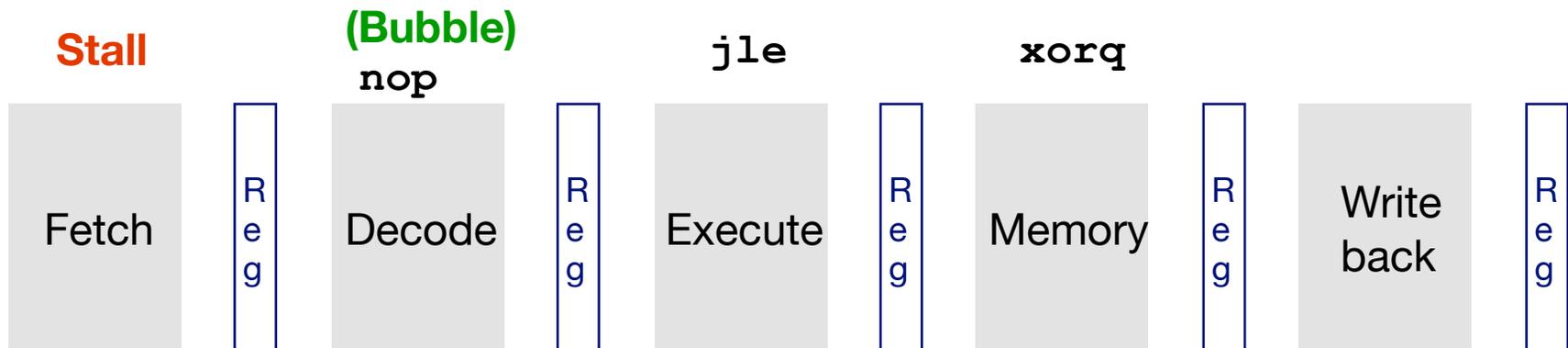
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



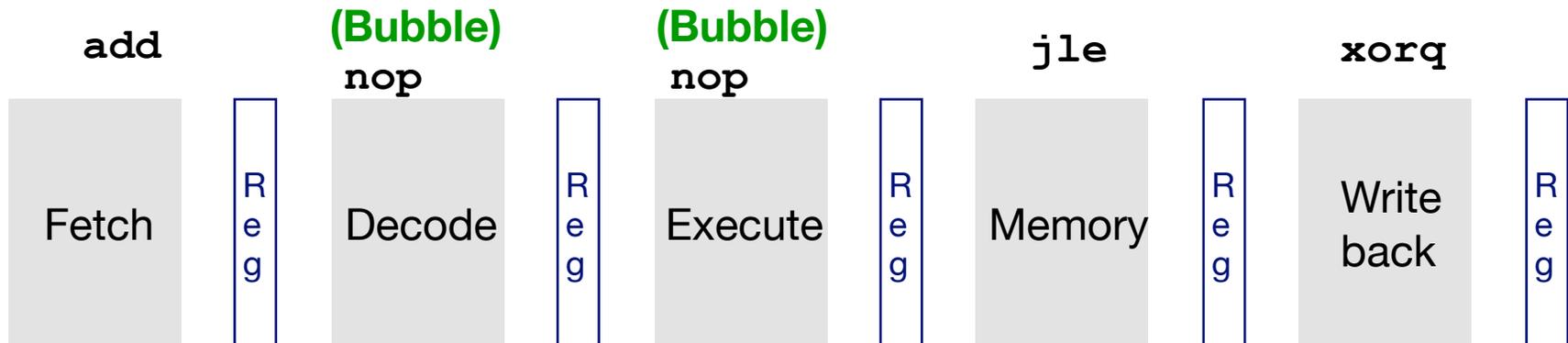
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



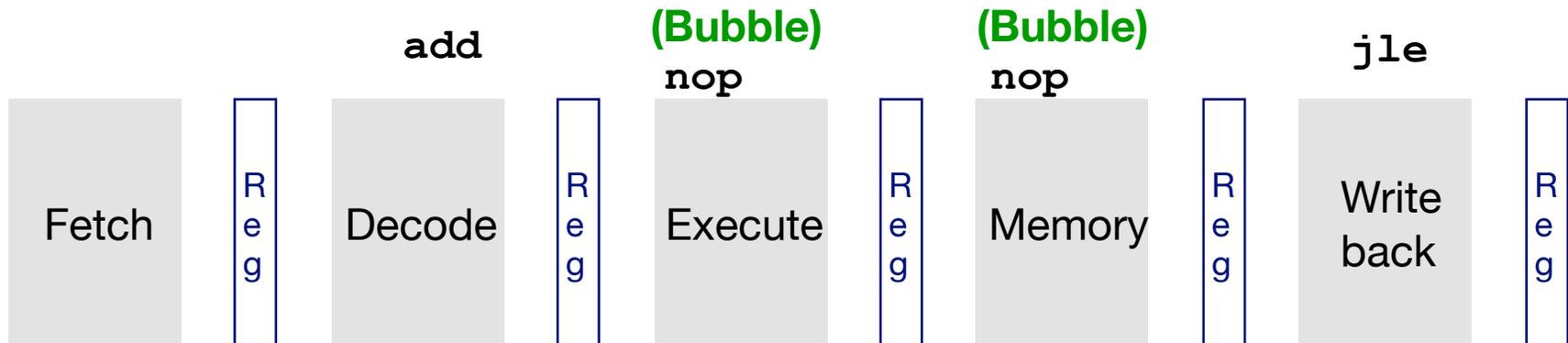
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



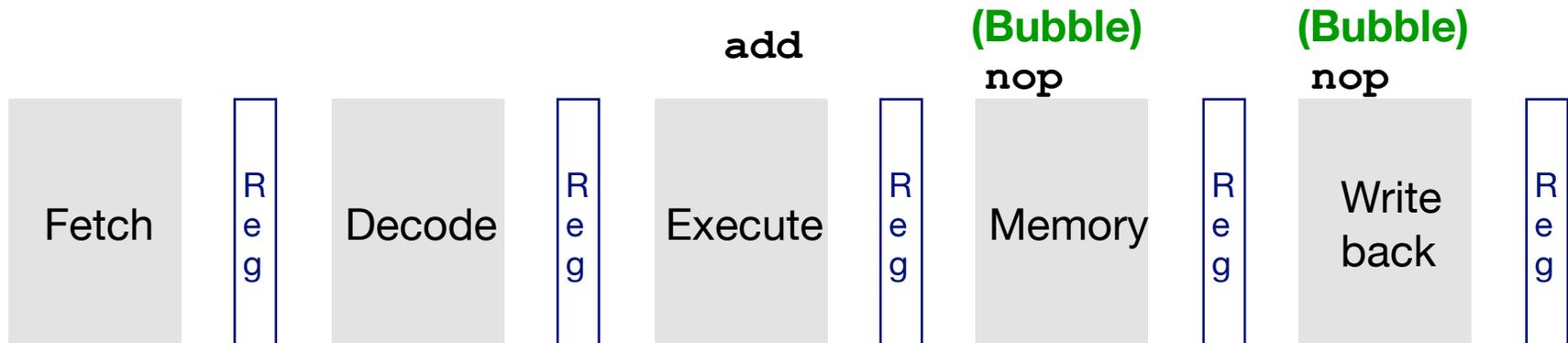
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



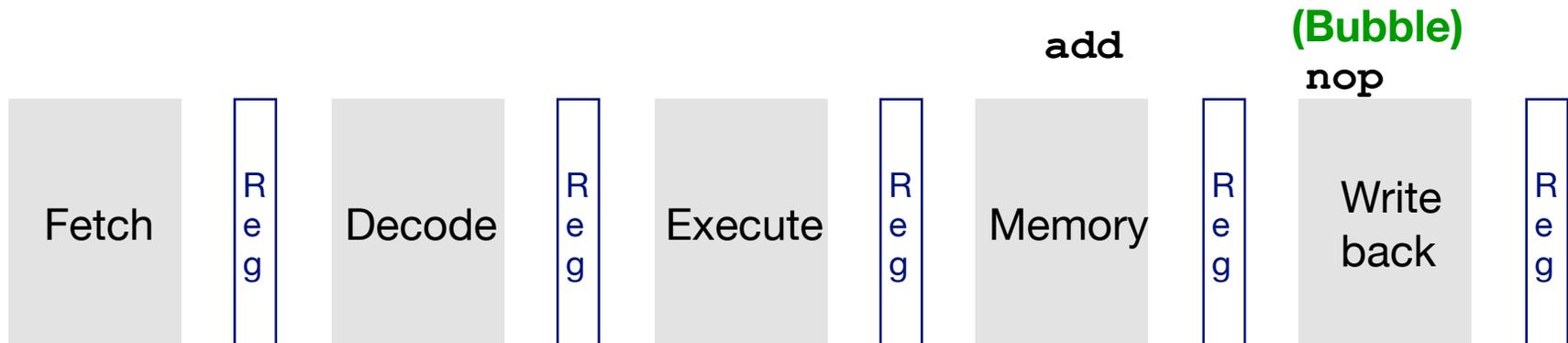
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



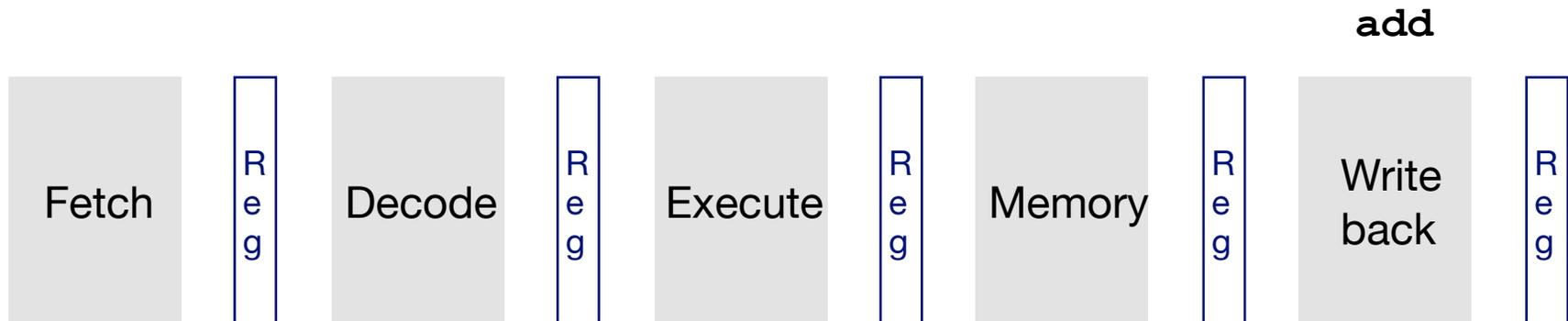
# Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

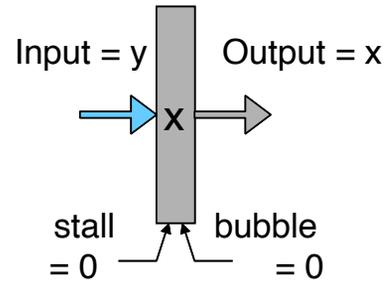
- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



# How are Stall and Bubble Implemented in Hardware?

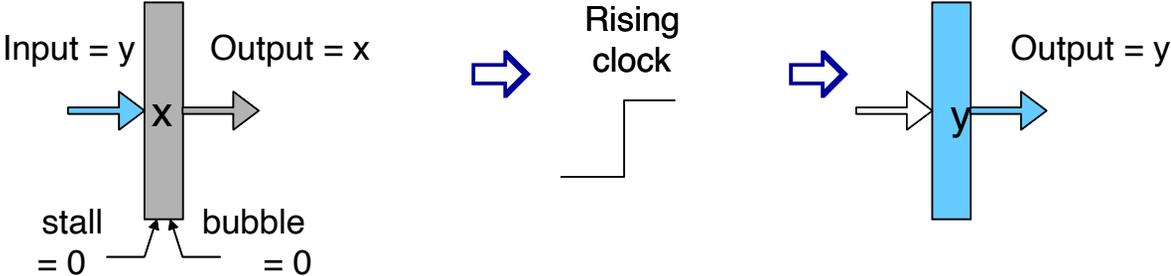
# How are Stall and Bubble Implemented in Hardware?

**Normal**



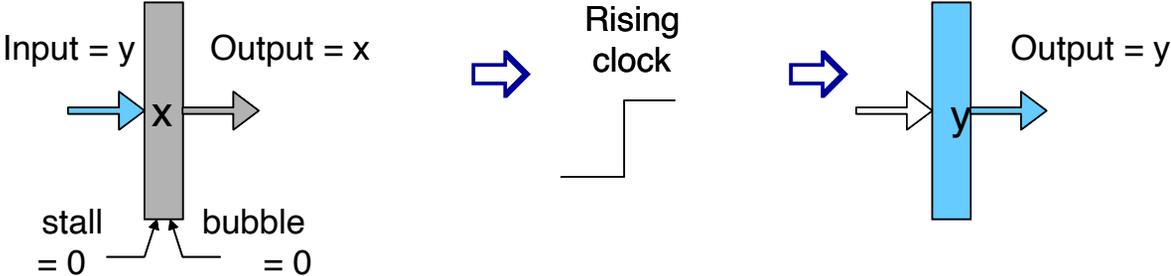
# How are Stall and Bubble Implemented in Hardware?

**Normal**

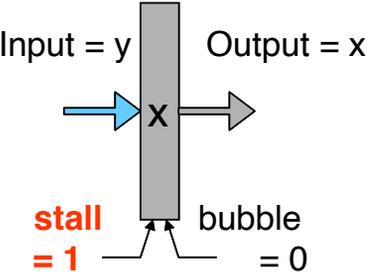


# How are Stall and Bubble Implemented in Hardware?

**Normal**

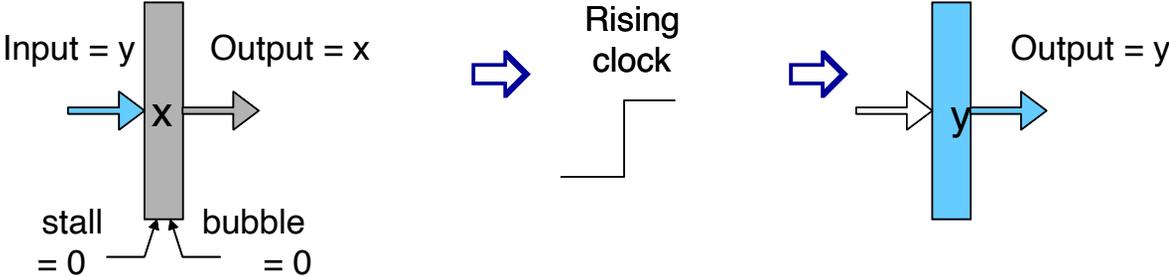


**Stall**

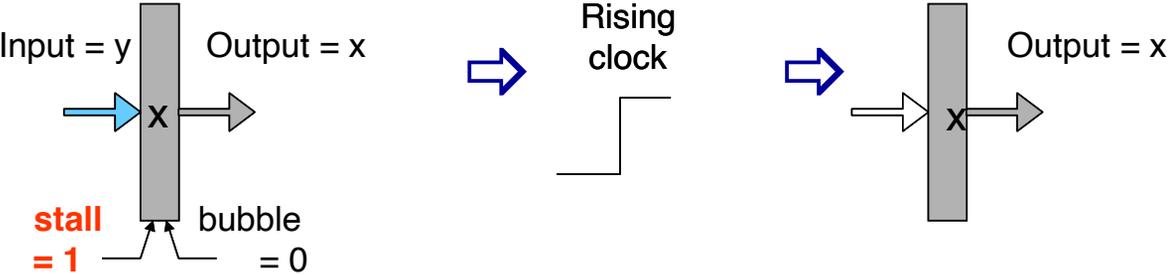


# How are Stall and Bubble Implemented in Hardware?

**Normal**

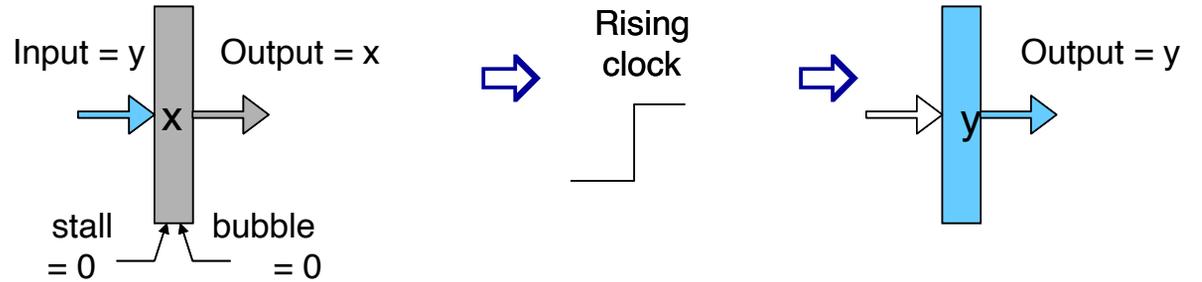


**Stall**

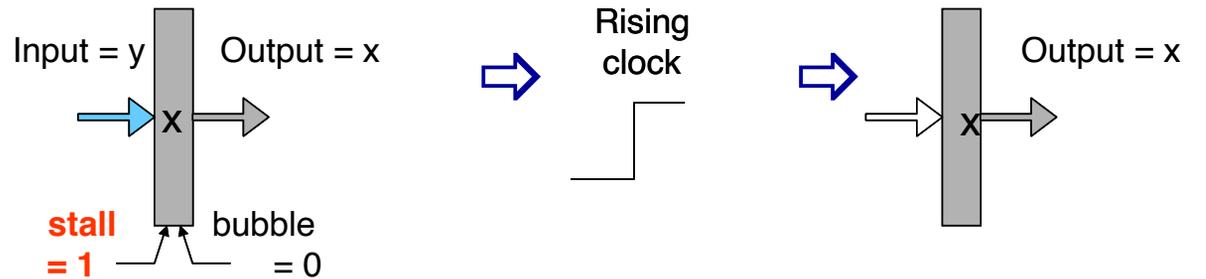


# How are Stall and Bubble Implemented in Hardware?

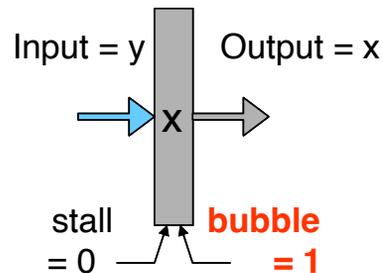
**Normal**



**Stall**

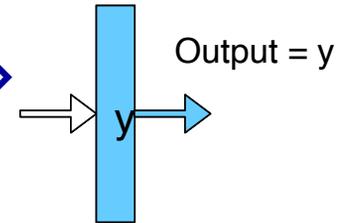
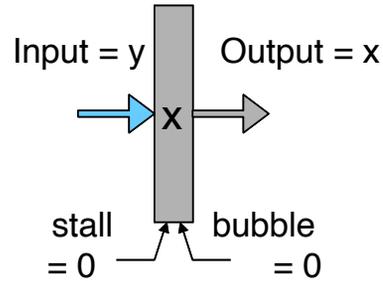


**Bubble**

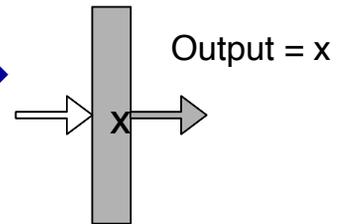
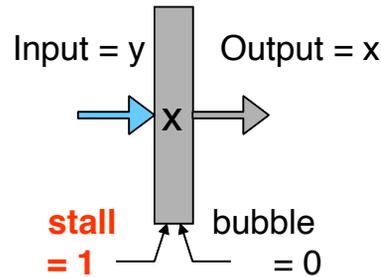


# How are Stall and Bubble Implemented in Hardware?

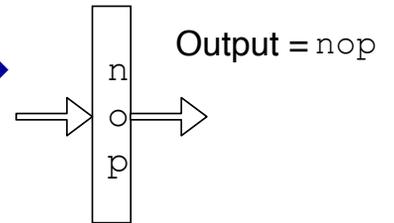
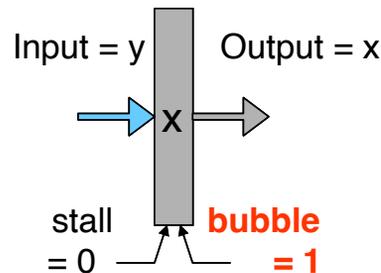
**Normal**



**Stall**



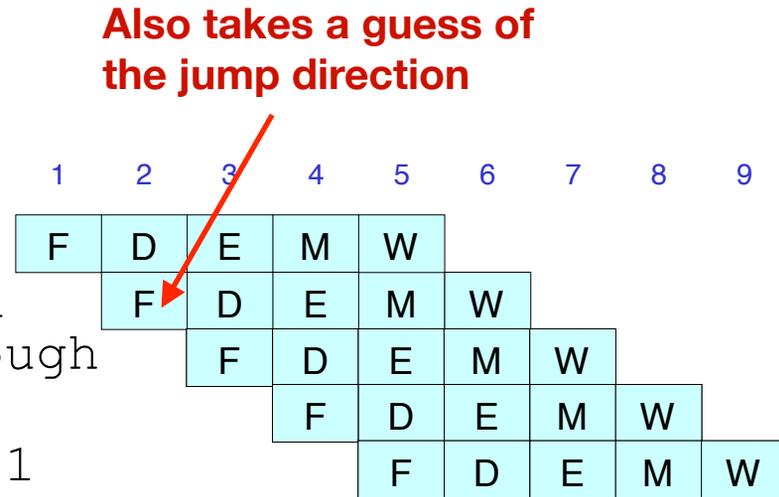
**Bubble**



# Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

```
xorg %rax, %rax
jne L1 # Not taken
irmovq $1, %rax # Fall Through
L1: irmovq $4, %rcx # Target
    irmovq $3, %rax # Target + 1
```

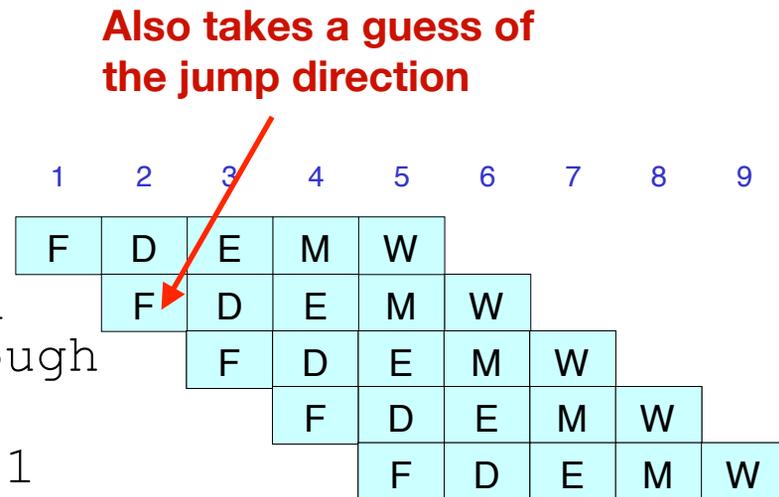


# Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

```
L1  xorg %rax, %rax
     jne L1          # Not taken
     irmovq $1, %rax # Fall Through
     irmovq $4, %rcx # Target
     irmovq $3, %rax # Target + 1
```



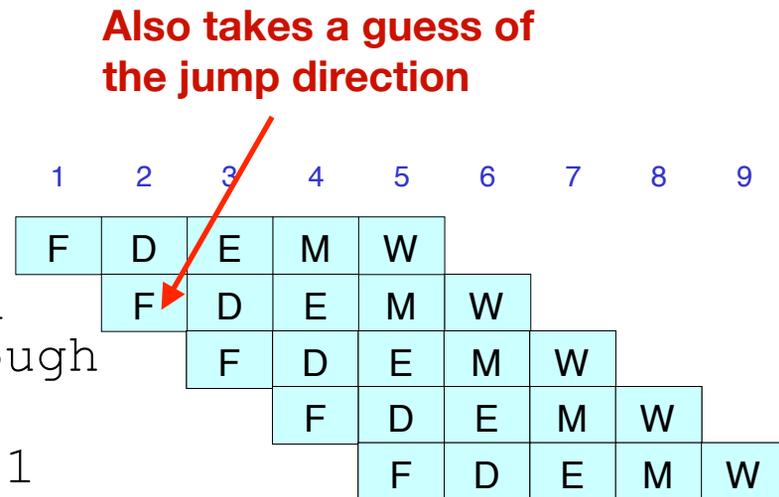
# Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

```
xorg %rax, %rax
jne L1 # Not taken
irmovq $1, %rax # Fall Through
L1: irmovq $4, %rcx # Target
    irmovq $3, %rax # Target + 1
```



# Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

## Static Prediction

- Always Taken
- Always Not-taken

## Dynamic Prediction

- Dynamically predict taken/not-taken for each specific jump instruction

# Static Prediction

# Static Prediction

Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

# Static Prediction

## Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

```
    cmpq    %rsi,%rdi
    jle    .corner_case
    <do_A>
.corner_case:
    <do_B>
    ret
```

# Static Prediction

## Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

```
    cmpq    %rsi, %rdi
    jle    .corner_case
    <do_A>
.corner_case:
    <do_B>
    ret
```

 **Mostly not taken**

# Static Prediction

## Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

```
    cmpq    %rsi, %rdi
    jle    .corner_case
    <do_A>
.corner_case:
    <do_B>
    ret

    .L1:   <body>
          cmpq B, A
          jl  .L1
          <after>
```

 **Mostly not taken**

# Static Prediction

## Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

```
    cmpq    %rsi, %rdi
    jle    .corner_case
    <do_A>
.corner_case:
    <do_B>
    ret
```

 **Mostly not taken**

```
    <before>
.L1:  <body>
      cmpq  B, A
      jl   .L1
    <after>
```

 **Mostly taken**

# Static Prediction

## Observation (Assumption really): Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

## Strategy:

- Forward jumps (i.e., `if-else`): always predict not-taken
- Backward jumps (i.e., `loop`): always predict taken

```
cmpq    %rsi, %rdi    <before>
jle     .corner_case  .L1: <body>
<do_A>
.corner_case:
<do_B>
ret
```

**Mostly not taken** (arrow pointing to `.corner_case`)

**Mostly taken** (arrow pointing to `jl .L1`)

# Static Prediction

Knowing branch prediction strategy helps us write faster code

- Any difference between the following two code snippets?
- What if you know that hardware uses the always non-taken branch prediction?

```
if (cond) {  
    do_A()  
} else {  
    do_B()  
}
```

```
if (!cond) {  
    do_B()  
} else {  
    do_A()  
}
```

# Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

# Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

```
for (i=0; i <5; i++) {...}
```

# Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

```
for (i=0; i <5; i++) {...}
```

Iteration #1	0	1	2	3	4
Predicted Outcome	<b>N</b>	T	T	T	<b>T</b>
Actual Outcome	T	T	T	T	N

# Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

```
for (i=0; i <5; i++) {...}
```

Iteration #1	0	1	2	3	4
Predicted Outcome	<b>N</b>	T	T	T	<b>T</b>
Actual Outcome	T	T	T	T	N



# Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

# Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```

Predict with 1-bit	<b>N</b>	T	T	T	<b>T</b>
Actual Outcome	T	T	T	T	N
Predict with 2-bit	<b>N</b>	<b>N</b>	T	T	<b>T</b>

# Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```

Predict with 1-bit	<b>N</b>	T	T	T	<b>T</b>	<b>N</b>	T	T	T	<b>T</b>
Actual Outcome	T	T	T	T	N	T	T	T	T	N
Predict with 2-bit	<b>N</b>	<b>N</b>	T	T	<b>T</b>	T	T	T	T	<b>T</b>

# Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```

Predict with 1-bit	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T
Actual Outcome	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N
Predict with 2-bit	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T

# Dynamic Prediction

- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {...}
```

Predict with 1-bit	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T
Actual Outcome	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N
Predict with 2-bit	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

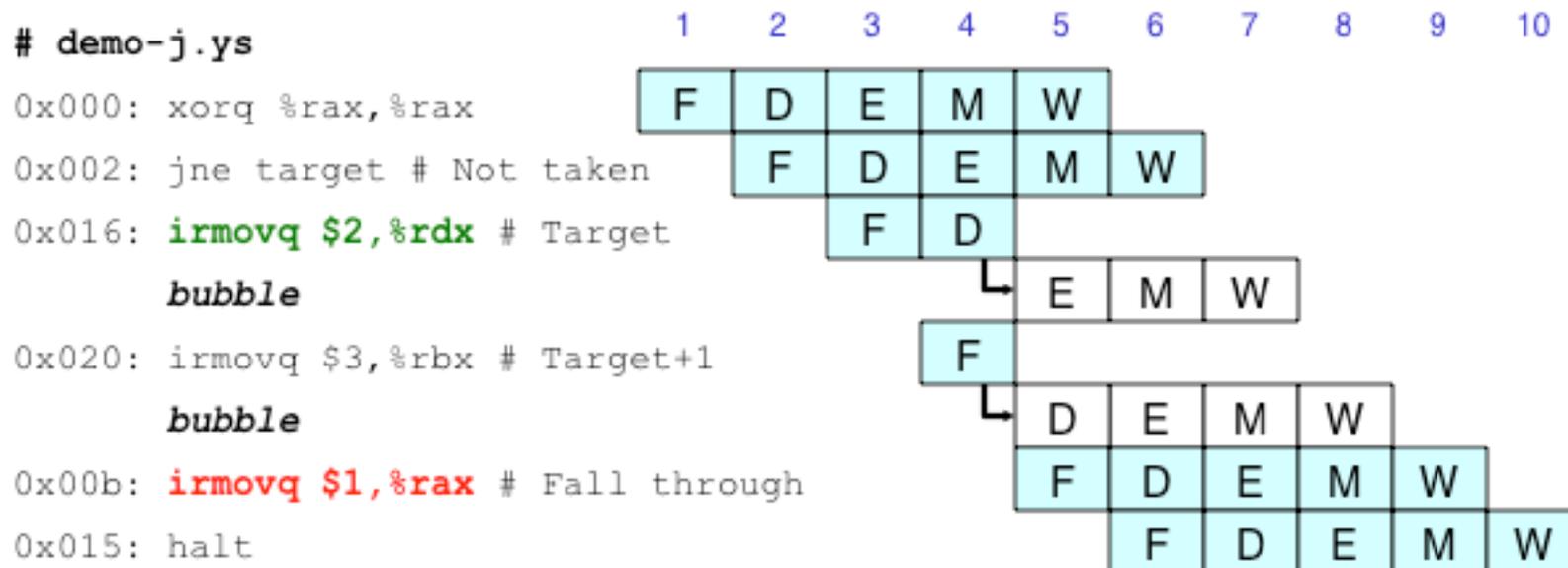
# More Advanced Dynamic Prediction

- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

cond1 branch not-  
taken means  $(x \leq 0)$   
branch taken

```
x = 0
if (cond1) x = 3
if (cond2) y = 19
if (x <= 0) z = 13
```

# What Happens If We Mispredict?



## Cancel instructions when mispredicted

- Assuming we detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by **bubbles**
- No side effects have occurred yet

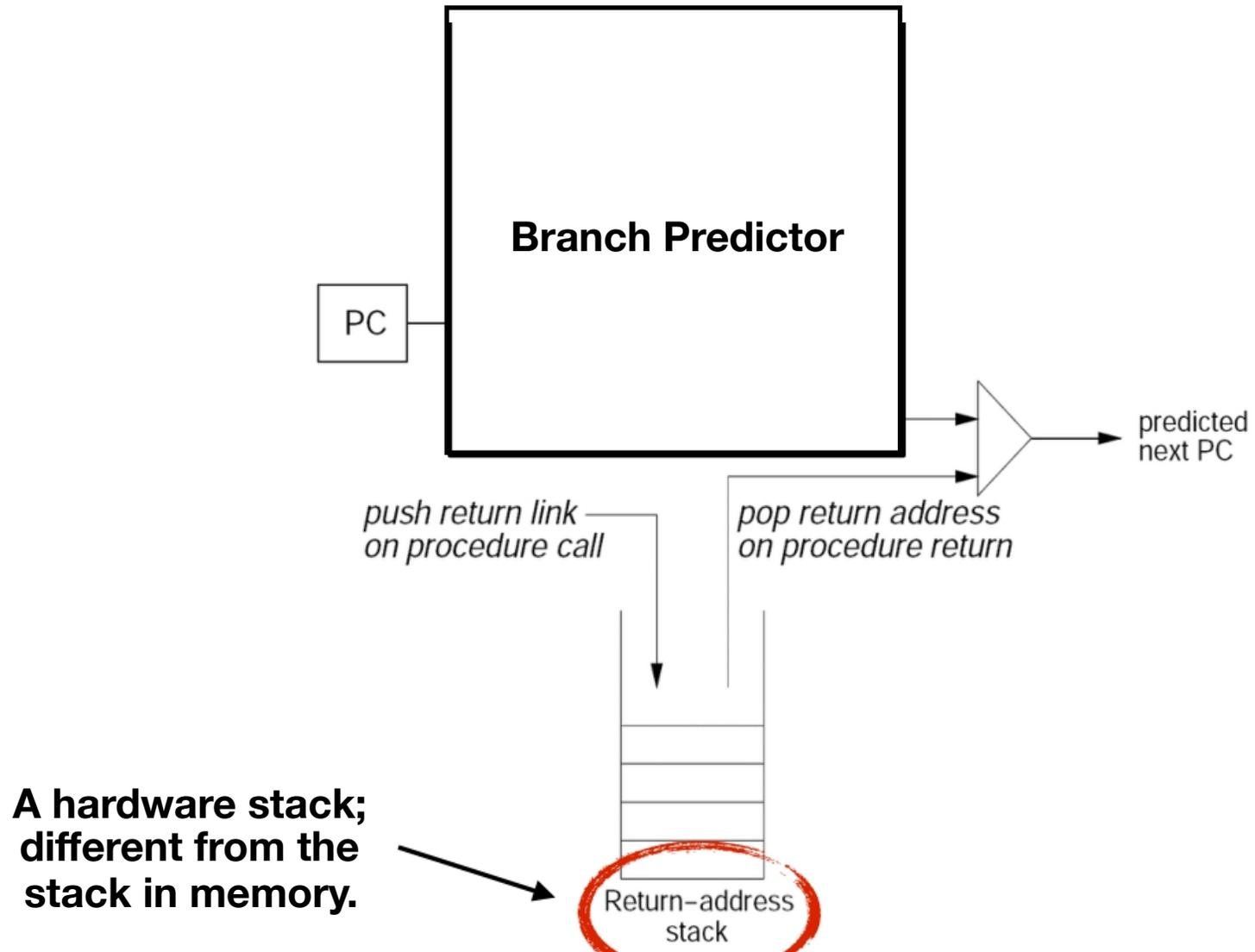
# Stalling for Return

- As `ret` passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage

# Return Address Stack (RAS)

- Stalling for return is silly since we know where exactly we need to jump to, except the jump target is retrieved later in the memory stage.
- Can we get that sooner? Where should we get it?

# Return Address Stack (RAS)



# Today: Making the Pipeline Really Work

- Control Dependencies
  - Inserting Nops
  - Stalling
  - Delay Slots
  - Branch Prediction
- Data Dependencies
  - Inserting Nops
  - Stalling
  - Out-of-order execution

# Data Dependencies

```
1    irmovq $50, %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```

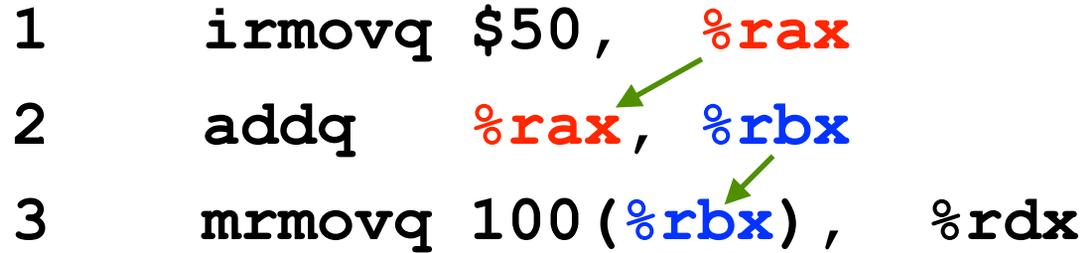
# Data Dependencies

```
1    irmovq $50, %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```



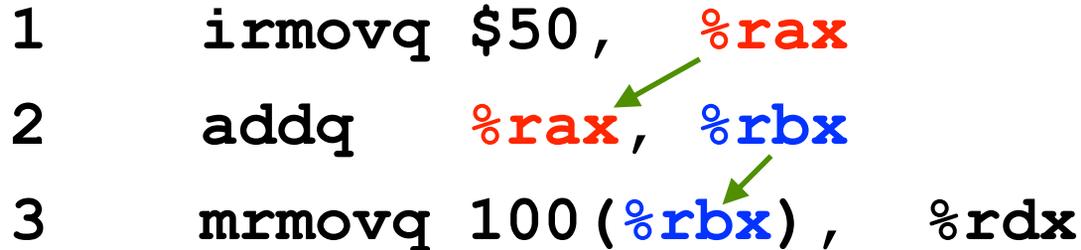
# Data Dependencies

```
1    irmovq $50, %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```



# Data Dependencies

```
1    irmovq $50, %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```



- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

# A Subtle Data Dependency

- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome **until after its Execute stage.**

**Why?**

```
xorg %rax, %rax
jne L1           # Not taken
irmovq $1, %rax # Fall Through
L1  irmovq $4, %rcx # Target
    irmovq $3, %rax # Target + 1
```

# A Subtle Data Dependency

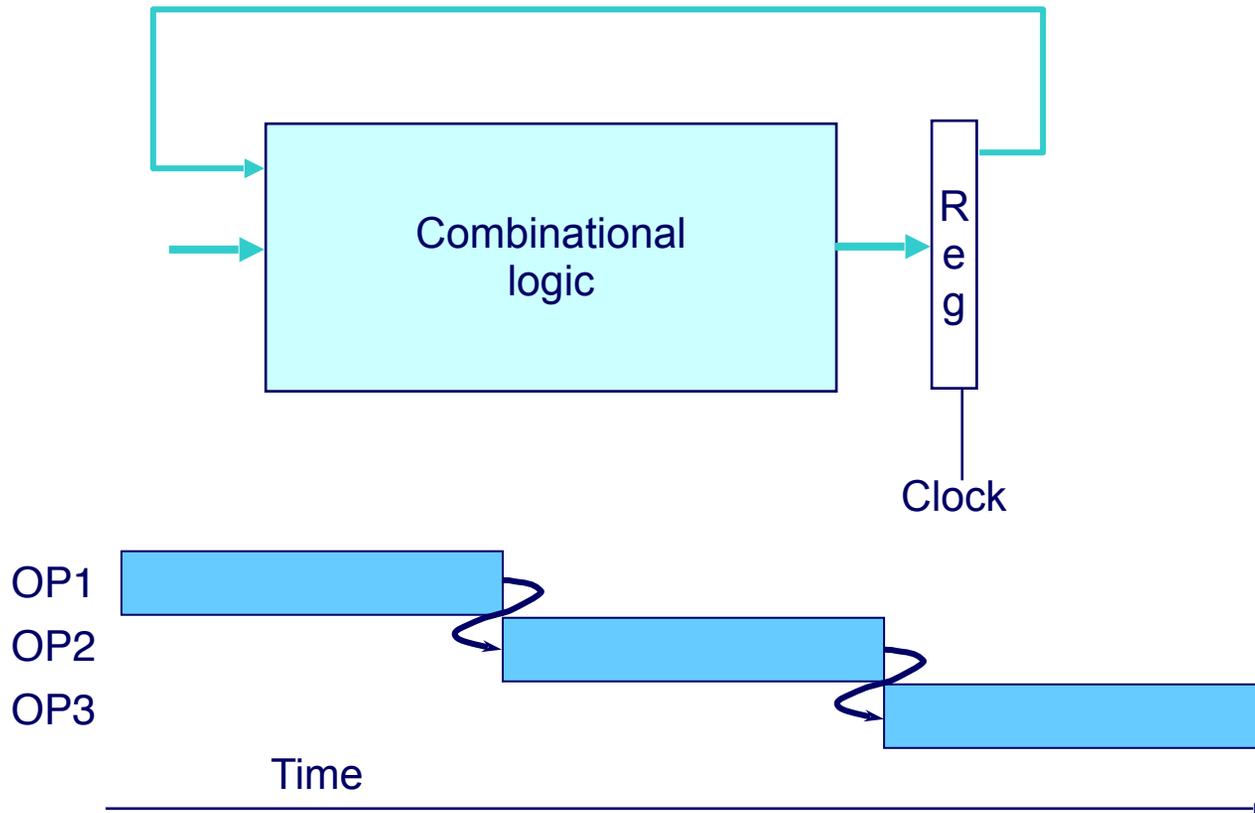
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome **until after its Execute stage.**

## Why?

- There is a data dependency between `xorg` and `jne`. The “data” is the status flags.

```
xorg %rax, %rax
jne L1           # Not taken
irmovq $1, %rax # Fall Through
L1  irmovq $4, %rcx # Target
    irmovq $3, %rax # Target + 1
```

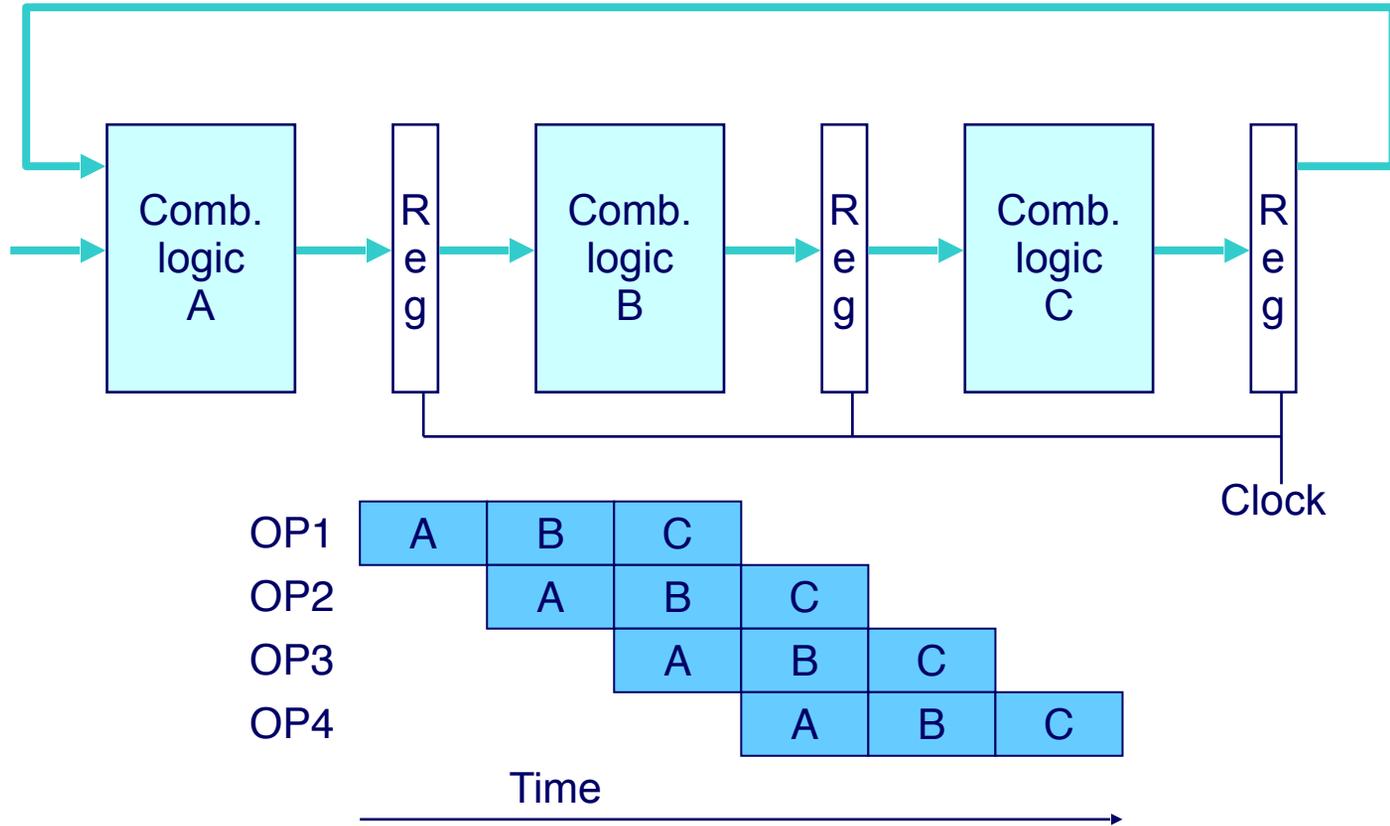
# Data Dependencies in Single-Cycle Machines



## In Single-Cycle Implementation:

- Each operation starts only after the previous operation finishes. Dependency always satisfied.

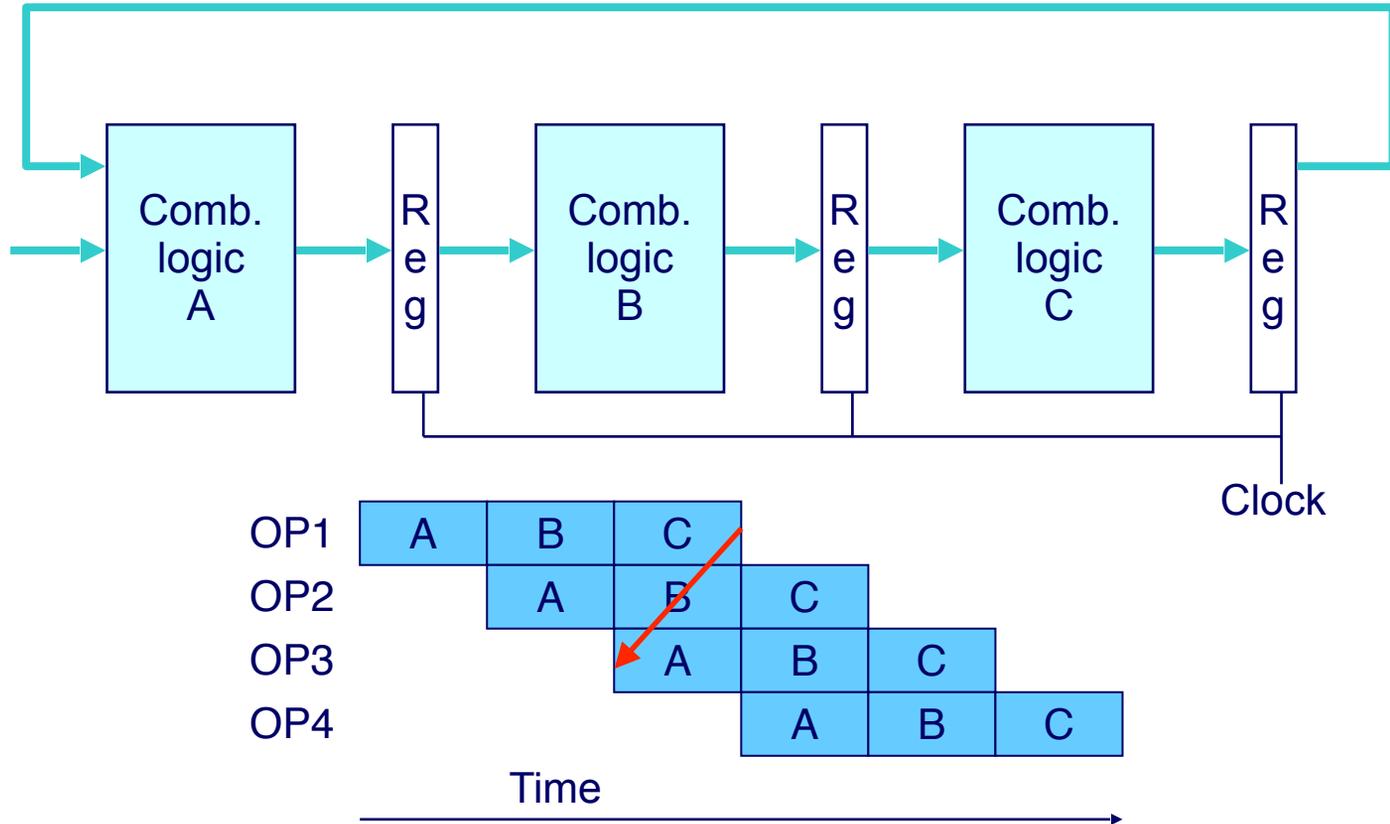
# Data Dependencies in Pipeline Machines



Data Hazards happen when:

- Result does not feed back around in time for next operation

# Data Dependencies in Pipeline Machines



Data Hazards happen when:

- Result does not feed back around in time for next operation

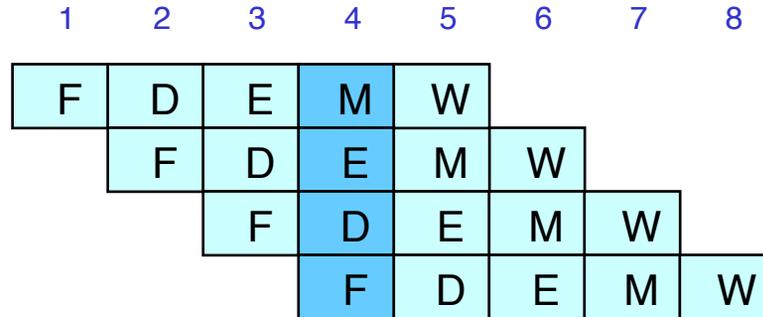
# Data Dependencies: No Nop

0x000: `irmovq $10,%rdx`

0x00a: `irmovq $3,%rax`

0x014: `addq %rdx,%rax`

0x016: `halt`



Remember registers get updated in the Write-back stage

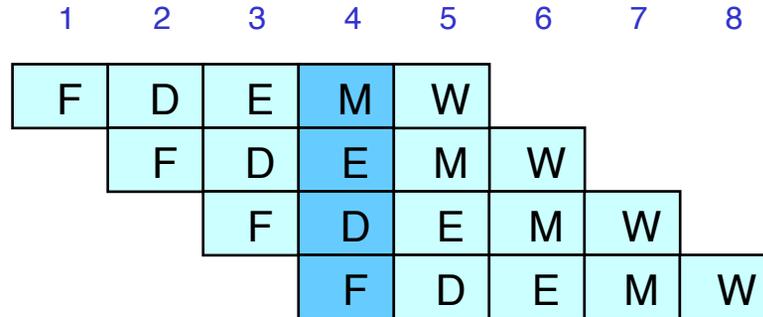
# Data Dependencies: No Nop

0x000: `irmovq $10,%rdx`

0x00a: `irmovq $3,%rax`

0x014: `addq %rdx,%rax`

0x016: `halt`

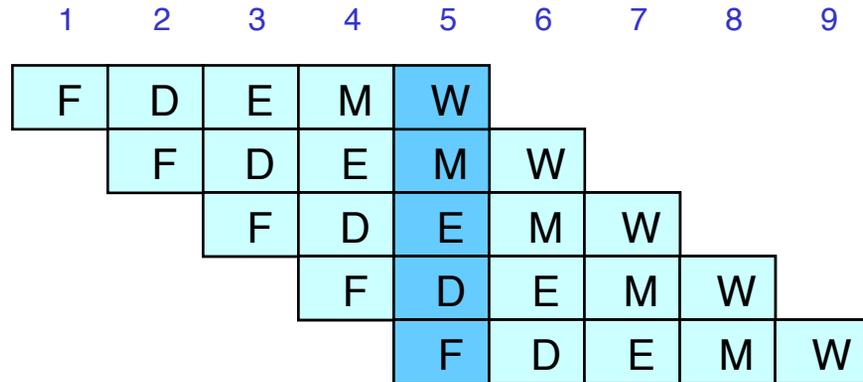


Remember registers get updated in the Write-back stage

**addq reads wrong %rdx and %rax**

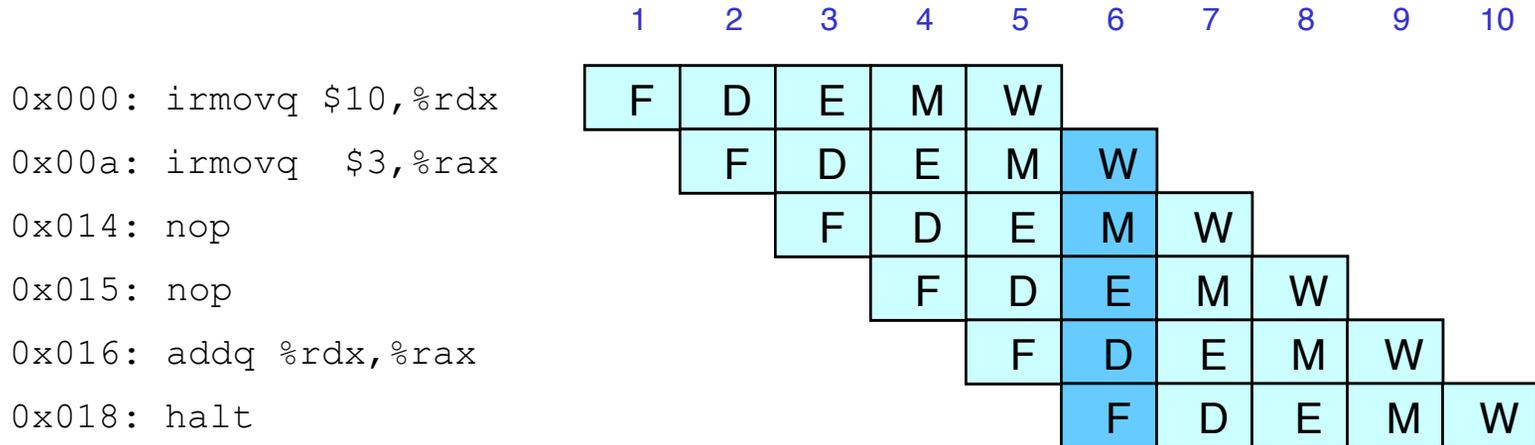
# Data Dependencies: 1 Nop

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



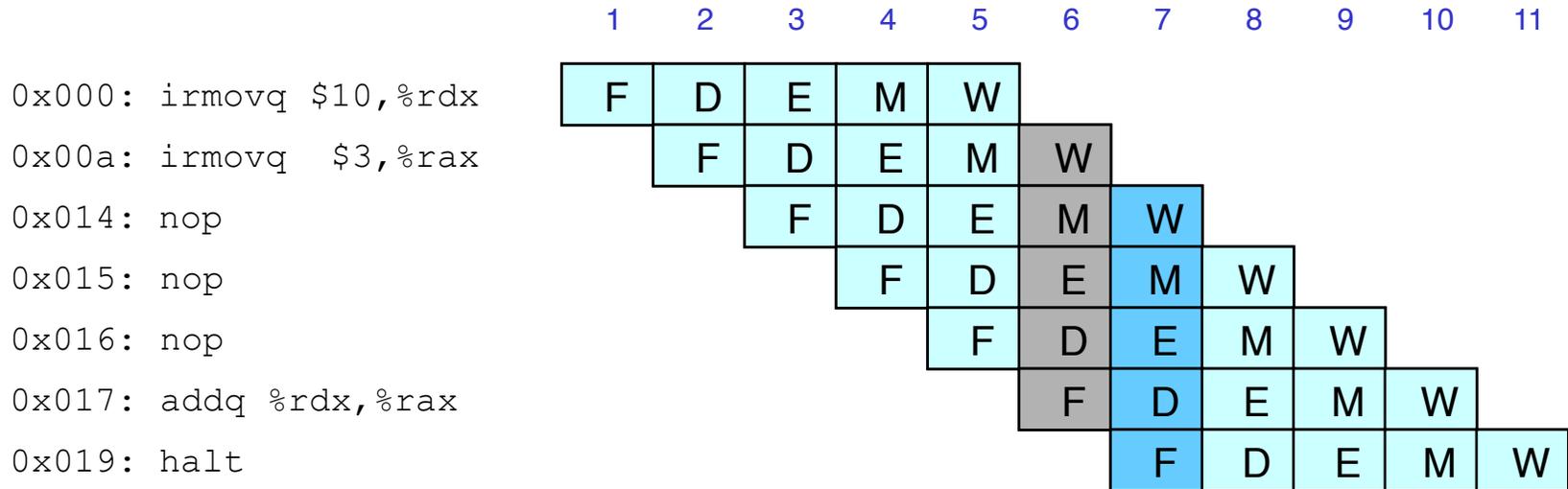
**addq still reads wrong %rdx and %rax**

# Data Dependencies: 2 Nop's



**addq reads the correct %rdx,  
but %rax still wrong**

# Data Dependencies: 3 Nop's



**addq reads the correct %rdx  
and %rax**

# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?

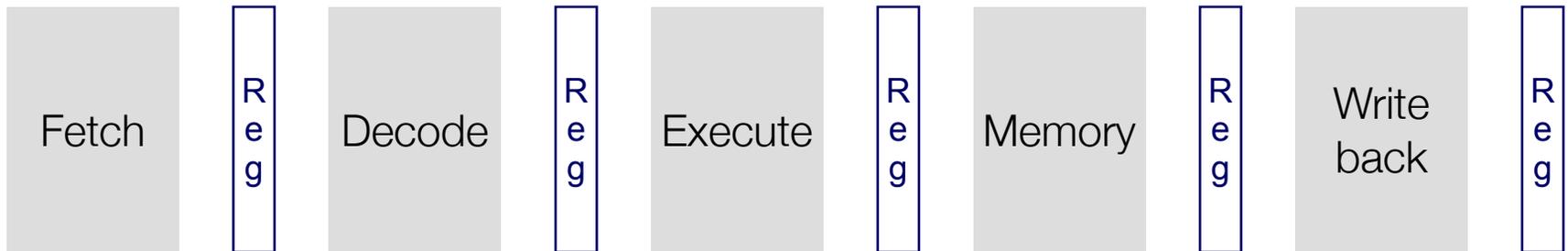


# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?

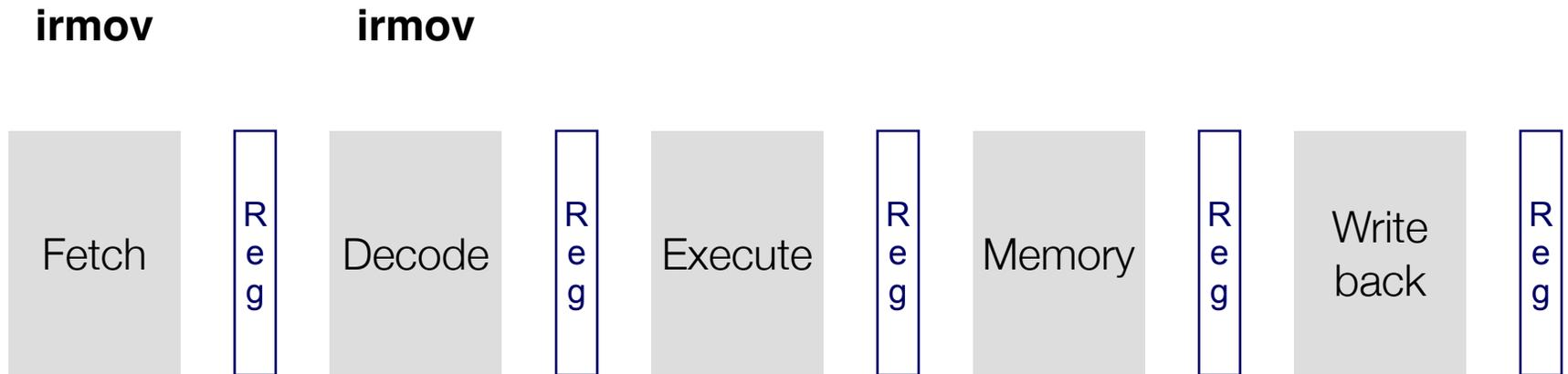
**irmov**



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

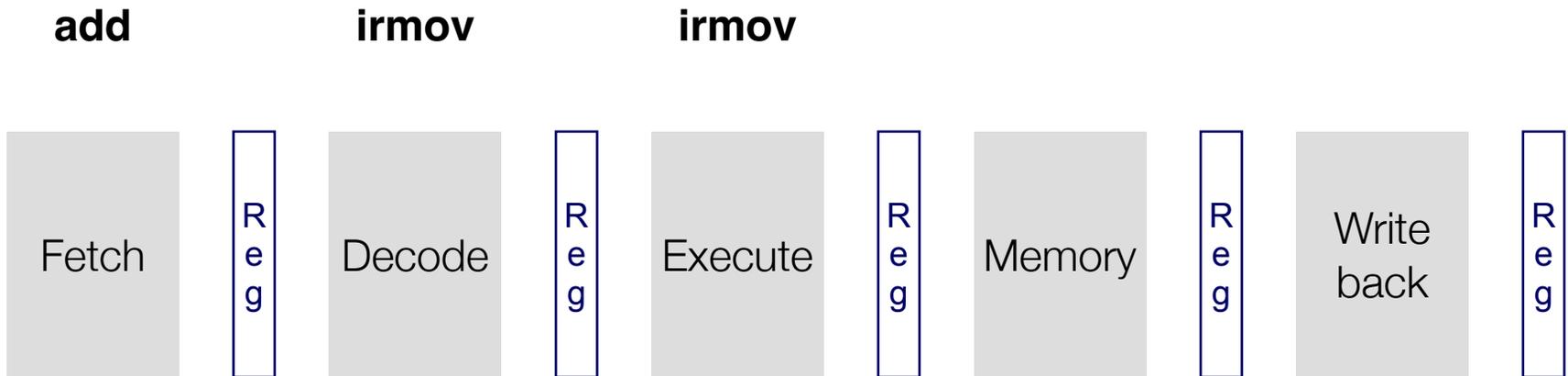
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

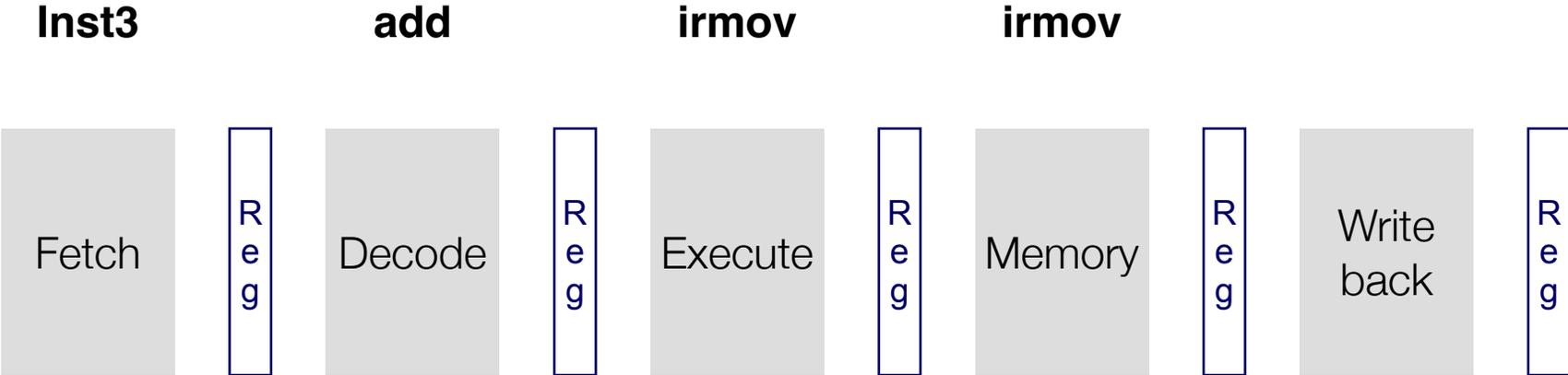
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

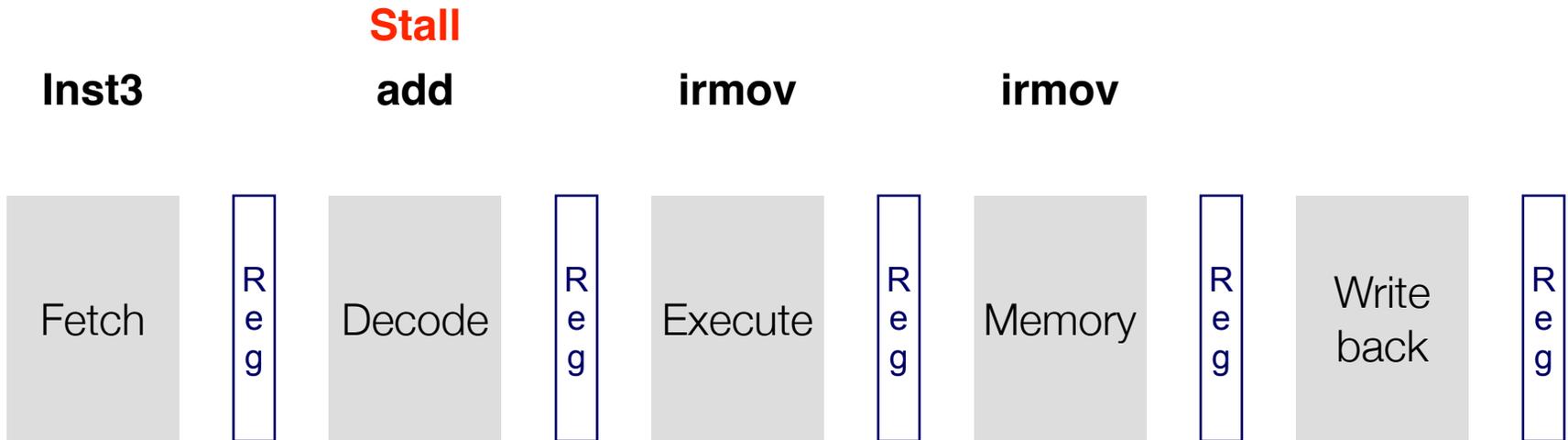
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

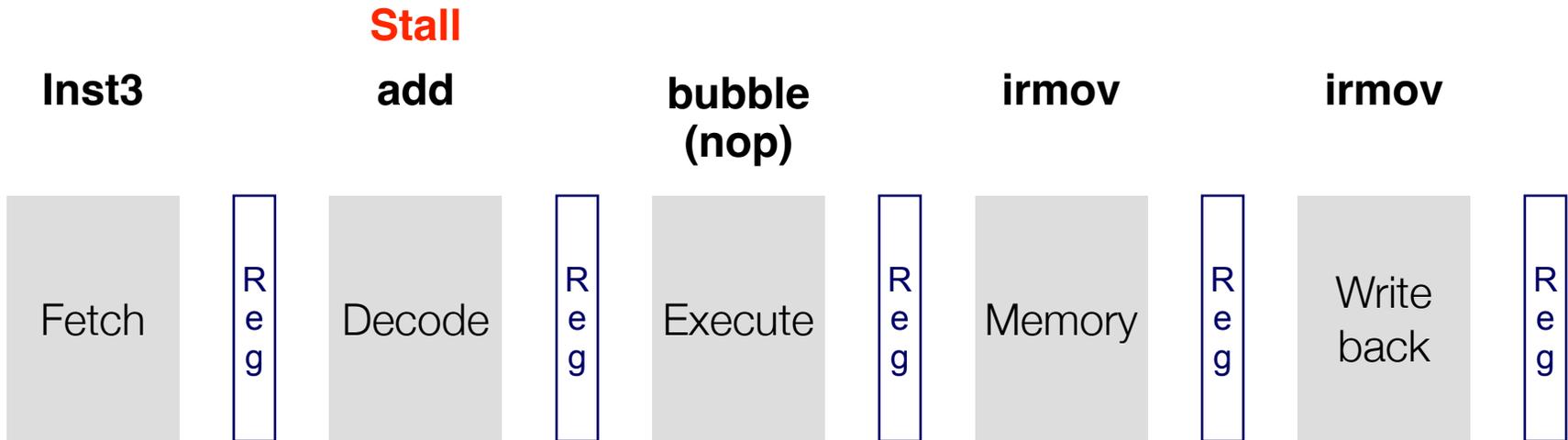
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

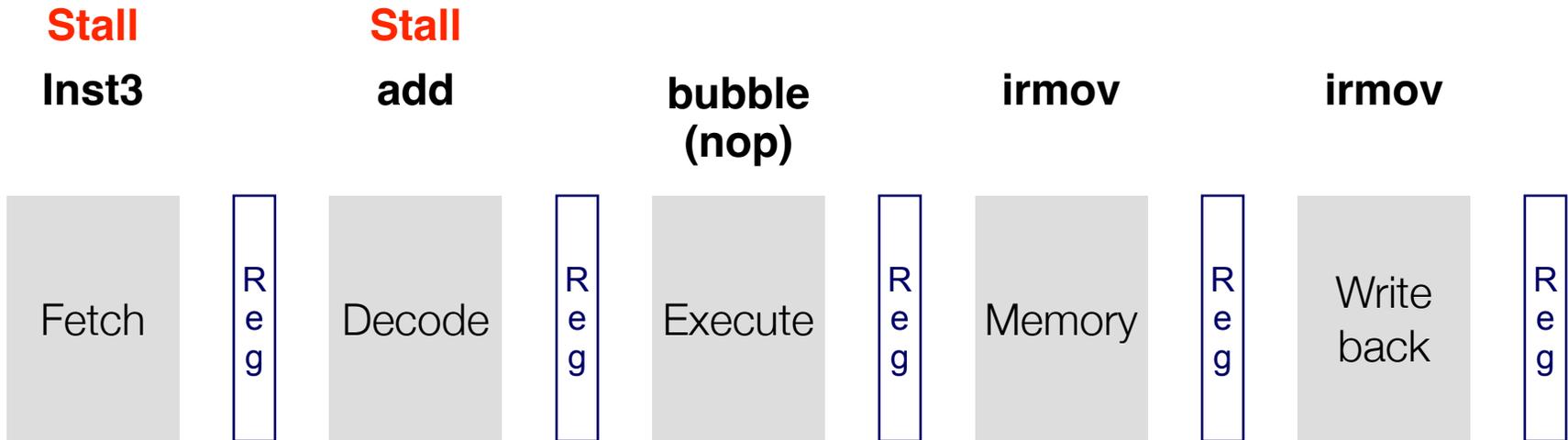
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

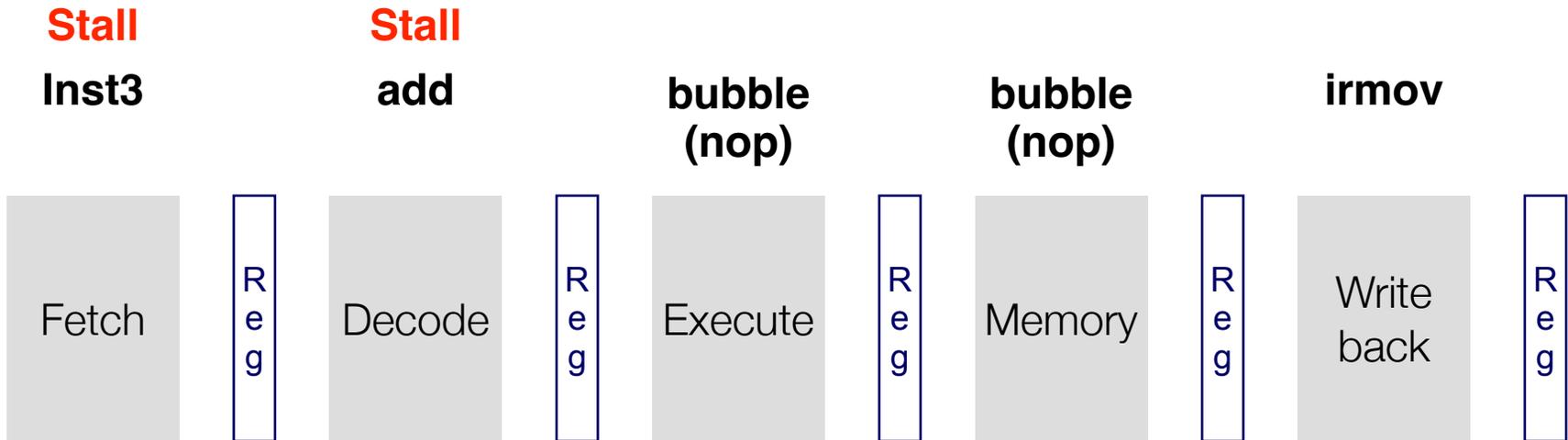
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

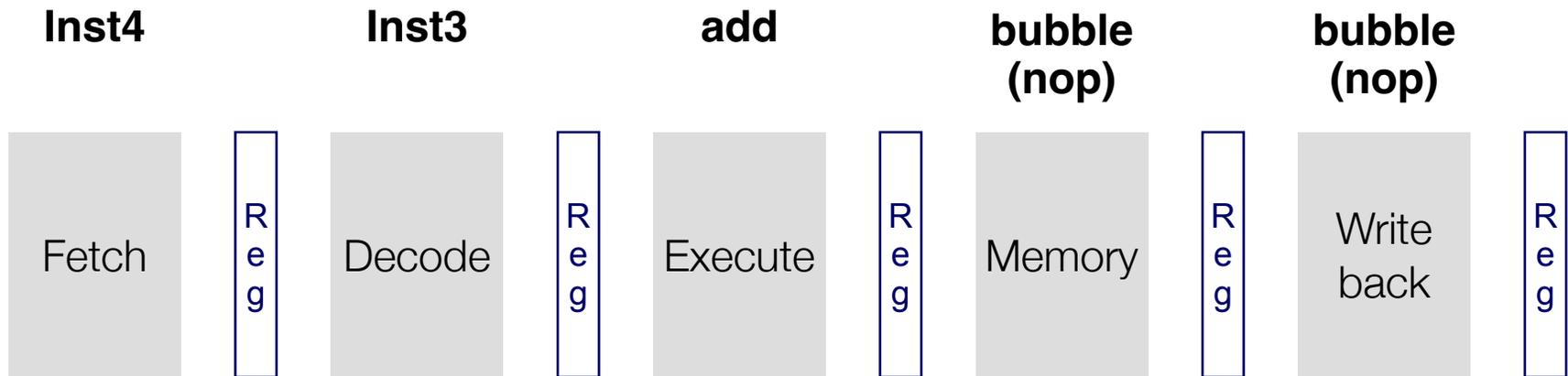
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

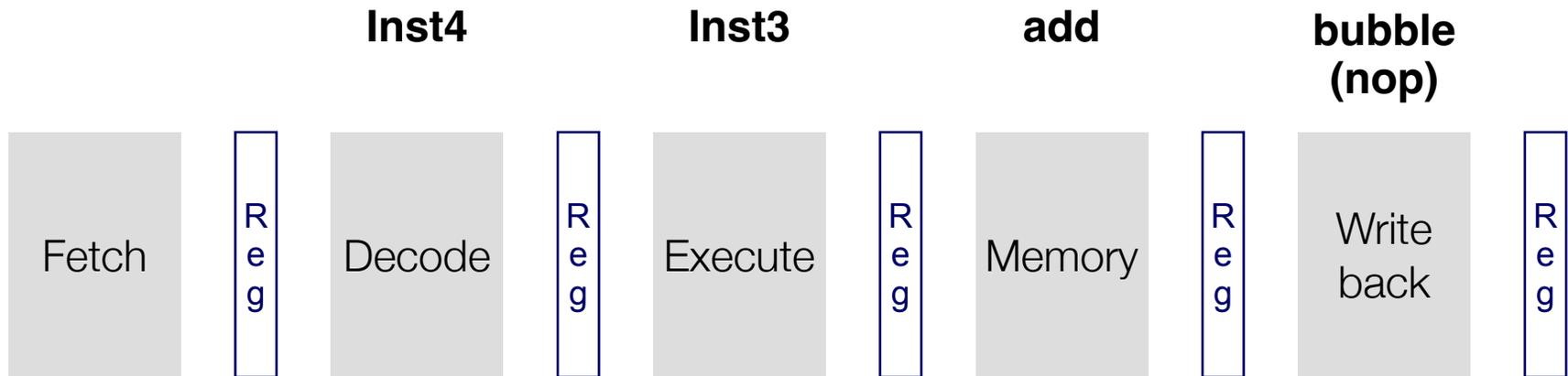
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

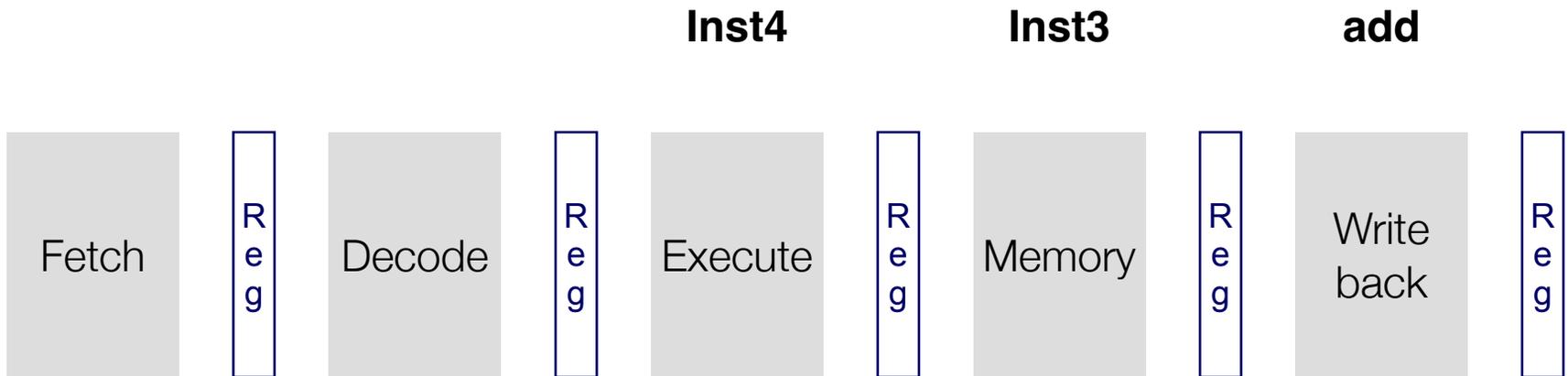
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

- Why is it good for the hardware to do so anyways?



# Detecting Stall Condition

- Using a “**scoreboard**”. Each register has a bit.
- Every instruction that writes to a register sets the bit.
- Every instruction that reads a register would have to check the bit first.
  - If the bit is set, then generate a bubble
  - Otherwise, free to go!!

# Data Forwarding

## Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
- The decode stage can't start until the write-back stage finishes

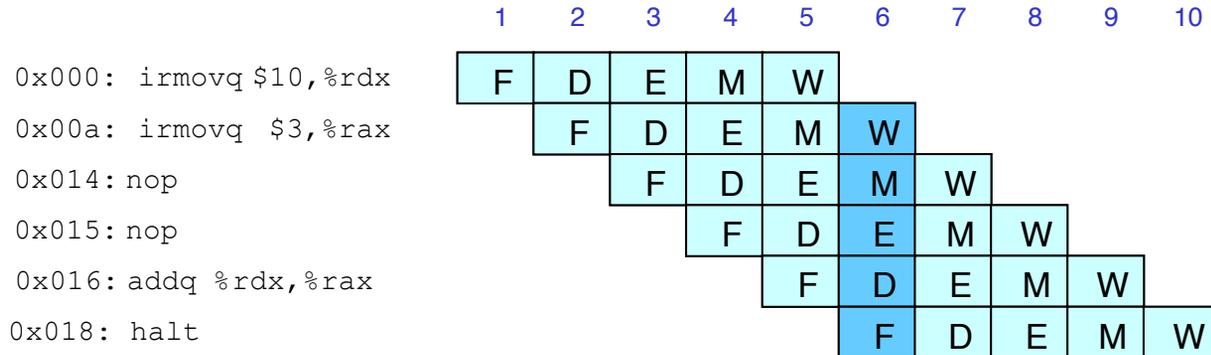
## Observation

- Value generated in execute or memory stage

## Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

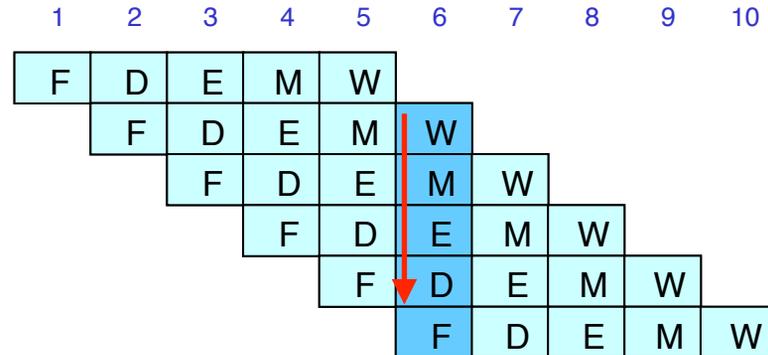
# Data Forwarding Example



- `irmovq` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

# Data Forwarding Example

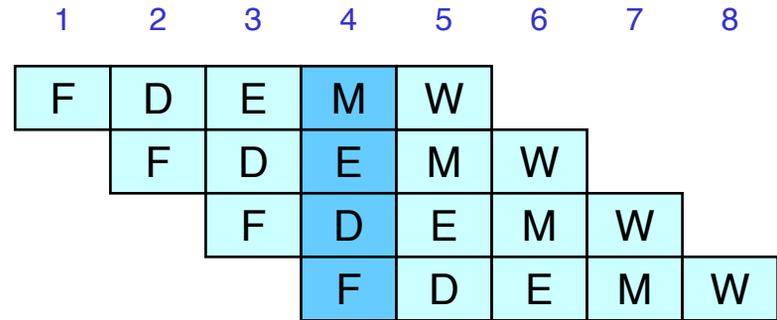
```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `irmovq` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

# Data Forwarding Example #2

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Register `%rdx`

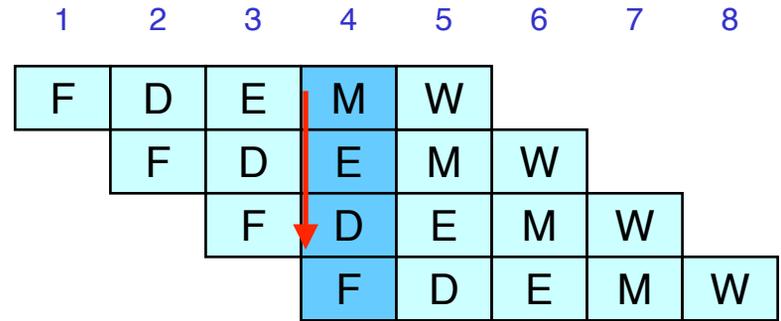
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

# Data Forwarding Example #2

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Register `%rdx`

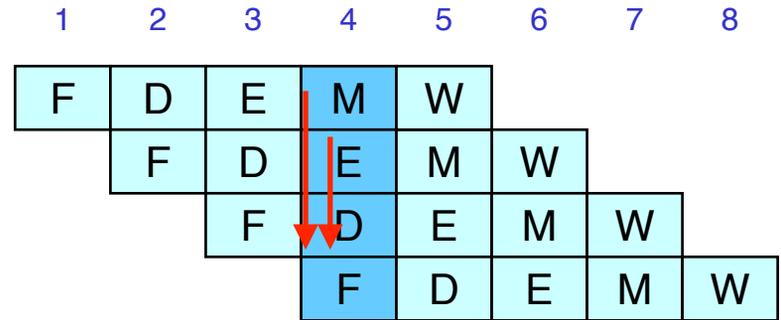
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

# Data Forwarding Example #2

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Register `%rdx`

- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

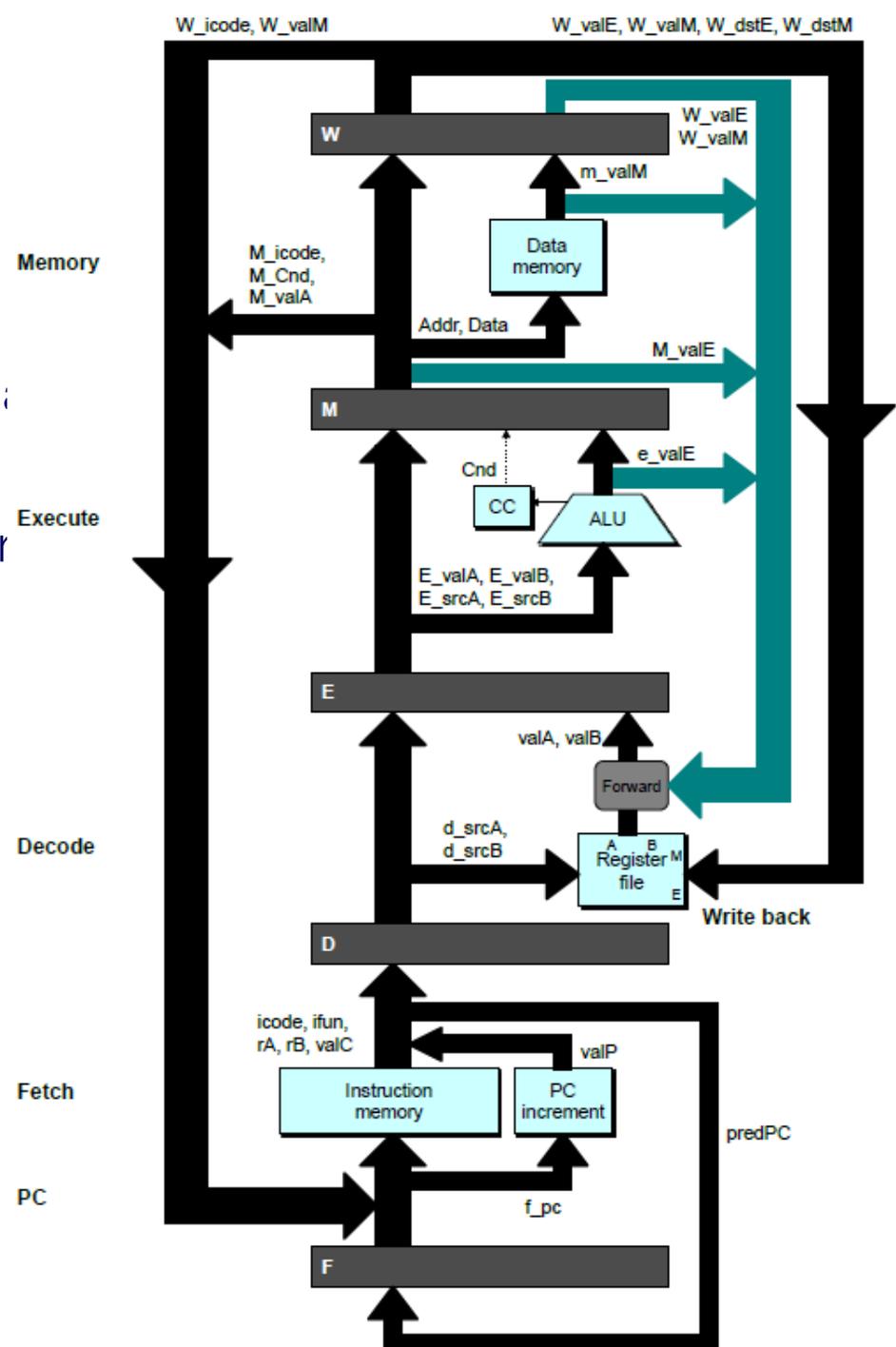
# Bypass Paths

## Decode Stage

- Forwarding logic selects valA :
- Normally from register file
- Forwarding: get valA or valB fr

## Forwarding Sources

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



# Out-of-order Execution

- Compiler could do this, but has limitations
- Generally done in hardware

Long-latency instruction.  
Forces the pipeline to stall.

**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r4 = **r3** + r6  
r7 = r5 + r1  
...



**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r7 = r5 + r1  
...  
r4 = **r3** + r6

# Out-of-order Execution

- Compiler could do this, but has limitations
- Generally done in hardware

Long-latency instruction.  
Forces the pipeline to stall.

**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r4 = **r3** + r6  
r7 = r5 + r1  
...

**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r7 = r5 + r1  
...  
r4 = **r3** + r6

# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r6 = r5 + r1
...
r4 = r3 + r6
```

# Out-of-order Execution

```
r0 = r1 + r2  
r3 = MEM[r0]  
r4 = r3 + r6  
r6 = r5 + r1  
...
```

Is this correct?



```
r0 = r1 + r2  
r3 = MEM[r0]  
r6 = r5 + r1  
...  
r4 = r3 + r6
```

```
r0 = r1 + r2  
r3 = MEM[r0]  
r4 = r3 + r6  
r4 = r5 + r1  
...
```

# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r6 = r5 + r1
...
r4 = r3 + r6
```

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r4 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r5 + r1
...
r4 = r3 + r6
```

# Out-of-order Execution

$r_0 = r_1 + r_2$   
 $r_3 = \text{MEM}[r_0]$   
 $r_4 = r_3 + \mathbf{r_6}$   
 $\mathbf{r_6} = r_5 + r_1$   
...

Is this correct?



$r_0 = r_1 + r_2$   
 $r_3 = \text{MEM}[r_0]$   
 $\mathbf{r_6} = r_5 + r_1$   
...  
 $r_4 = r_3 + \mathbf{r_6}$

$r_0 = r_1 + r_2$   
 $r_3 = \text{MEM}[r_0]$   
 $\mathbf{r_4} = r_3 + r_6$   
 $\mathbf{r_4} = r_5 + r_1$   
...

Is this correct?



$r_0 = r_1 + r_2$   
 $r_3 = \text{MEM}[r_0]$   
 $\mathbf{r_4} = r_5 + r_1$   
...  
 $\mathbf{r_4} = r_3 + r_6$

“**Tomasolu Algorithm**” is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.