CSC 252: Computer Organization
Spring 2022: Lecture 14

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Programming assignment 3 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2022/labs/assignment3.html
  - Due on **March 3**, 11:59 PM
  - You (may still) have 3 slip days

13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	Mar 1 **Today**	2	3 **Due Mid-term**	4	5

Announcements

- Grades for Lab 2 are posted.

- Programming assignment 3 is in x86 assembly language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

- Mid-term exam: March 3; online.
- Past exam & Problem set: https://www.cs.rochester.edu/courses/252/spring2022/handouts.html
- Exam will be electronic using Gradescope.

Announcements

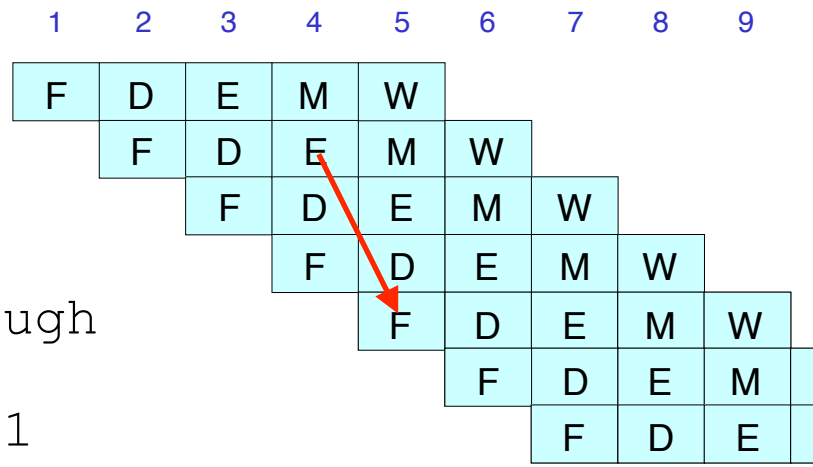
Announcements

- Open book test: any sort of paper-based product, e.g., book, **notes**, magazine, old tests.
- Exams are designed to test your ability to apply what you have learned and not your memory (though a good memory could help).
- **Nothing electronic (including laptop, cell phone, calculator, etc) other than the computer you use to take the exam**.
- **Nothing biological**, including your roommate, husband, wife, your hamster, another professor, etc.
- "**I don't know**" gets15% partial credit. Must erase everything else.

Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- 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

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



Resolving Control Dependencies

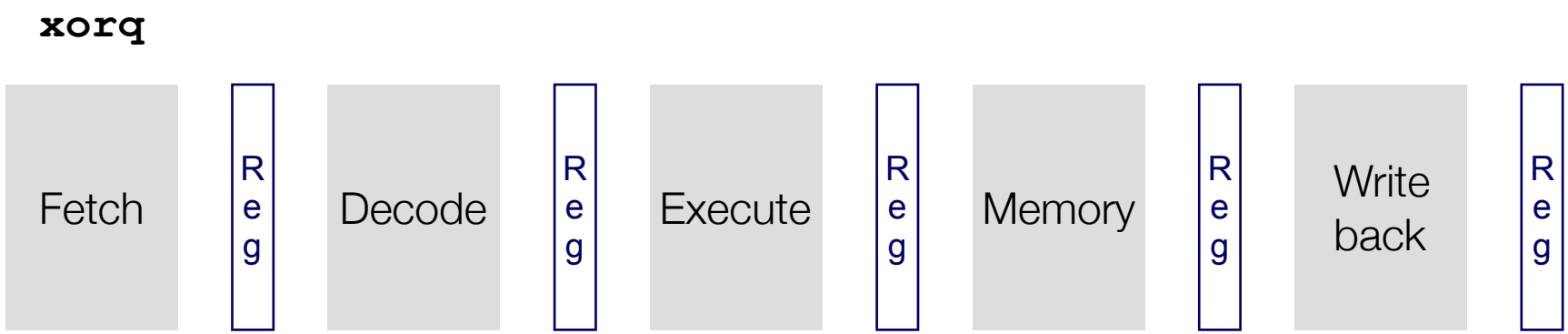
- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
  - Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach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

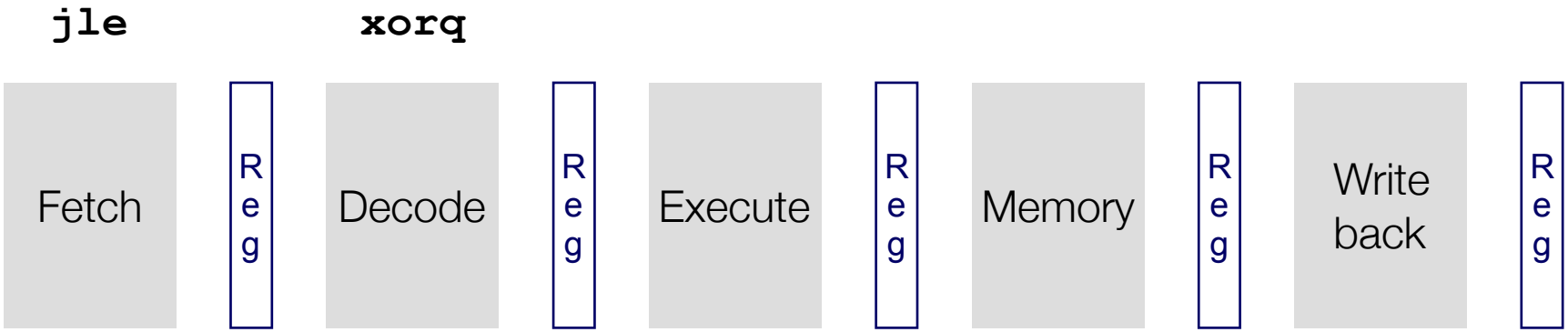
Hardware Generated Nops (Bubble and Stalling)

- **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?



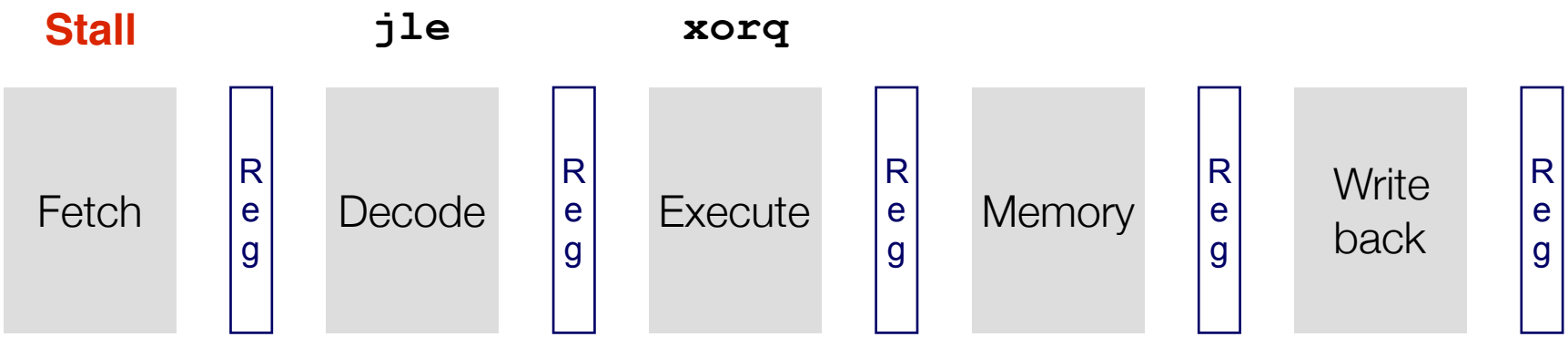
Hardware Generated Nops (Bubble and Stalling)

- **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?



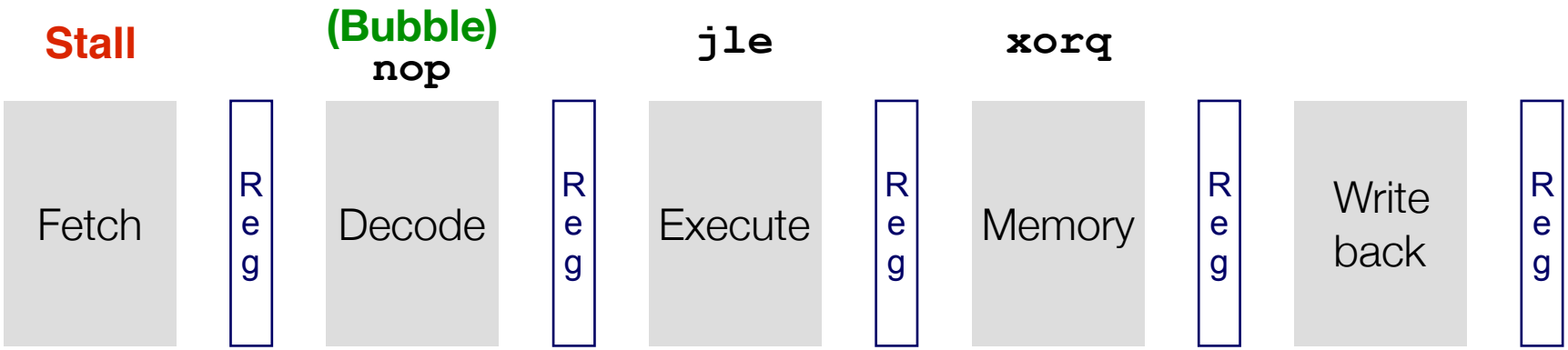
Hardware Generated Nops (Bubble and Stalling)

- **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?



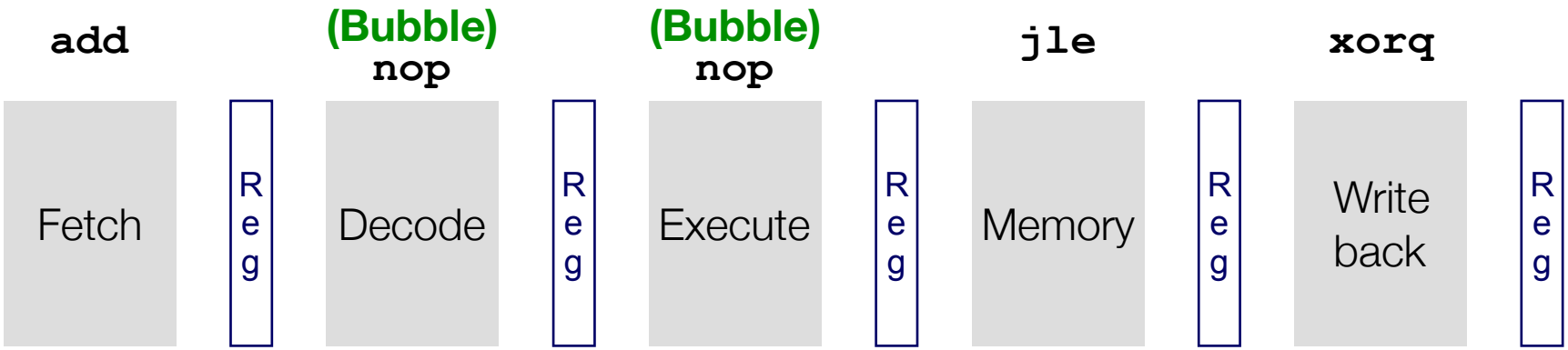
Hardware Generated Nops (Bubble and Stalling)

- **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?



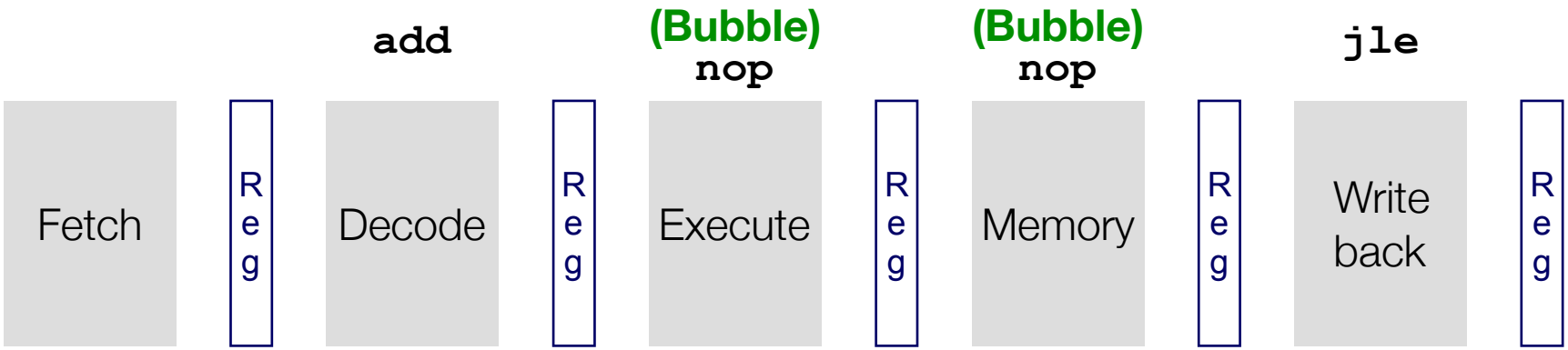
Hardware Generated Nops (Bubble and Stalling)

- **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?



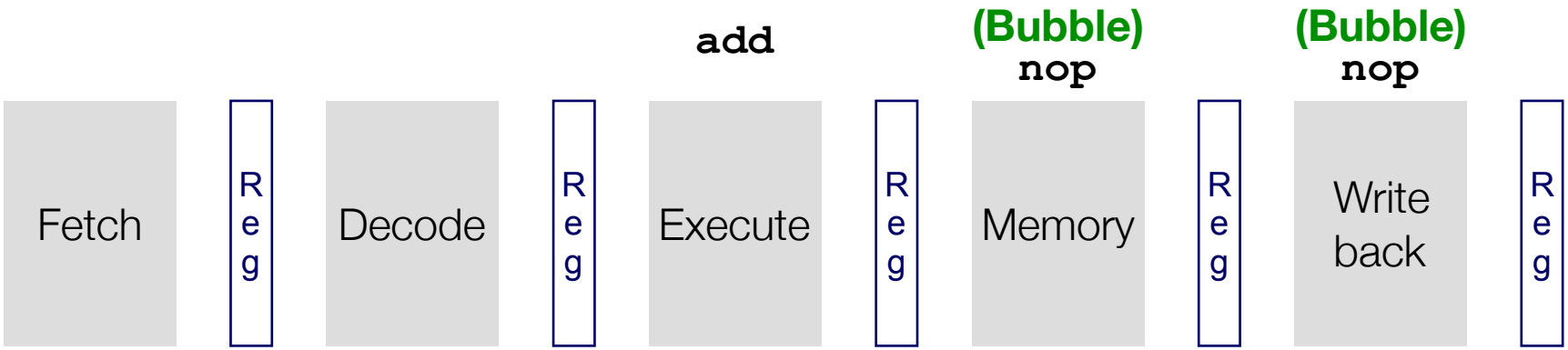
Hardware Generated Nops (Bubble and Stalling)

- **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?



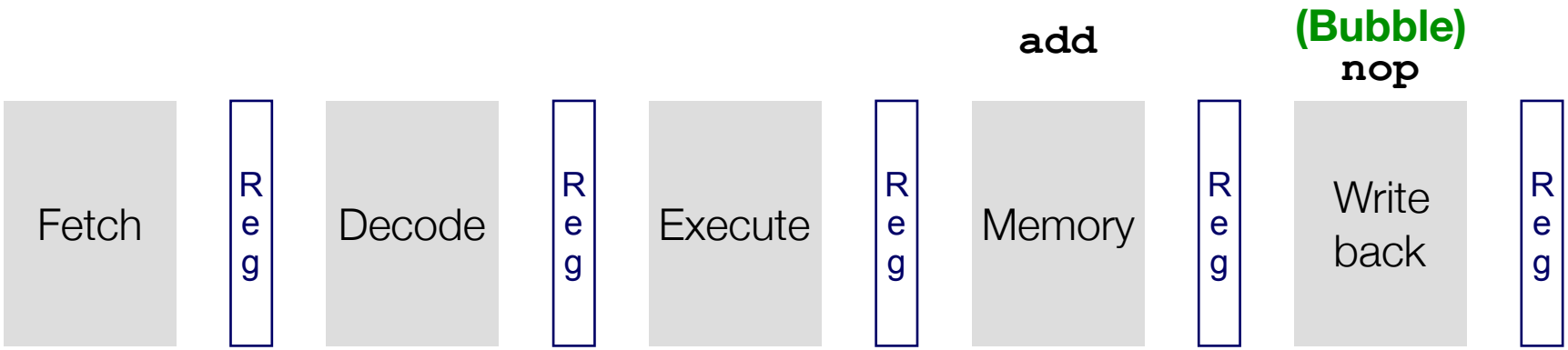
Hardware Generated Nops (Bubble and Stalling)

- **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?



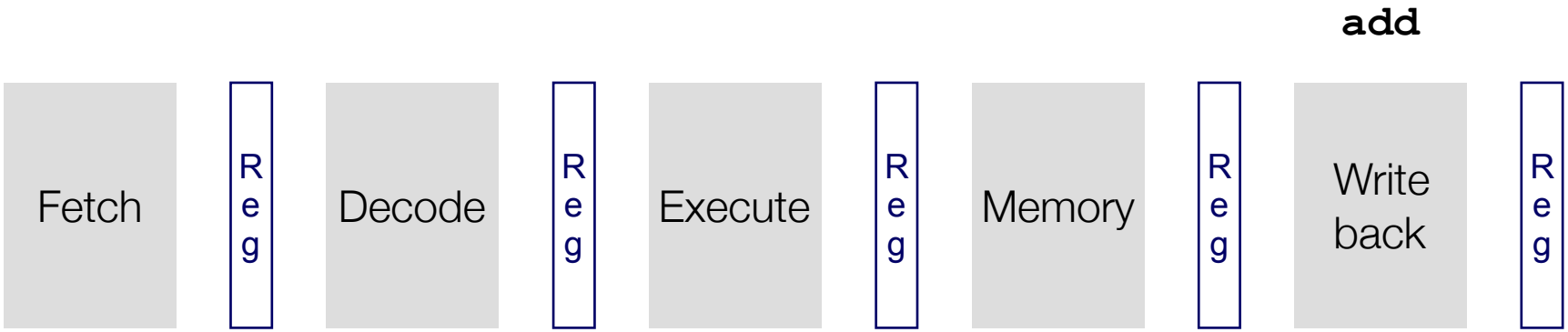
Hardware Generated Nops (Bubble and Stalling)

- **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?



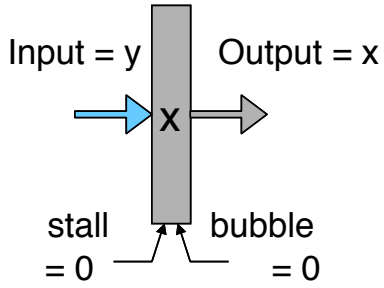
Hardware Generated Nops (Bubble and Stalling)

- **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?

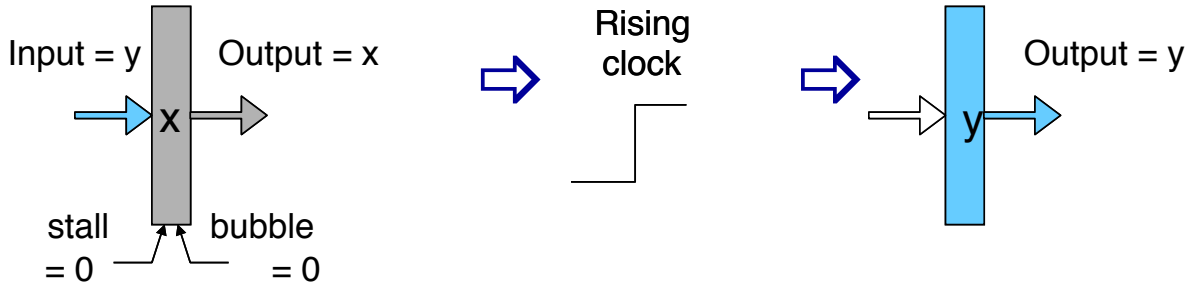
How are Stall and Bubble Implemented in Hardware?

Normal



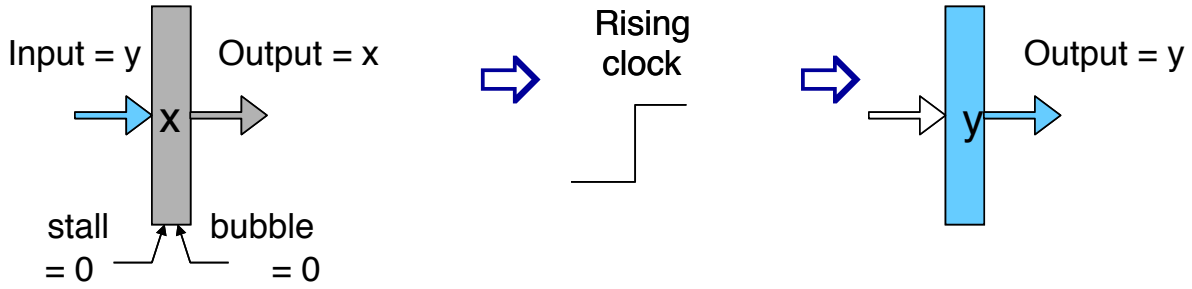
How are Stall and Bubble Implemented in Hardware?

Normal

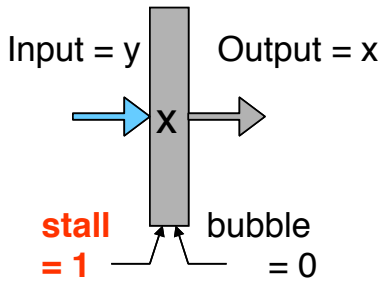


How are Stall and Bubble Implemented in Hardware?

Normal

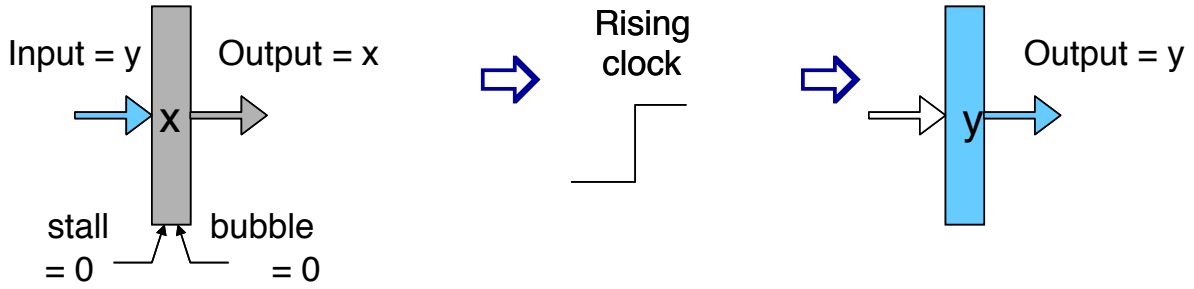


Stall

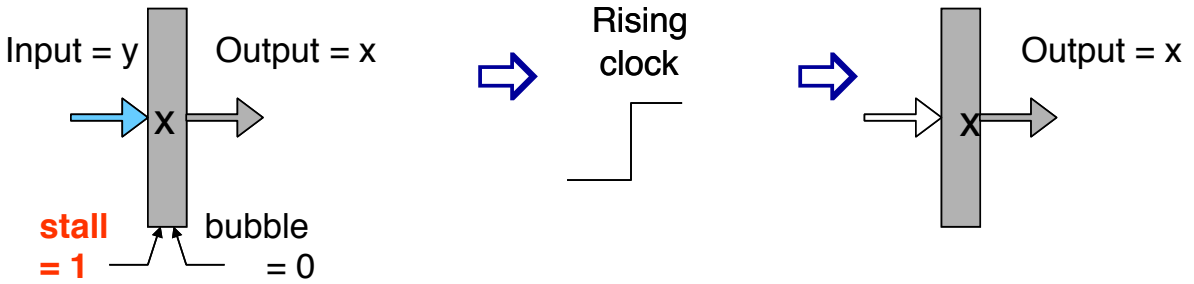


How are Stall and Bubble Implemented in Hardware?

Normal

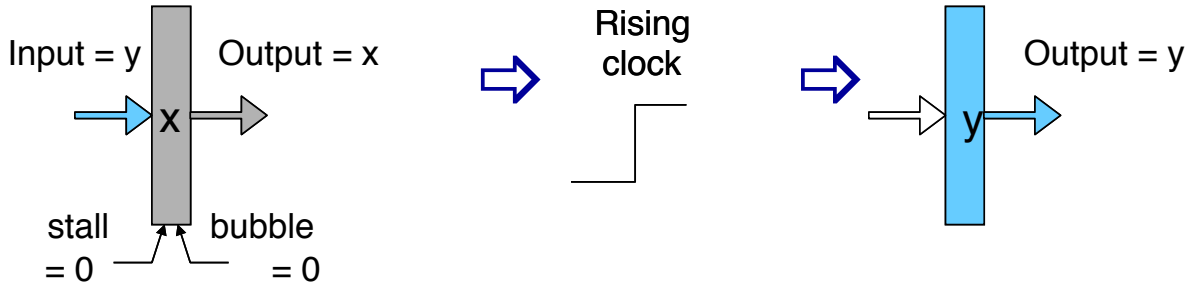


Stall

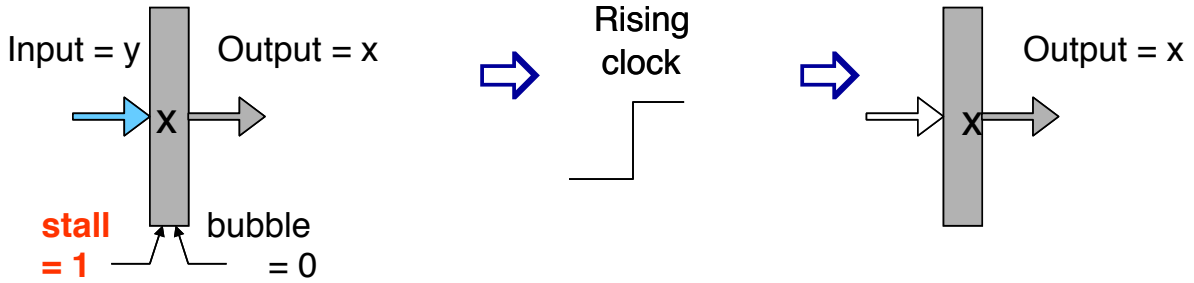


How are Stall and Bubble Implemented in Hardware?

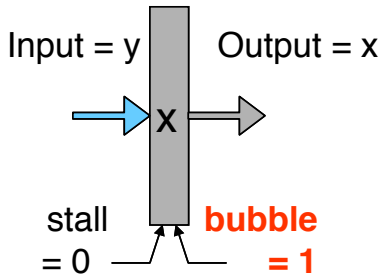
Normal



Stall

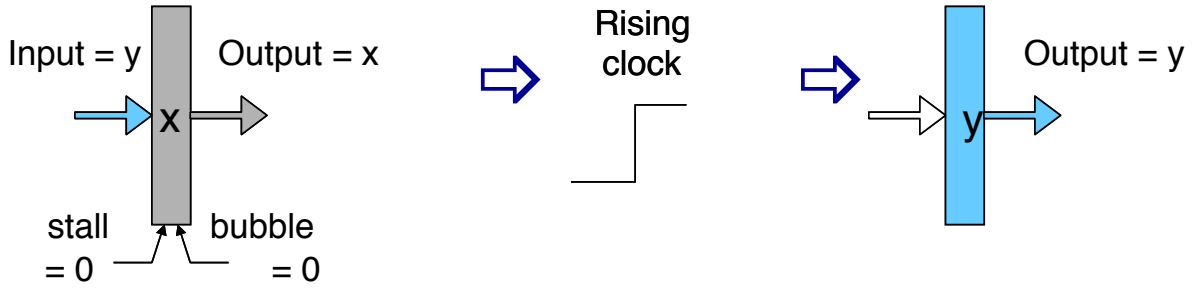


Bubble

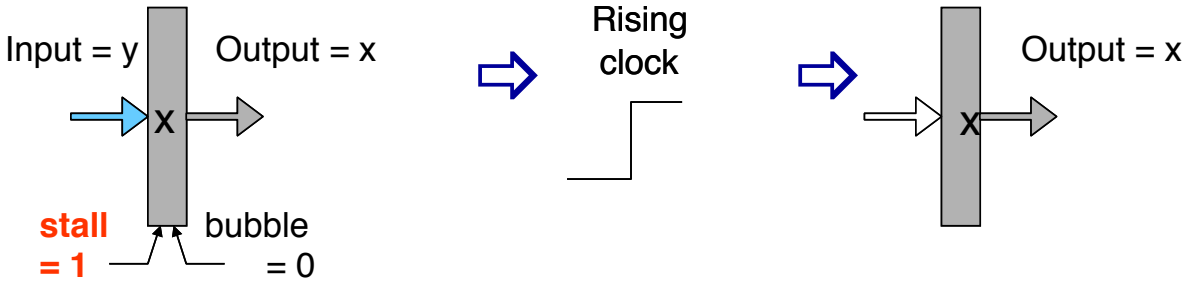


How are Stall and Bubble Implemented in Hardware?

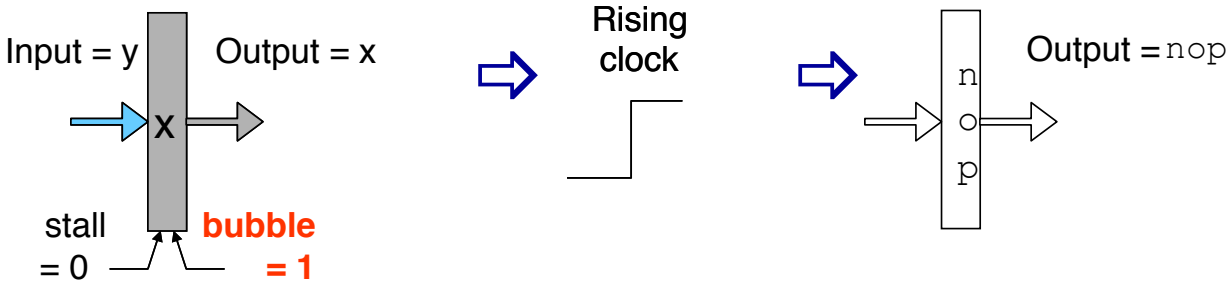
Normal



Stall



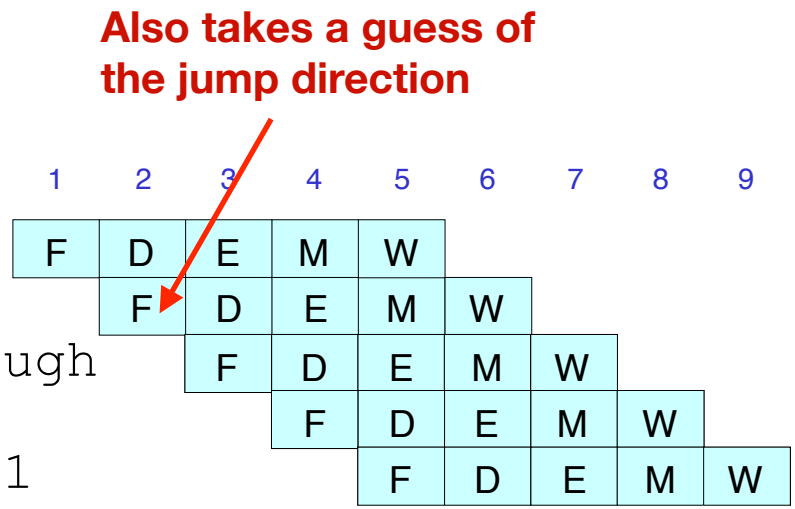
Bubble



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
```

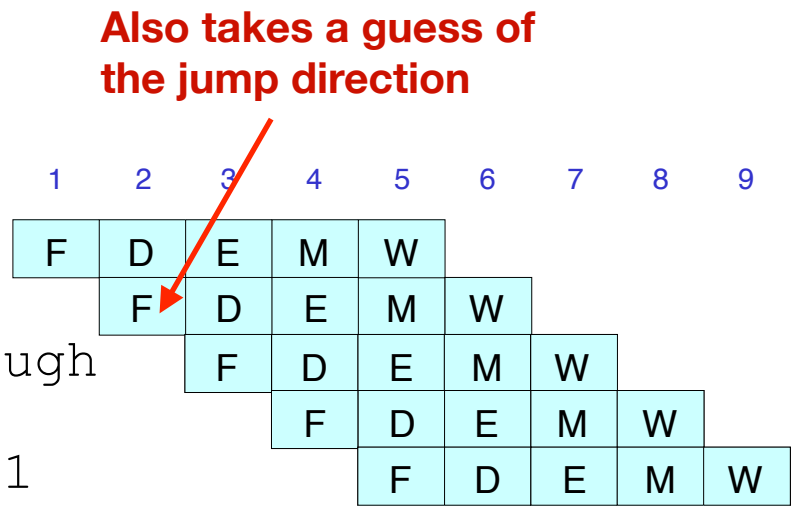


Branch Prediction

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

If prediction is correct: pipeline moves forward without stalling

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



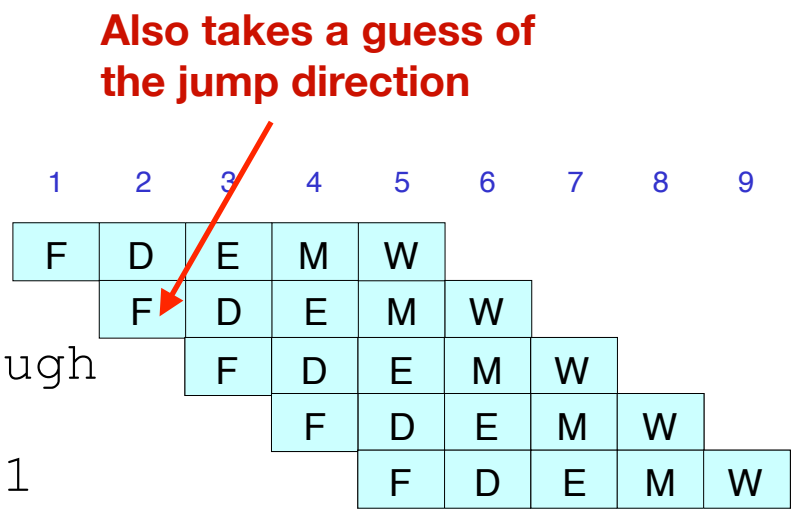
Branch Prediction

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

```
     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.

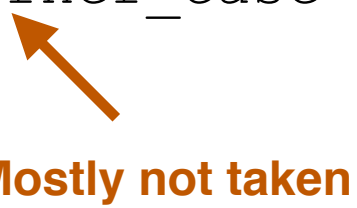
```
    cmpq     %rsi,%rdi
    jle      .corner_case
    <do_A>
.corner_case:
    <do_B>
    ret
```

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
```



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>
```

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
    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

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

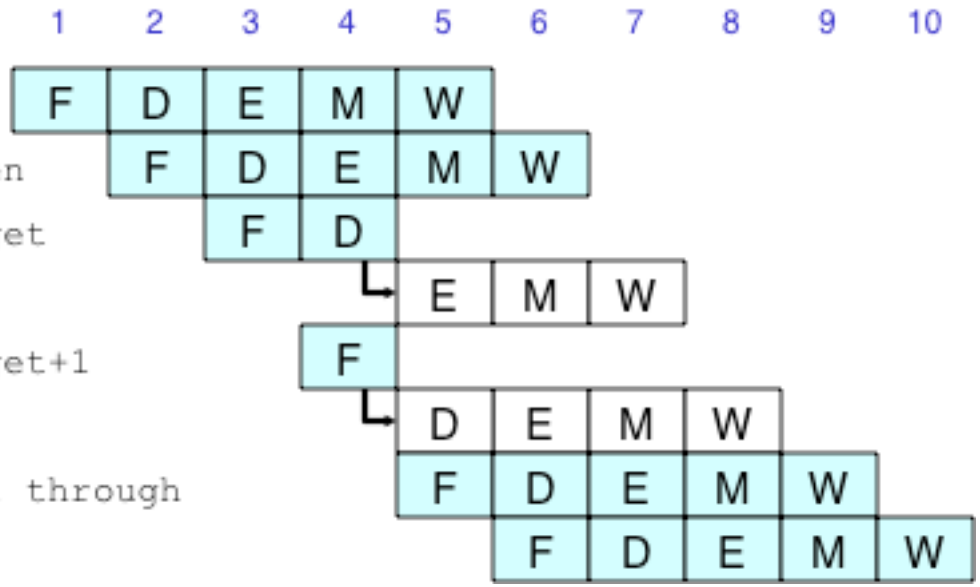
- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

`cond1` branch not-taken means `(x <=0)` branch taken

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

What Happens If We Mispredict?

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



Cancel instructions when mispredicted

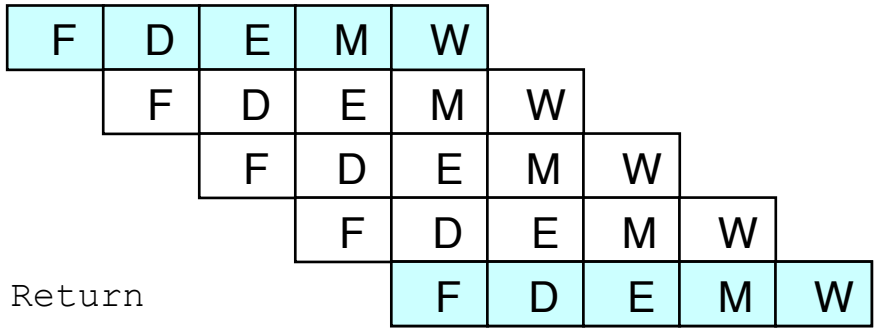
- Assuming we detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by **bubbles**
- No side effects have occurred yet

Return Instruction

```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020: .pos 0x20
0x020: p:  irmovq $-1,%rdi    # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100: .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

Stalling for Return

```
0x026:     ret
           nop
           nop
           nop
0x013:     irmovq $5,%rsi  # Return
```

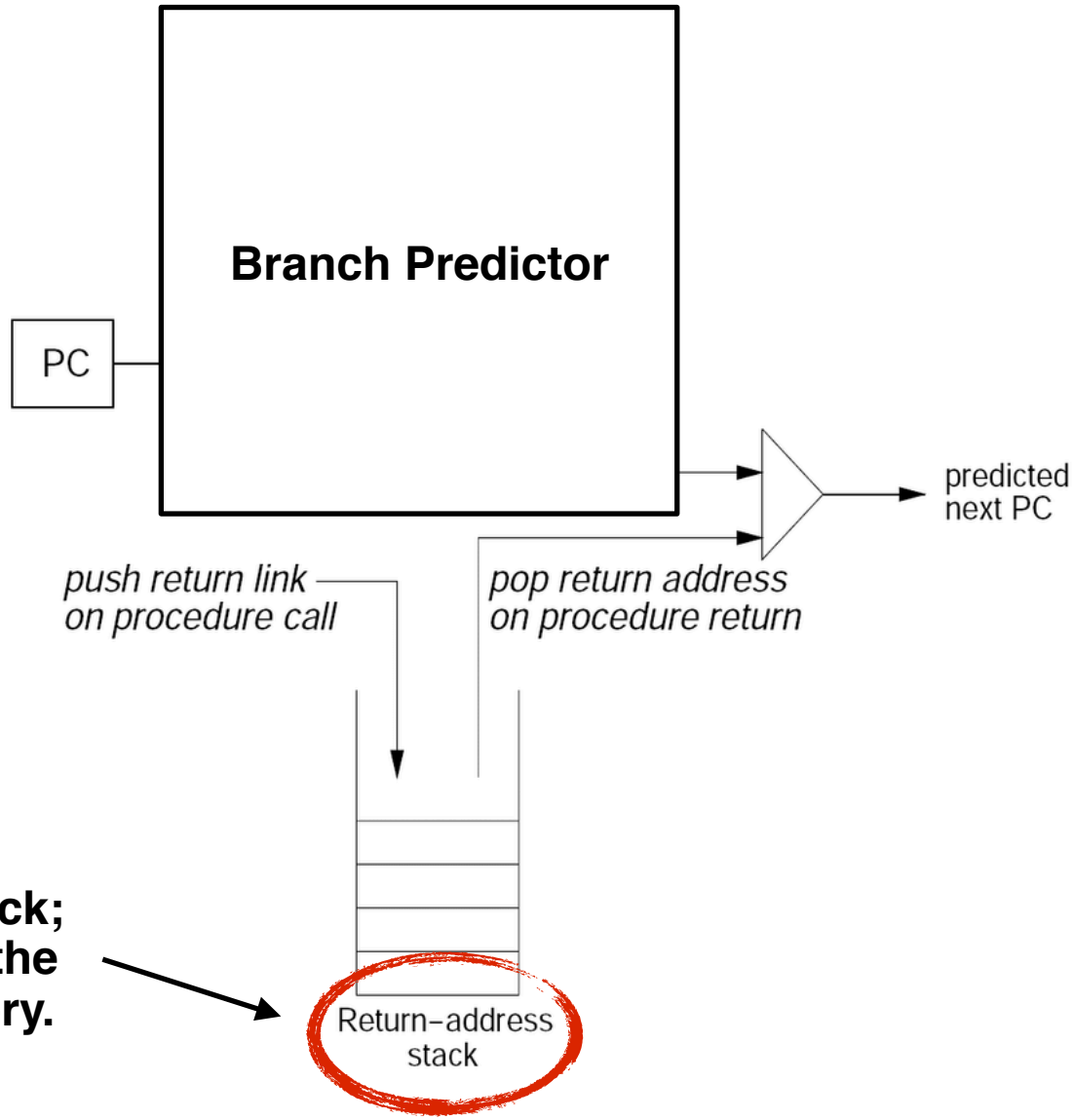


- As `ret` passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage

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)



A hardware stack; different from the stack in memory.

Today: Making the Pipeline Really Work

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

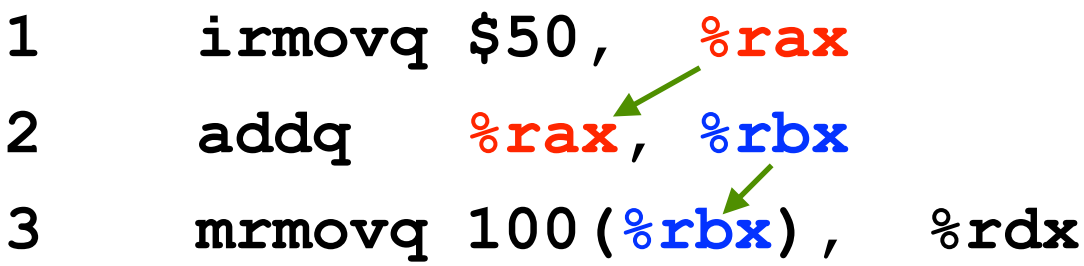
Data Dependencies

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

Data Dependencies

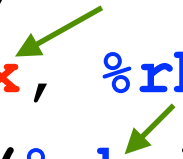
```
1     irmovq $50,   %rax
2     addq   %rax, %rbx
3     mrmovq 100(%rbx),   %rdx
```

Data Dependencies



Data Dependencies

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

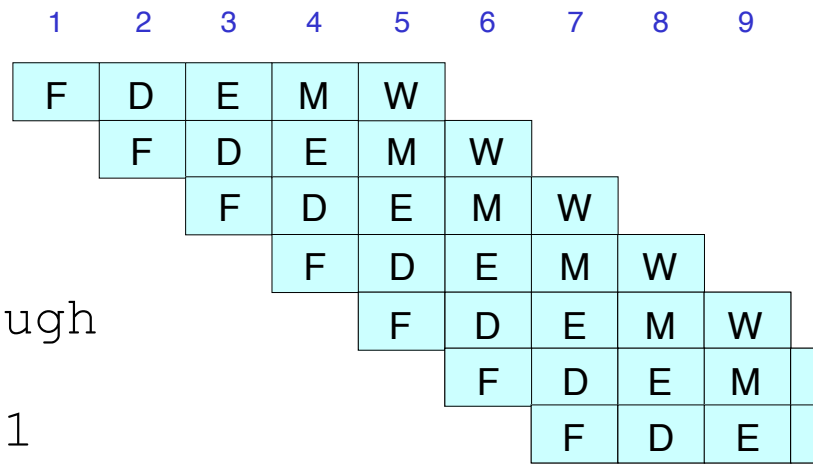


- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

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?**

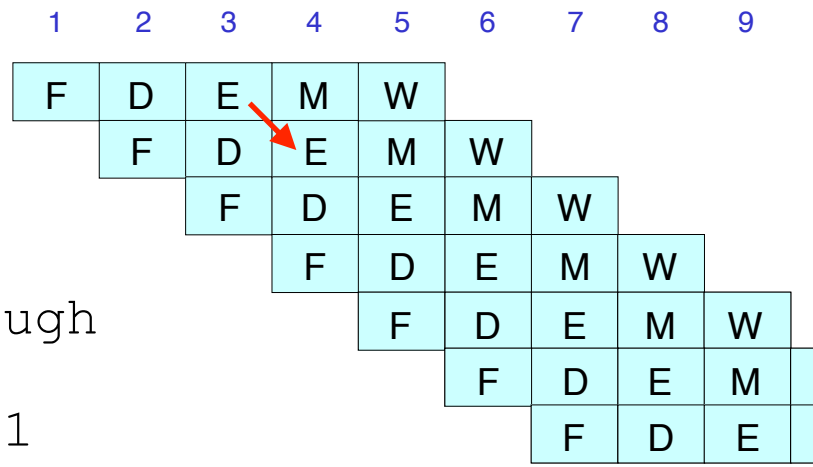
```
     xorg %rax, %rax
     jne L1                 # Not taken
     nop
     nop
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```



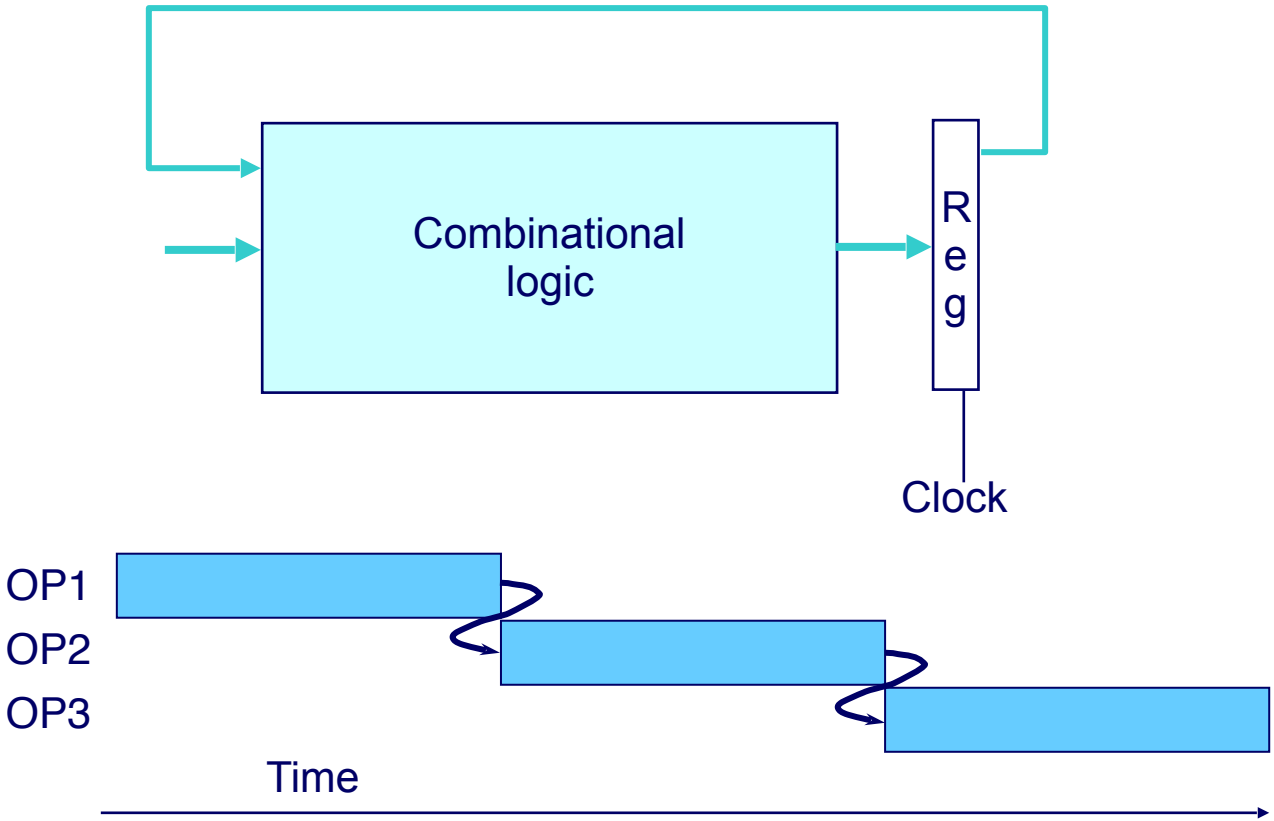
A Subtle Data Dependency

- 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
     nop
     nop
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```



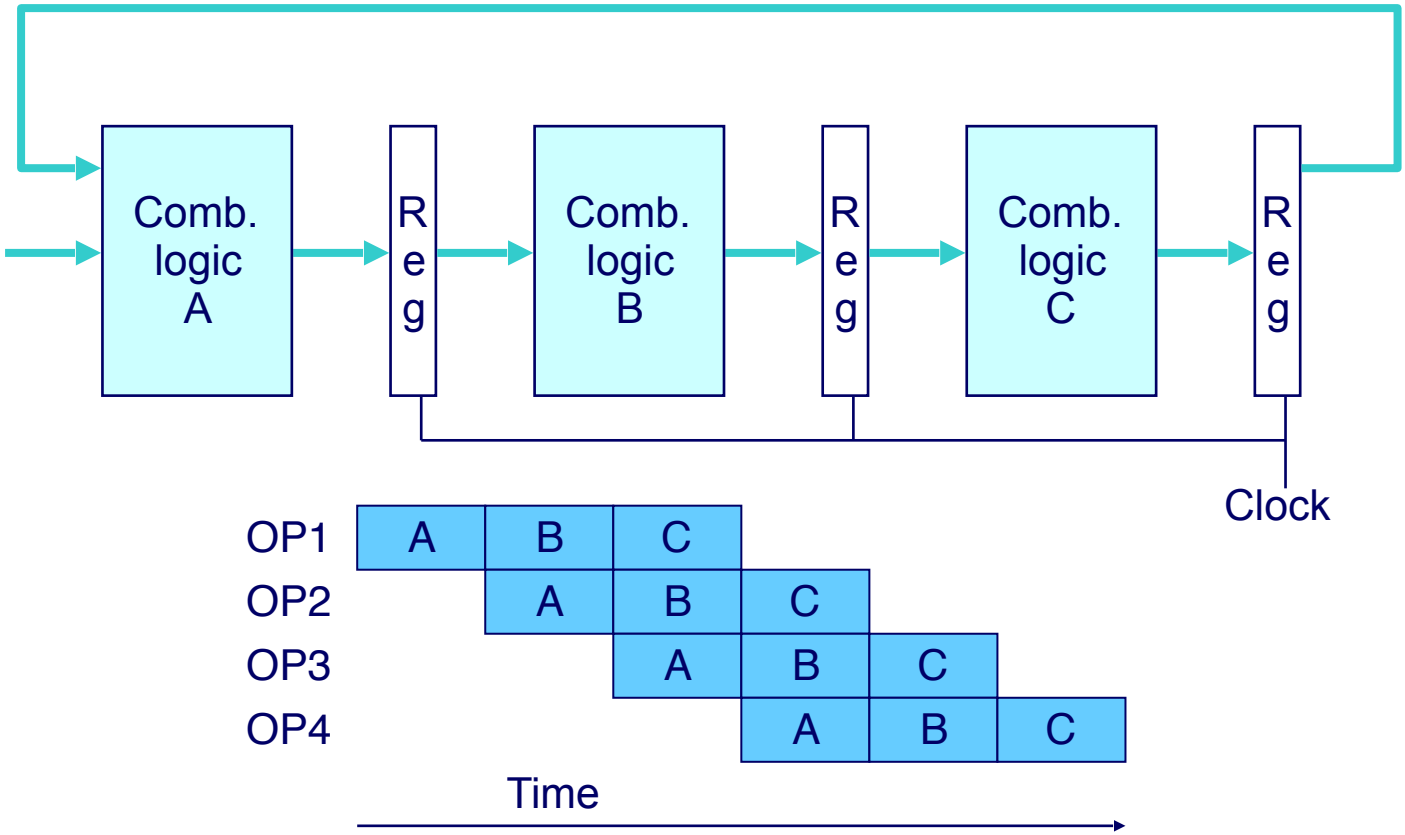
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

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

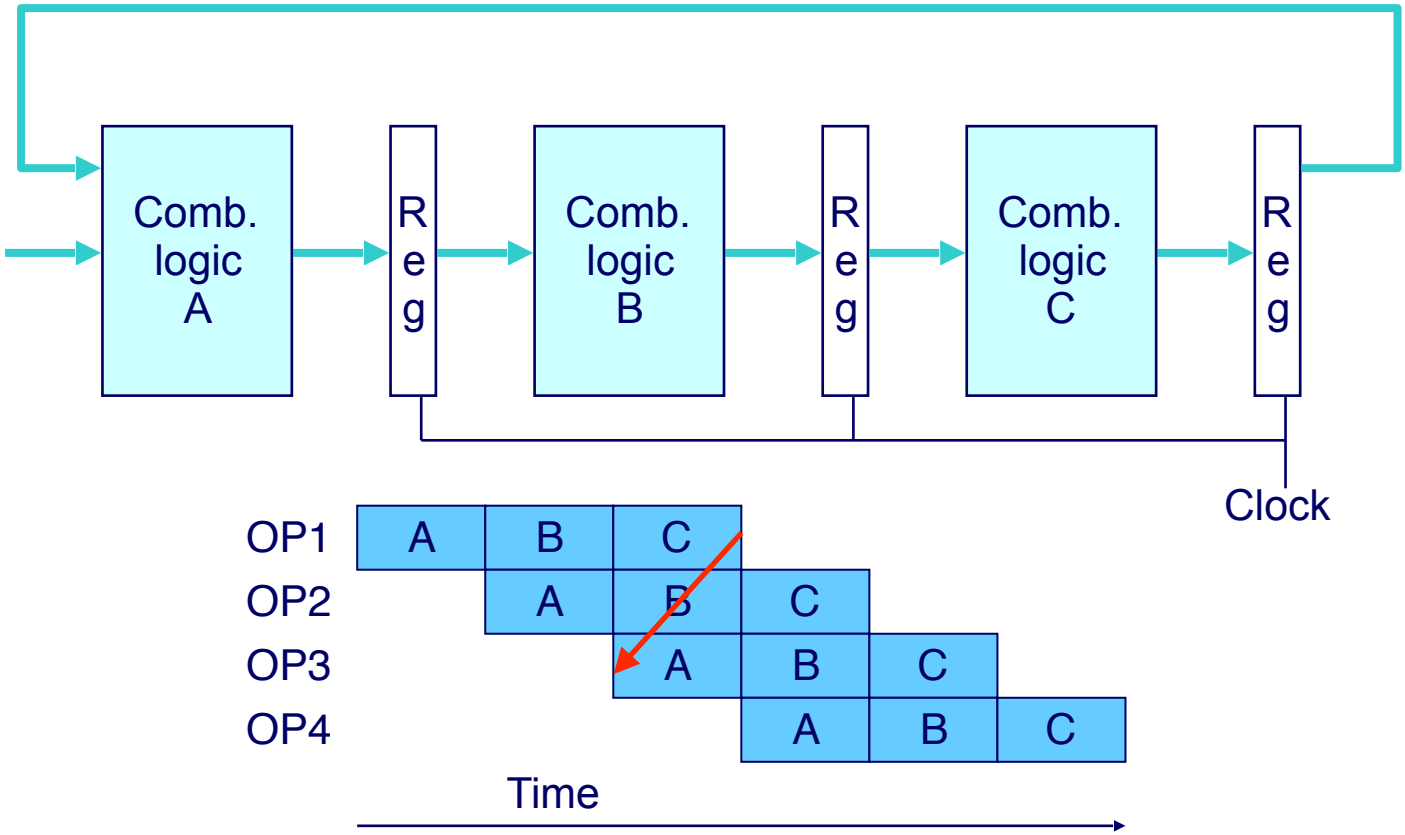
Data Dependencies in Pipeline Machines



Data Hazards happen when:

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

Data Dependencies in Pipeline Machines

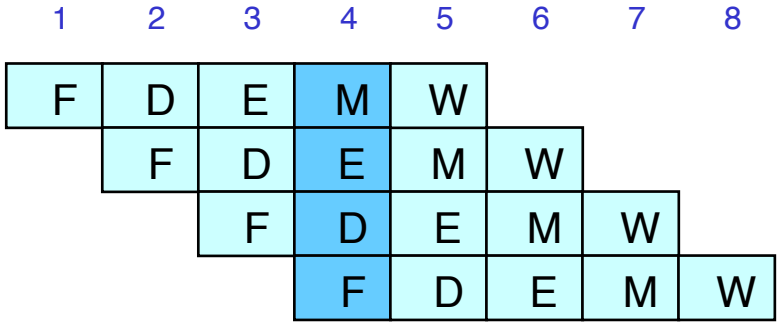


Data Hazards happen when:

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

Data Dependencies: No Nop

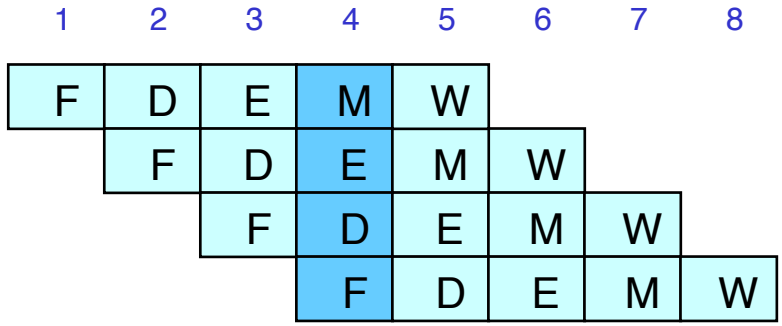
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



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
```

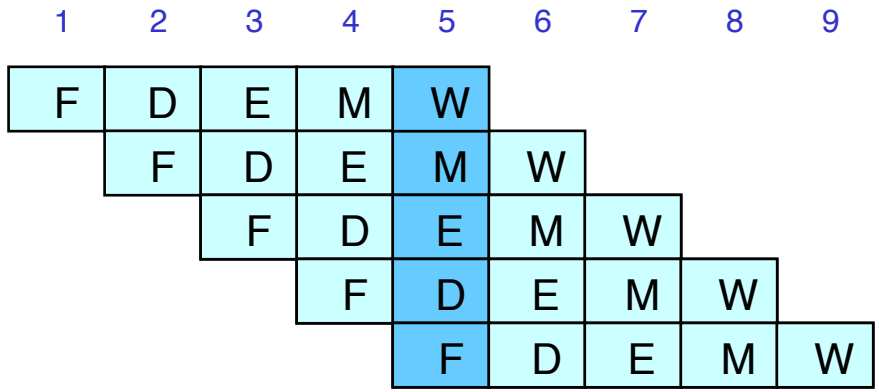


Remember registers get
updated in the Write-back stage

addq reads wrong %rdx and %rax

Data Dependencies: 1 Nop

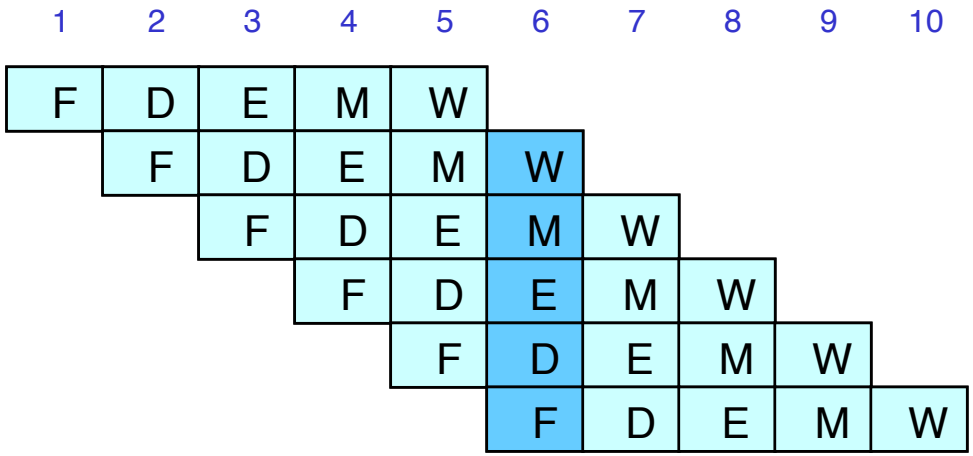
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



addq still reads wrong %rdx and %rax

Data Dependencies: 2 Nop's

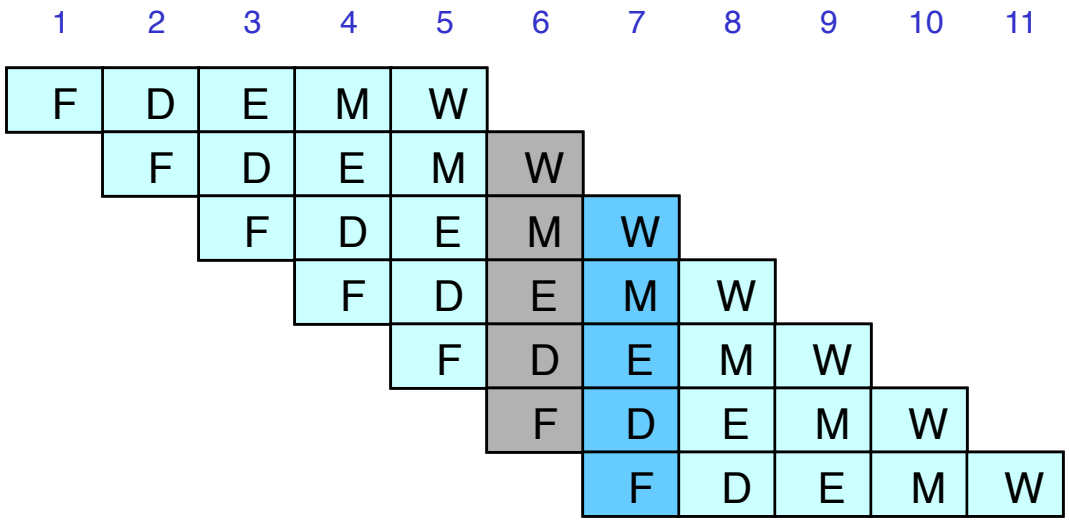
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



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

Data Dependencies: 3 Nop's

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```



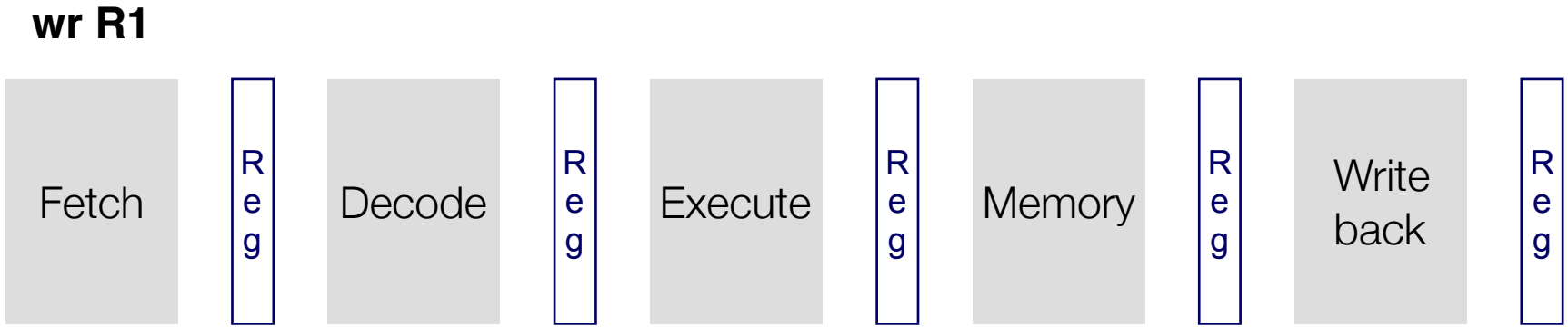
addq reads the correct %rdx and %rax

Resolving Data Dependencies

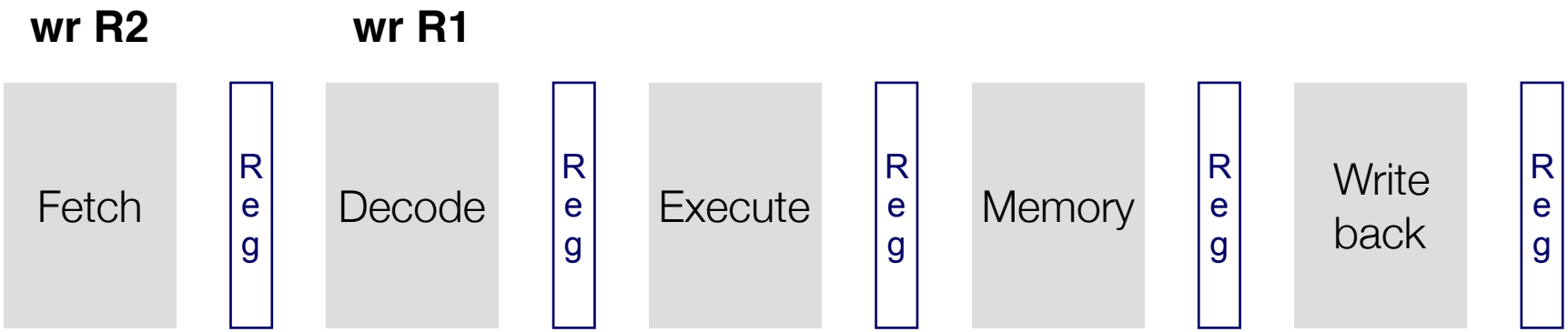
- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Hardware mechanisms
  - Stalling
  - Forwarding
  - Out-of-order execution

Hardware Generated Nops (Bubble and Stalling)

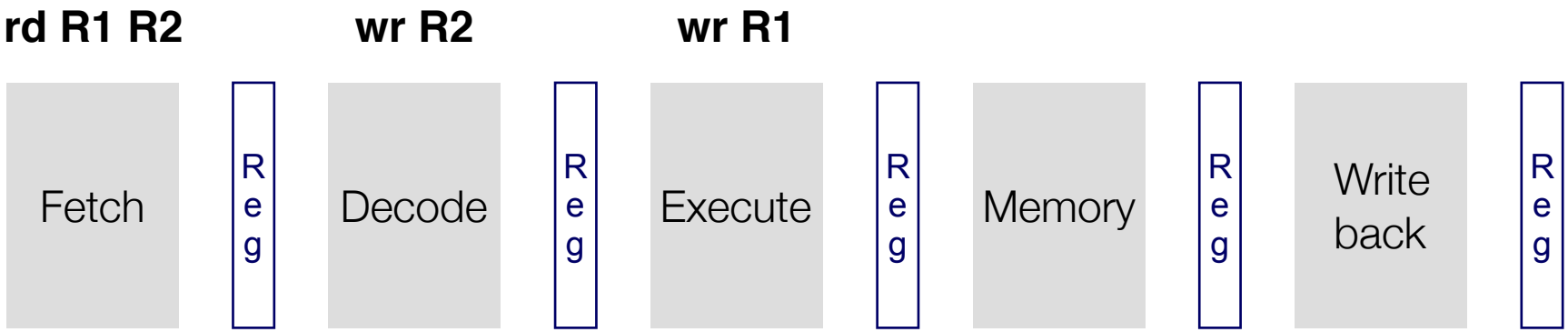
Hardware Generated Nops (Bubble and Stalling)



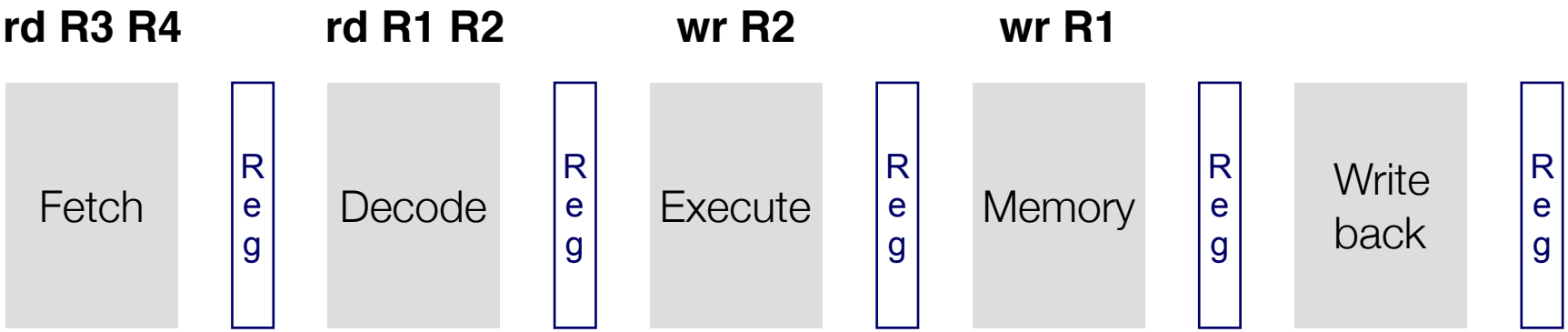
Hardware Generated Nops (Bubble and Stalling)



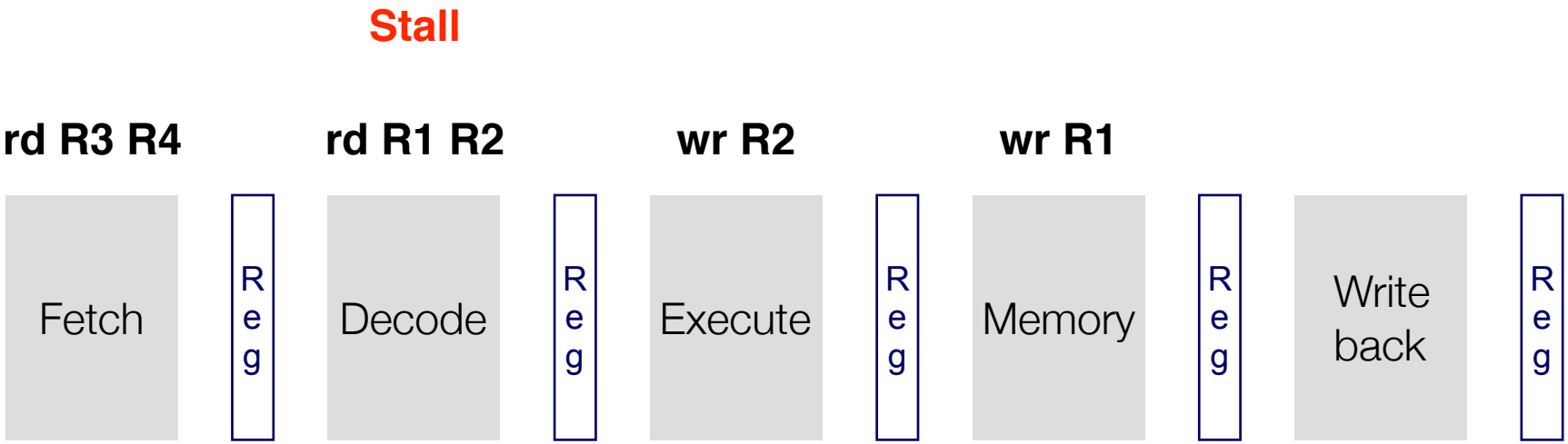
Hardware Generated Nops (Bubble and Stalling)



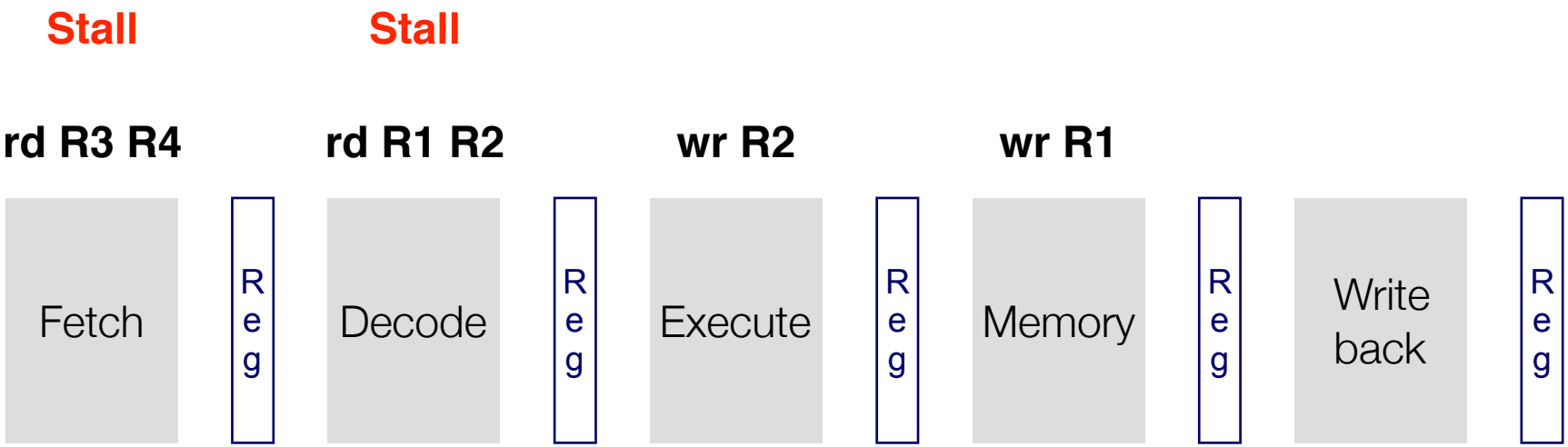
Hardware Generated Nops (Bubble and Stalling)



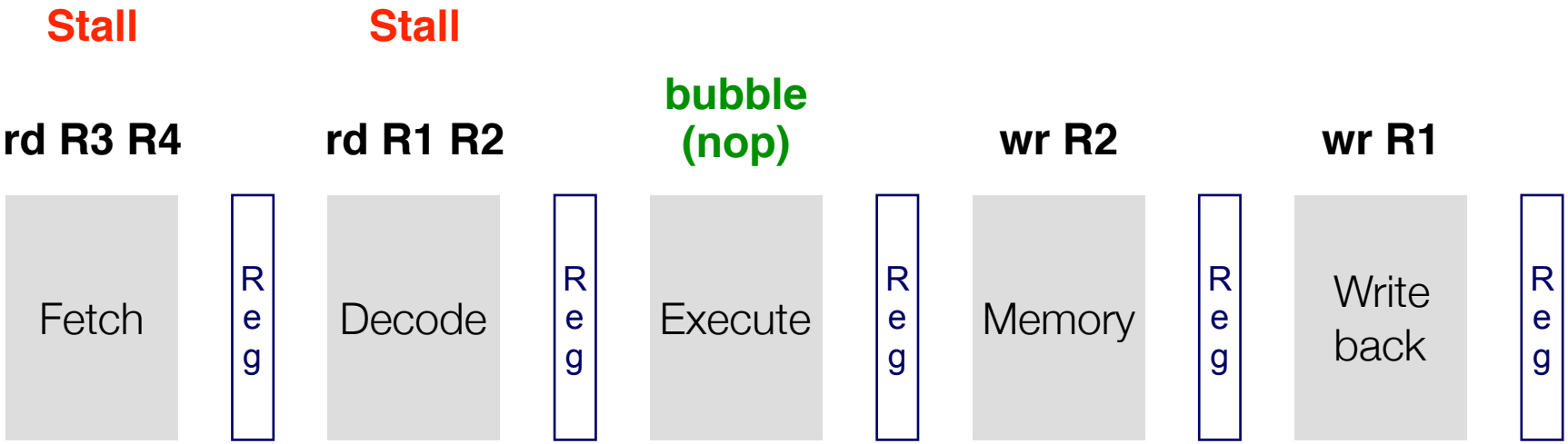
Hardware Generated Nops (Bubble and Stalling)



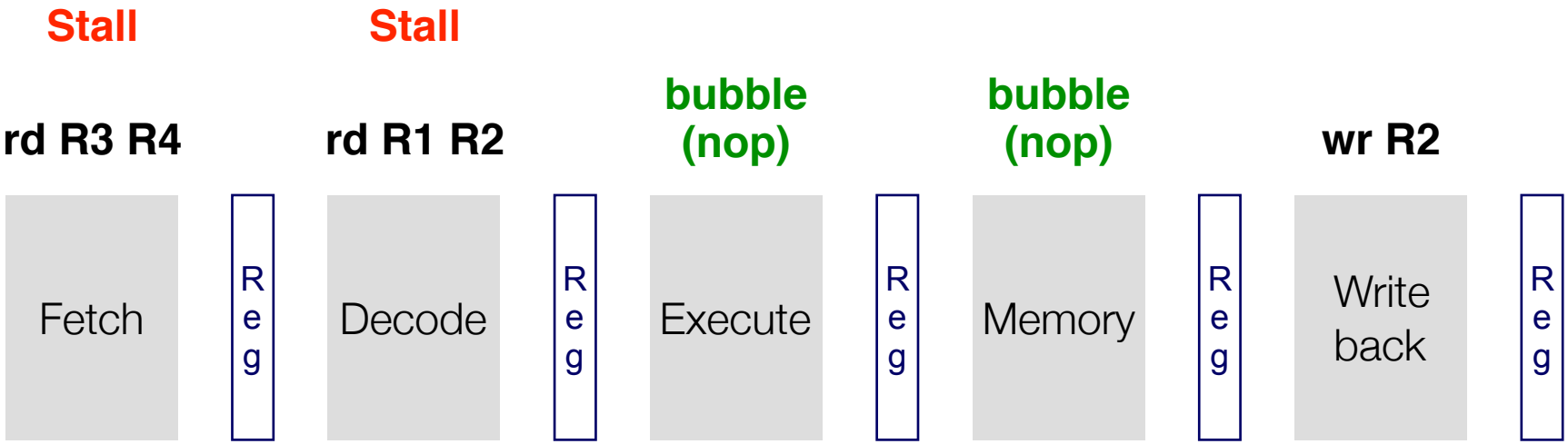
Hardware Generated Nops (Bubble and Stalling)



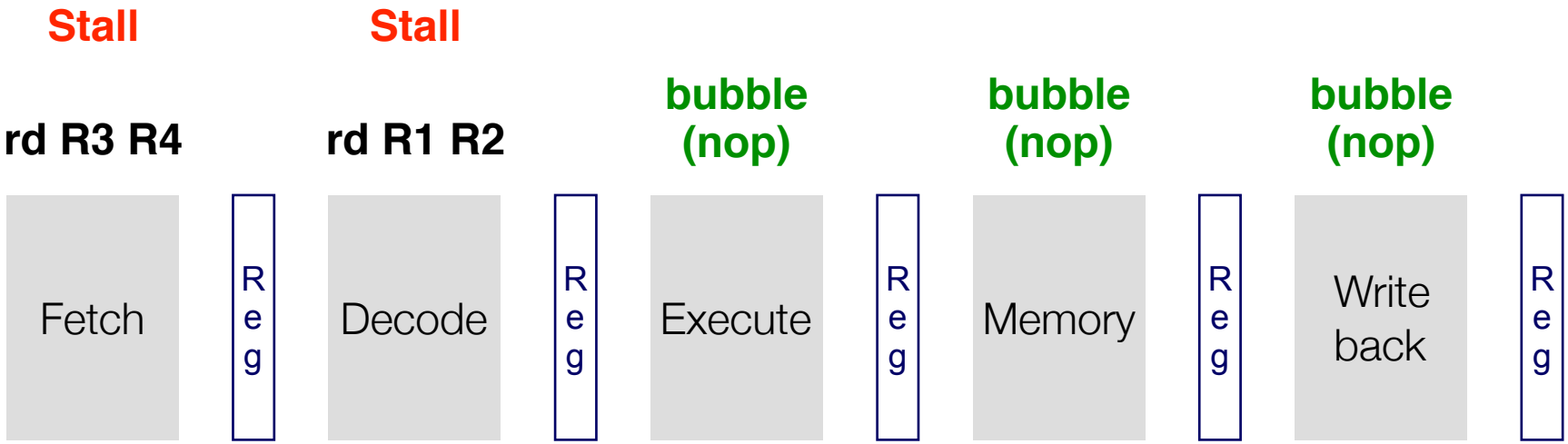
Hardware Generated Nops (Bubble and Stalling)



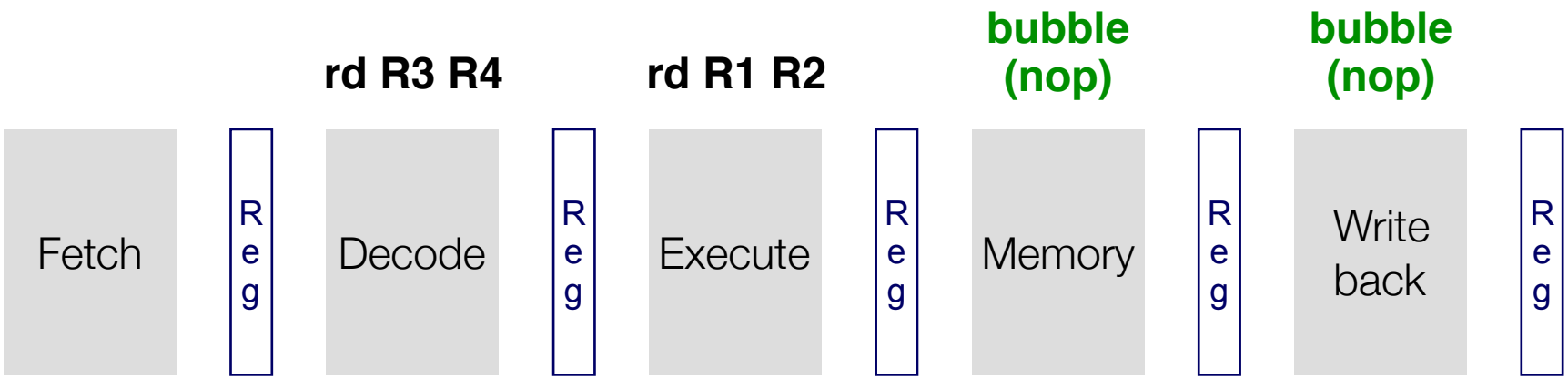
Hardware Generated Nops (Bubble and Stalling)



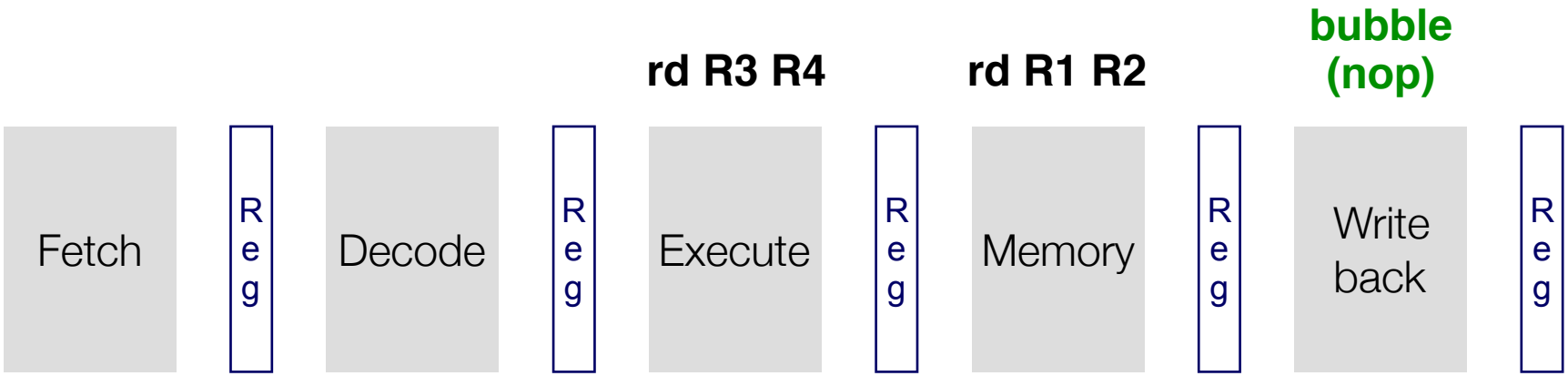
Hardware Generated Nops (Bubble and Stalling)



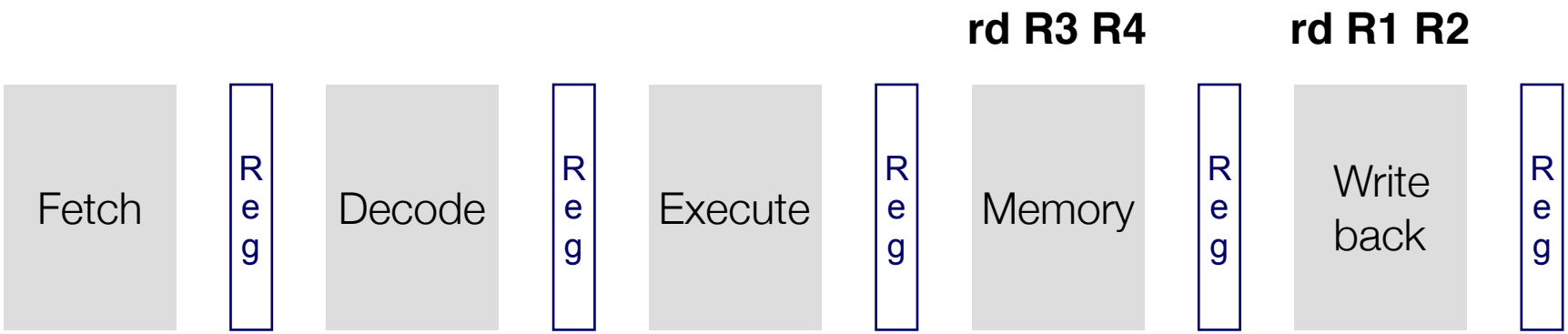
Hardware Generated Nops (Bubble and Stalling)



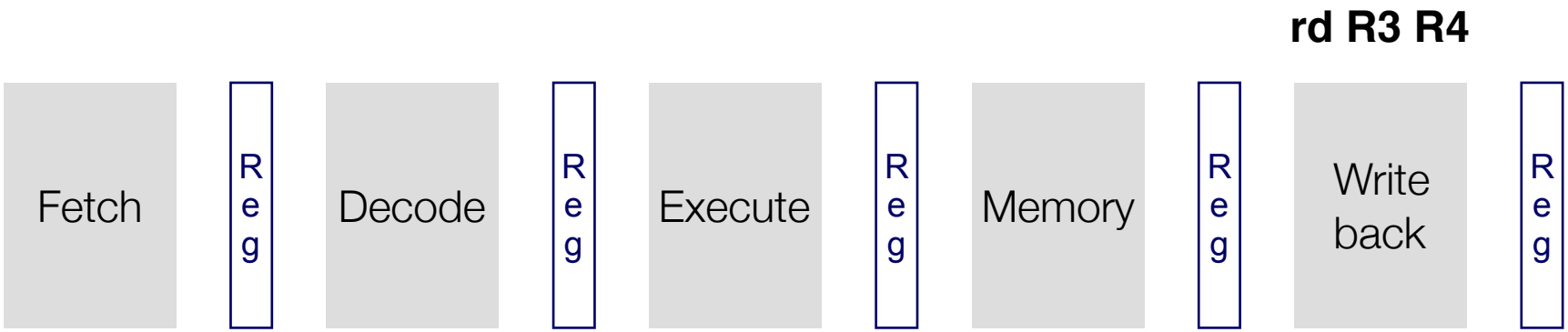
Hardware Generated Nops (Bubble and Stalling)



Hardware Generated Nops (Bubble and Stalling)



Hardware Generated Nops (Bubble and Stalling)



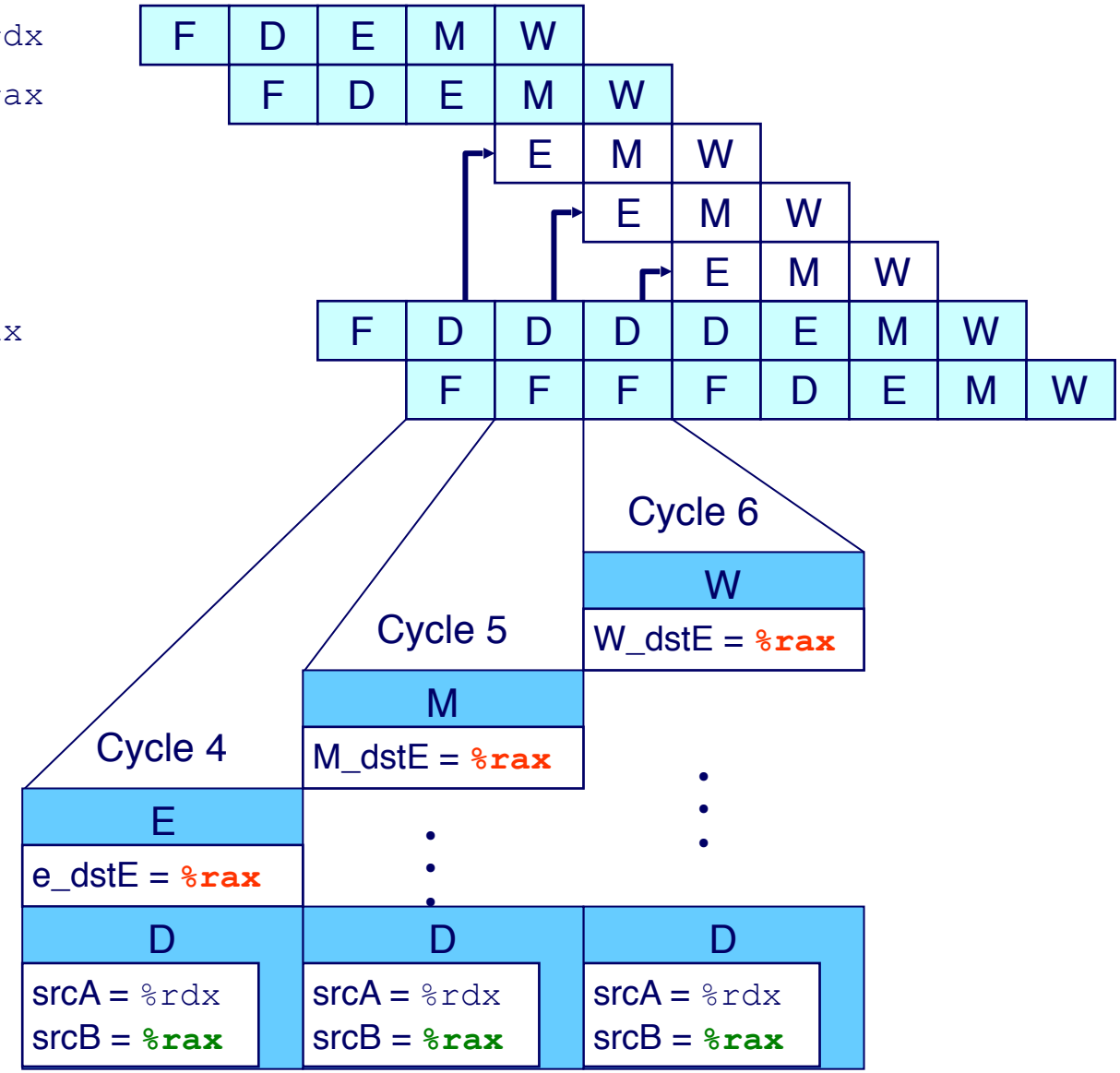
Hardware Generated Nops (Bubble and Stalling)

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!!

Detecting Stall Condition

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



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

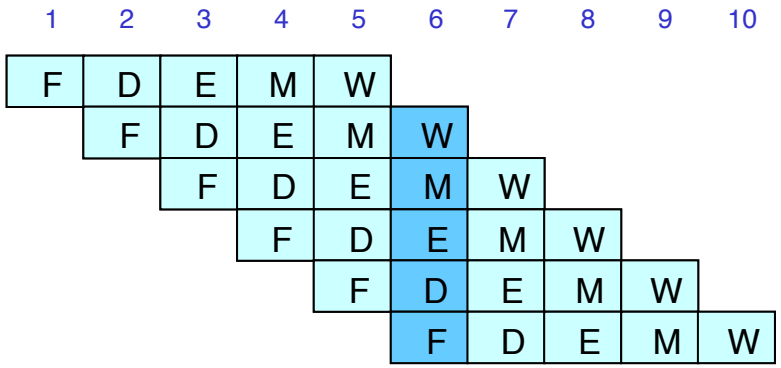
- Value generated in execute or memory stage

Trick

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

Data Forwarding Example

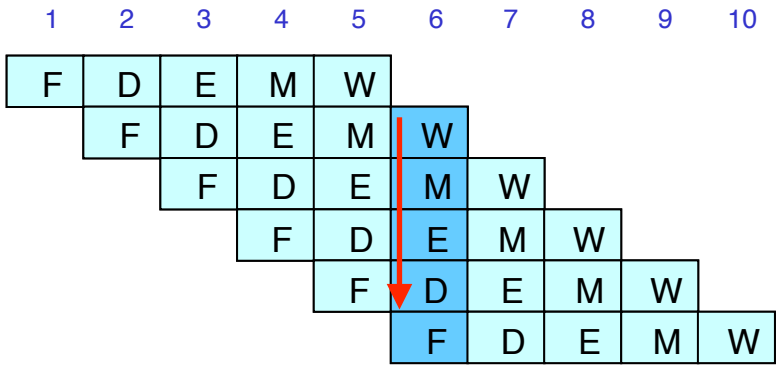
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `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

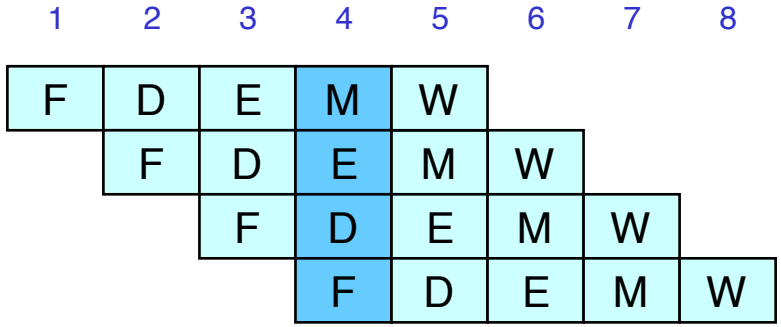
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `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`

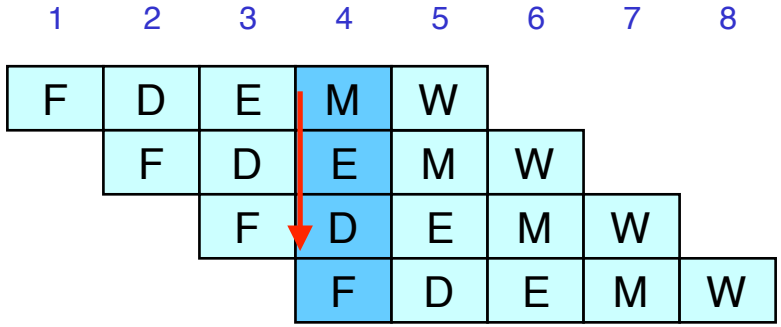
- Forward from the memory stage

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`

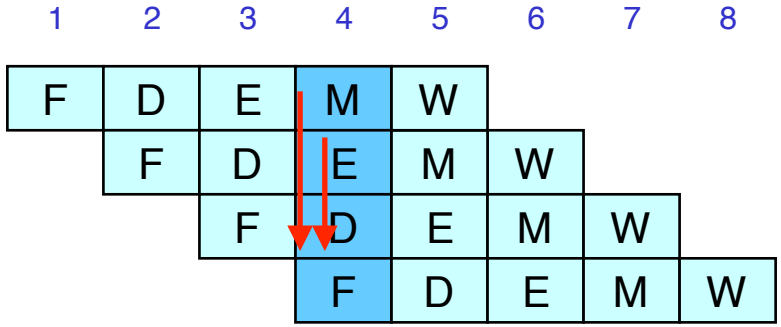
- Forward from the memory stage

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
```



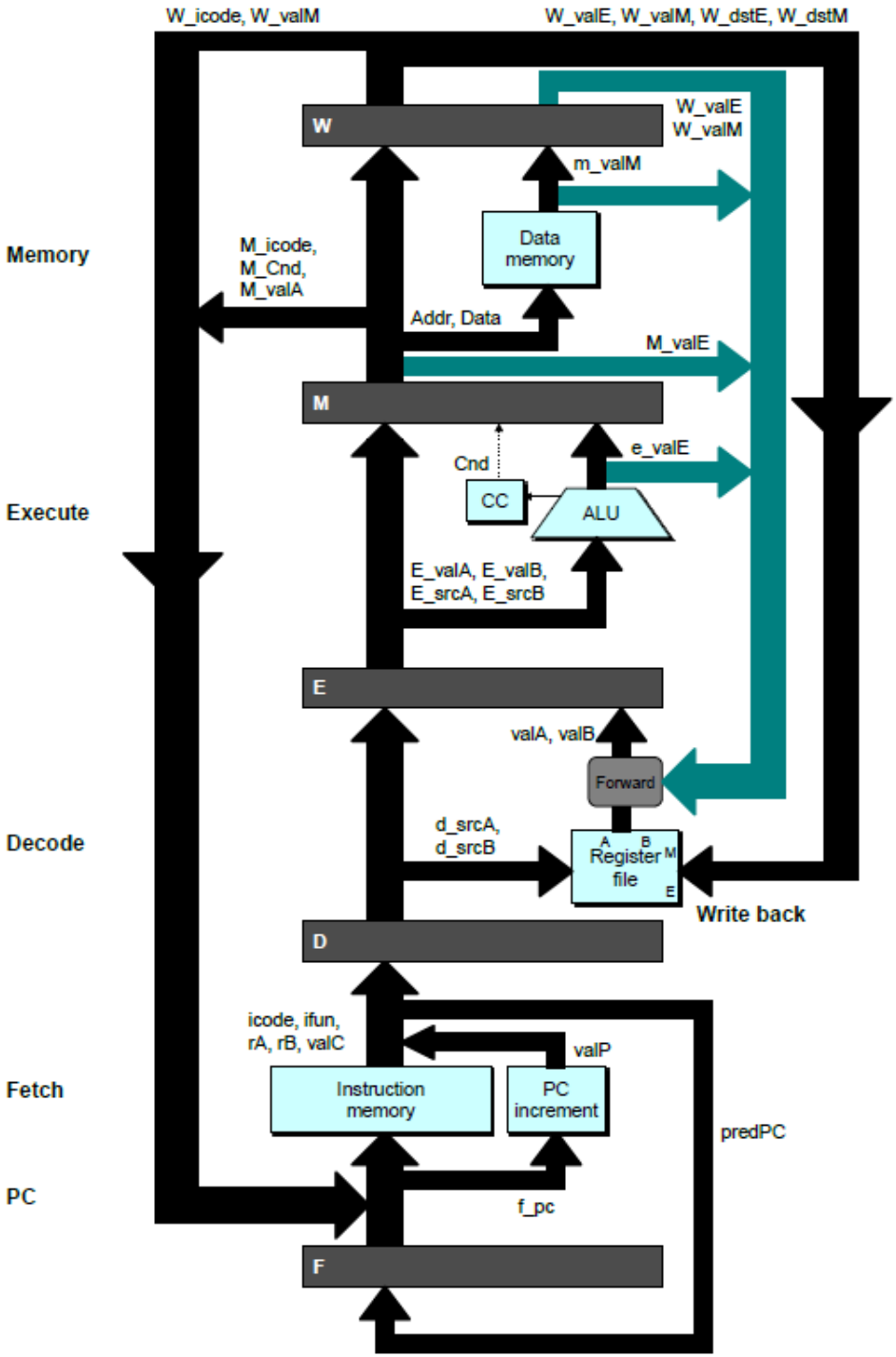
Register `%rdx`

- Forward from the memory stage

Register `%rax`

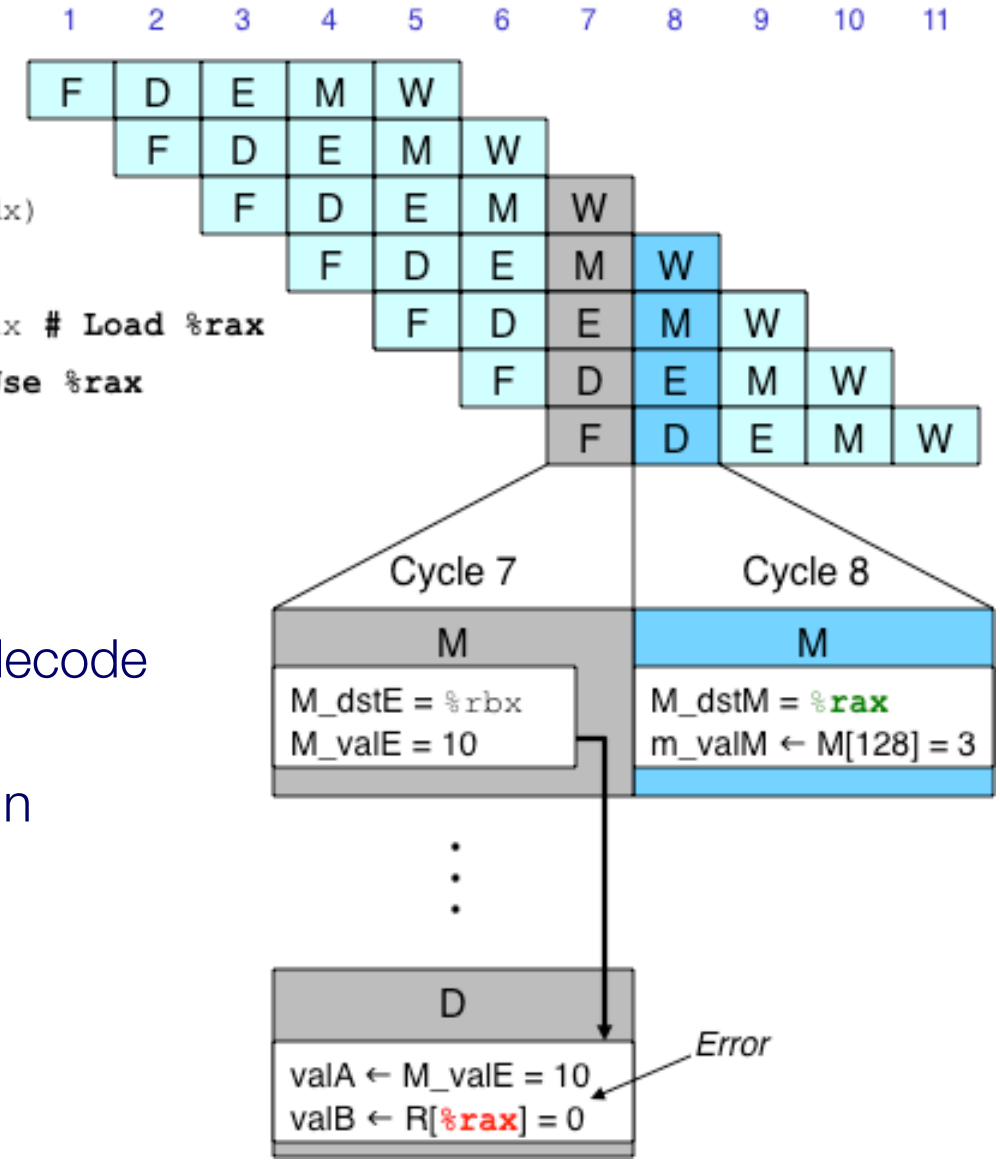
- Forward from the execute stage

Hardware Design



Limitation of Forwarding

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

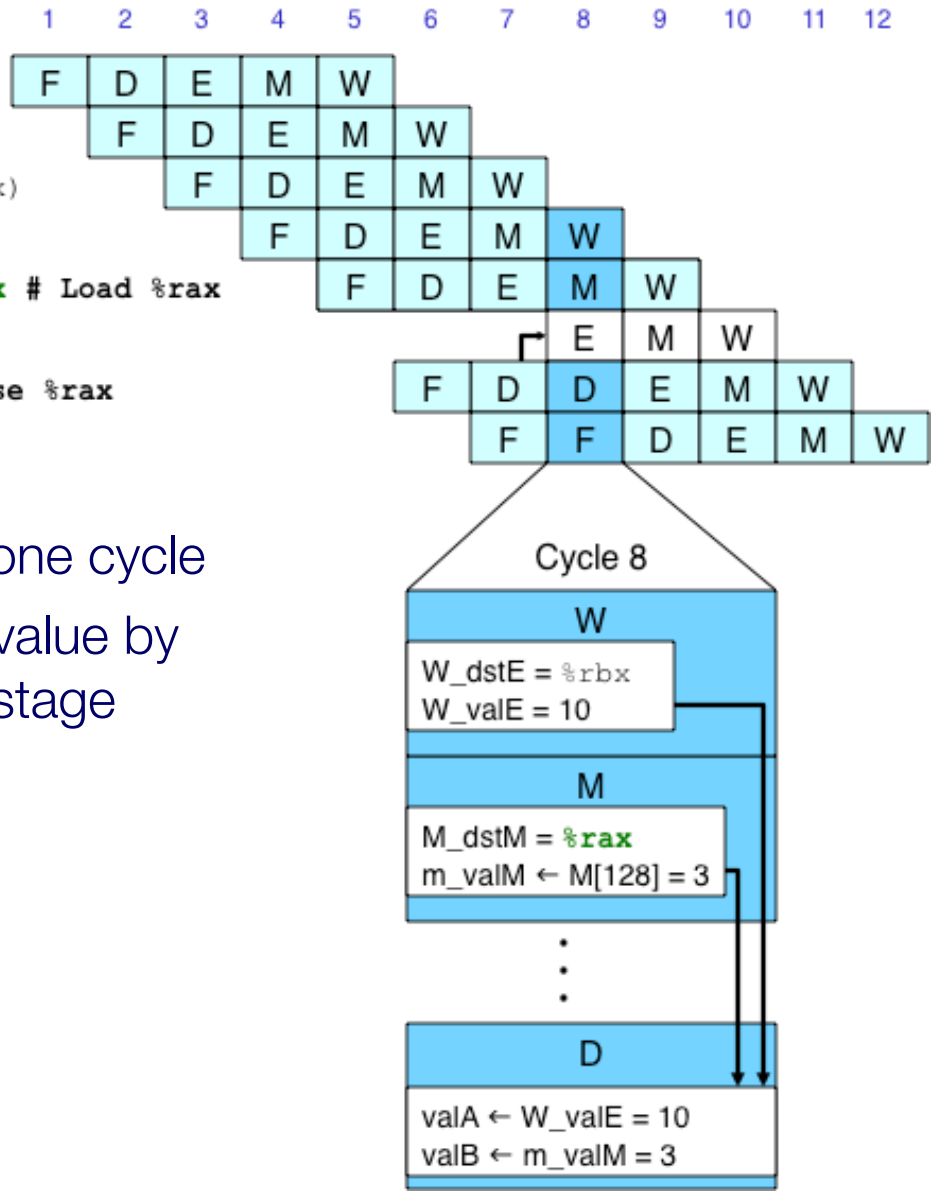


Load-use dependency

- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8

Avoiding Load/Use Hazard

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
     bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

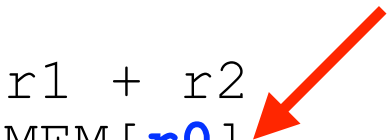


- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

Out-of-order Execution

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

Long-latency instruction.
Forces the pipeline to stall.

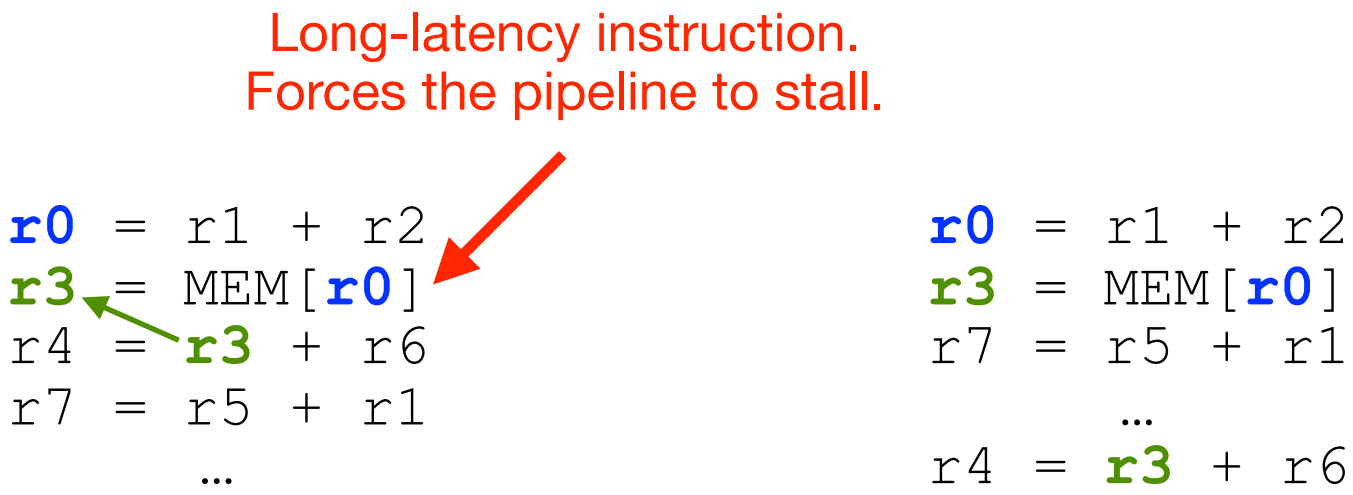


```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r7 = r5 + r1
     …
```

```
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



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

```
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
```

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