CSC 252: Computer Organization
Spring 2020: Lecture 14

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Lab 3 grades are out.

Announcements

- Lab 3 grades are out.

- Mid-term exam: March 5; in class. 75 mins.
- Past exam and problem set: https://www.cs.rochester.edu/courses/252/spring2020/handouts.html

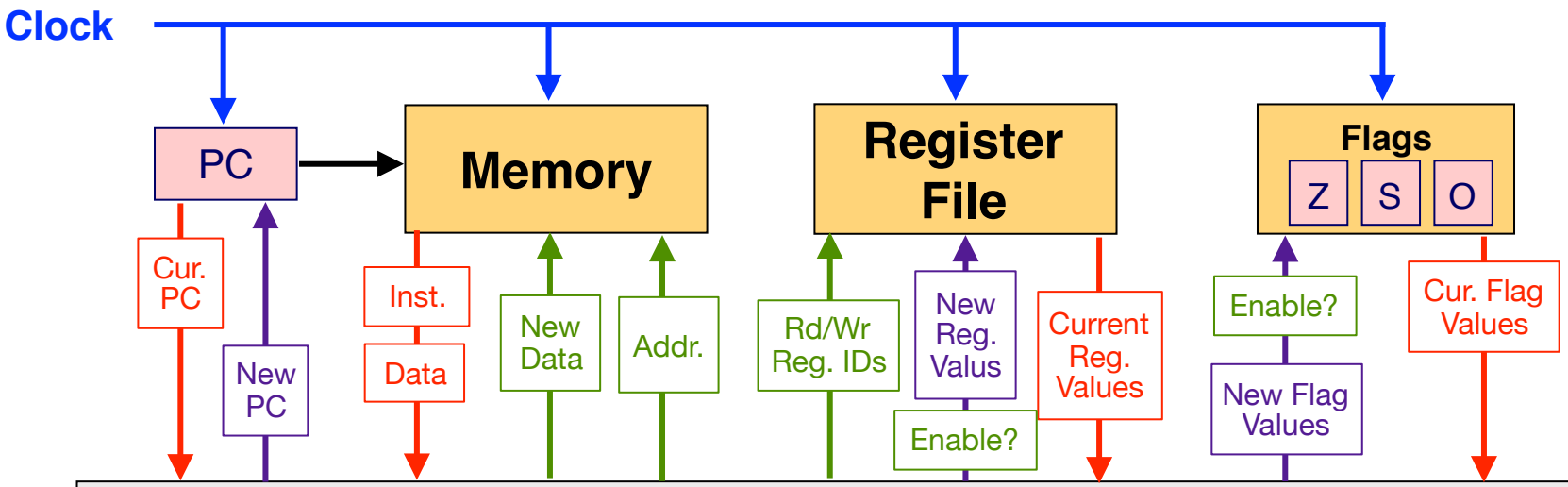
Announcements

- Lab 3 grades are out.

- Mid-term exam: March 5; in class. 75 mins.
- Past exam and problem set: https://www.cs.rochester.edu/courses/252/spring2020/handouts.html

- 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.
- **Nothing biological**, including your roommate, husband, wife, your hamster, another professor, etc.
- "**I don't know**" gets15% partial credit. Must erase everything else.

Single-Cycle Microarchitecture



Combinational Logic

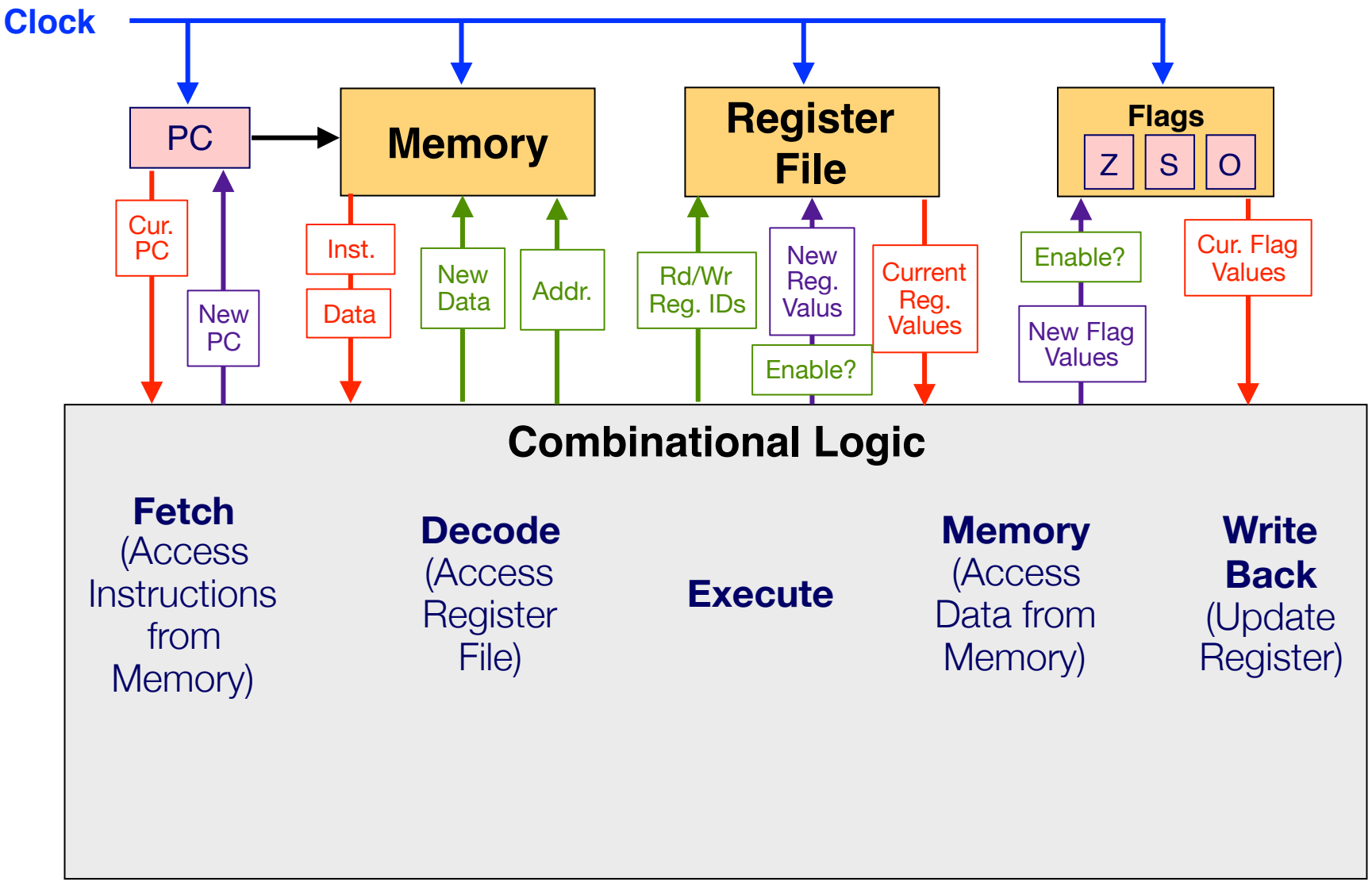
Read current_states;

next_states = f(current_states);

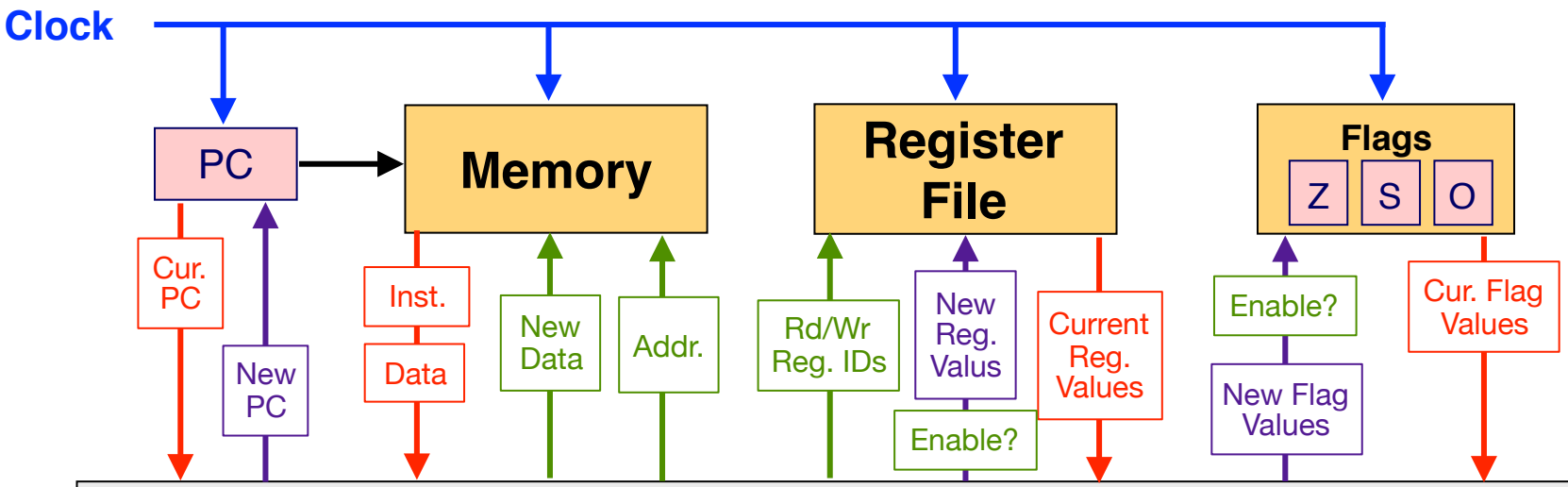
When clock rises, current_states = next_states;

next_states has to be ready before the close rises

Single-Cycle Microarchitecture



Single-Cycle Microarchitecture



Fetch (Access Instructions from Memory)

Decode (Access Register File)

Execute

Memory (Access Data from Memory)

Write Back (Update Register)

The execution time of an instruction is the delay of the combinational logic.
The next instruction can't start until the previous instruction has finished.

A Way to Look At the Microarchitecture

Principles:

- Execute each instruction one at a time, one after another
- Express every instruction as series of simple steps
- Dedicated hardware structure for completing each step
- Follow same general flow for each instruction type

Fetch: Read instruction from instruction memory

Decode: Read program registers

Execute: Compute value or address

Memory: Read or write data

Write Back: Write program registers

PC: Update program counter

Pipeline Stages

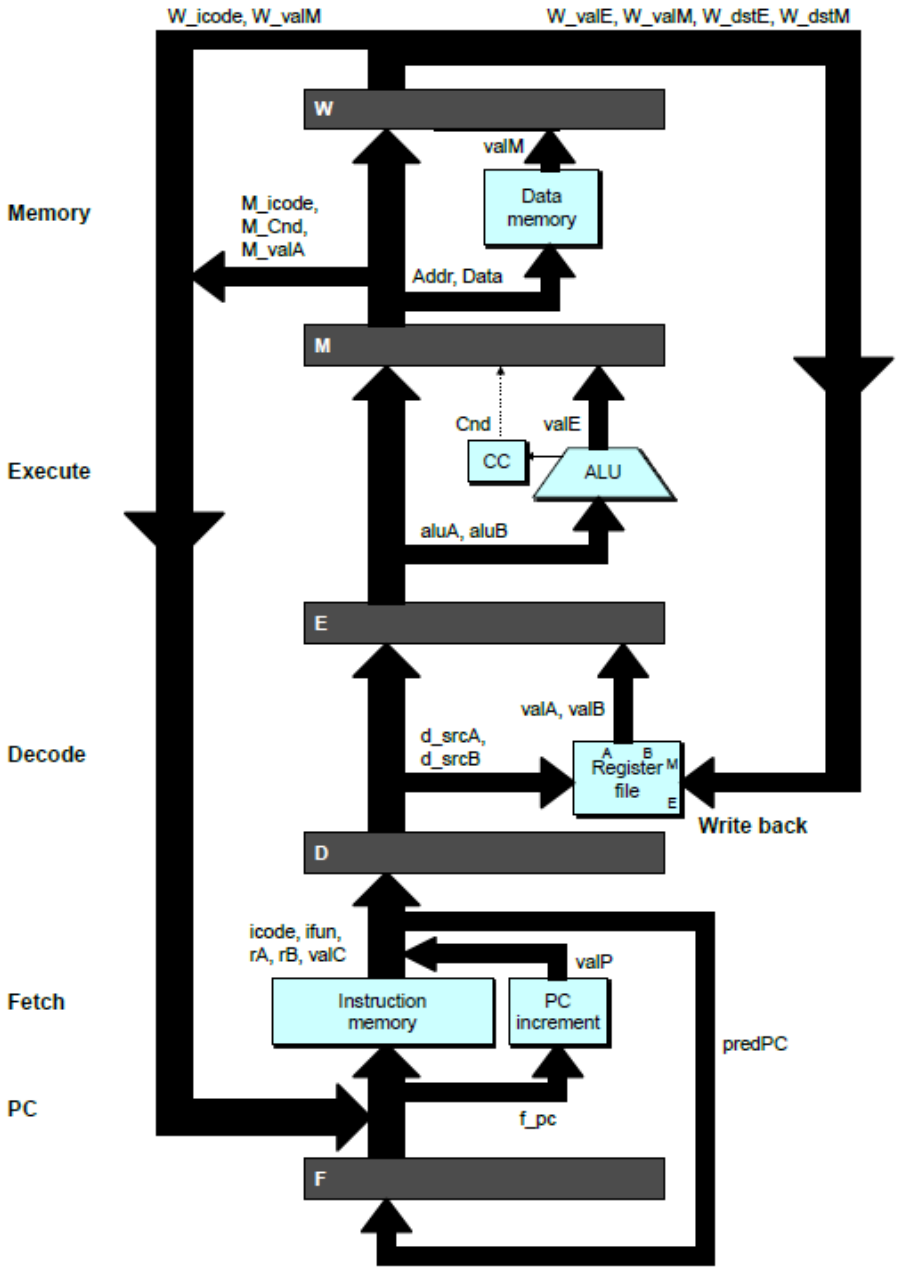
Fetch
- Select current PC
- Read instruction
- Compute incremented PC

Decode
- Read program registers

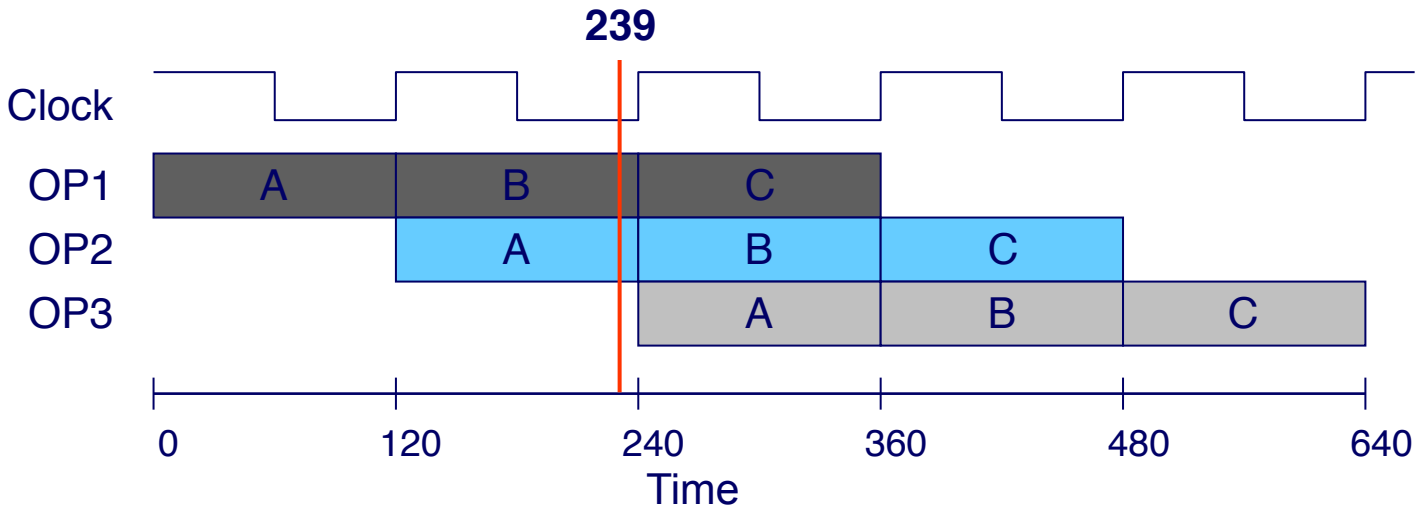
Execute
- Operate ALU

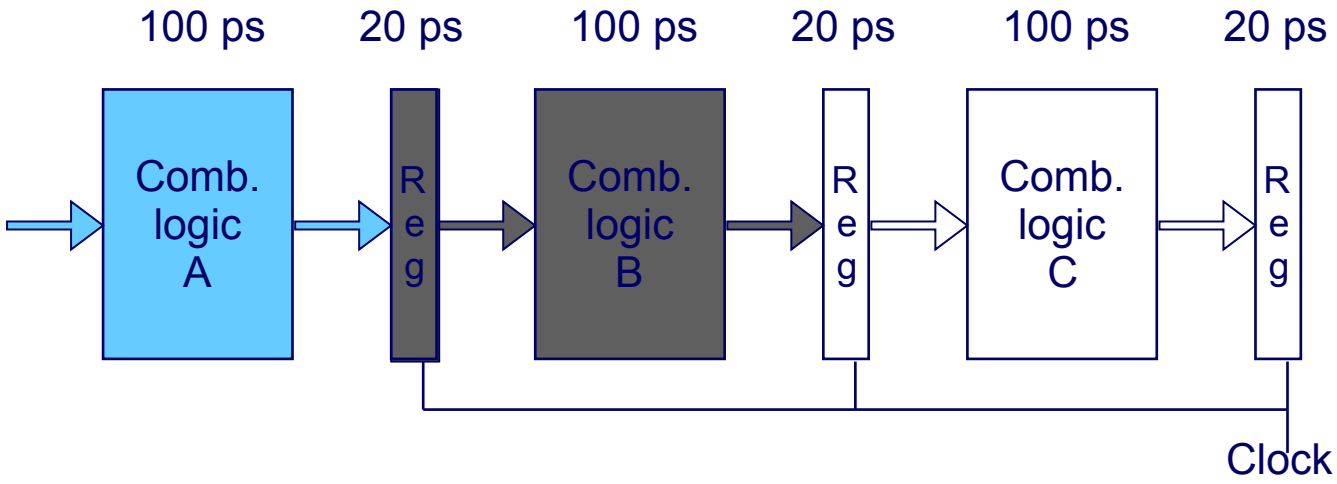
Memory
- Read or write data memory

Write Back
- Update register file

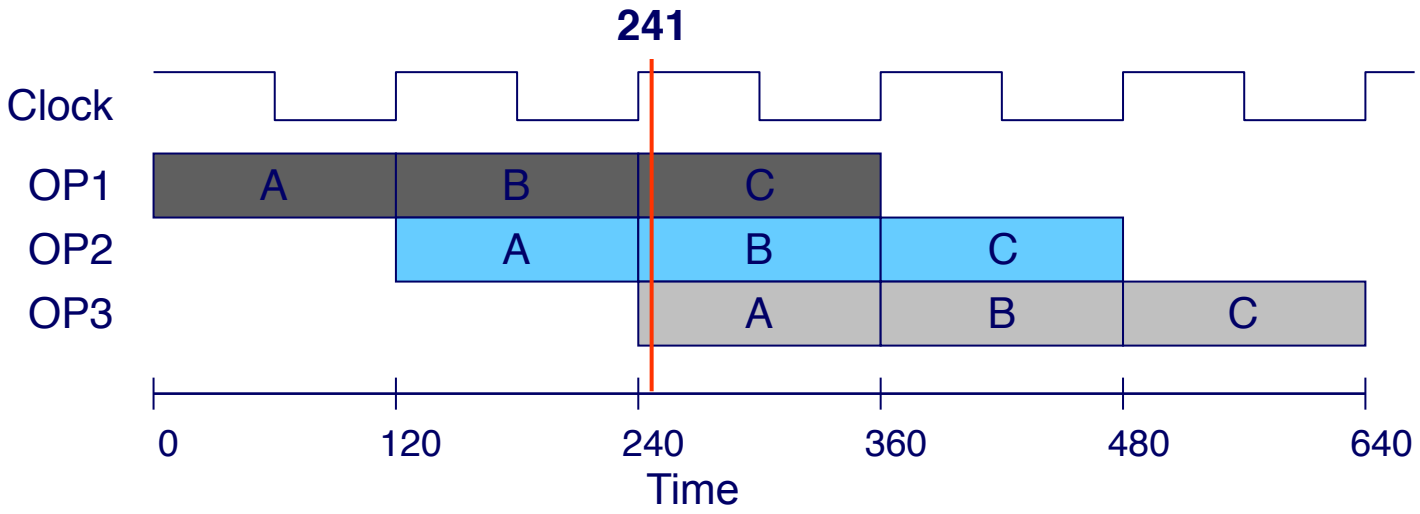


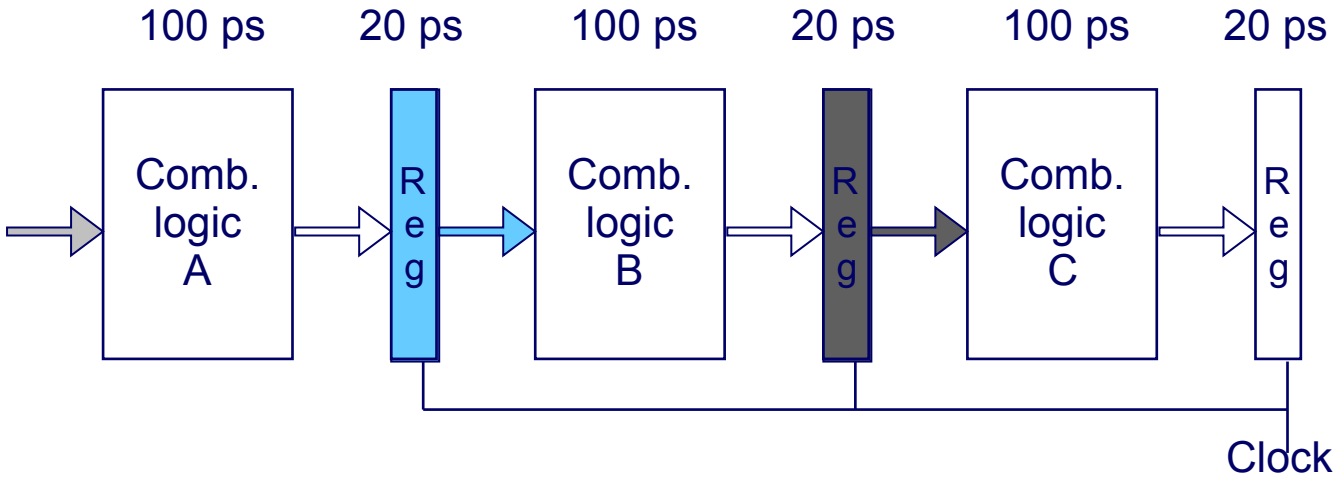
Operating a Pipeline



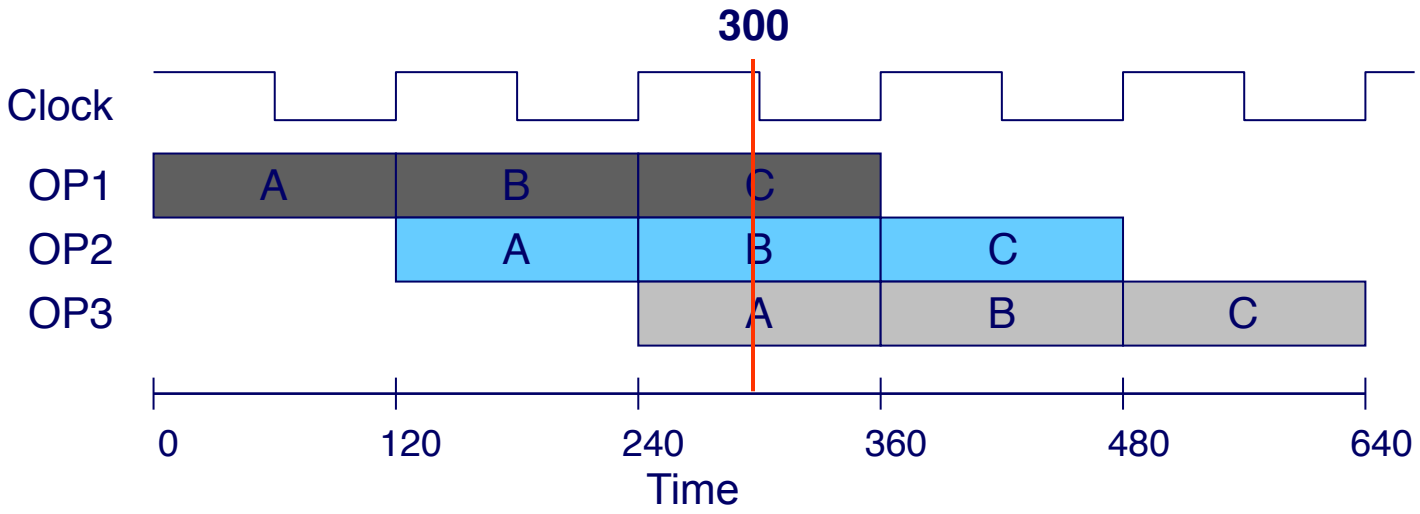


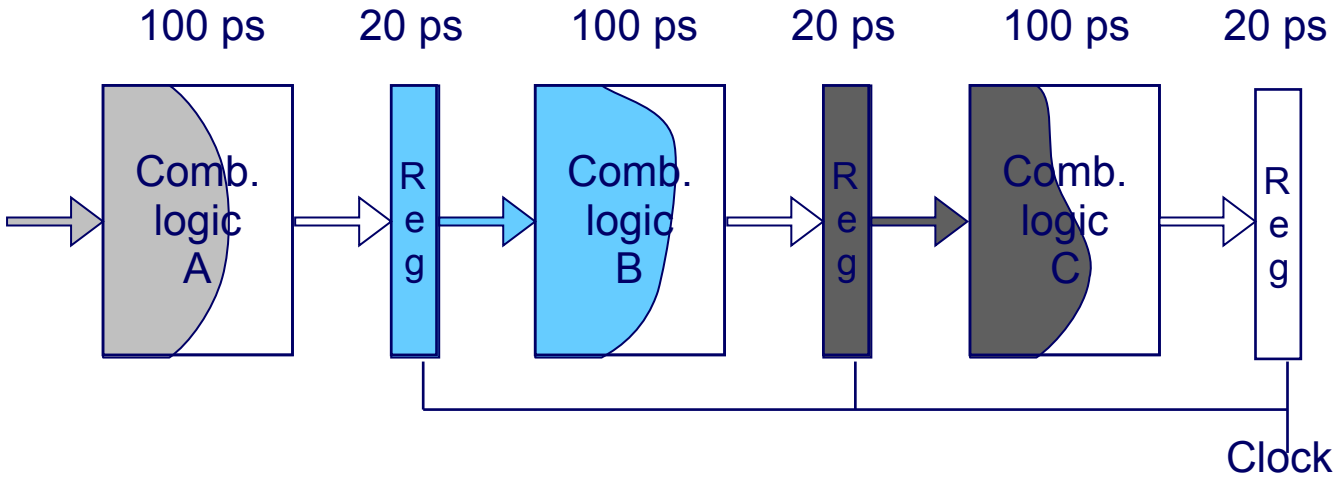
Operating a Pipeline



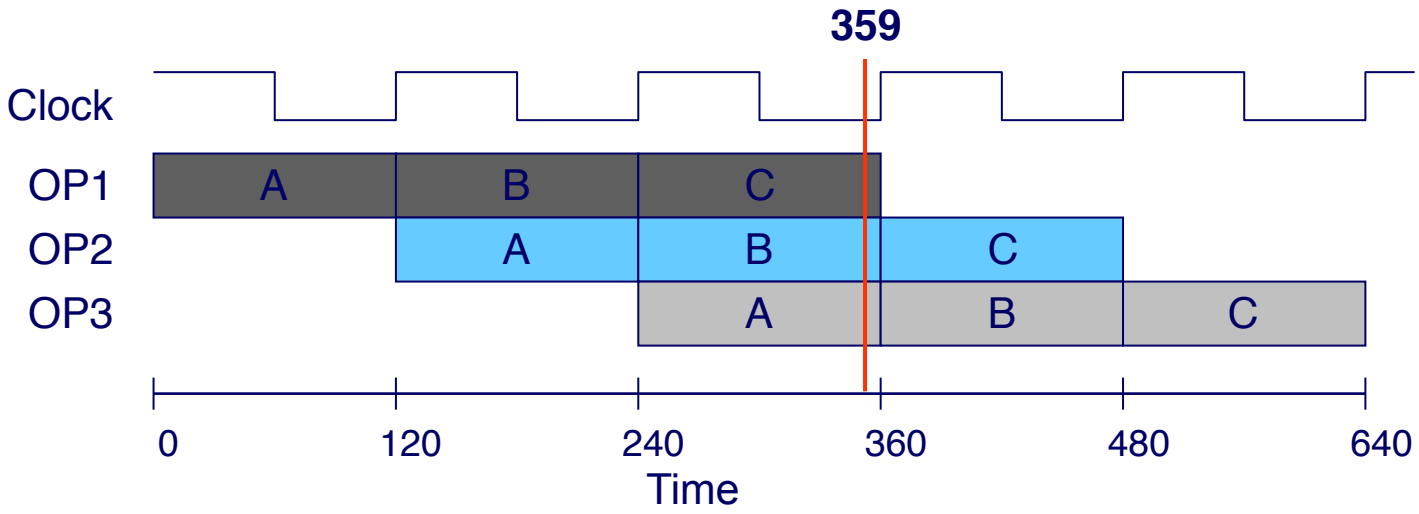


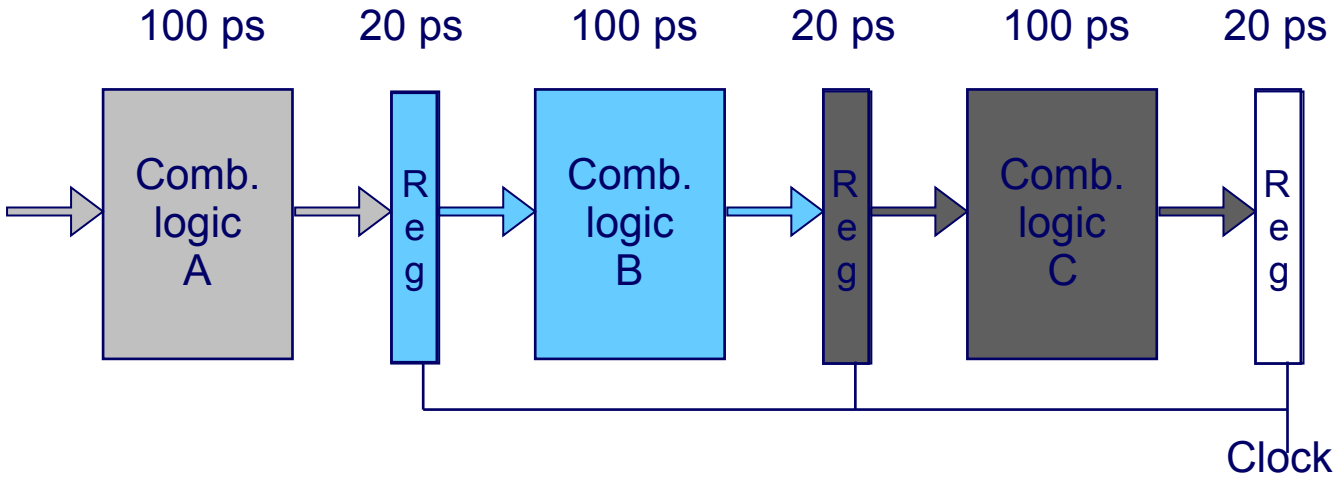
Operating a Pipeline





Operating a Pipeline





Pipeline Stages

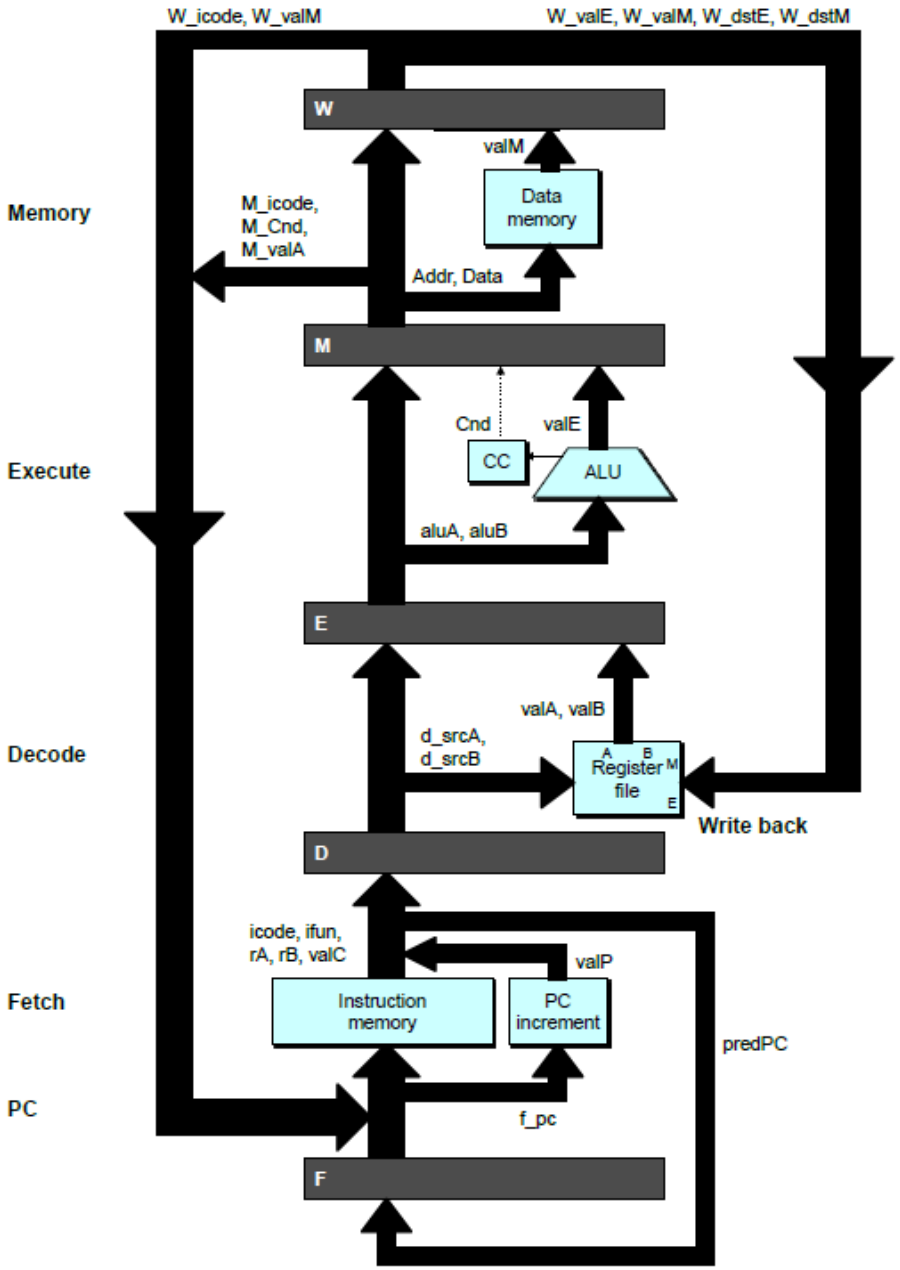
Fetch
- Select current PC
- Read instruction
- Compute incremented PC

Decode
- Read program registers

Execute
- Operate ALU

Memory
- Read or write data memory

Write Back
- Update register file

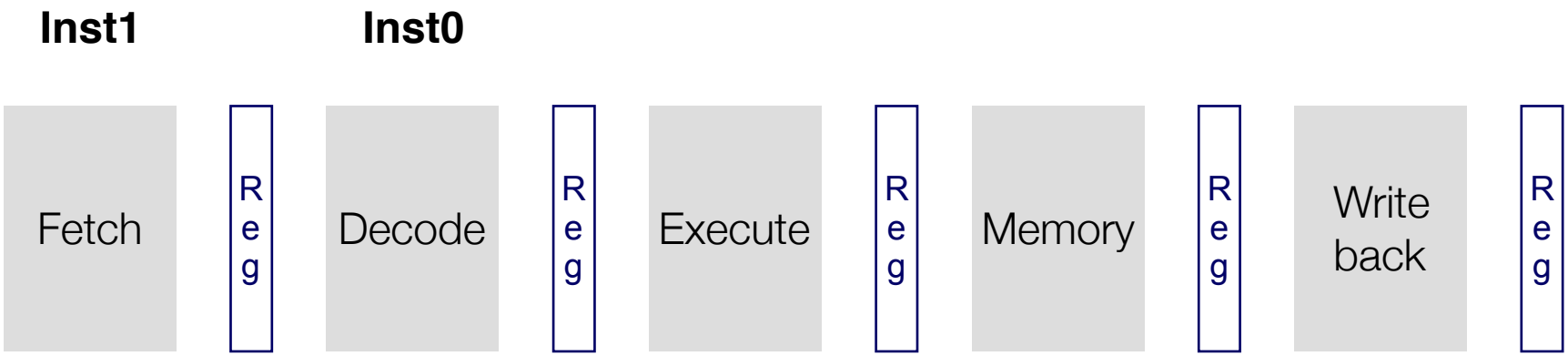


Pipeline Illustration

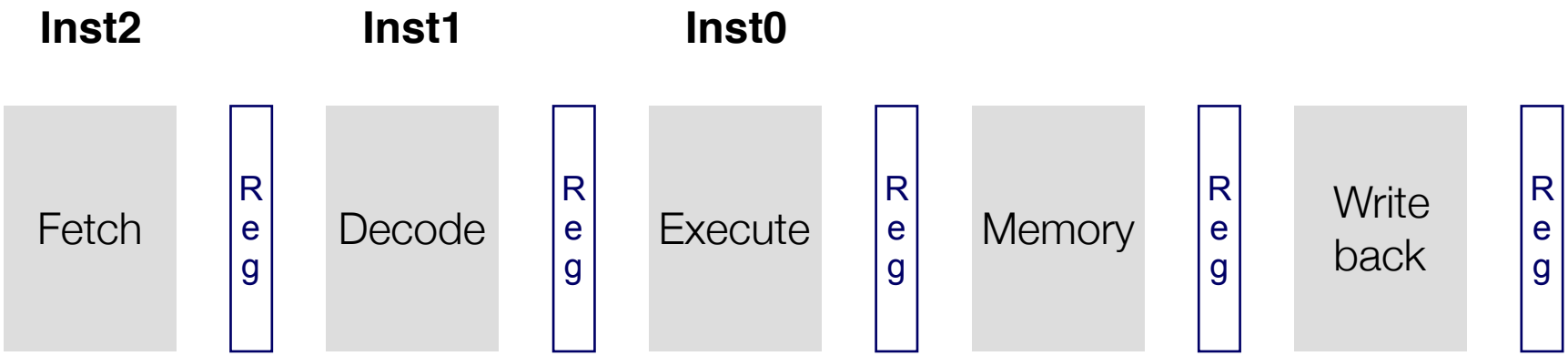
Pipeline Illustration

Inst0

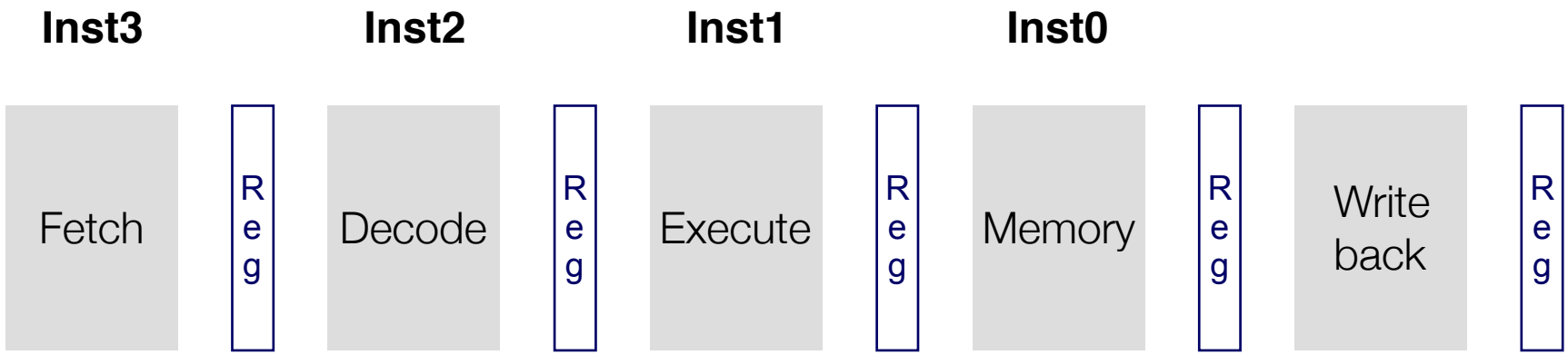
Pipeline Illustration



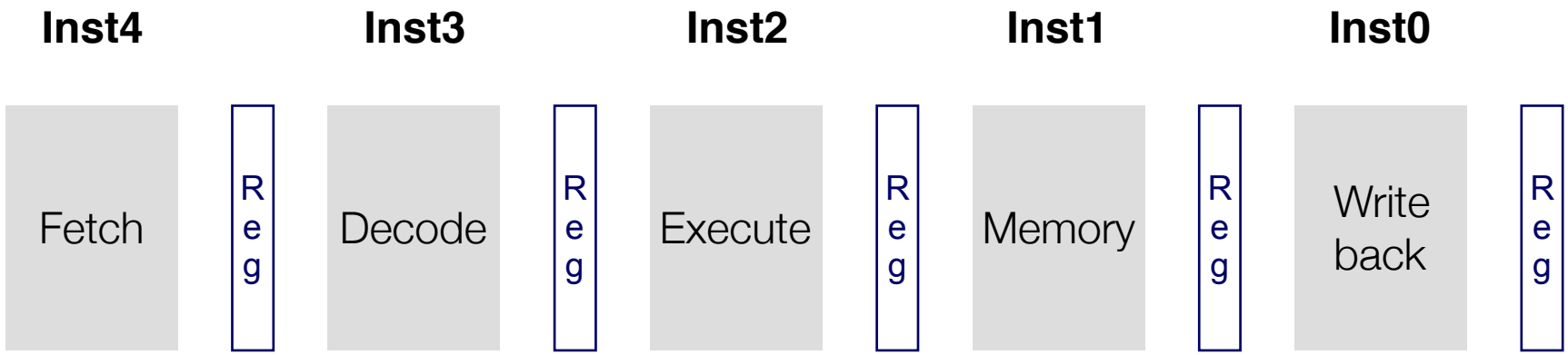
Pipeline Illustration



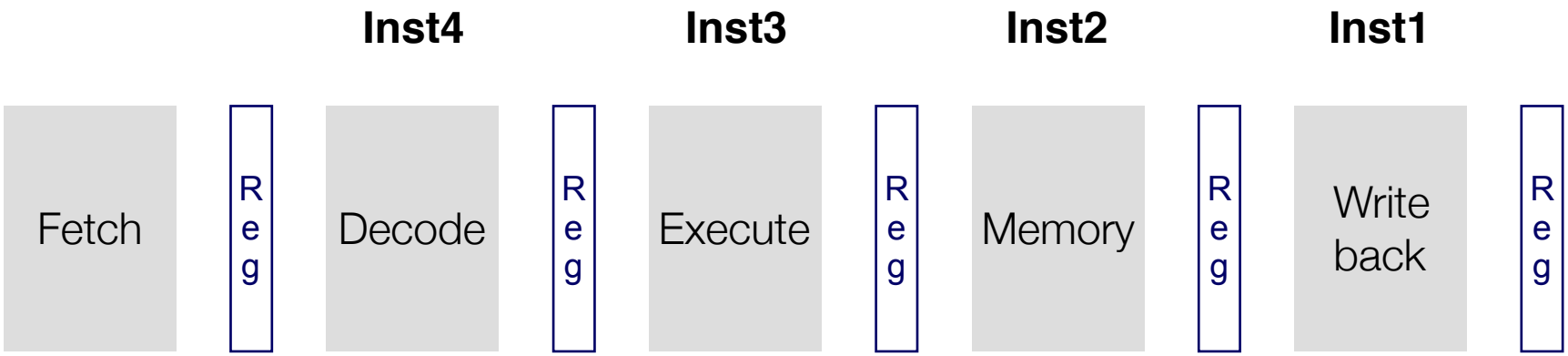
Pipeline Illustration



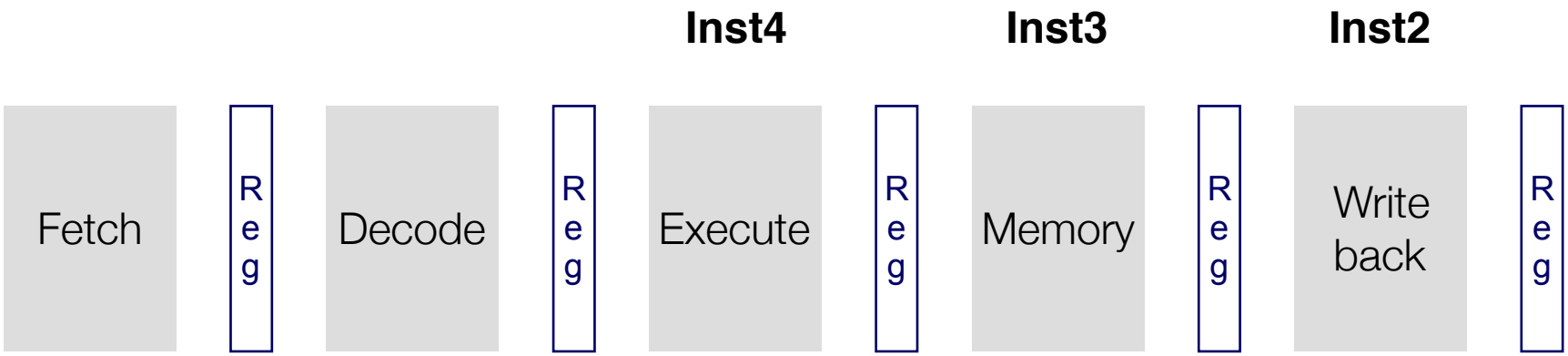
Pipeline Illustration



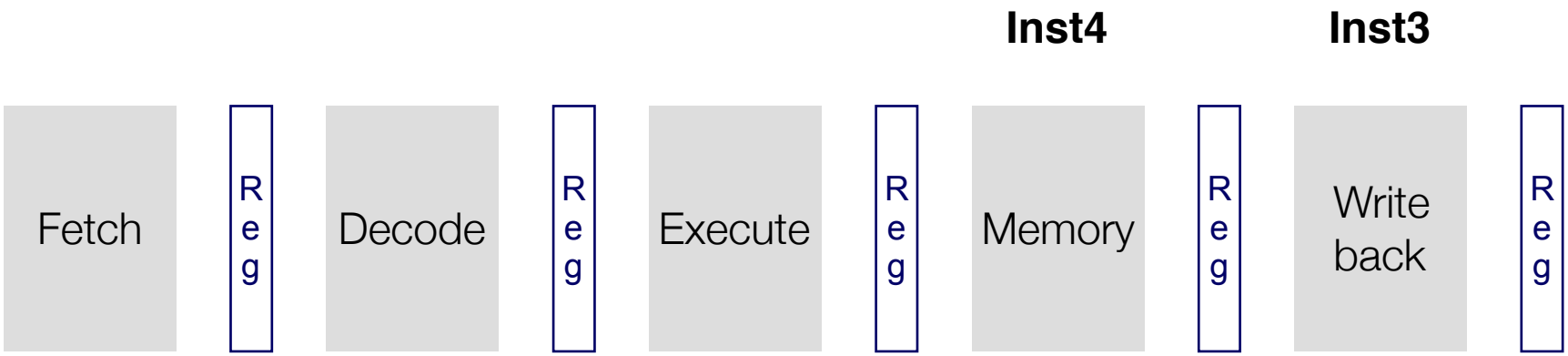
Pipeline Illustration



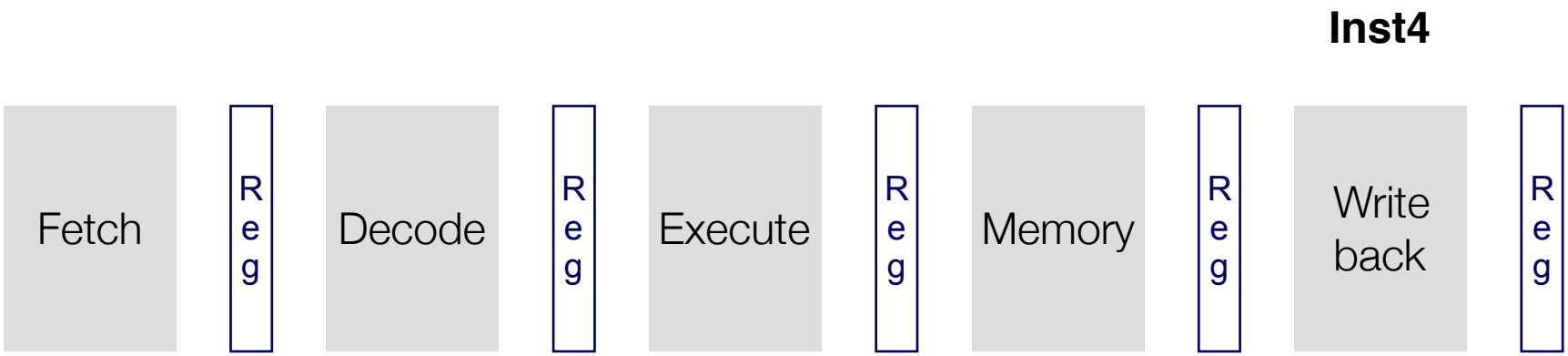
Pipeline Illustration



Pipeline Illustration



Pipeline Illustration



Today: Making the Pipeline Really Work

- Control Dependencies
  - What is it?
  - Software mitigation: Inserting Nops
  - Software mitigation: Delay Slots
- Data Dependencies
  - What is it?
  - Software mitigation: Inserting Nops

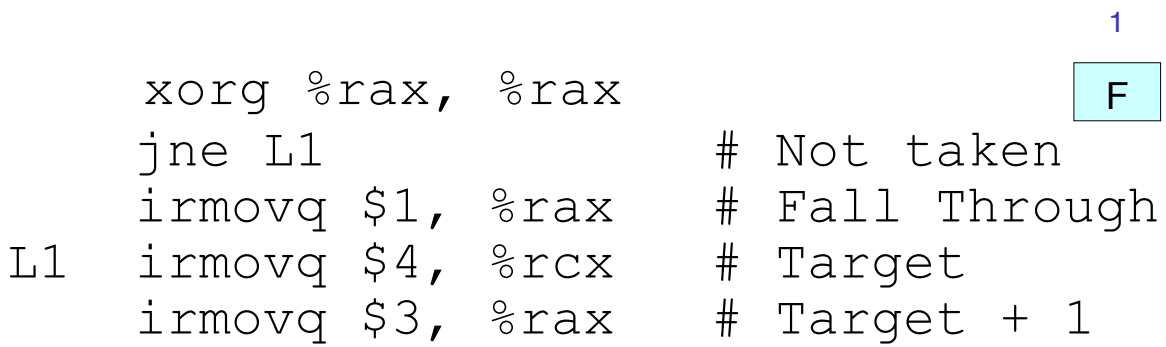
Control Dependency

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

Control Dependency

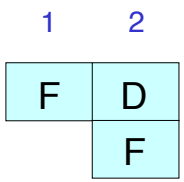
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



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

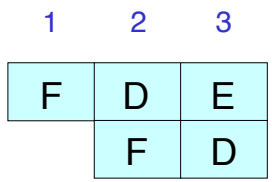
```
     xorg %rax, %rax
     jne L1              # Not taken
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```



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

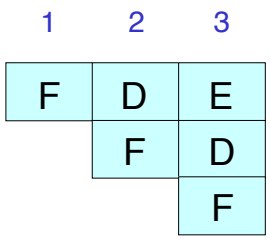
```
    xorg %rax, %rax
    jne L1              # Not taken
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



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

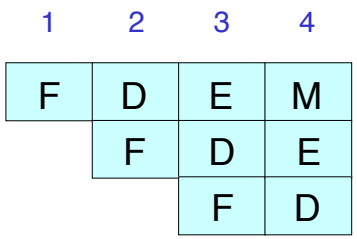
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



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

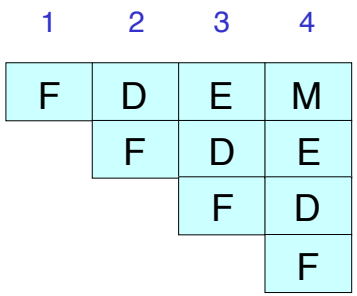
```
     xorg %rax, %rax
     jne L1              # Not taken
     nop
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```



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

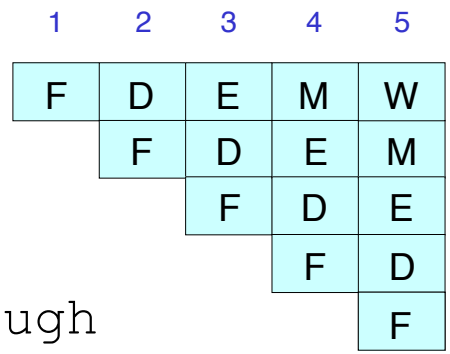
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



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

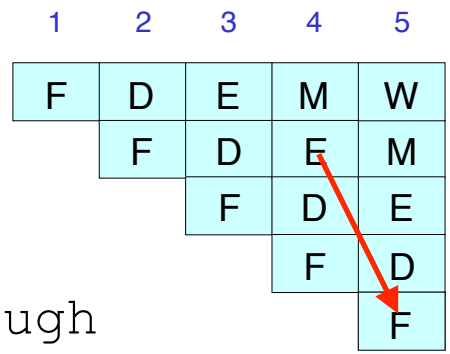
```
     xorg %rax, %rax
     jne L1              # Not taken
     nop
     nop
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```



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

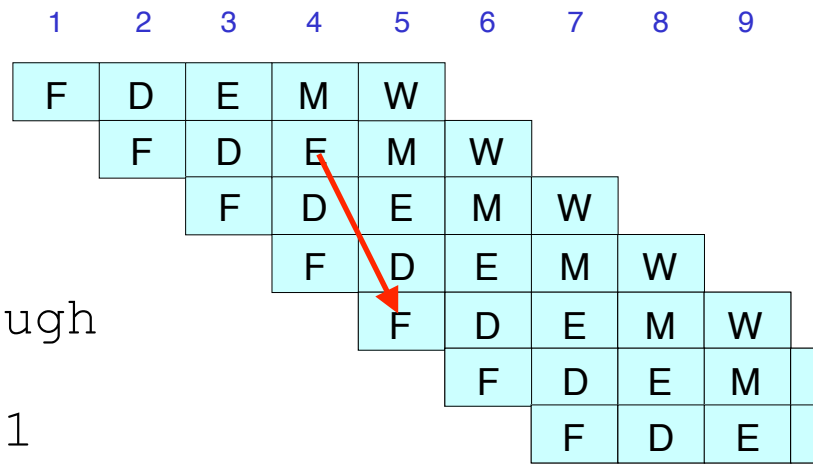
```
     xorg %rax, %rax
     jne L1                 # Not taken
     nop
     nop
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```



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

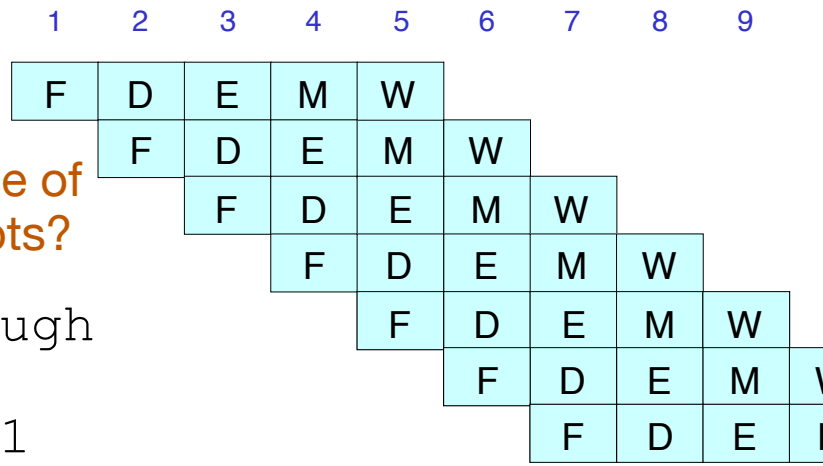
```
     xorg %rax, %rax
     jne L1                 # Not taken
     nop
     nop
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```



Delay Slots

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

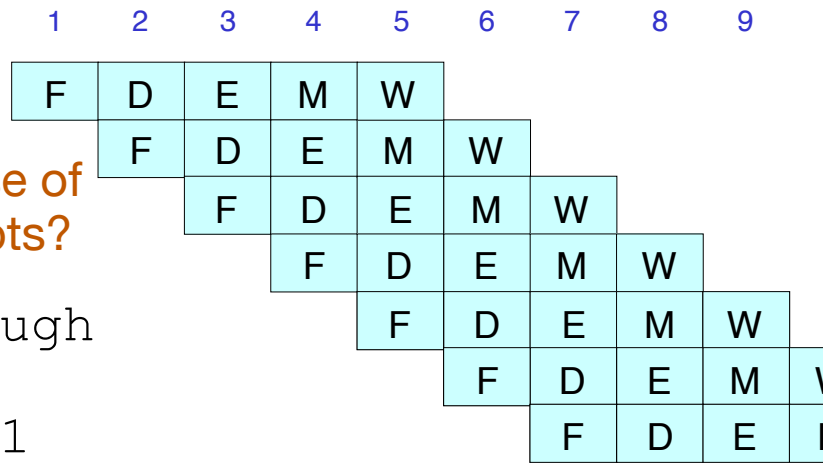
Can we make use of the 2 wasted slots?



Delay Slots

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

Can we make use of the 2 wasted slots?

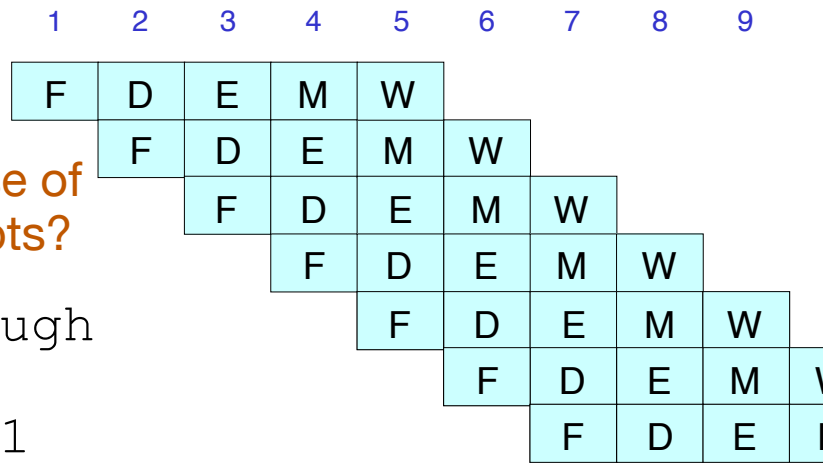



```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```

Delay Slots

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

Can we make use of the 2 wasted slots?



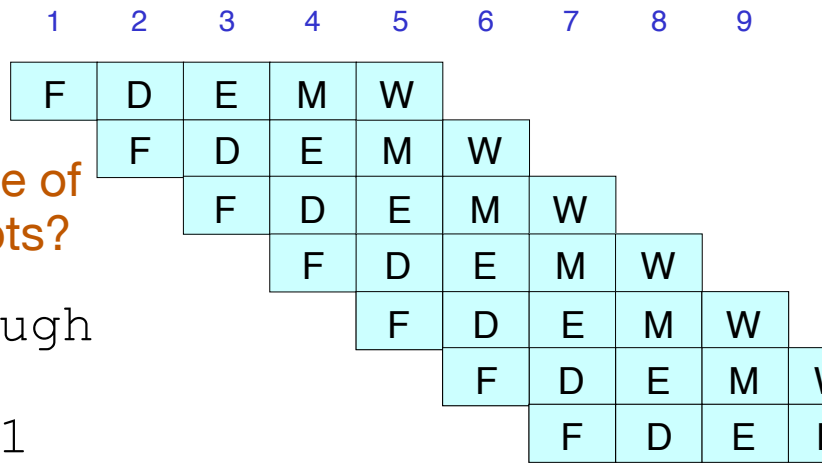
Have to make sure `do_C` doesn't depend on `do_A` and `do_B!!!`

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

Delay Slots

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

Can we make use of the 2 wasted slots?



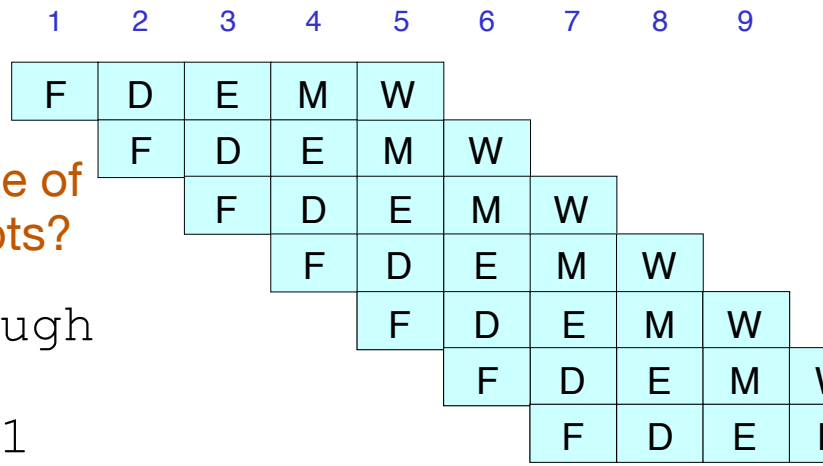
A less obvious example

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

Delay Slots

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

Can we make use of the 2 wasted slots?



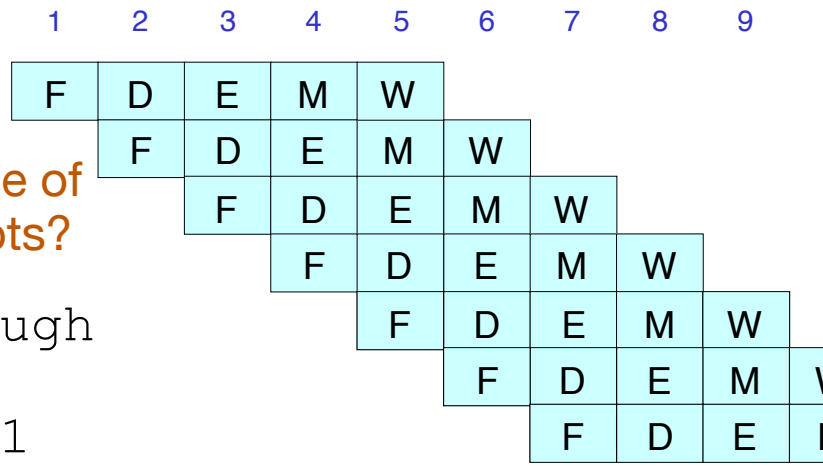
A less obvious example

```
do_C();                 |  add A, B
if (cond) {             |  or C, D
  do_A();               |  sub E, F
} else {                |  jle 0x200
  do_B();               |  add A, C
}                       |
```

Delay Slots

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

Can we make use of the 2 wasted slots?



A less obvious example

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

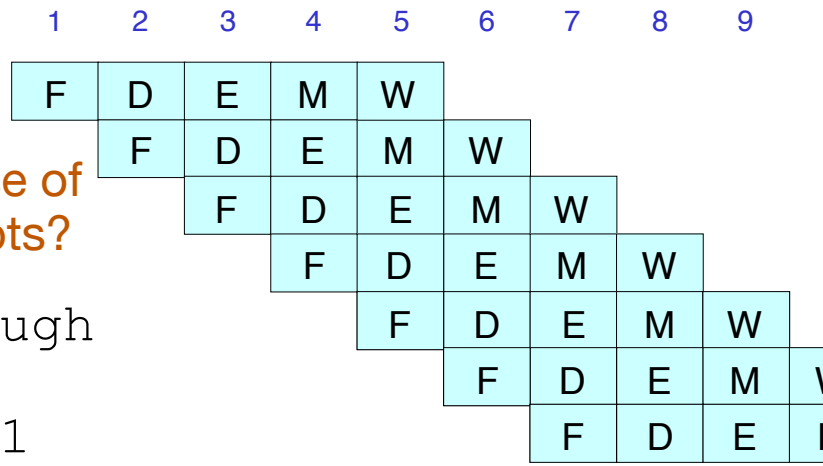
```
add A, B
or C, D
sub E, F
jle 0x200
add A, C
```

```
add A, B
sub E, F
jle 0x200
or C, D
add A, C
```

Delay Slots

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

Can we make use of the 2 wasted slots?



A less obvious example

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

```
add A, B
or C, D
sub E, F
jle 0x200
add A, C
```

```
add A, B
sub E, F
jle 0x200
or C, D
add A, C
```

Why don't we move the `sub` instruction?

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

Branch Prediction

Static Prediction

- Always Taken
- Always Not-taken

Dynamic Prediction

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

If prediction is correct: pipeline moves forward without stalling

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

Static Prediction

Static Prediction

Observation: 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: 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: 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: 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: 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: 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

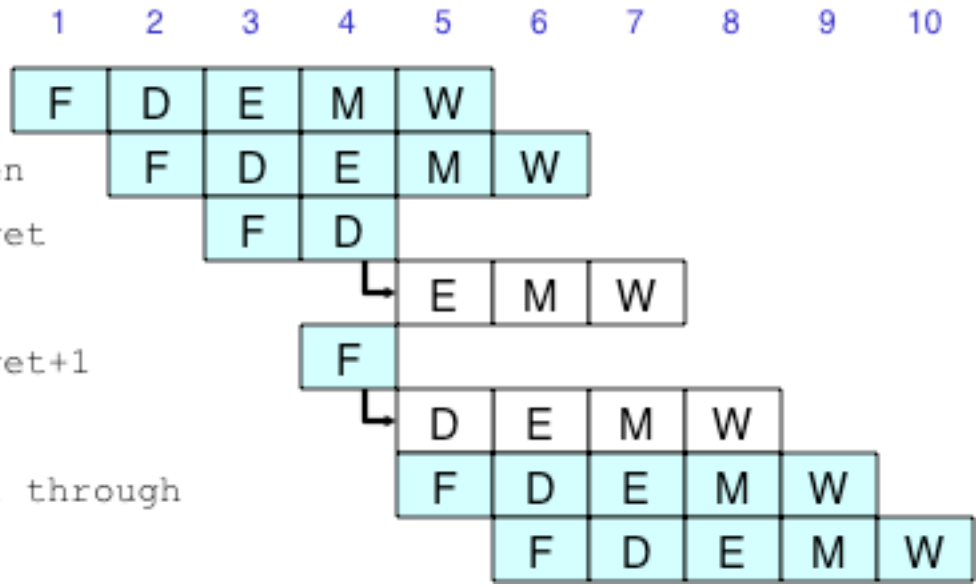
- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

`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

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

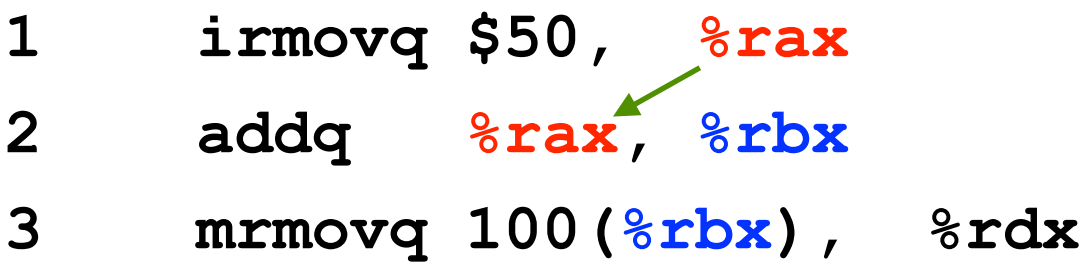
Today: Making the Pipeline Really Work

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

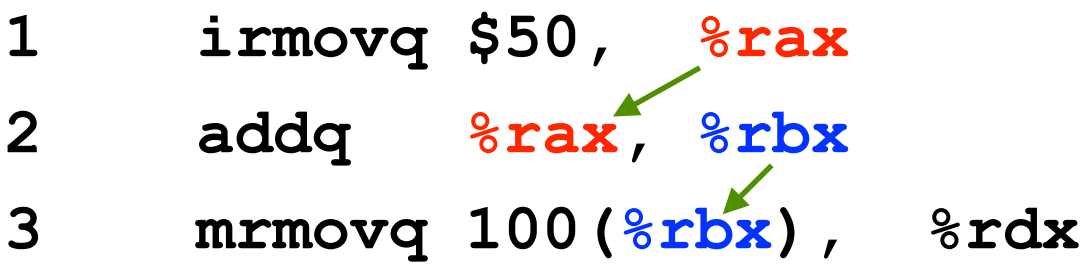
Data Dependencies

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

Data Dependencies

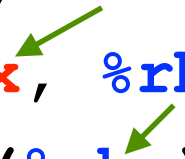


Data Dependencies



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???**

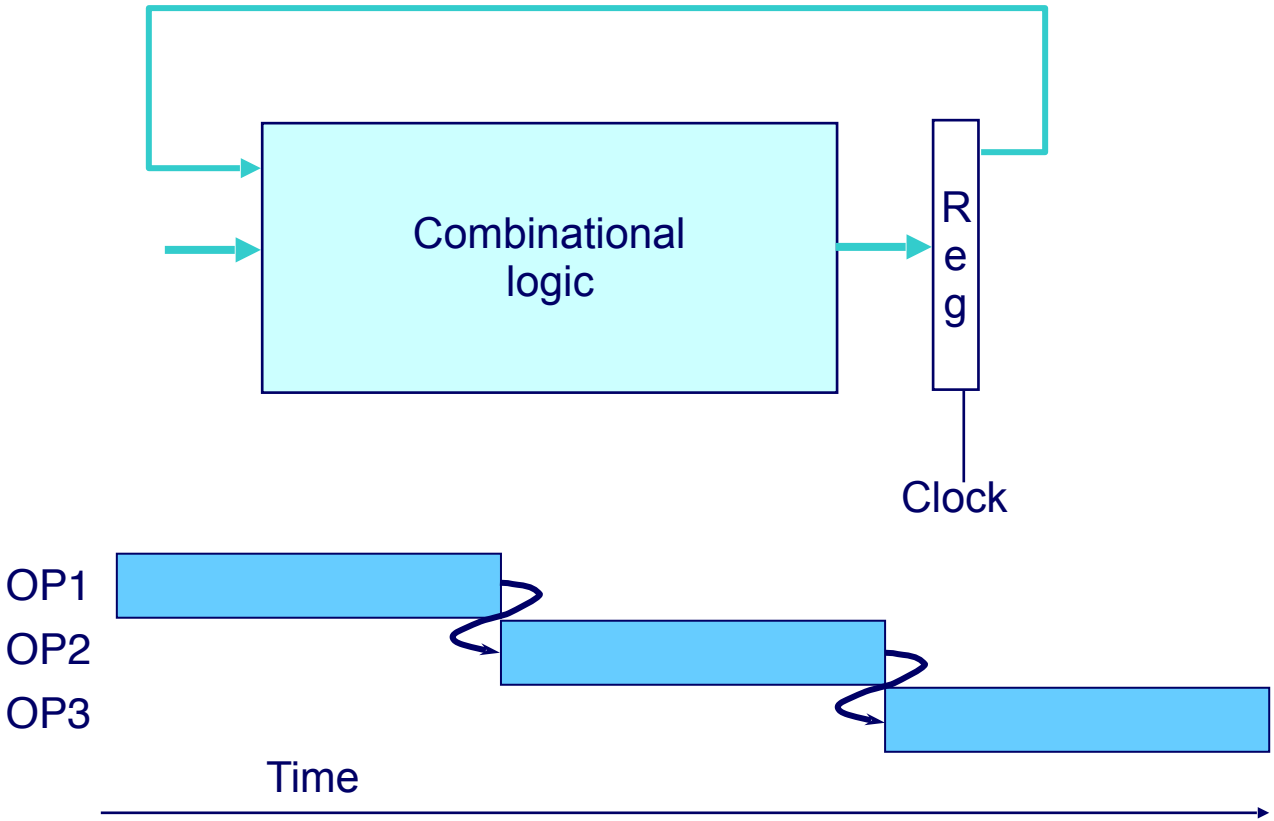
```
    xorg %rax, %rax
    jne L1              # Not taken
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```

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
```

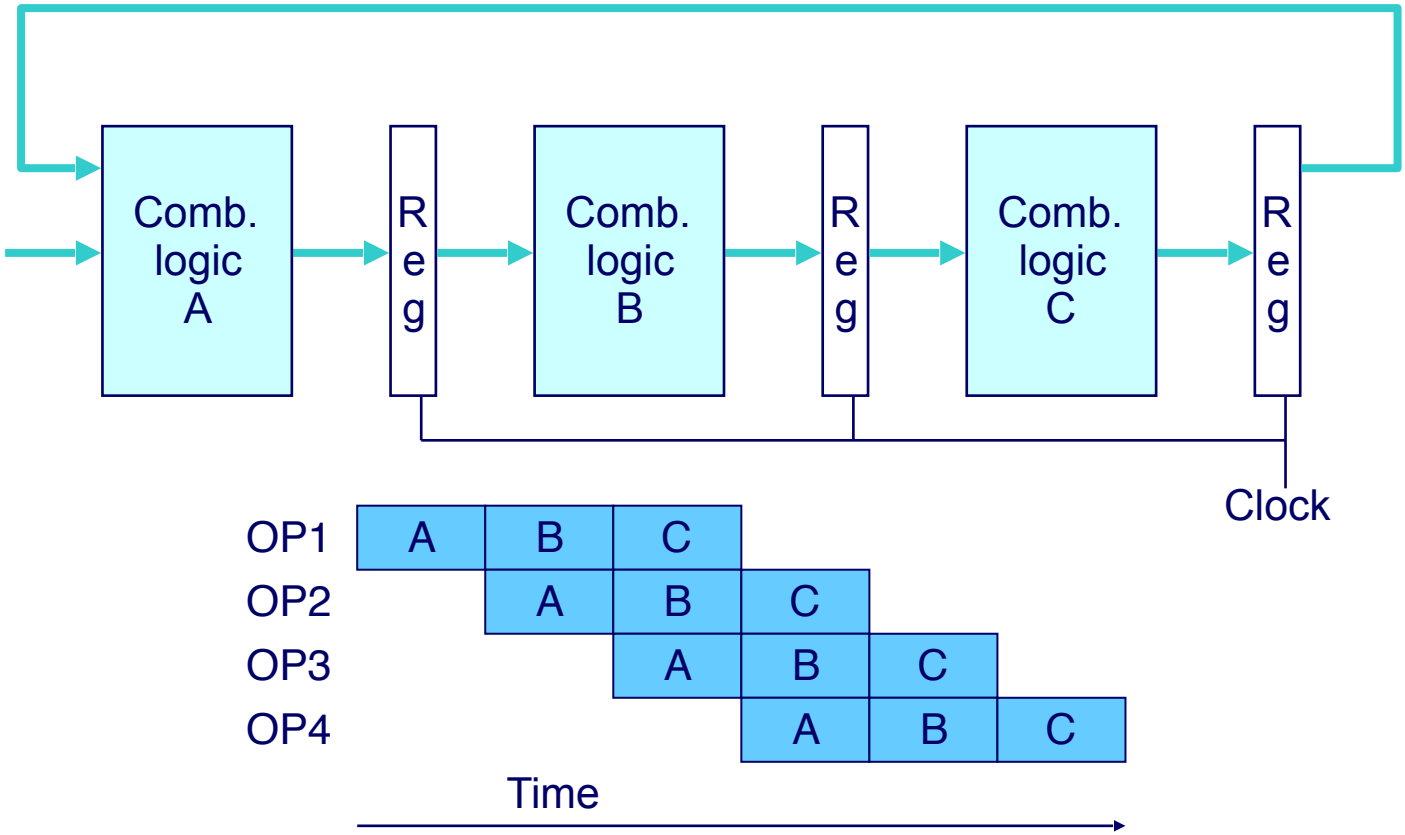
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

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

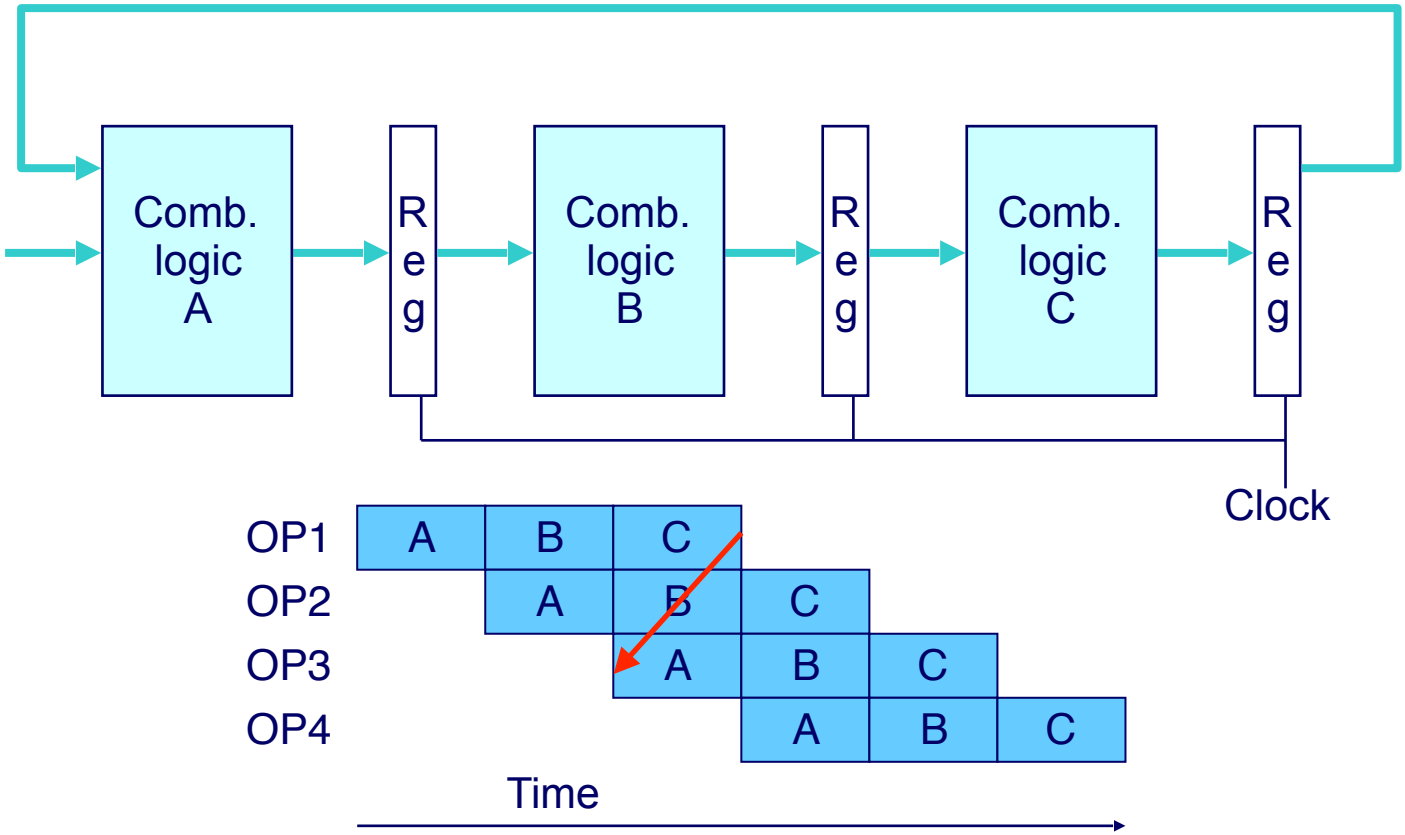
Data Dependencies in Pipeline Machines



Data Hazards happen when:

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

Data Dependencies in Pipeline Machines

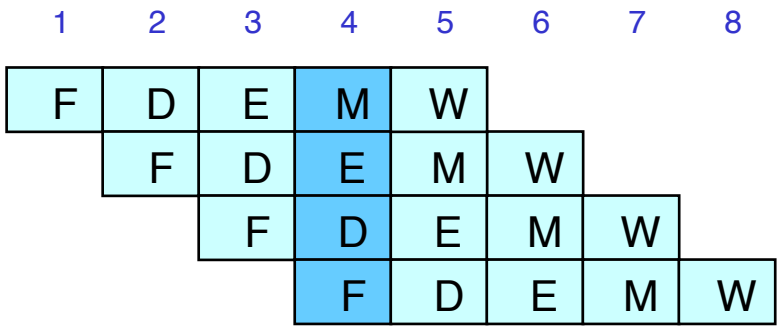


Data Hazards happen when:

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

Data Dependencies: No Nop

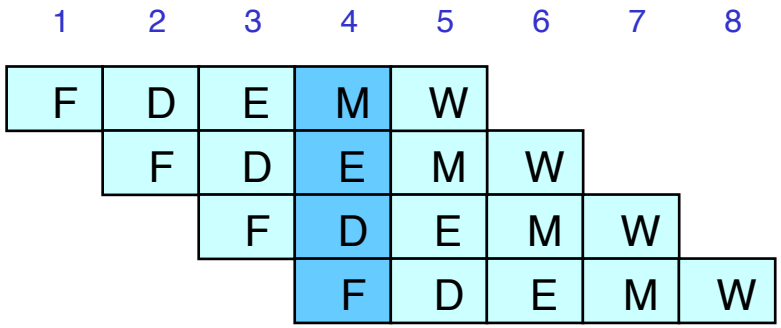
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



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
```

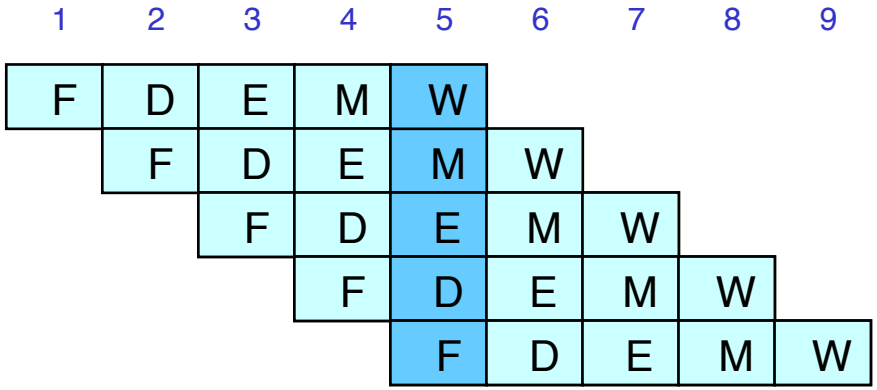


Remember registers get updated in the Write-back stage

addq reads wrong %rdx and %rax

Data Dependencies: 1 Nop

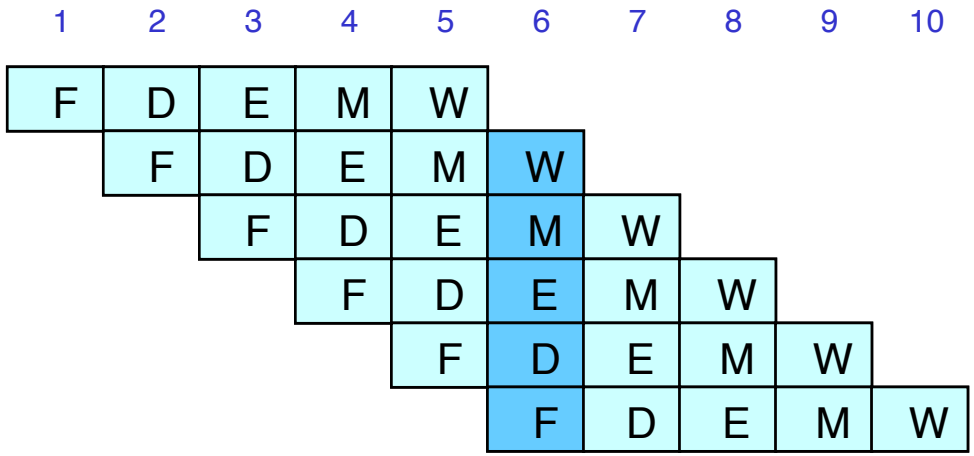
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



addq still reads wrong %rdx and %rax

Data Dependencies: 2 Nop's

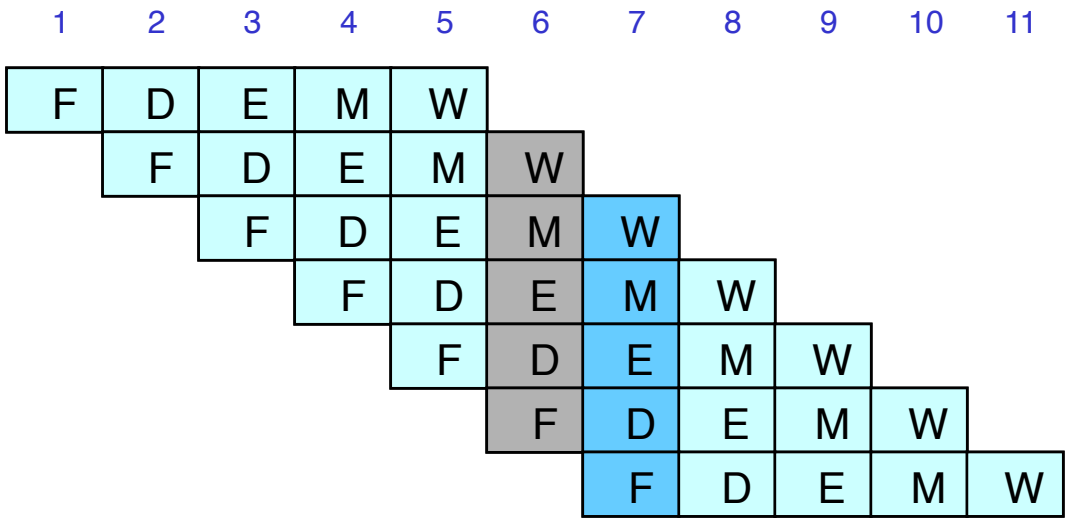
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



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

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?

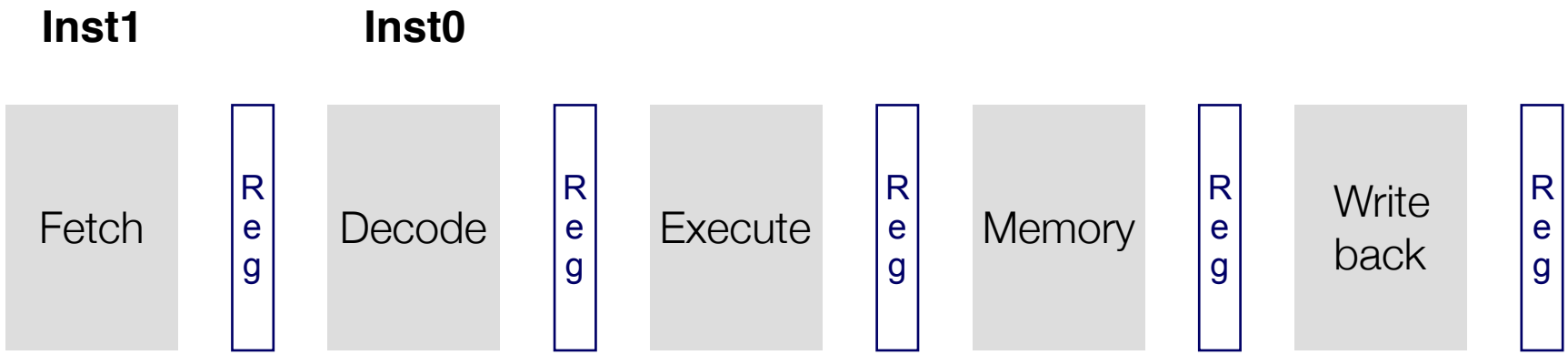
- Why is it good for the hardware to do so anyways?

Inst0

Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

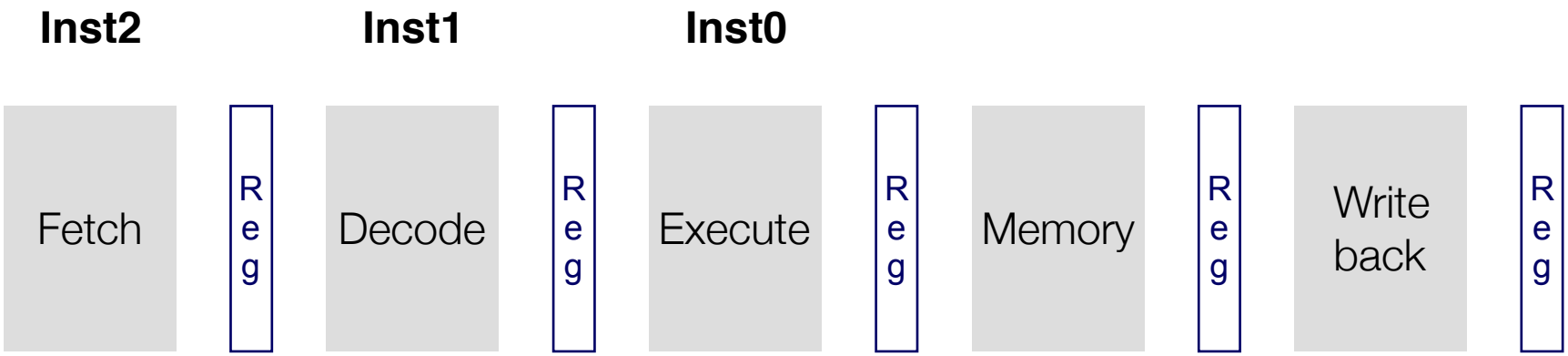
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

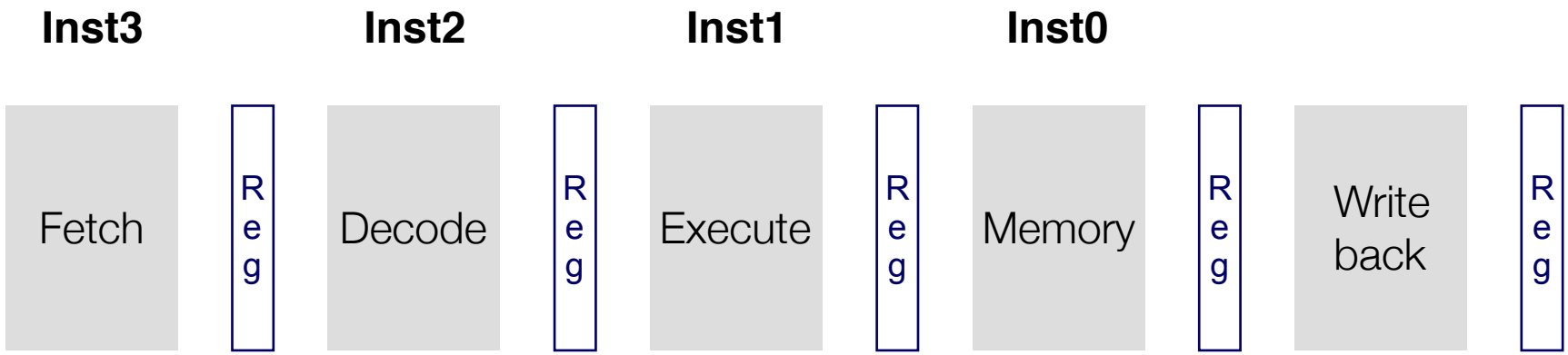
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

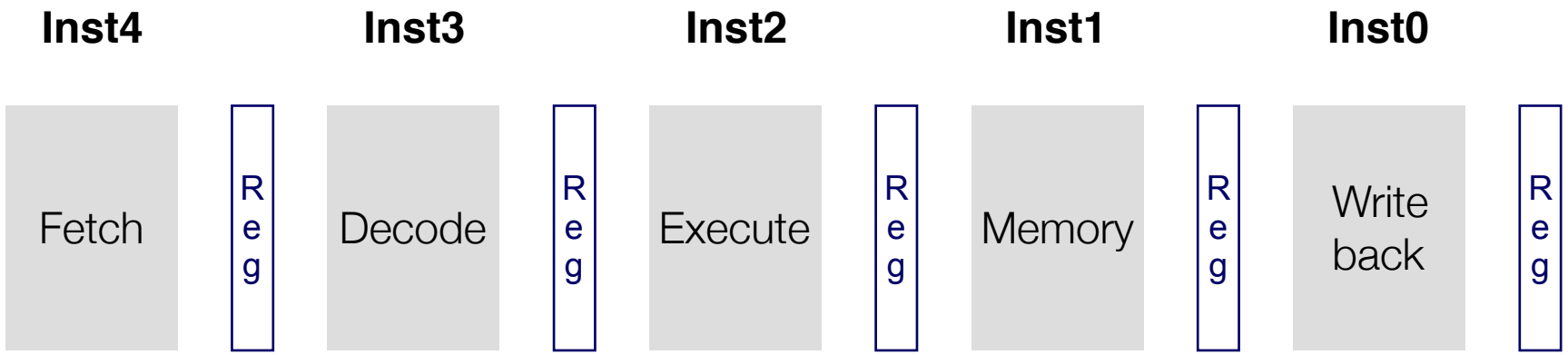
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

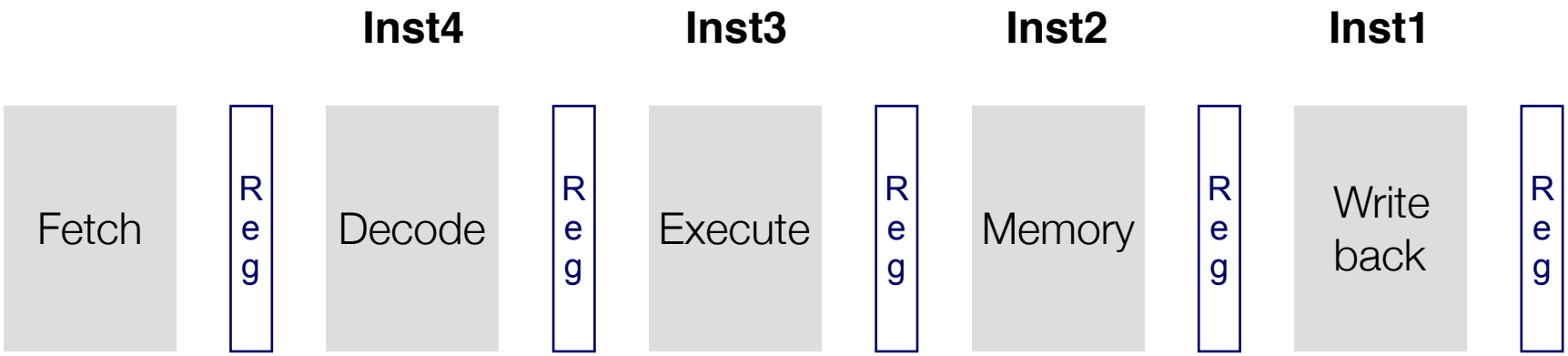
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

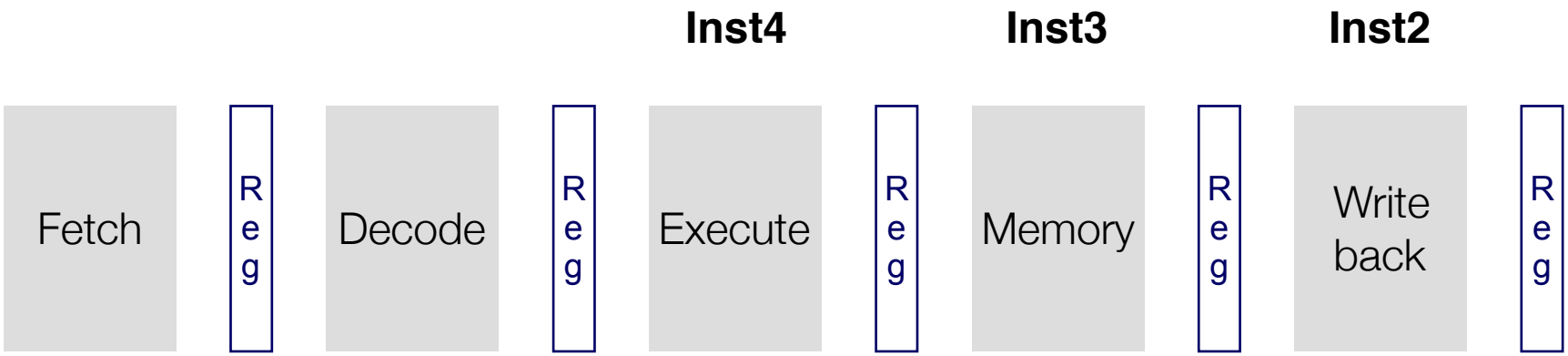
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

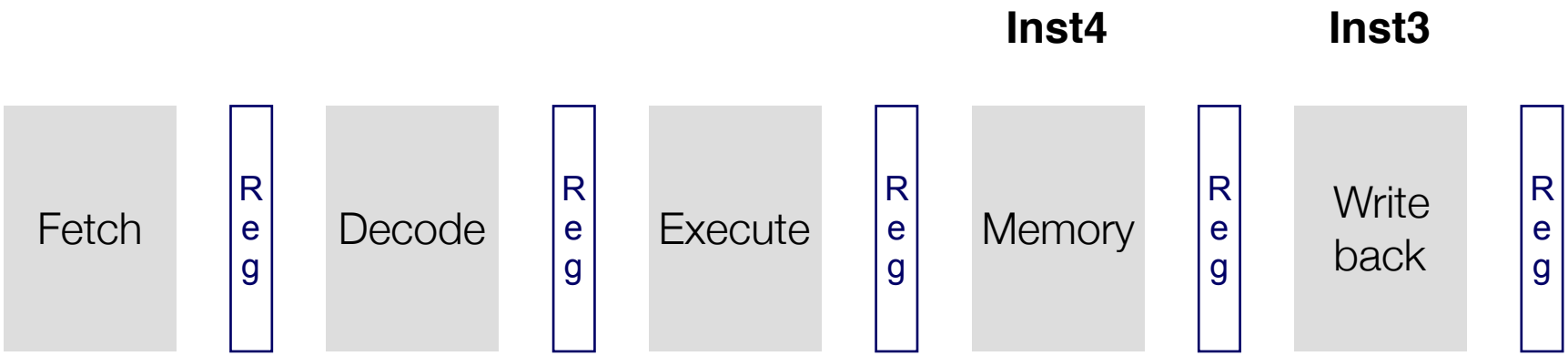
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

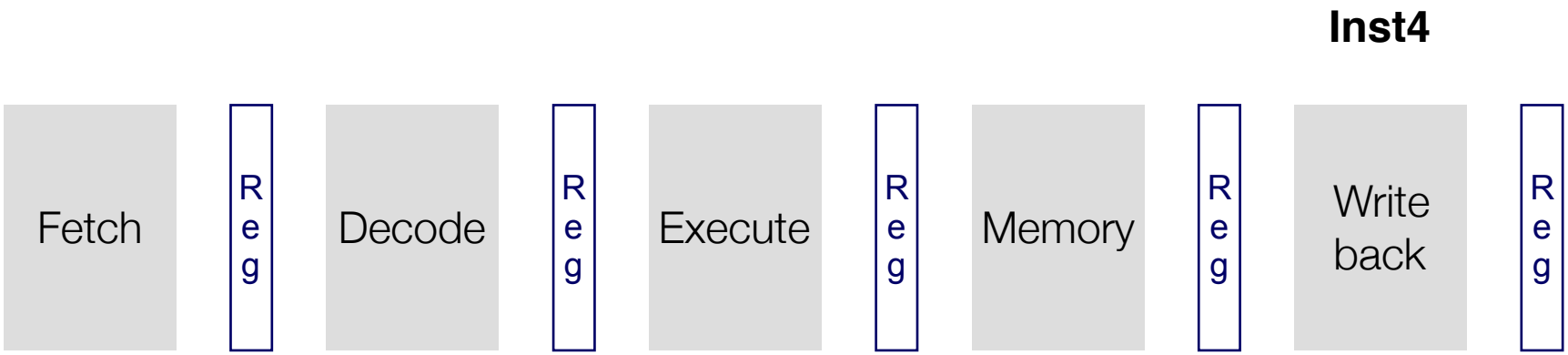
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

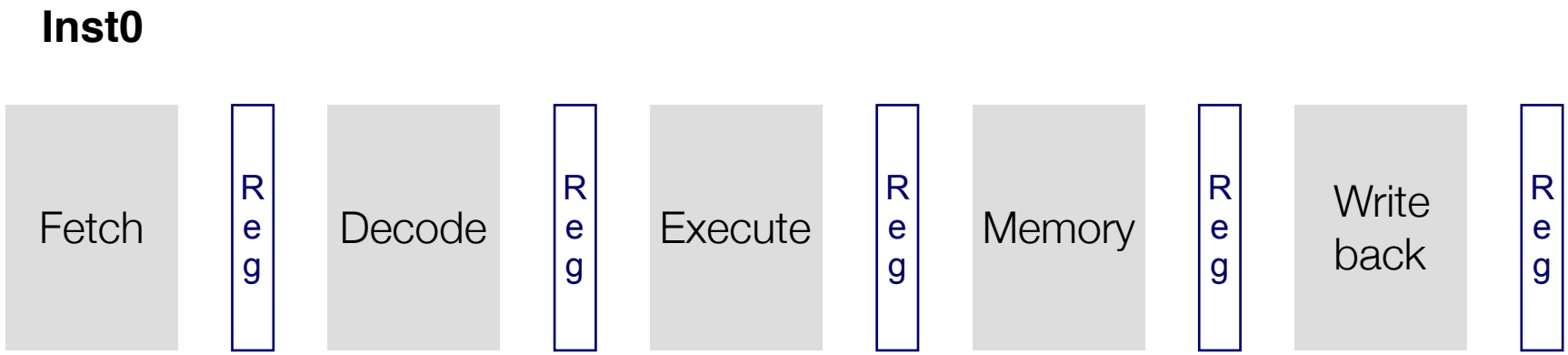
Can we have the hardware automatically generates a nop?

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

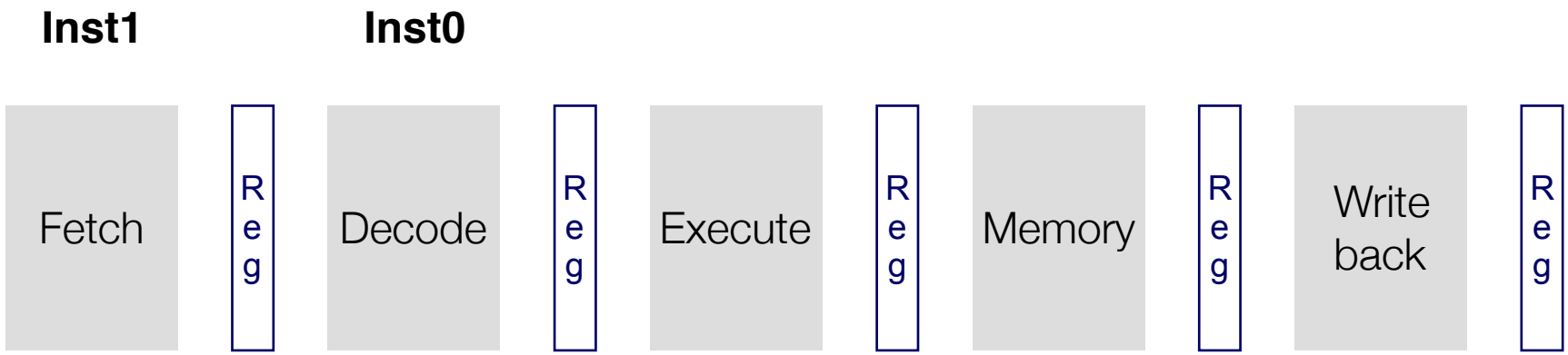


Stalling Illustration

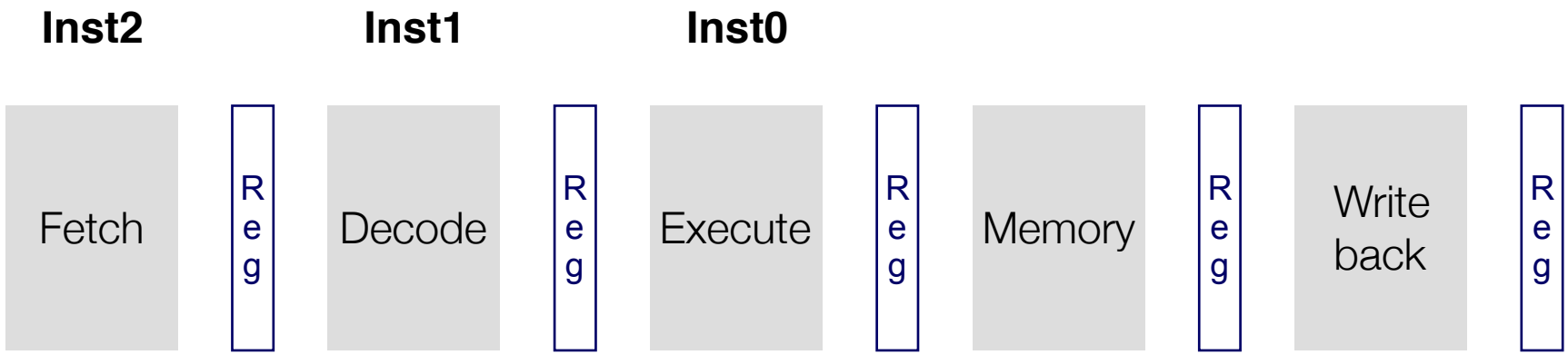
Stalling Illustration



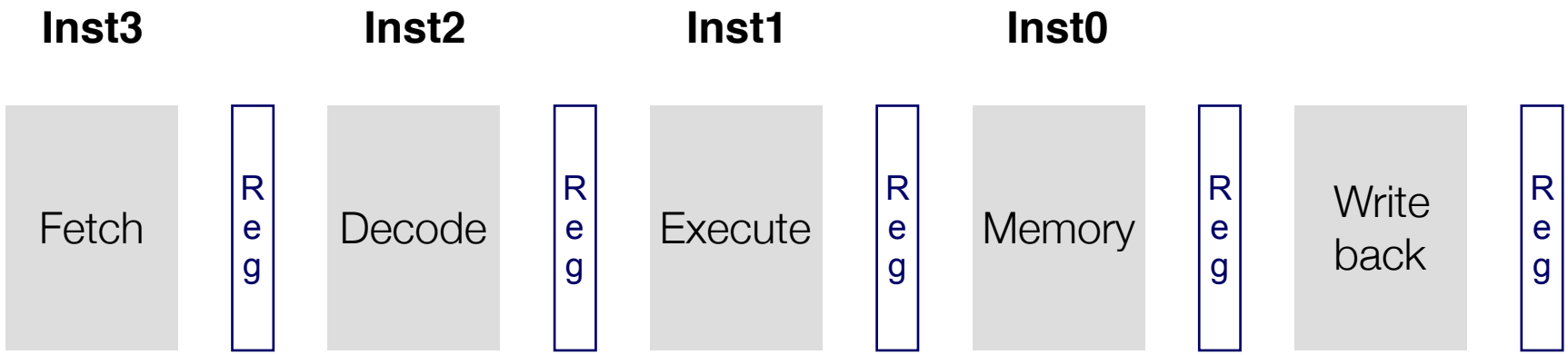
Stalling Illustration



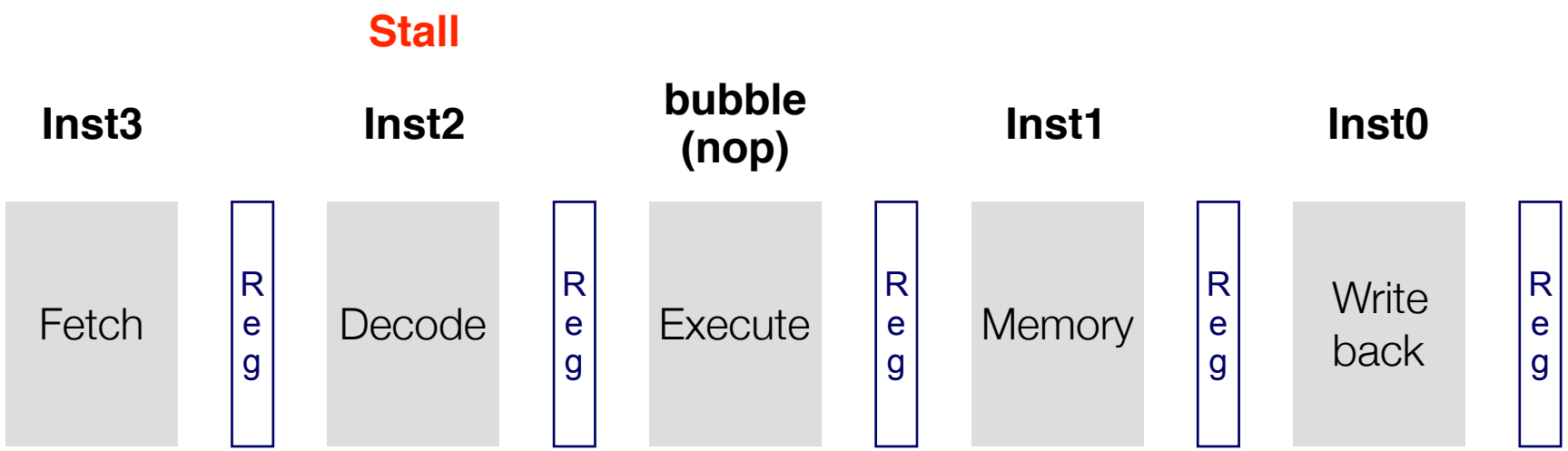
Stalling Illustration



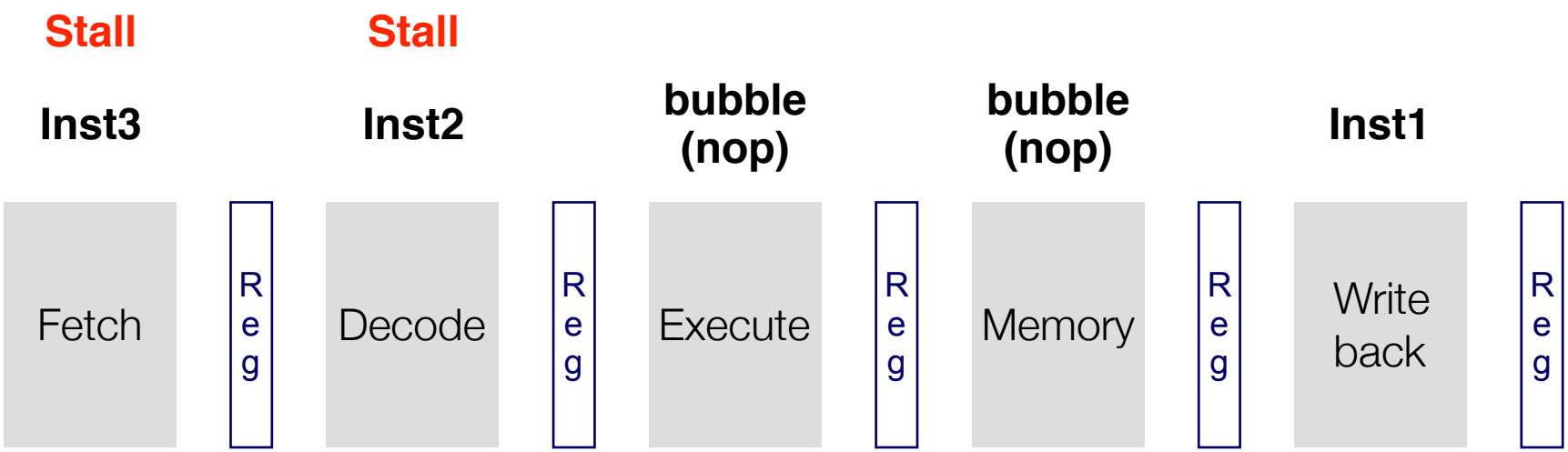
Stalling Illustration



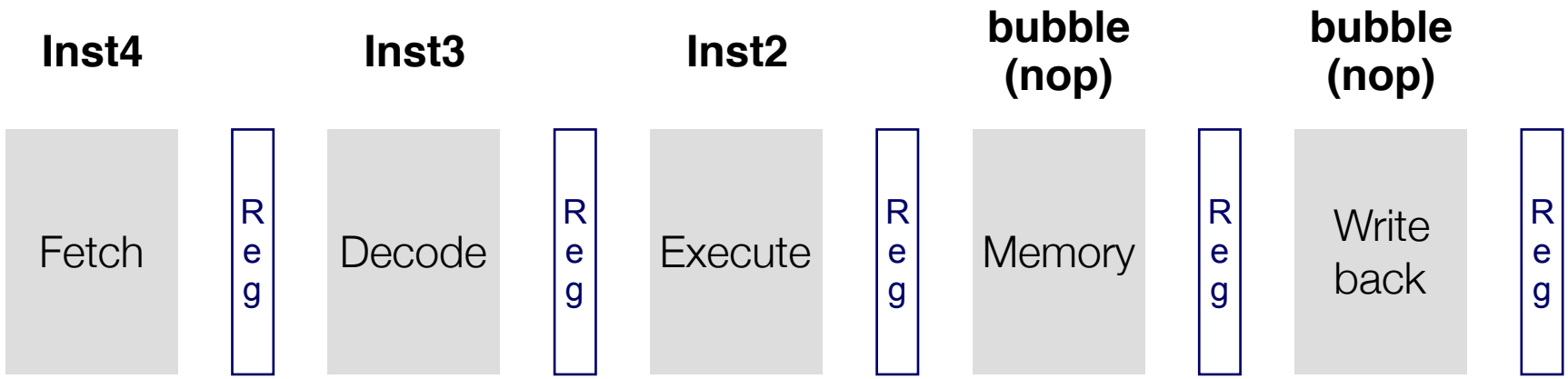
Stalling Illustration



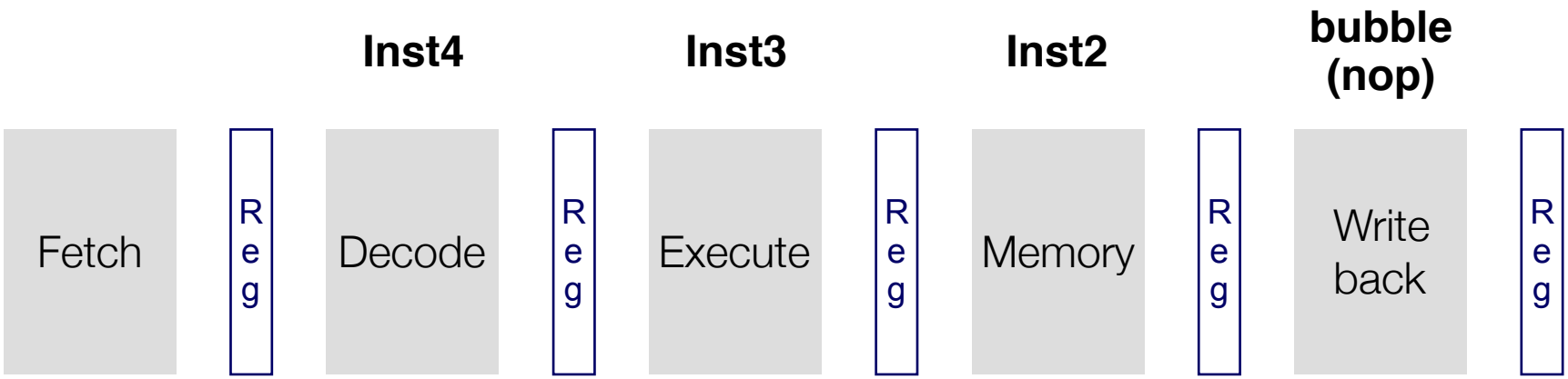
Stalling Illustration



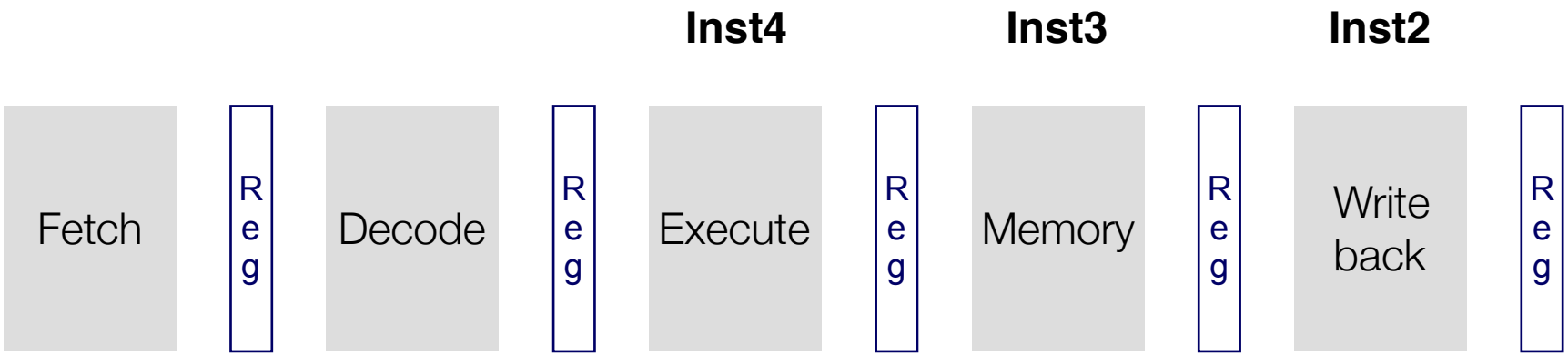
Stalling Illustration



Stalling Illustration

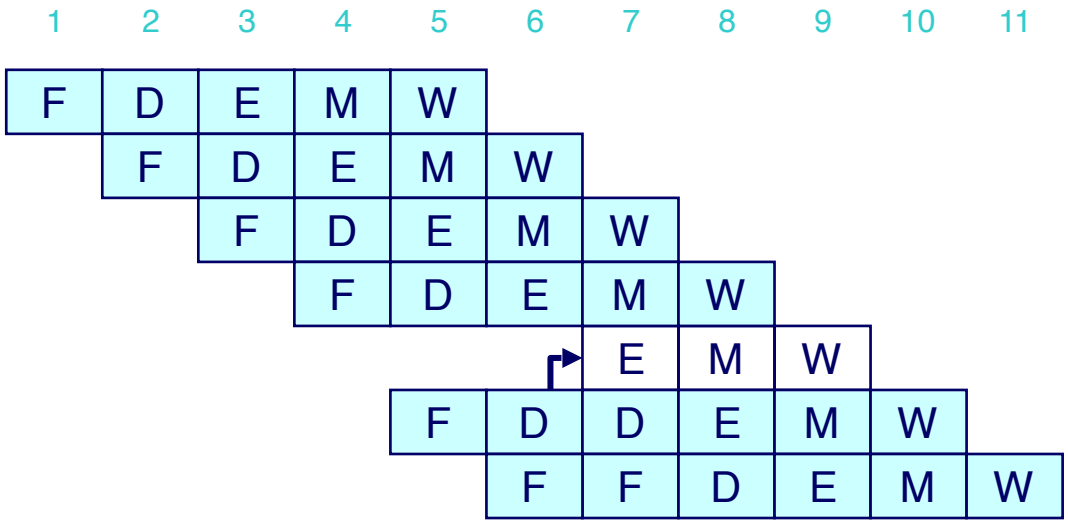


Stalling Illustration



Stalling for Data Dependencies

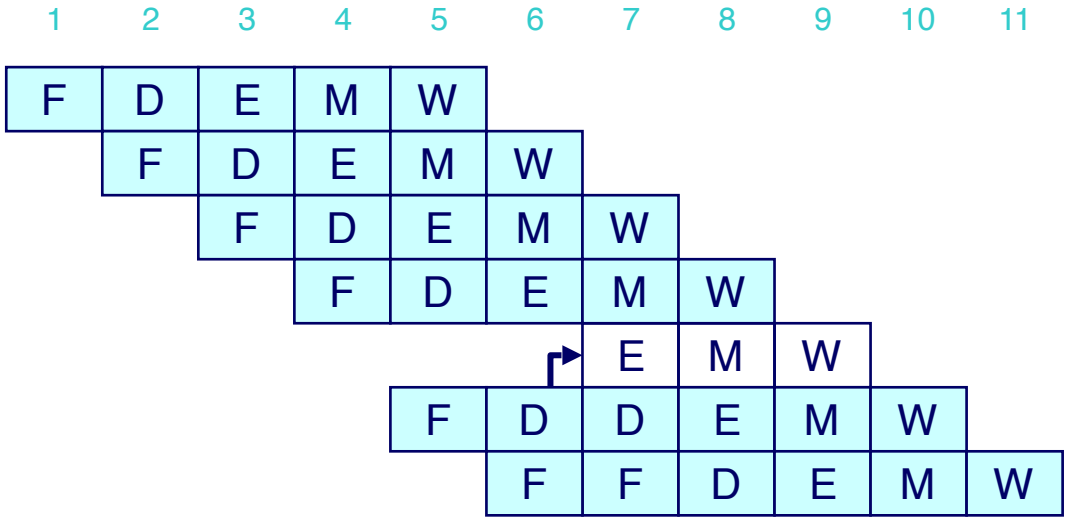
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode

Detecting Stall Condition

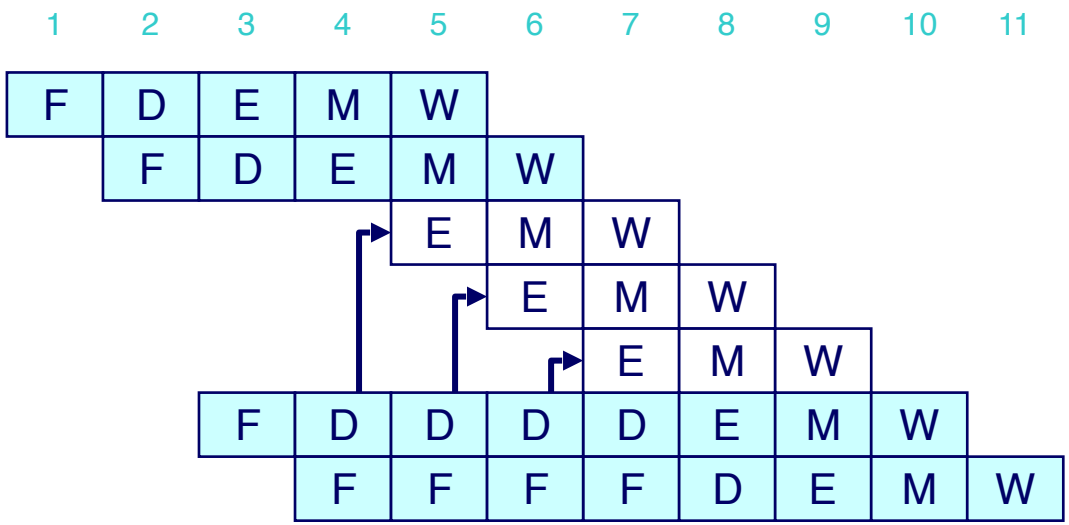
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



- 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!!

Stalling X3

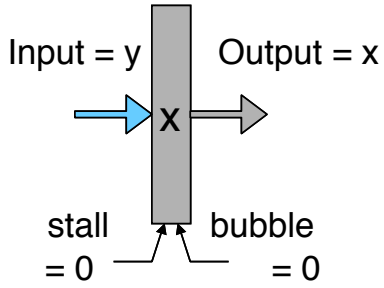
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



How are Stall and Bubble Implemented in Hardware?

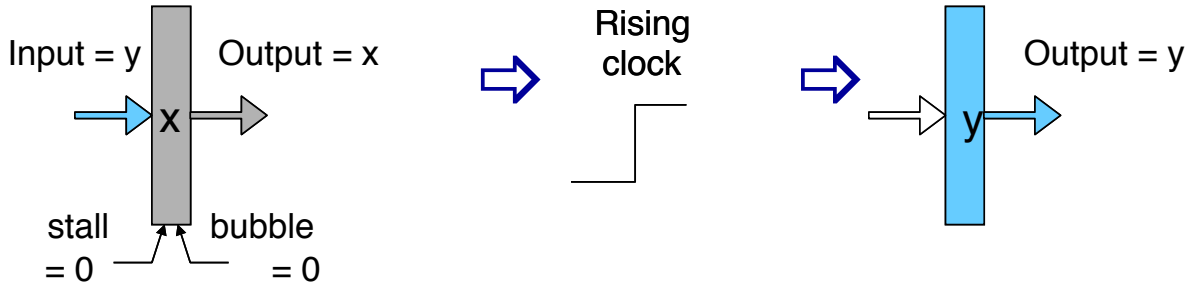
How are Stall and Bubble Implemented in Hardware?

Normal



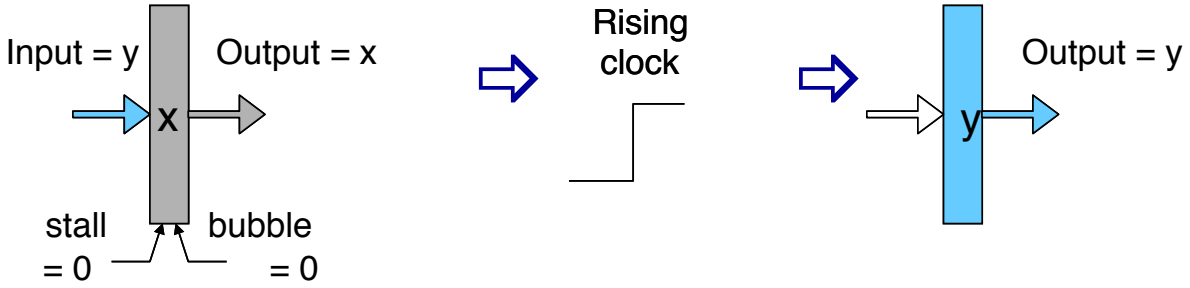
How are Stall and Bubble Implemented in Hardware?

Normal

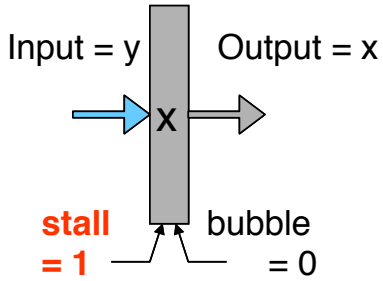


How are Stall and Bubble Implemented in Hardware?

Normal

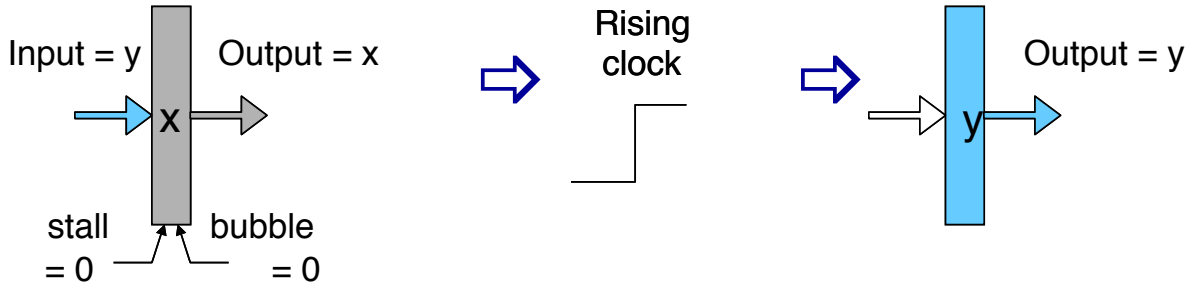


Stall

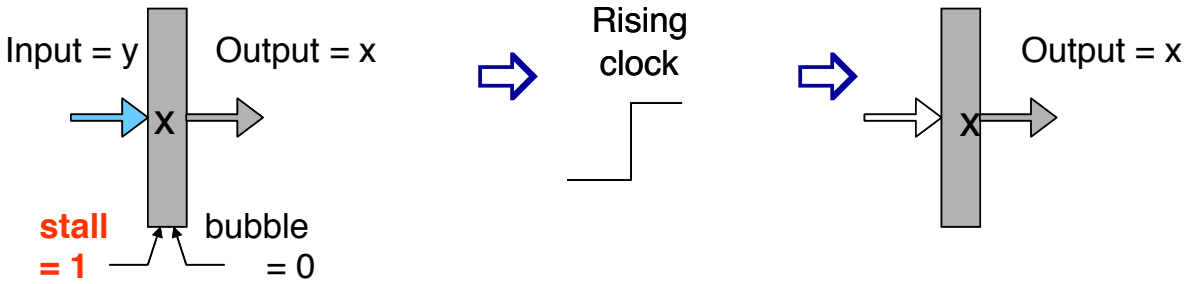


How are Stall and Bubble Implemented in Hardware?

Normal

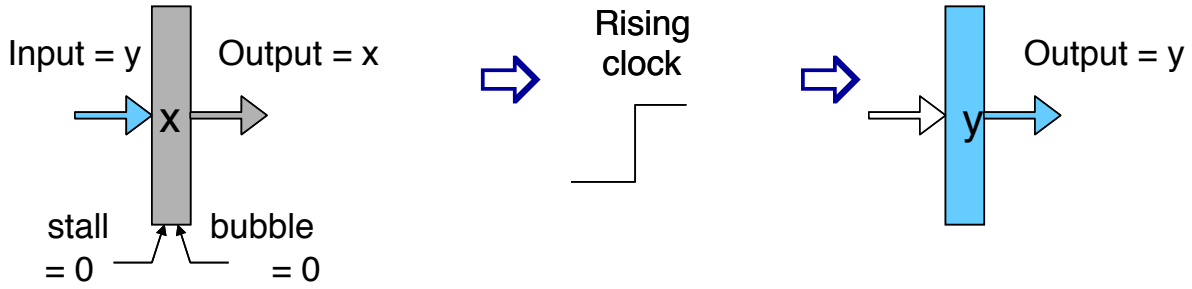


Stall

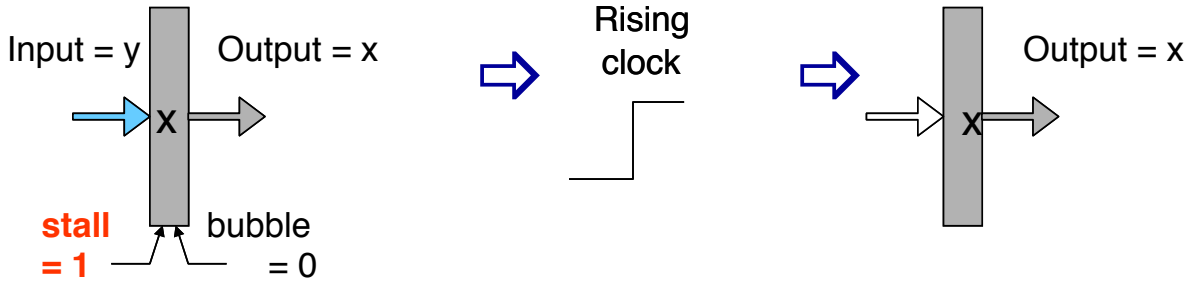


How are Stall and Bubble Implemented in Hardware?

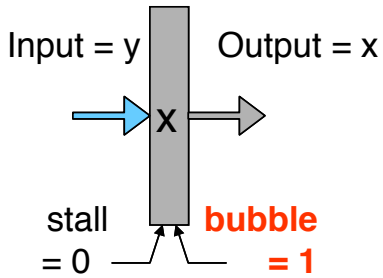
Normal



Stall

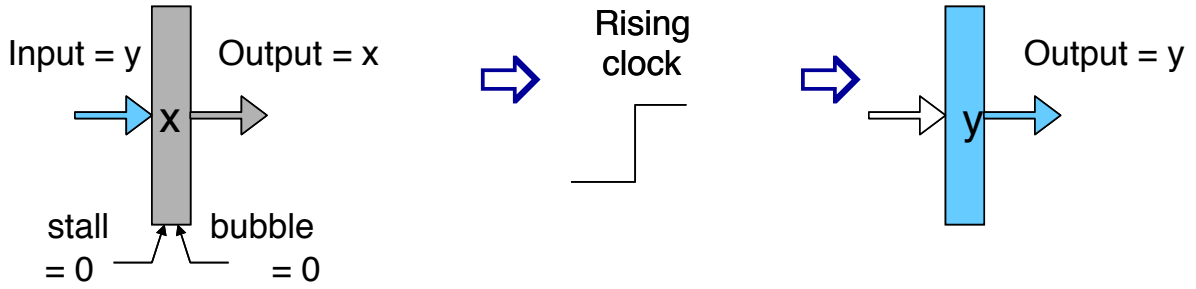


Bubble

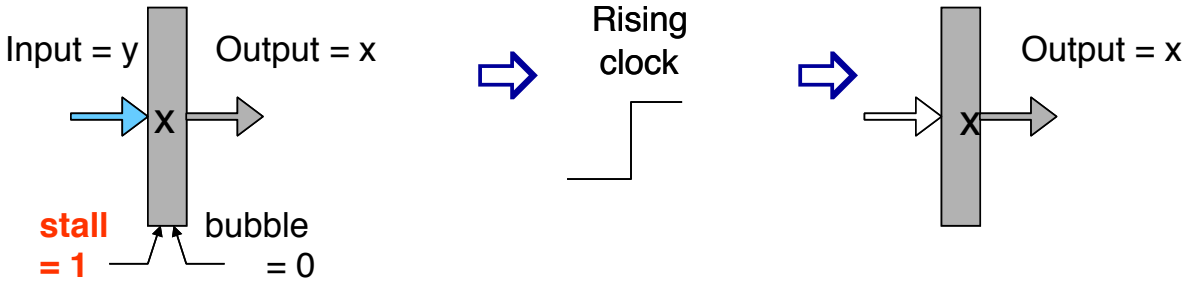


How are Stall and Bubble Implemented in Hardware?

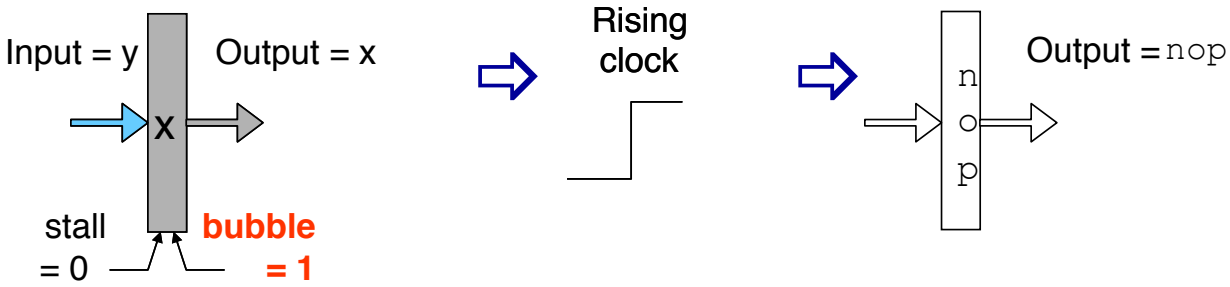
Normal



Stall



Bubble



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

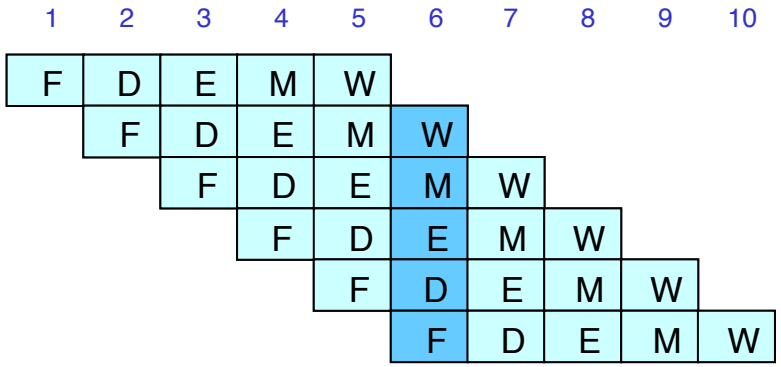
- Value generated in execute or memory stage

Trick

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

Data Forwarding Example

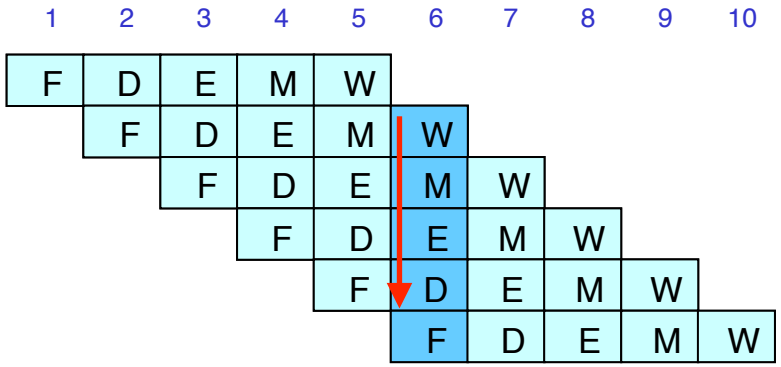
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `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

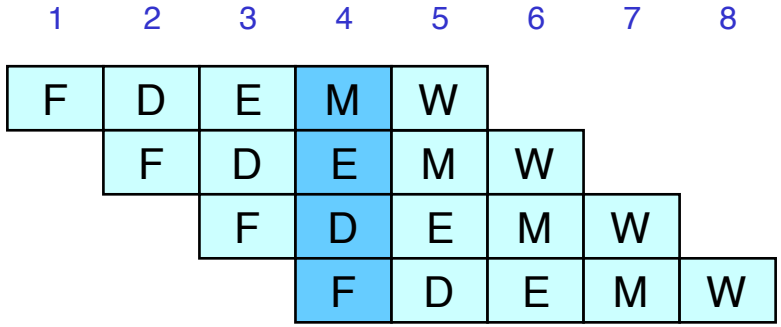
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `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`

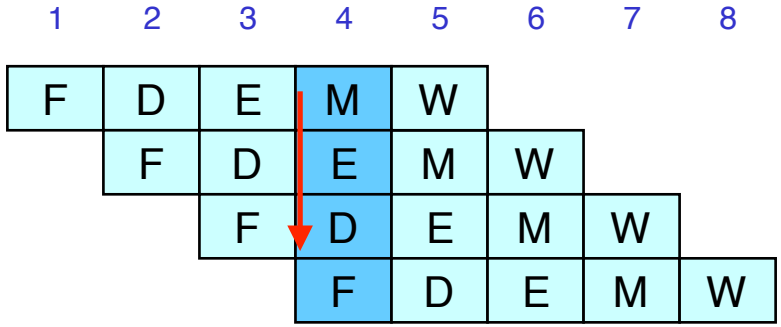
- Forward from the memory stage

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`

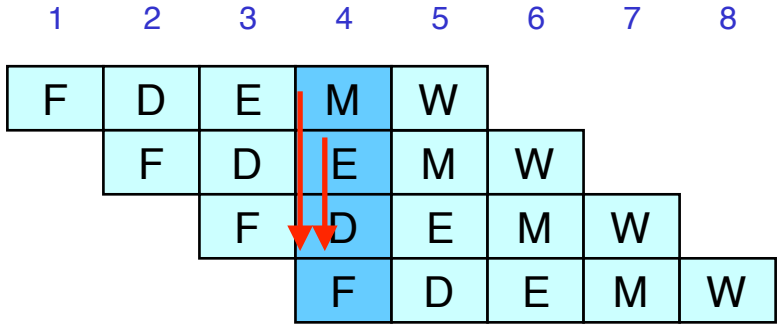
- Forward from the memory stage

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

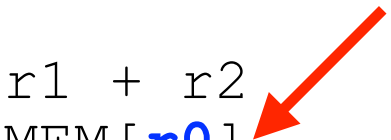
Register `%rax`

- Forward from the execute stage

Out-of-order Execution

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

Long-latency instruction.
Forces the pipeline to stall.



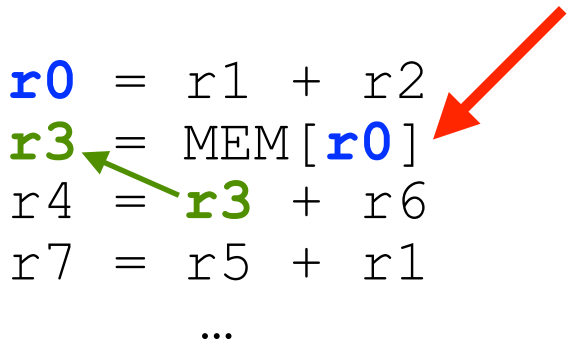
```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r7 = r5 + r1
     …
```

```
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

```
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.