CSC 252: Computer Organization
Spring 2018: Lecture 13

Instructor: Yuhao Zhu

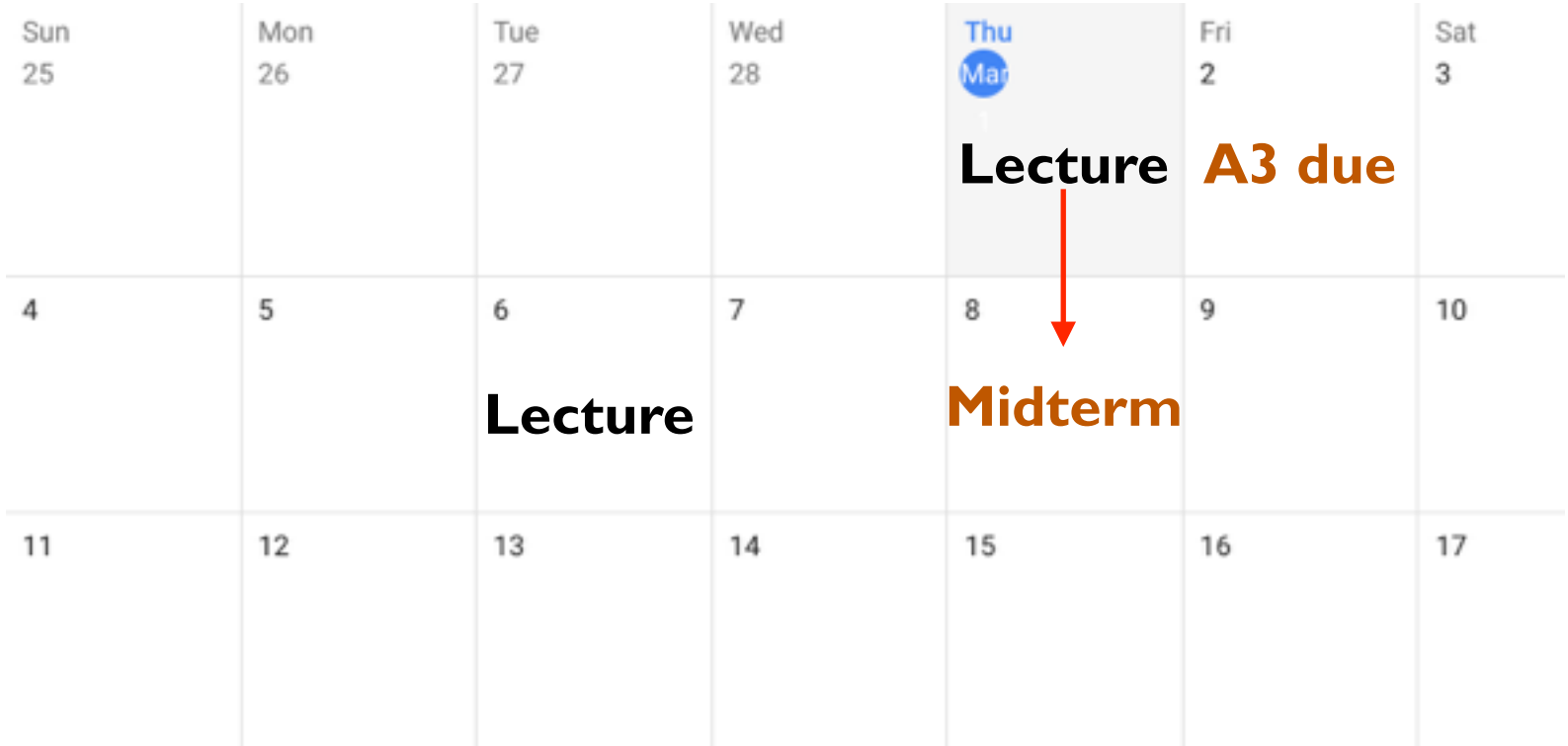
Department of Computer Science
University of Rochester

Action Items:

- **Assignment 3 is due tomorrow, midnight**
- **Mid-term: March 8**

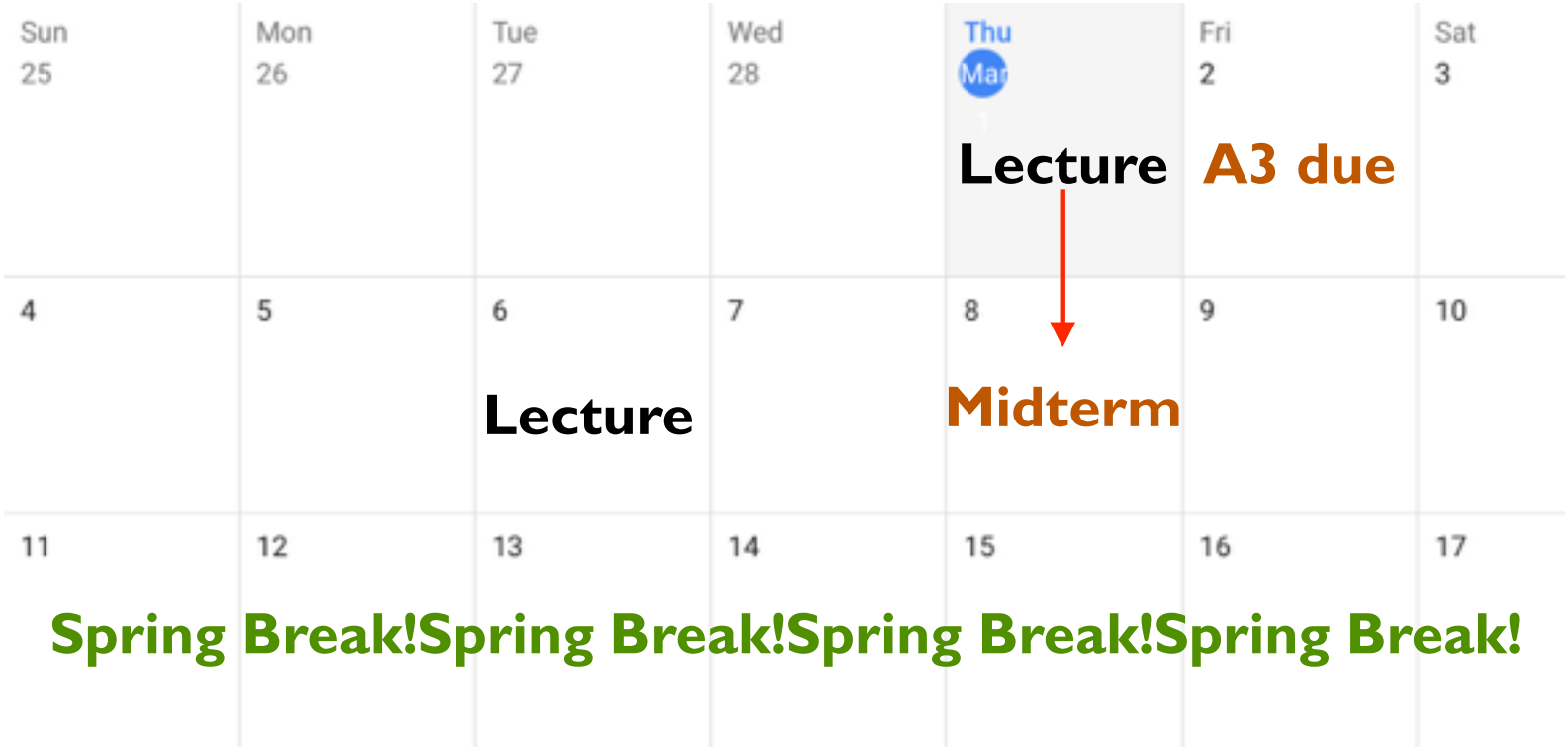
Announcement

- Programming Assignment 3 is due on tomorrow, midnight
- Mid-term exam: March 8; in class
- Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml

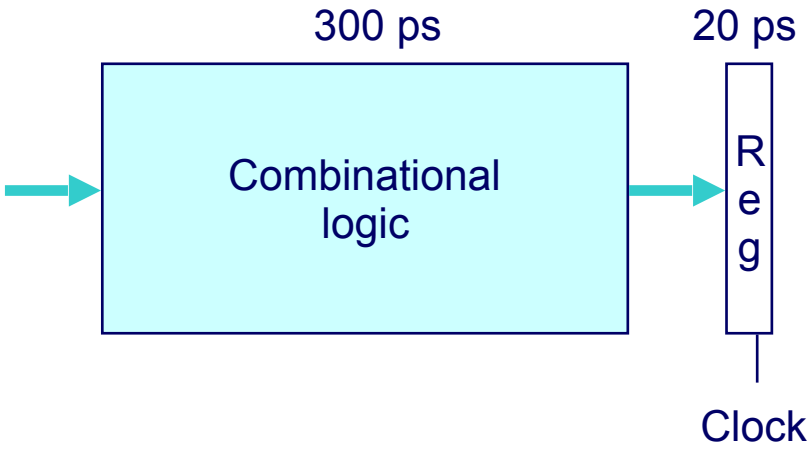


Announcement

- Programming Assignment 3 is due on tomorrow, midnight
- Mid-term exam: March 8; in class
- Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml



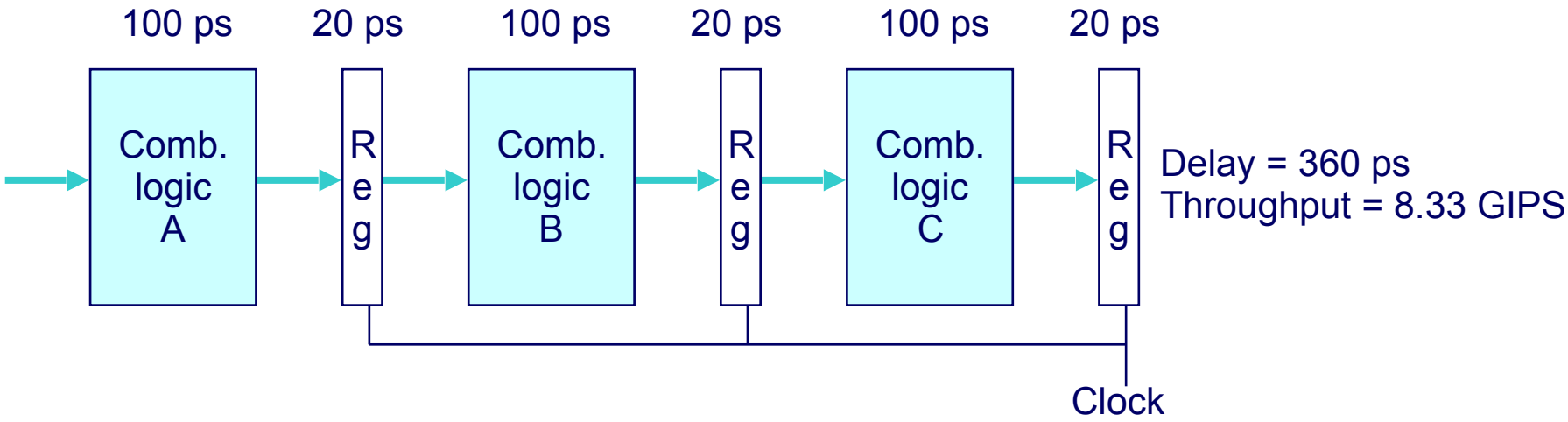
Pipeline Example



System Characteristics

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Delay for each instruction: 320 ps
- Can push a new instruction every 320 ps
- Throughput of the system: 3.12 Giga Instructions Per Second (GIPS)

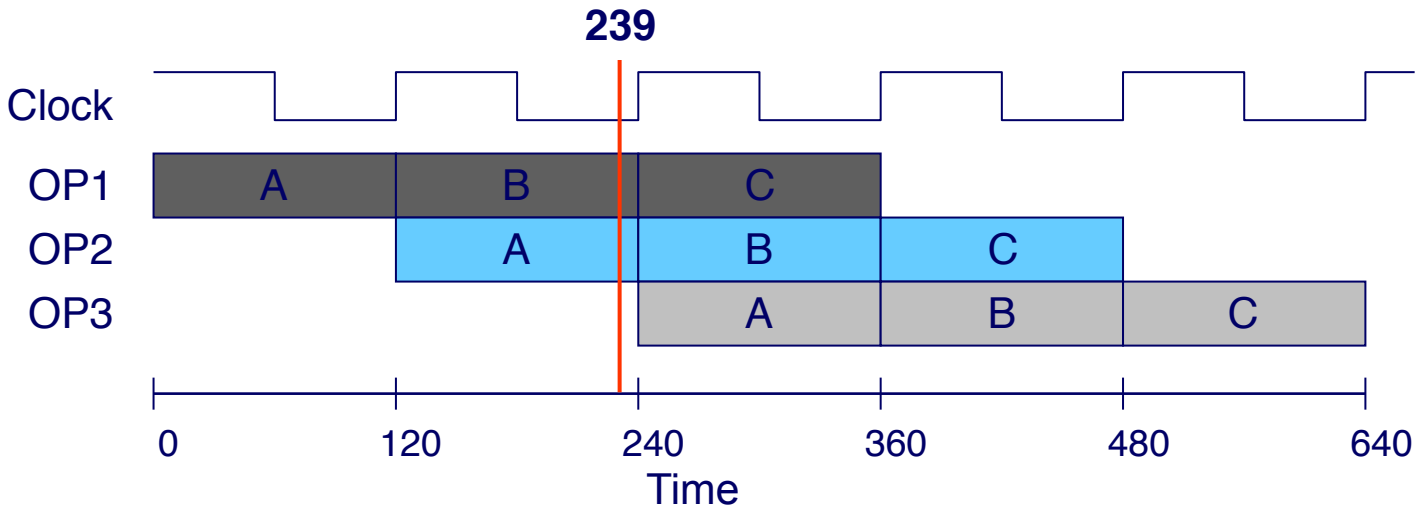
3-Stage Pipelined Version

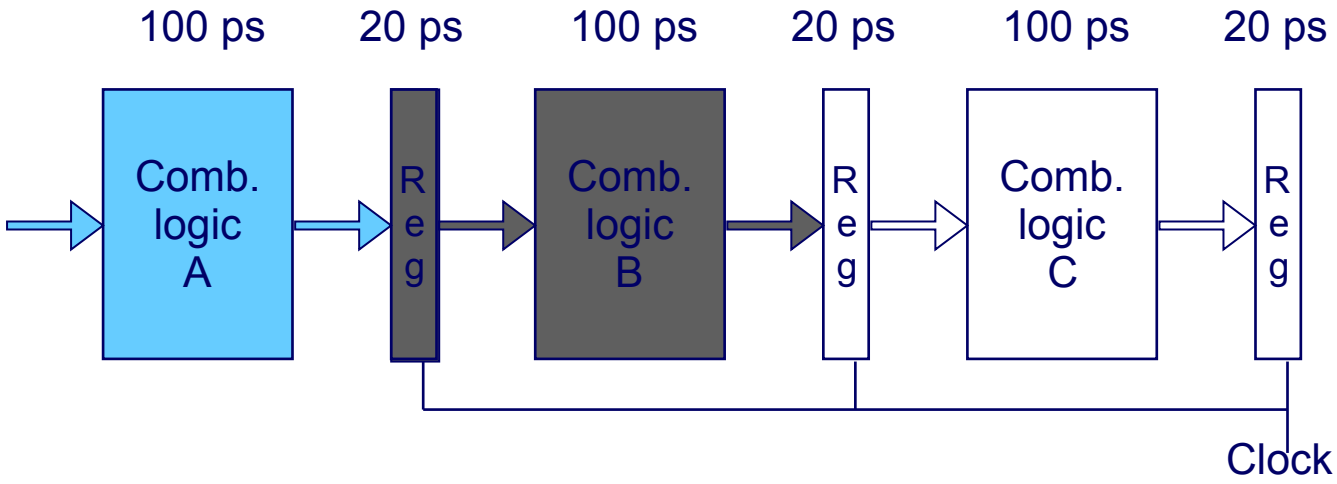


System Characteristics

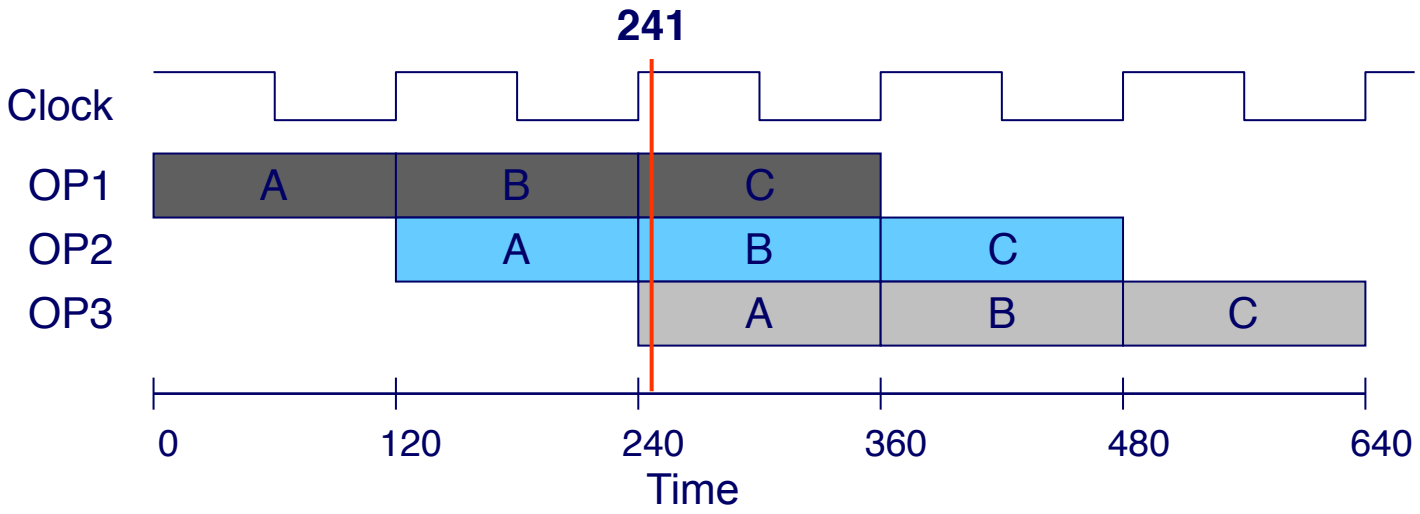
- Delay for each instruction: 360 ps (60 ps in loading registers)
- Can push a new instruction every 120 ps
- Throughput of the system: 8.33 Giga Instructions Per Second (GIPS)

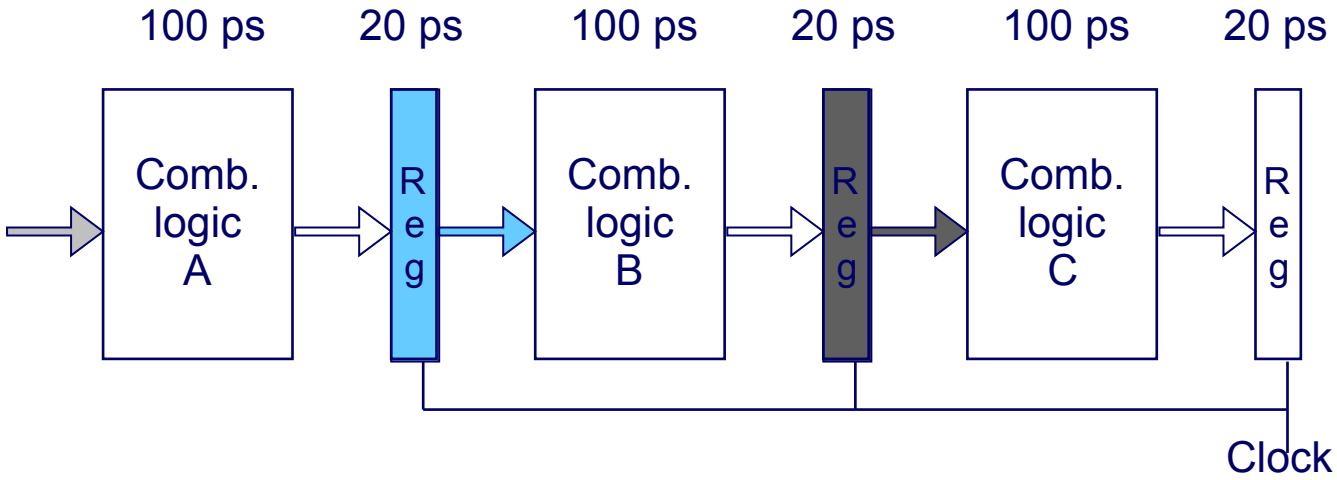
Operating a Pipeline



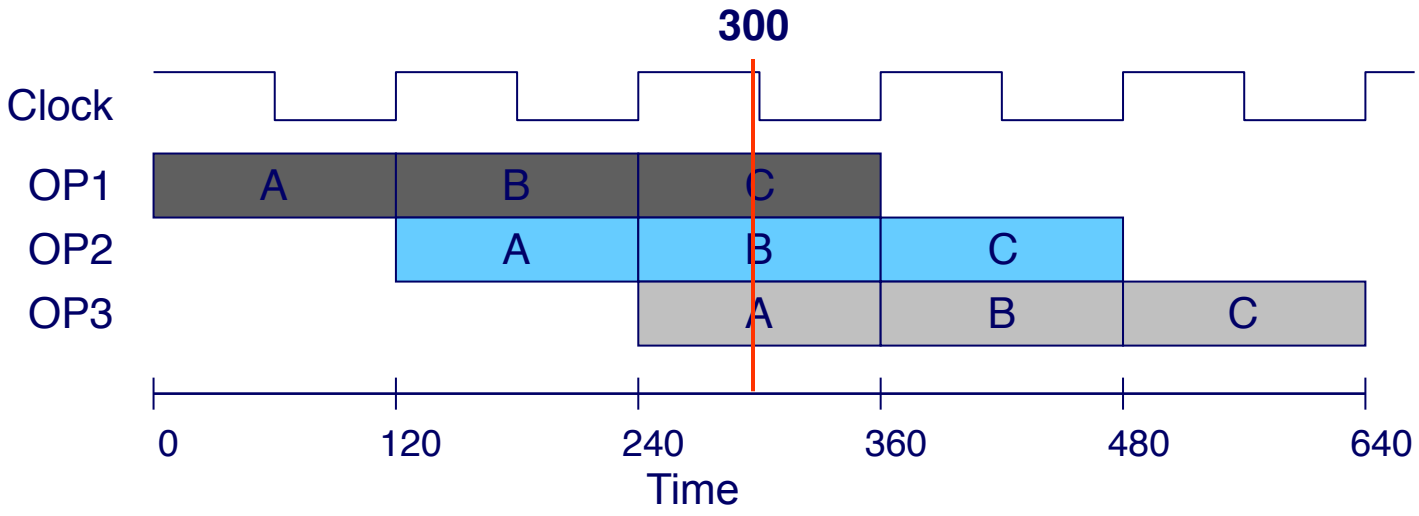


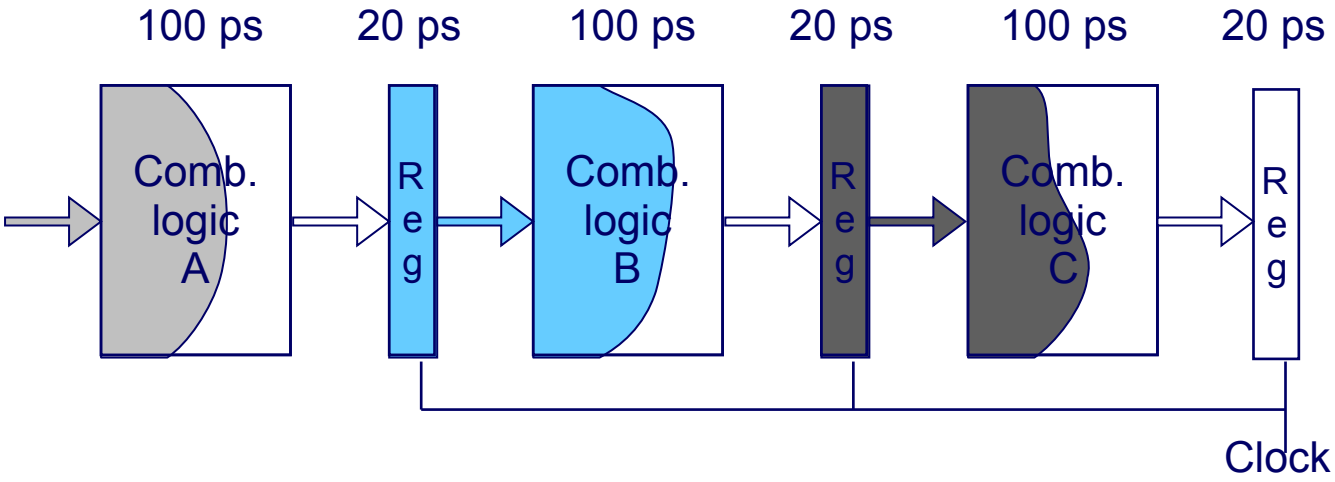
Operating a Pipeline



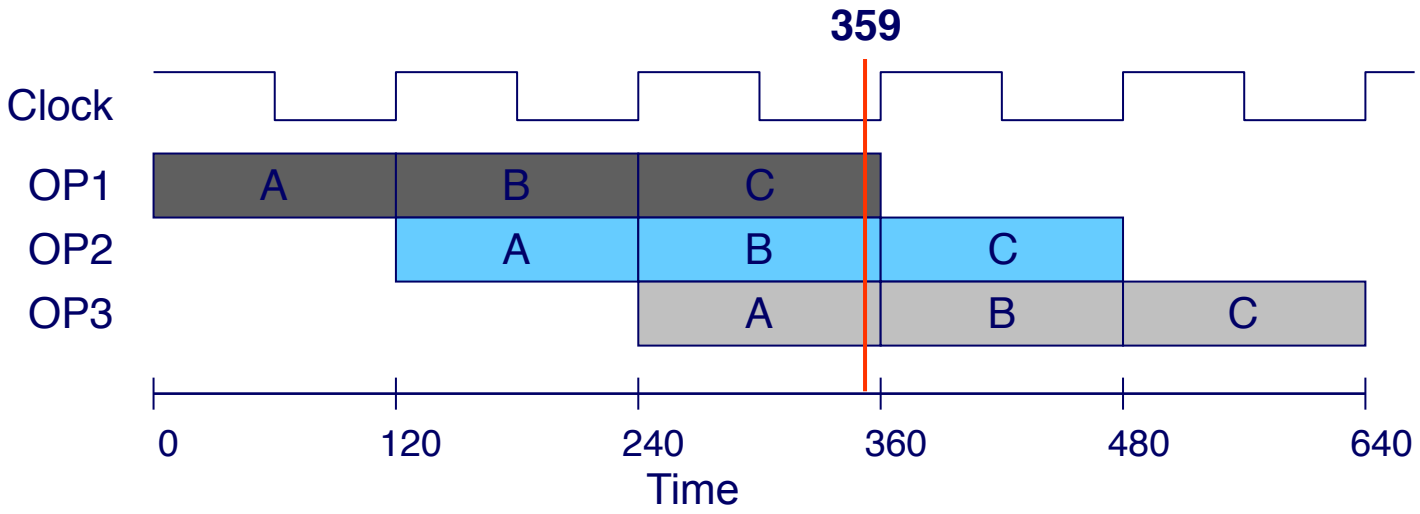


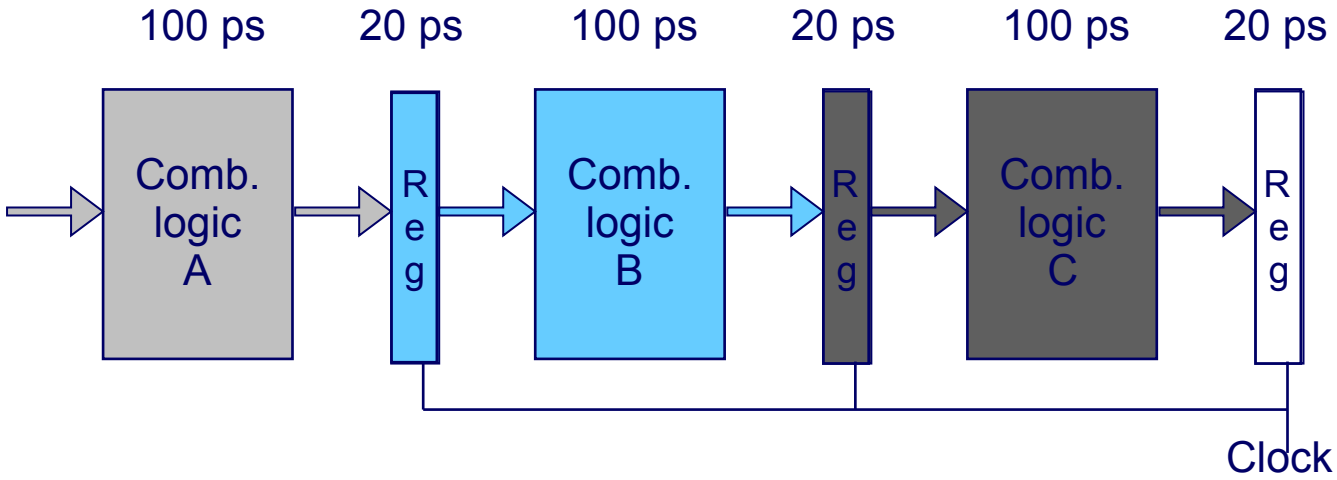
Operating a Pipeline





Operating a Pipeline





Pipeline Stages

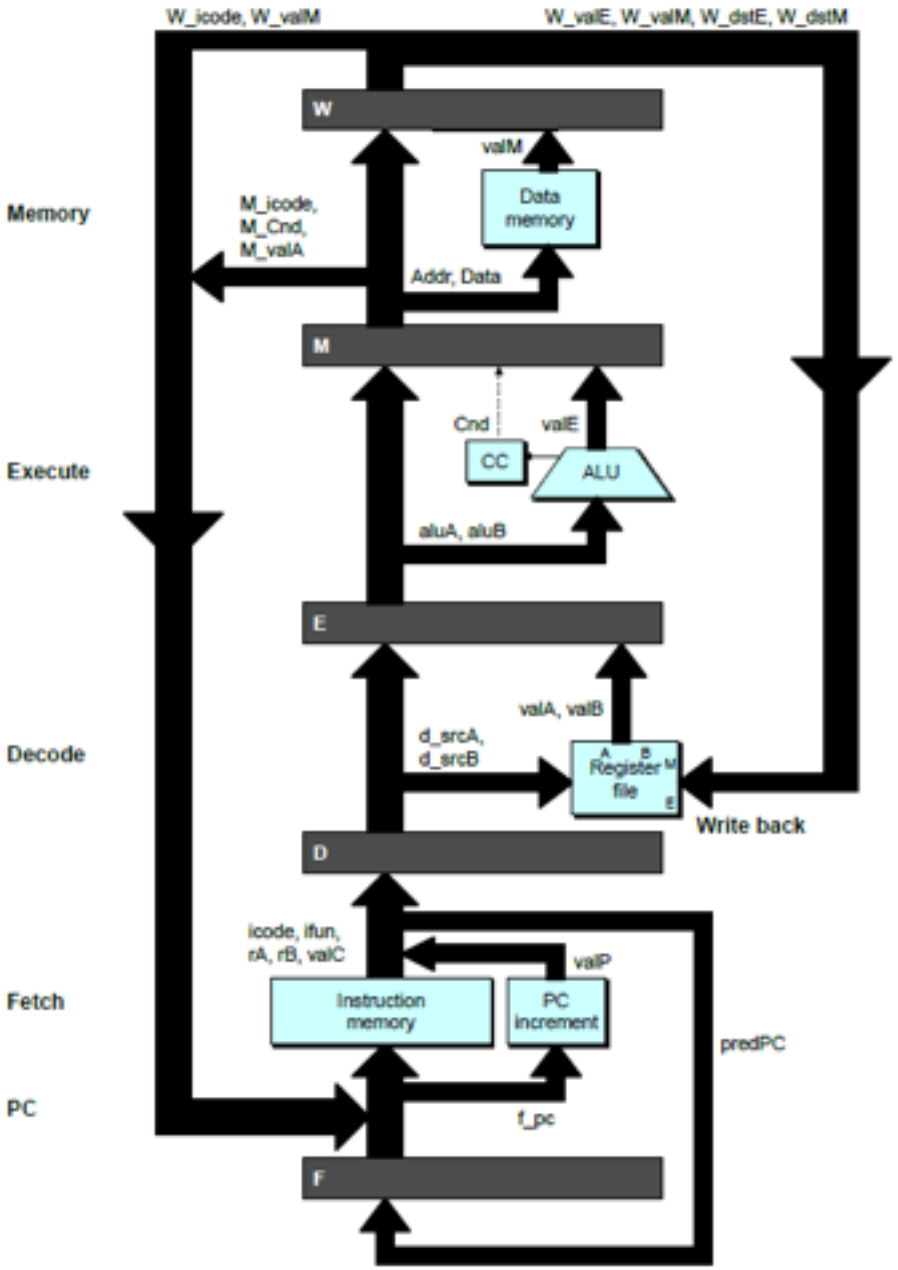
Fetch
- Select current PC
- Read instruction
- Compute incremented PC

Decode
- Read program registers

Execute
- Operate ALU

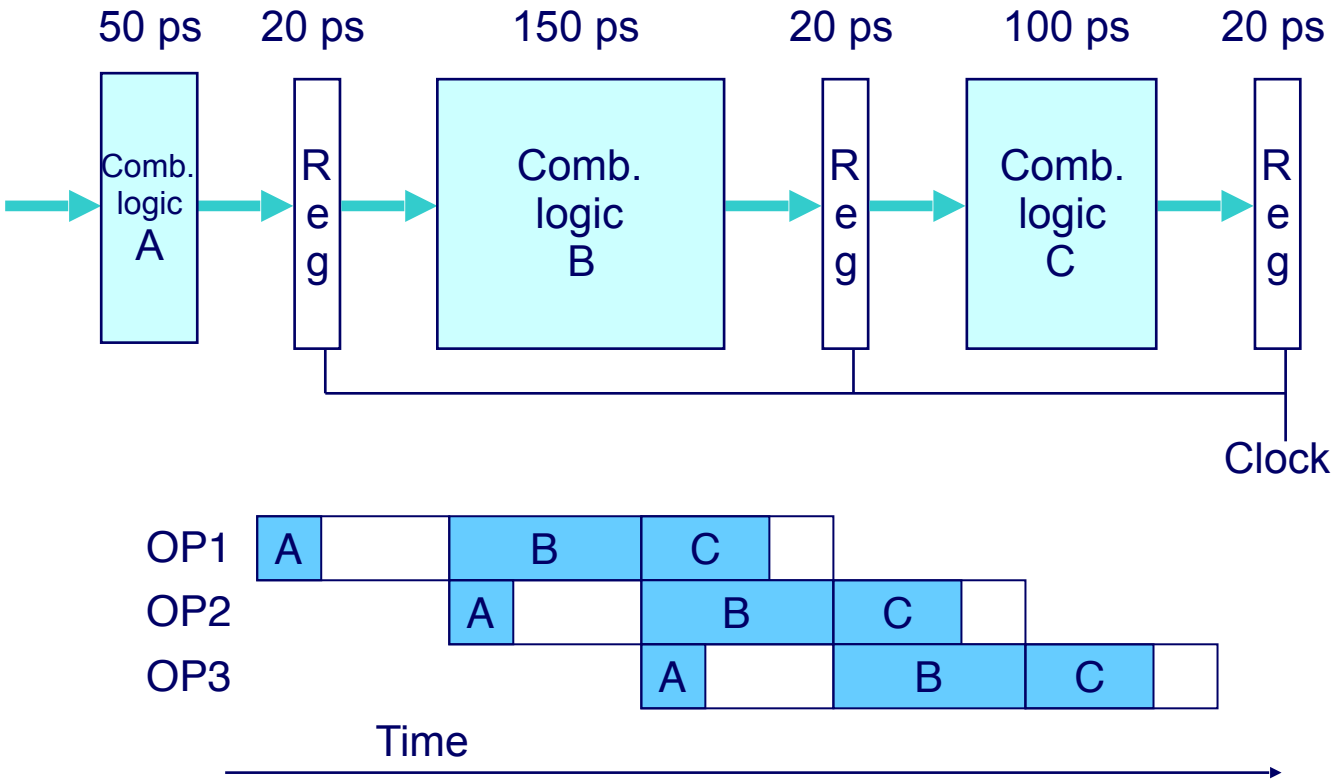
Memory
- Read or write data memory

Write Back
- Update register file

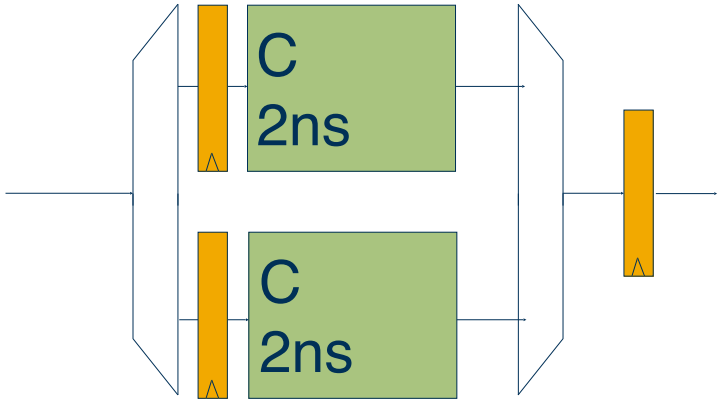


Basic Pipelining Summary

- **Pros:**Increase throughput by breaking up long combinational paths
- **Cons:** May increase latency. Need new registers
  - Can't do better than slowest component (bottleneck)



Interleaving



- Use multiple copies of the slow component
- Interleave between them
  - Cycle by cycle
- Throughput goes up
- N-way interleaving equivalent to N-stage pipeline in performance

Today: Making the Pipeline Really Work

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

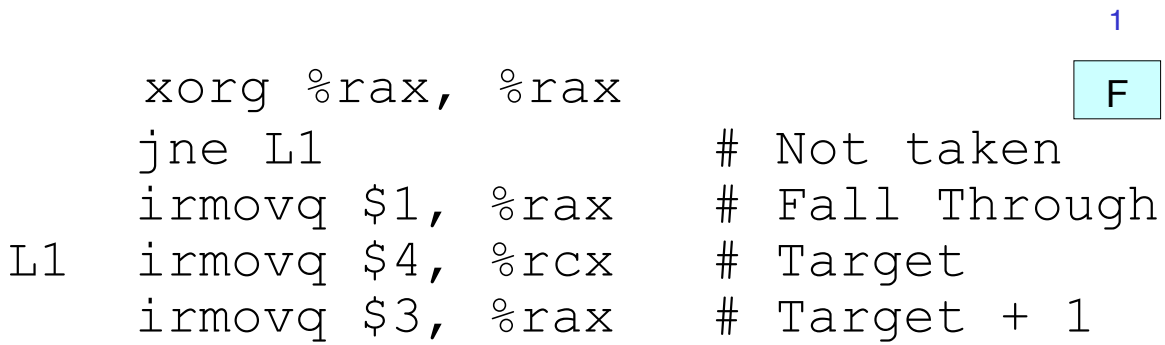
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

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

Control Dependency

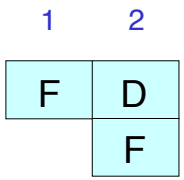
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

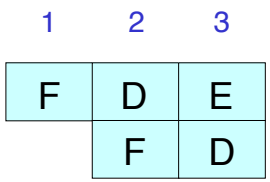
```
     xorg %rax, %rax
     jne L1              # Not taken
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

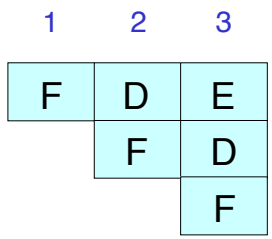
```
    xorg %rax, %rax
    jne L1              # Not taken
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

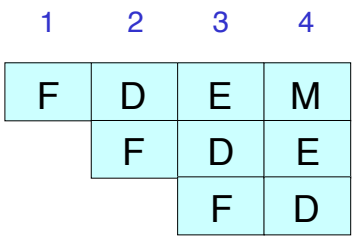
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

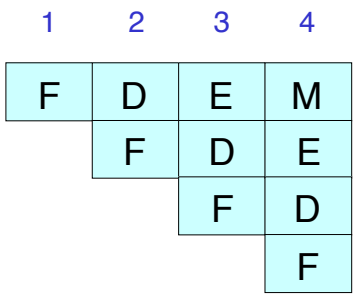
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

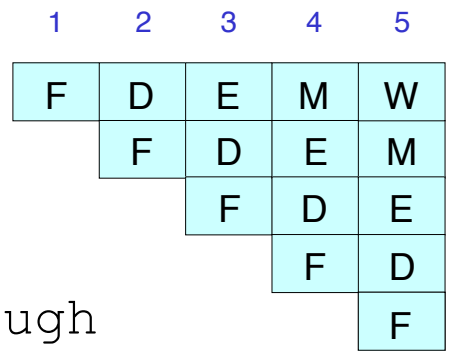
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

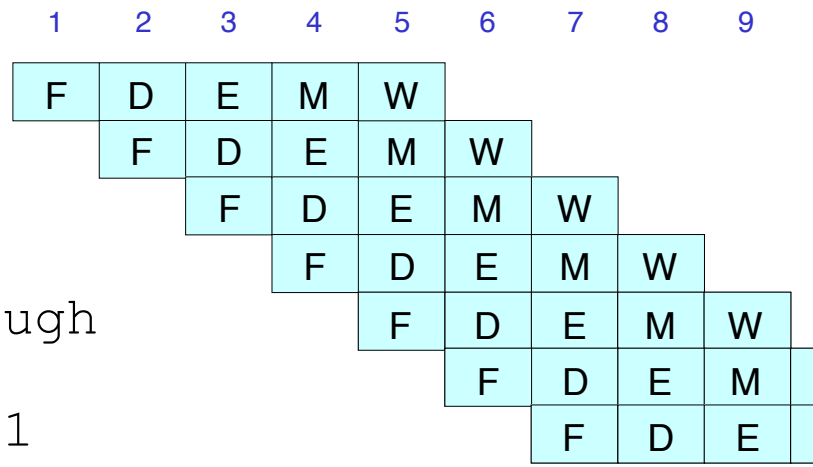
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Inserting Nops wastes pipeline slots
- Can we do better than that?
- Two strategies:
  - Delay slots
  - Branch Prediction

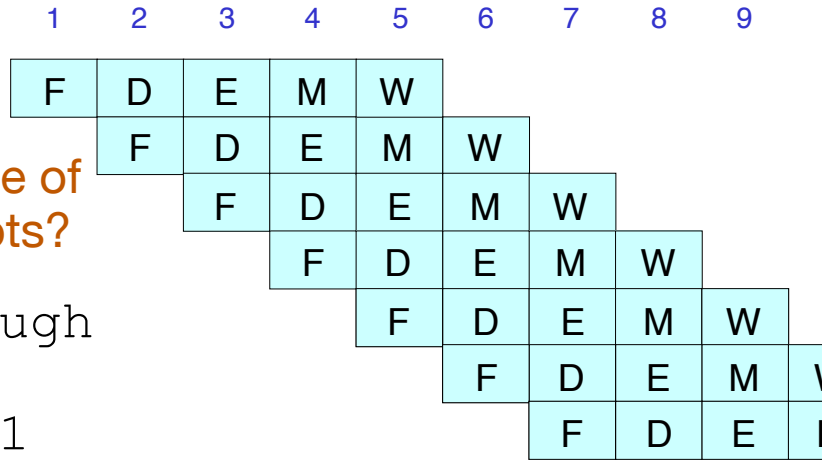
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Delay Slots

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

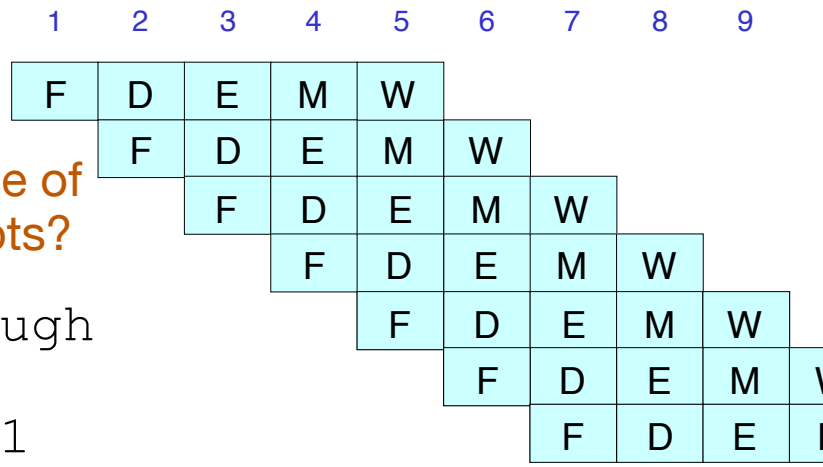
Can we make use of the 2 wasted slots?



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?

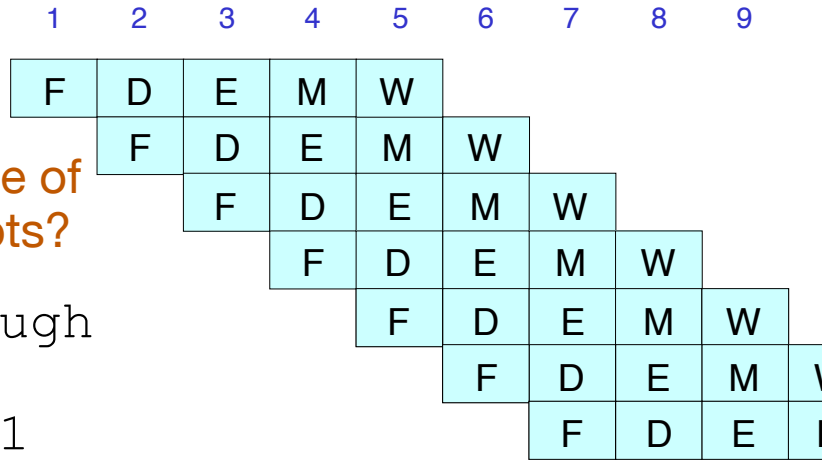



```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```

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?



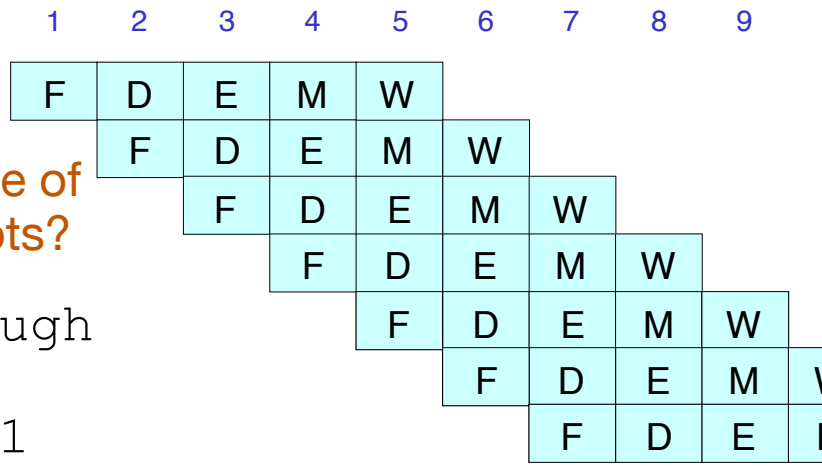
Have to make sure `do_C` doesn't depend on `do_A` and `do_B!!!`

```
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?



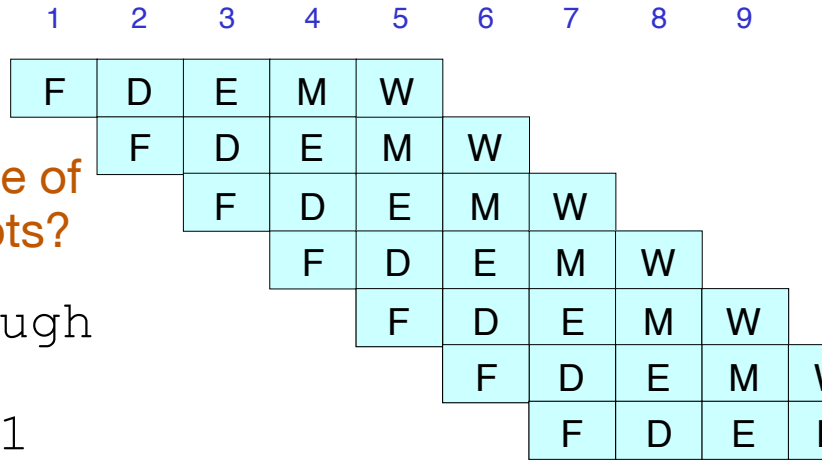
A less obvious example

```
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?



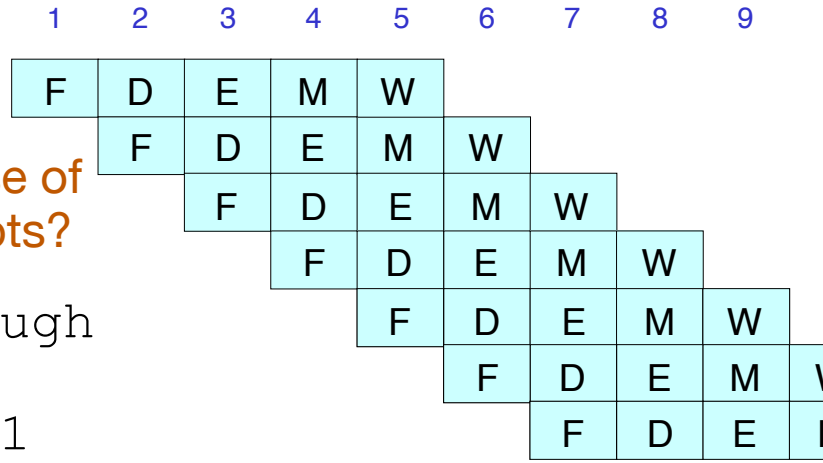
A less obvious example

```
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();
}
```

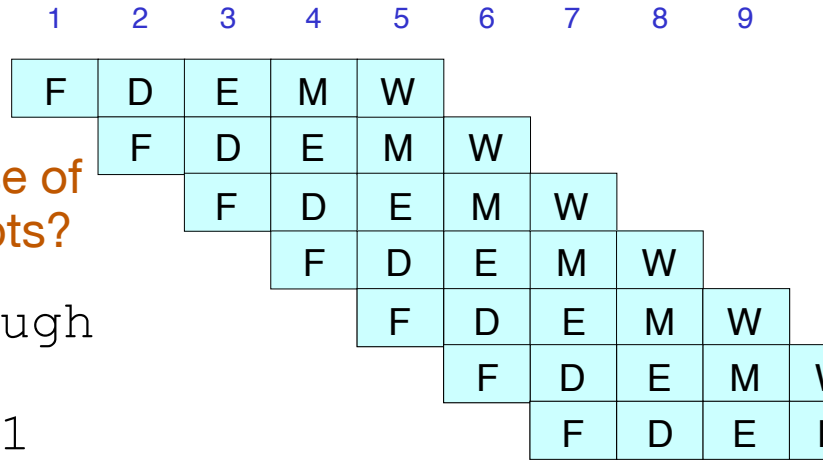
```
add A, B
or C, D
sub E, F
jle 0x200
add A, C
```

```
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?

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

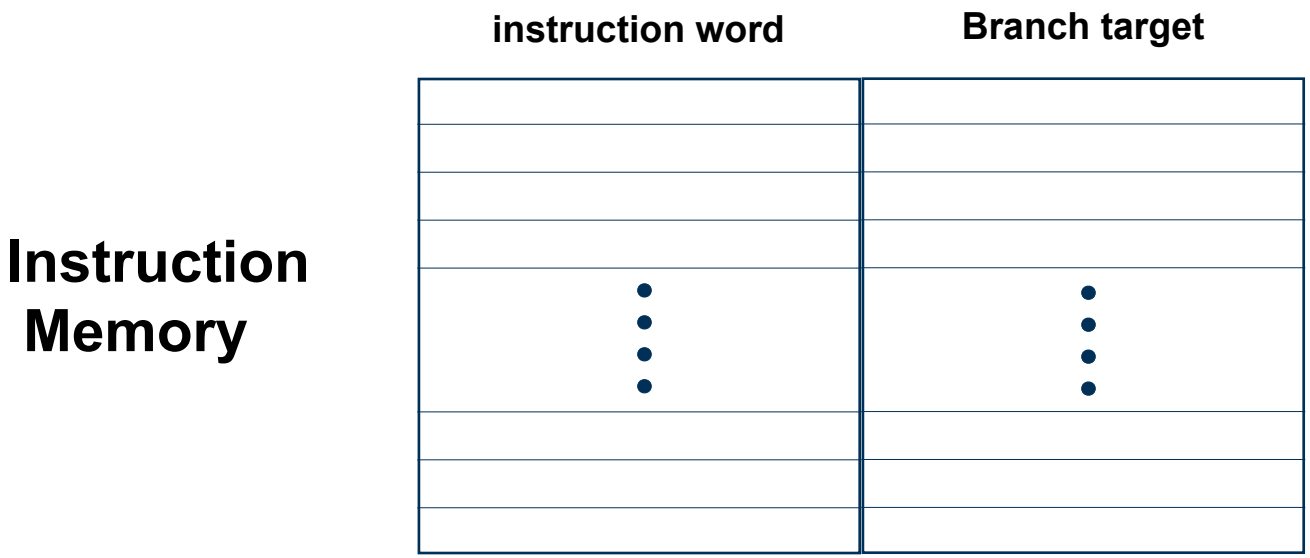
- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

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

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

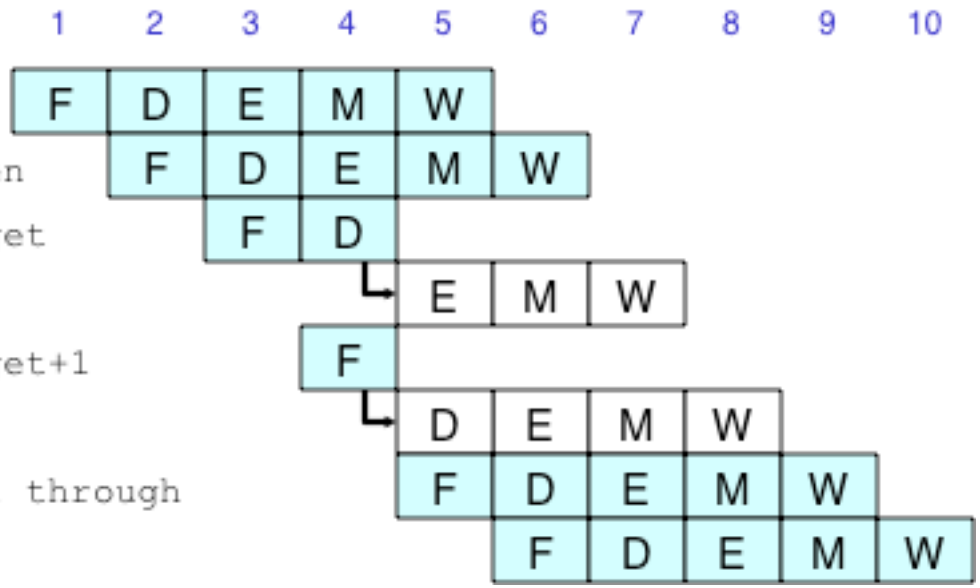
How to Keep The Predictions?

- Branch Target Buffer (BTB)
  - A separate memory that's not visible to programmers
  - Keep tracks of the target for each branch instruction



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

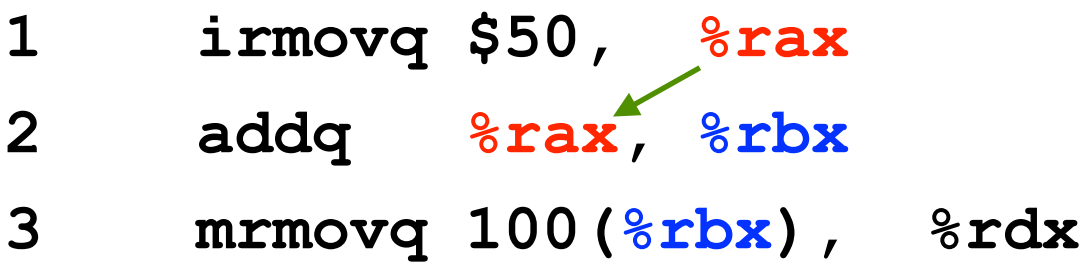
Today: Making the Pipeline Really Work

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

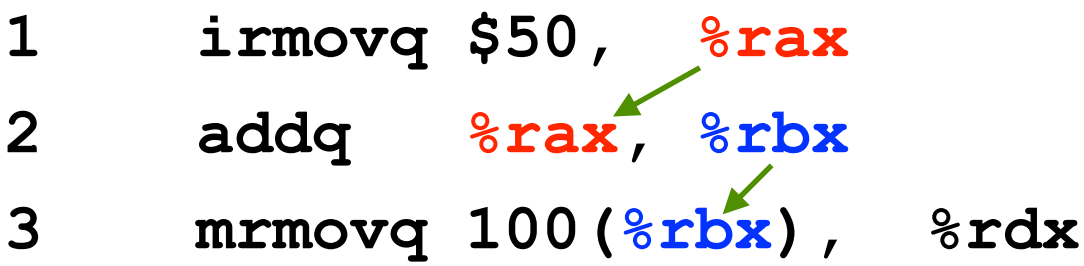
Data Dependencies

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

Data Dependencies

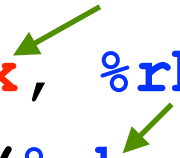


Data Dependencies



Data Dependencies

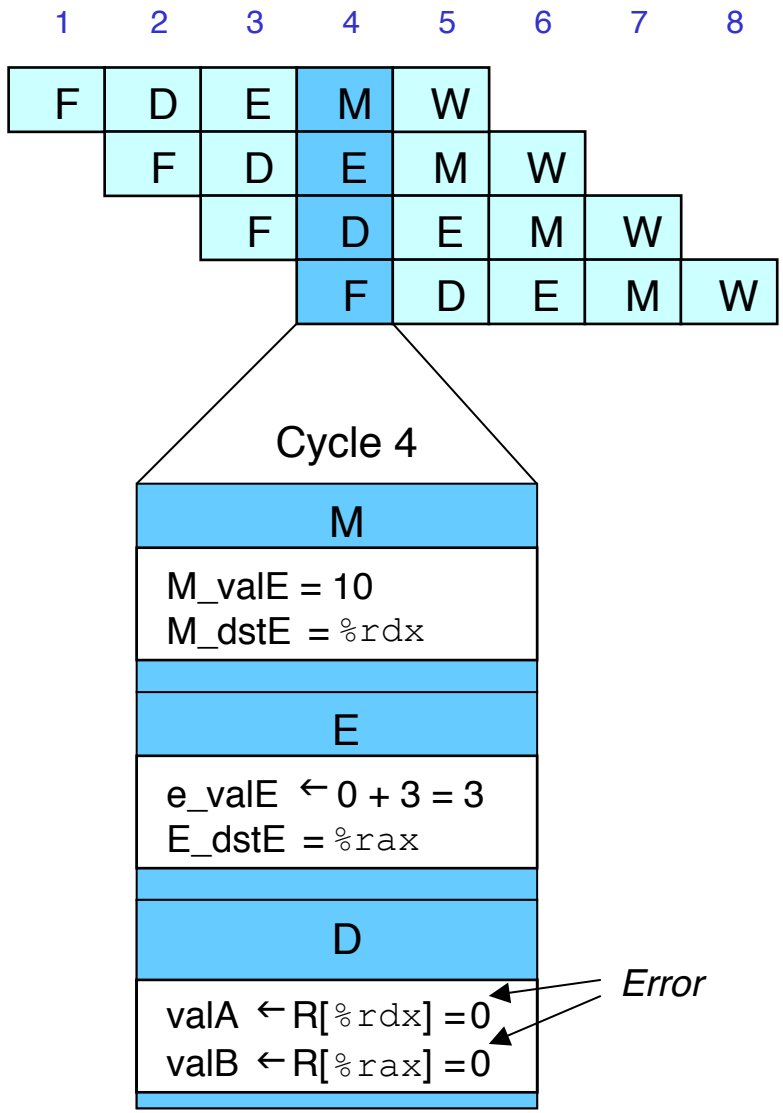
```
1     irmovq $50,   %rax
2     addq   %rax, %rbx
3     mrmovq 100(%rbx),   %rdx
```



- 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

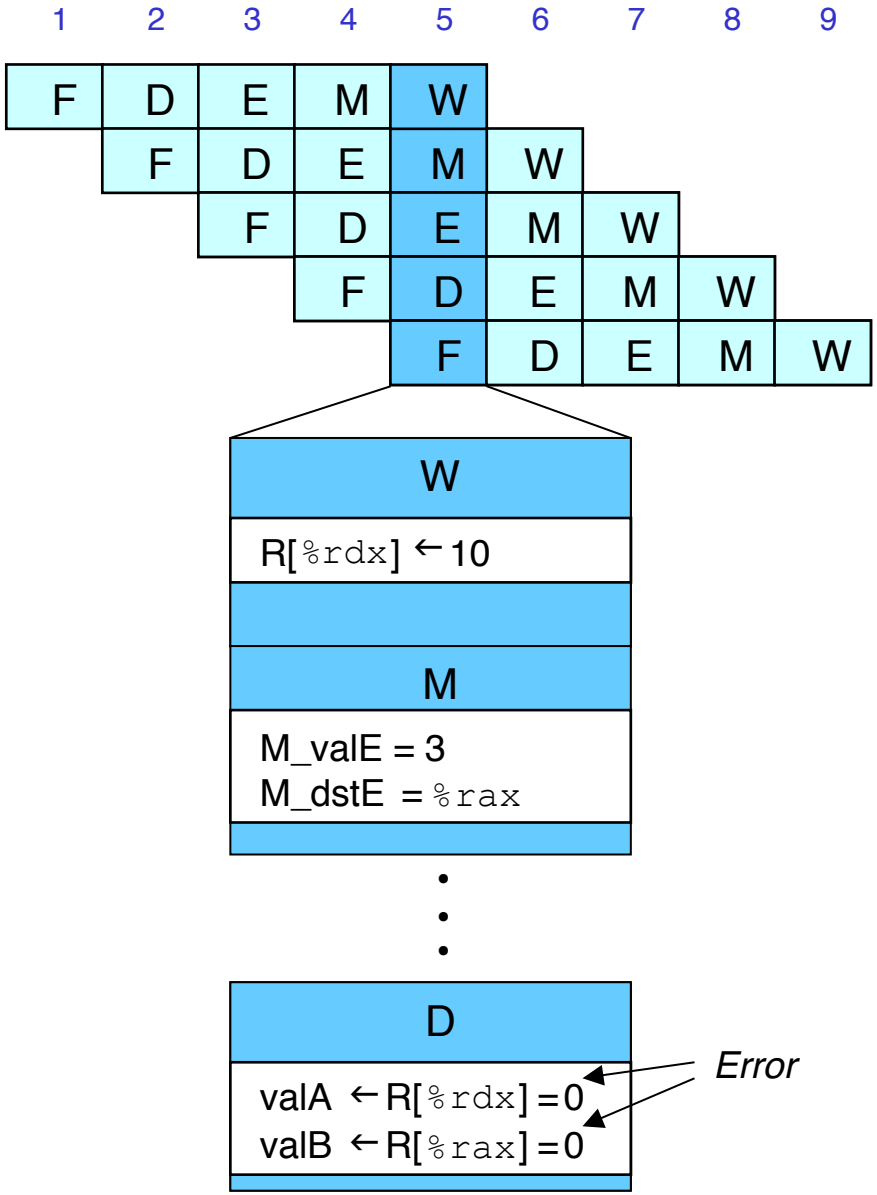
Data Dependencies: No Nop

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



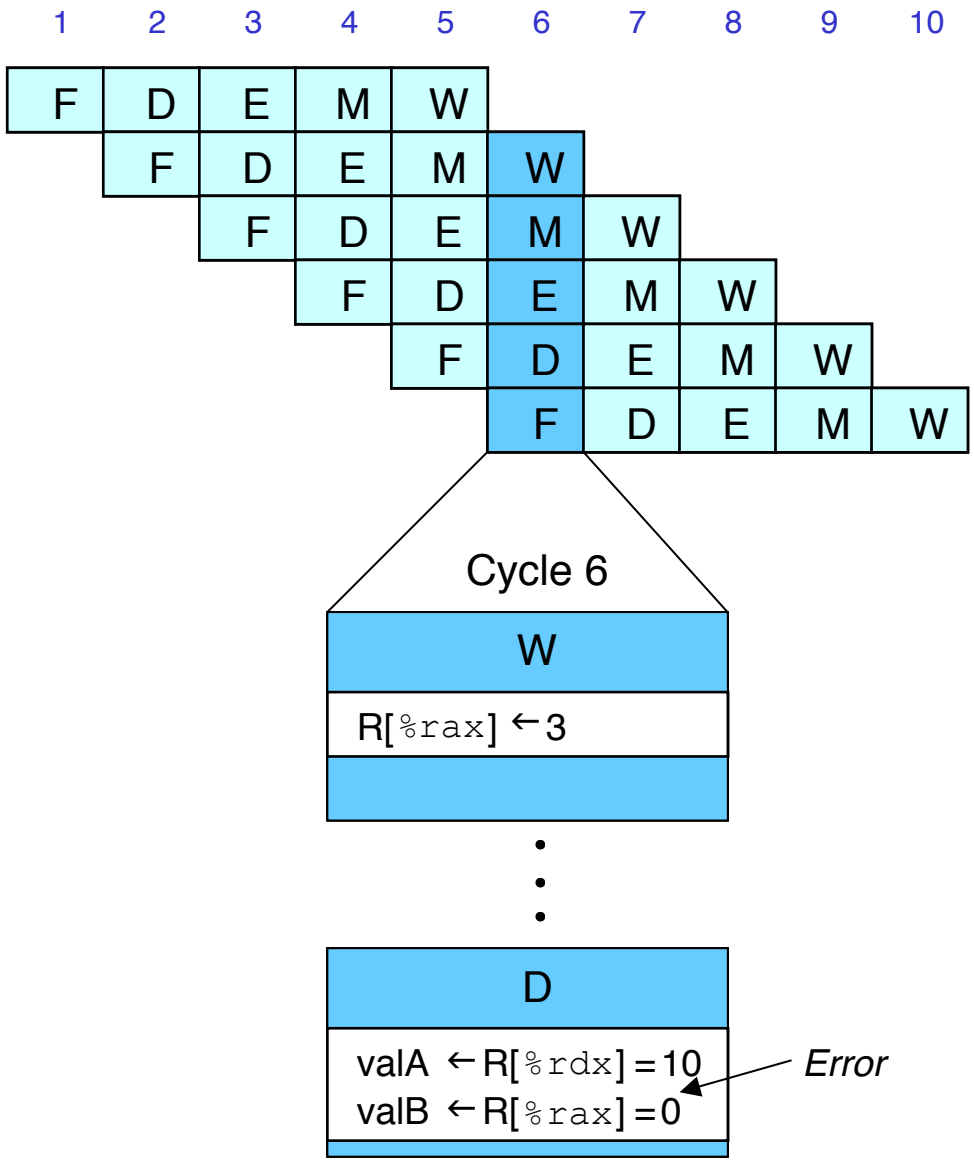
Data Dependencies: 1 Nop

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



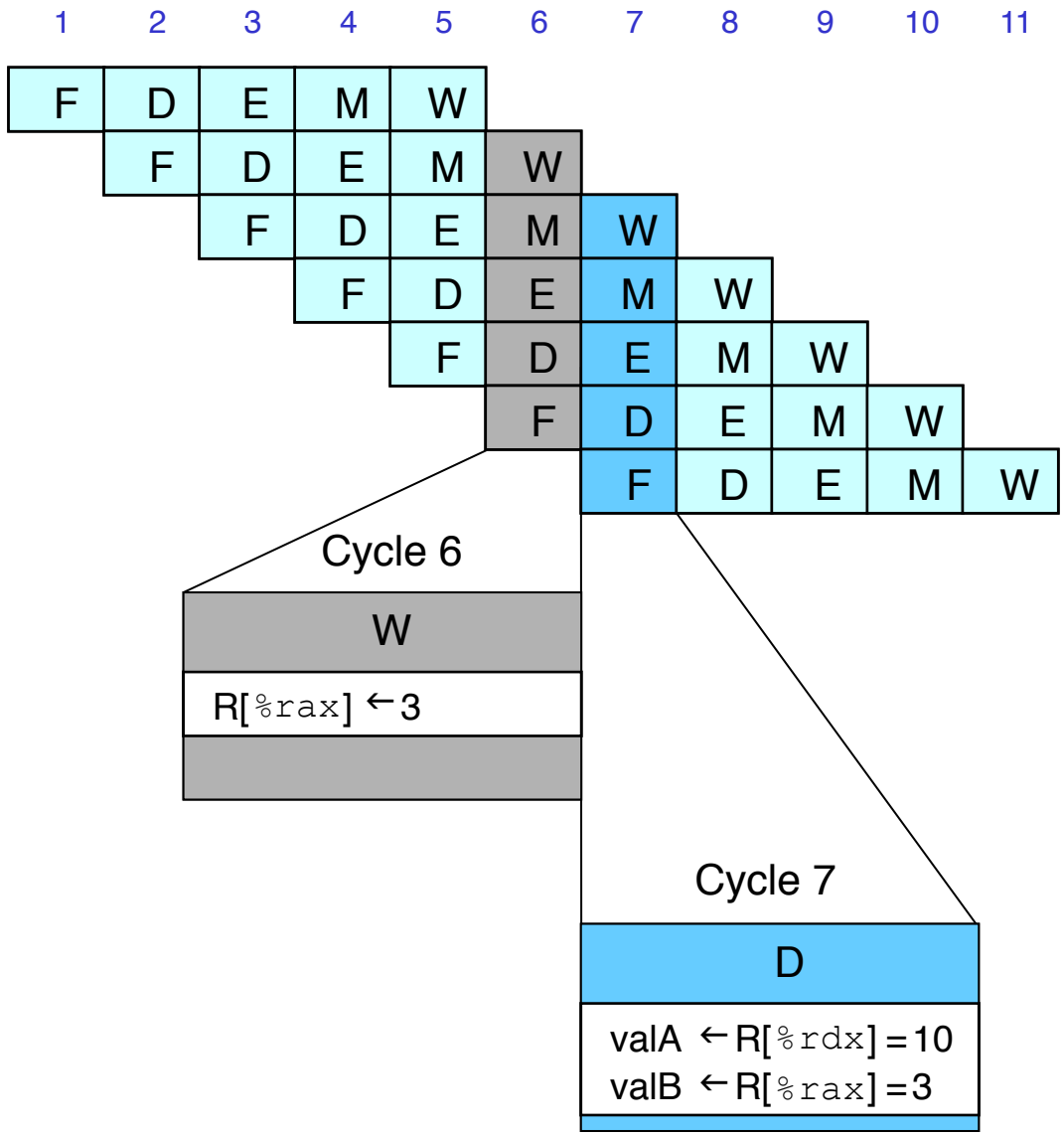
Data Dependencies: 2 Nop's

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



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
```

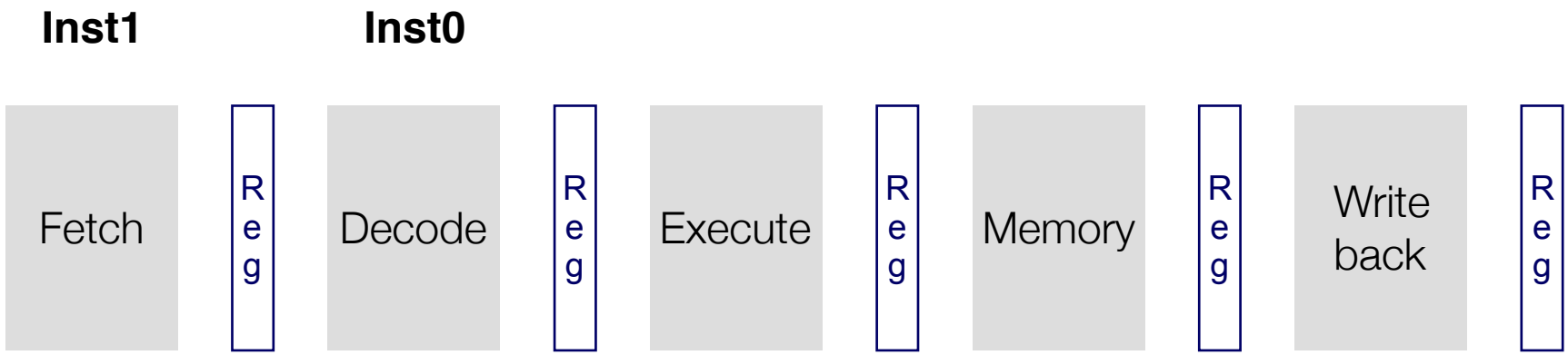


Normal Execution

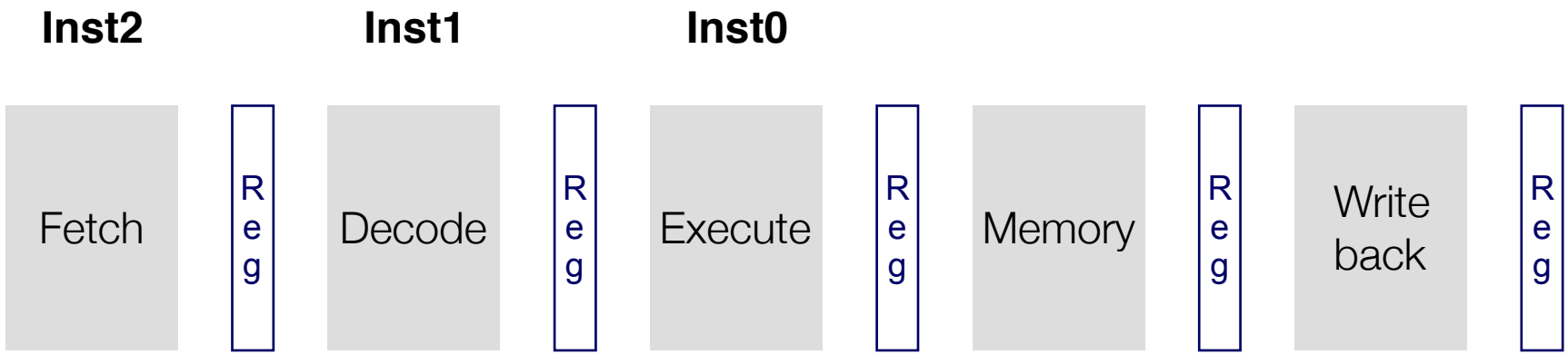
Normal Execution

Inst0

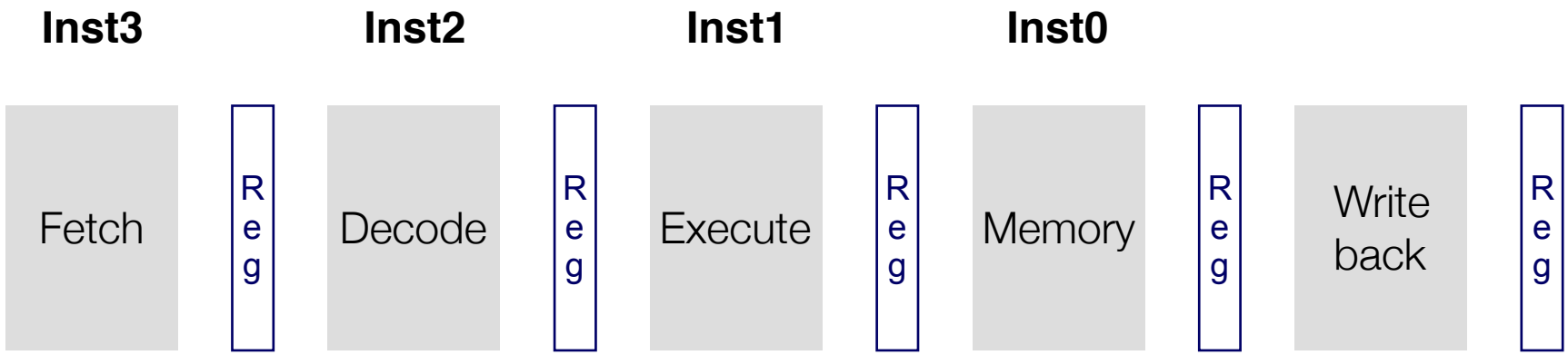
Normal Execution



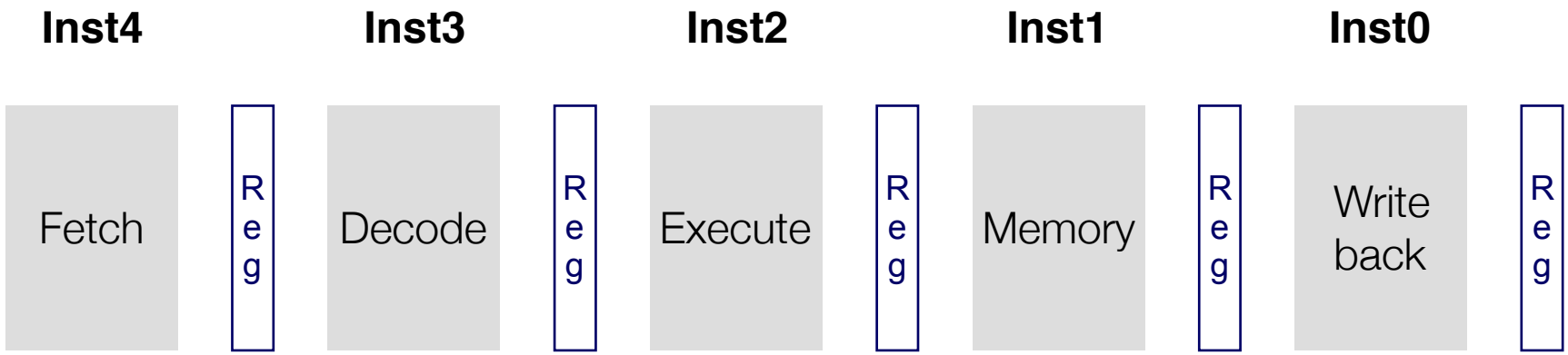
Normal Execution



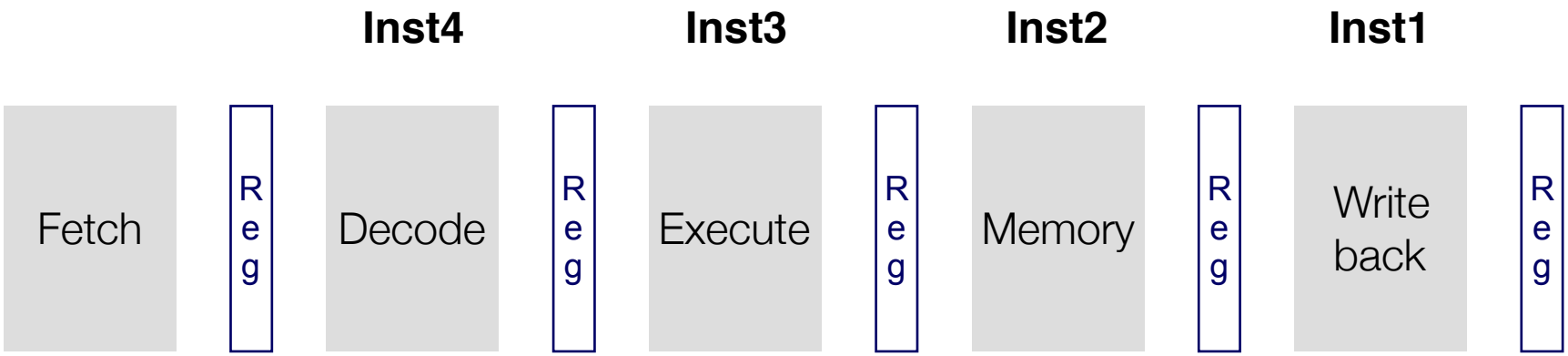
Normal Execution



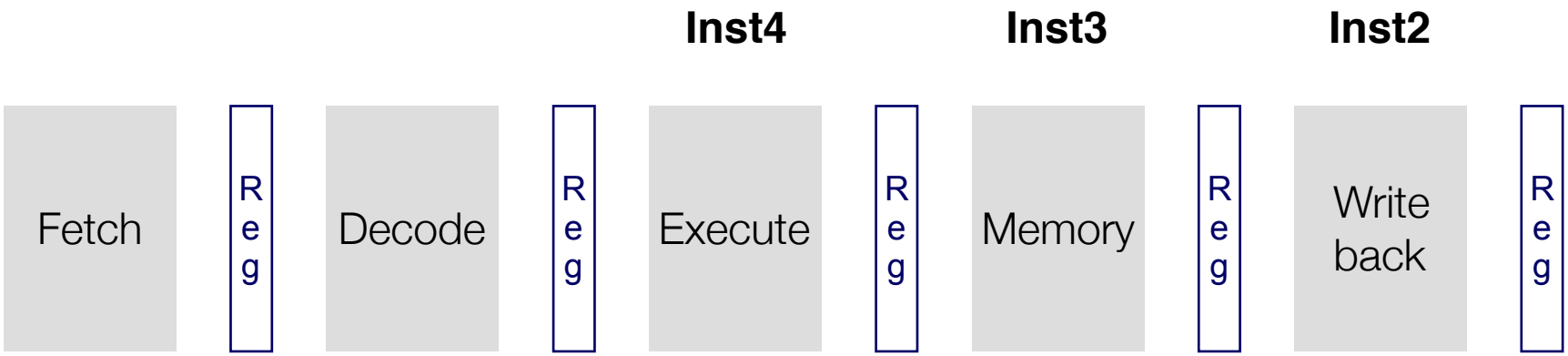
Normal Execution



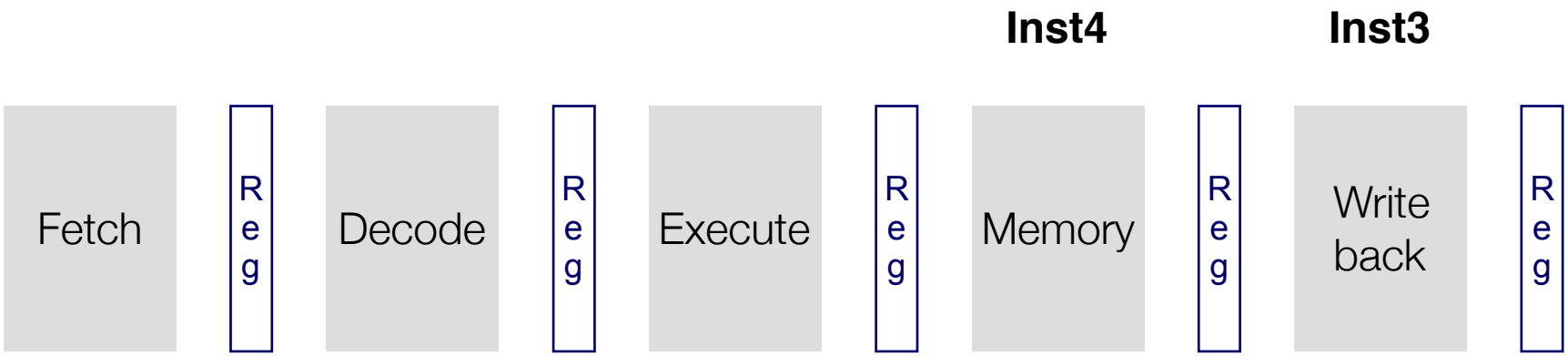
Normal Execution



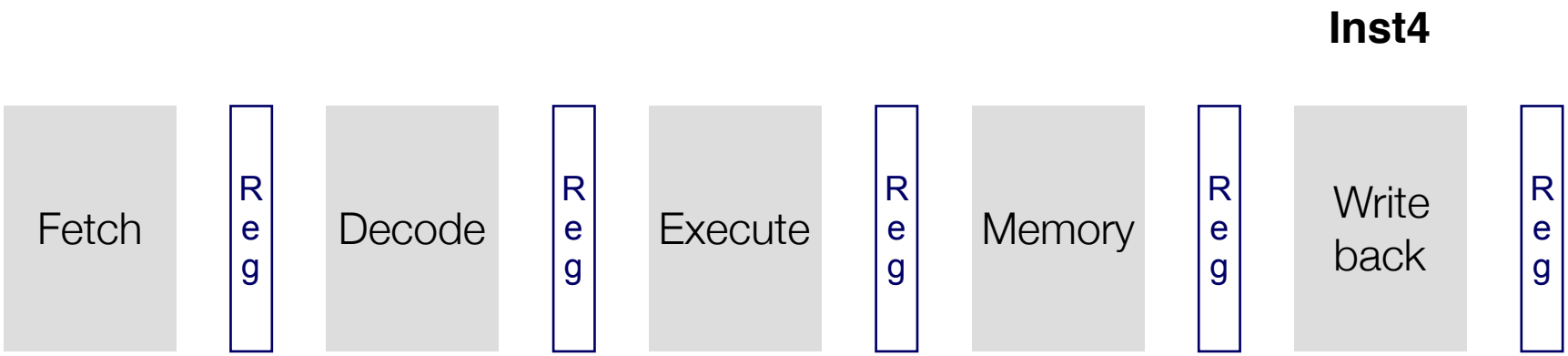
Normal Execution



Normal Execution

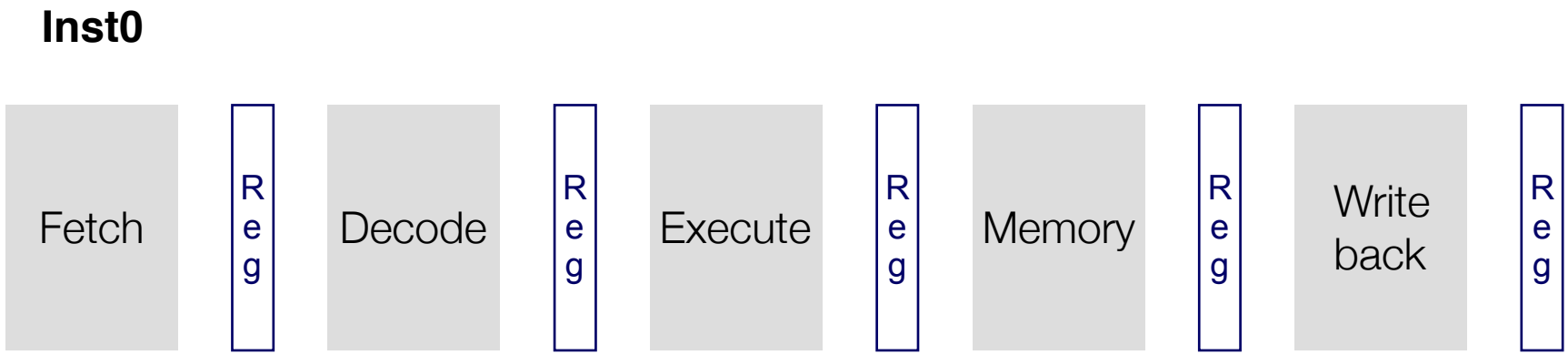


Normal Execution

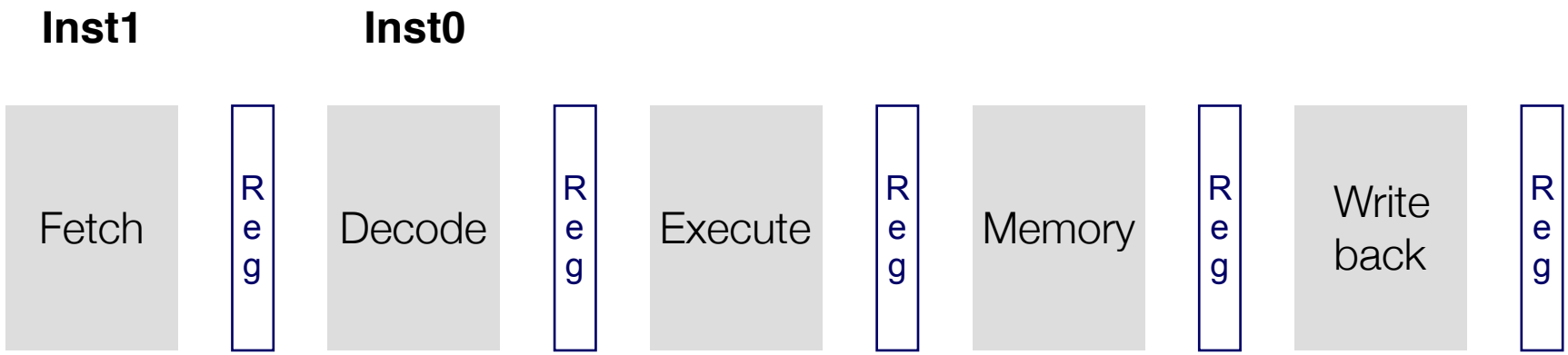


Stalling Illustration

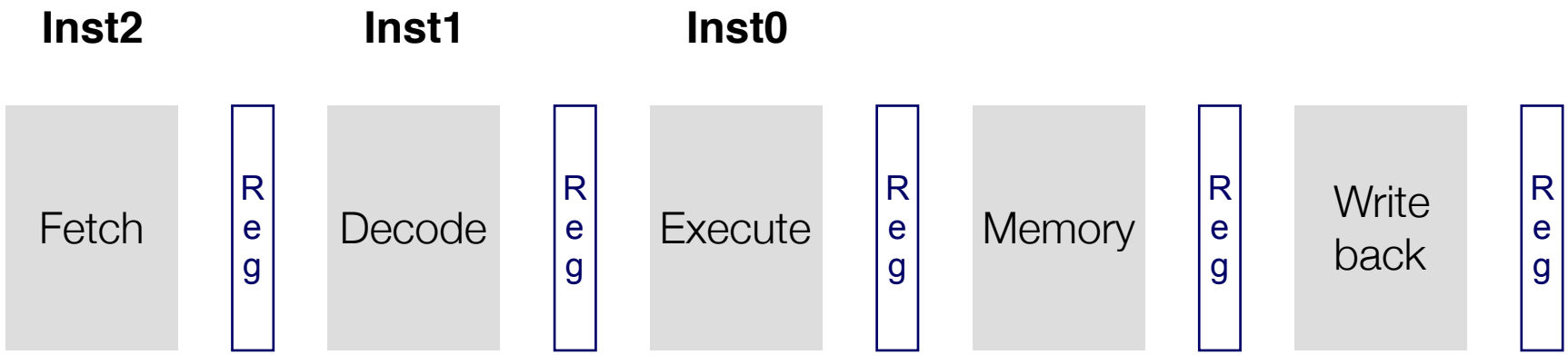
Stalling Illustration



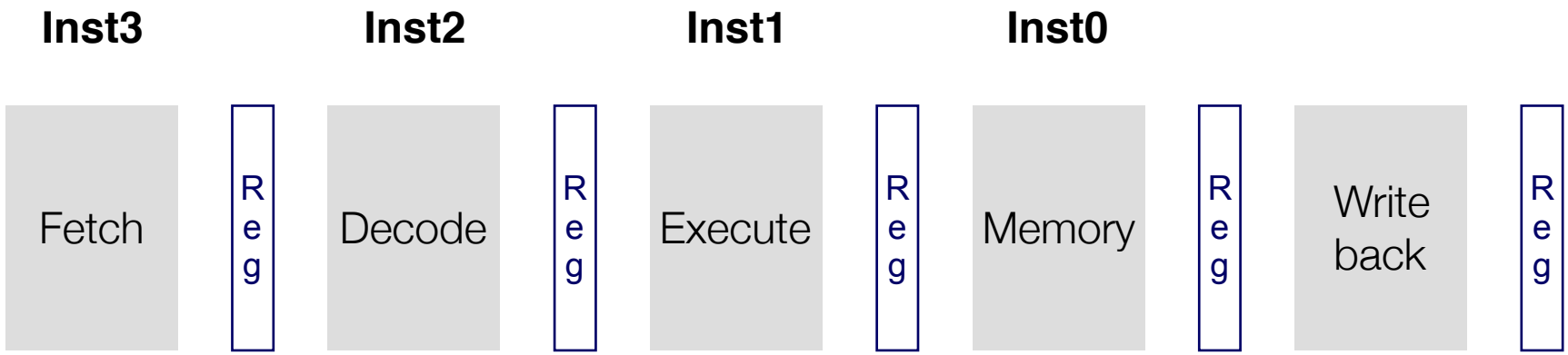
Stalling Illustration



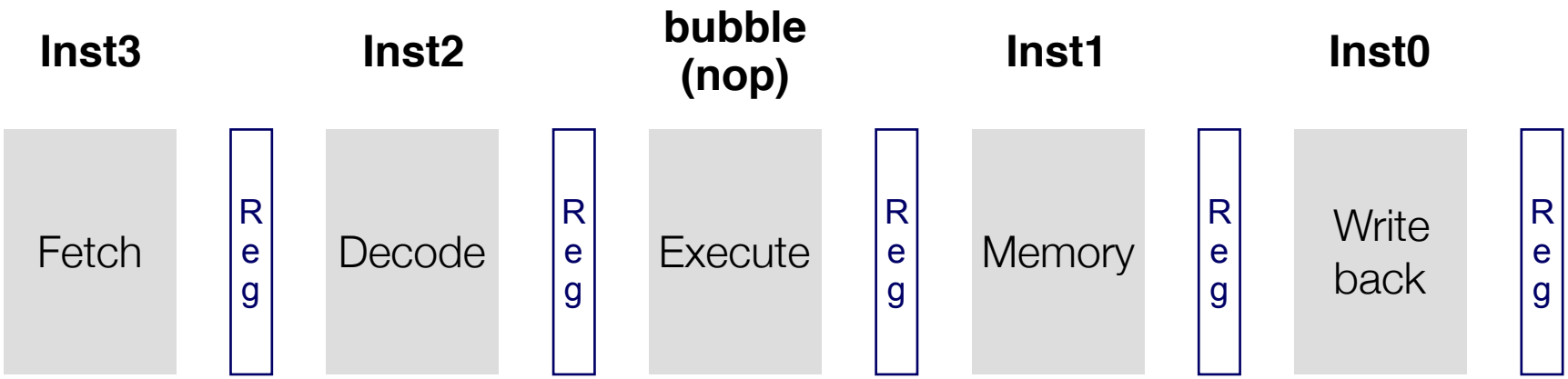
Stalling Illustration



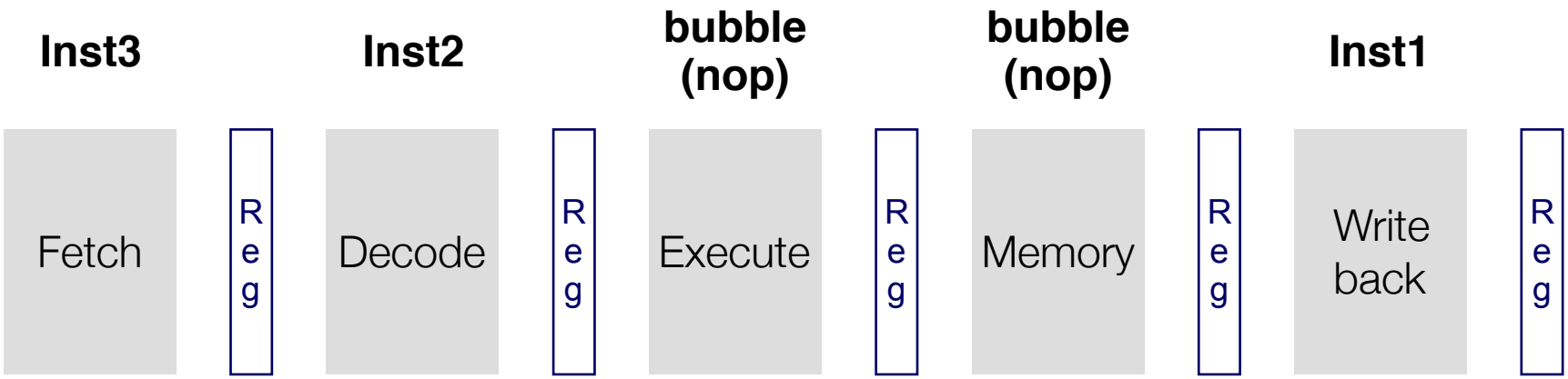
Stalling Illustration



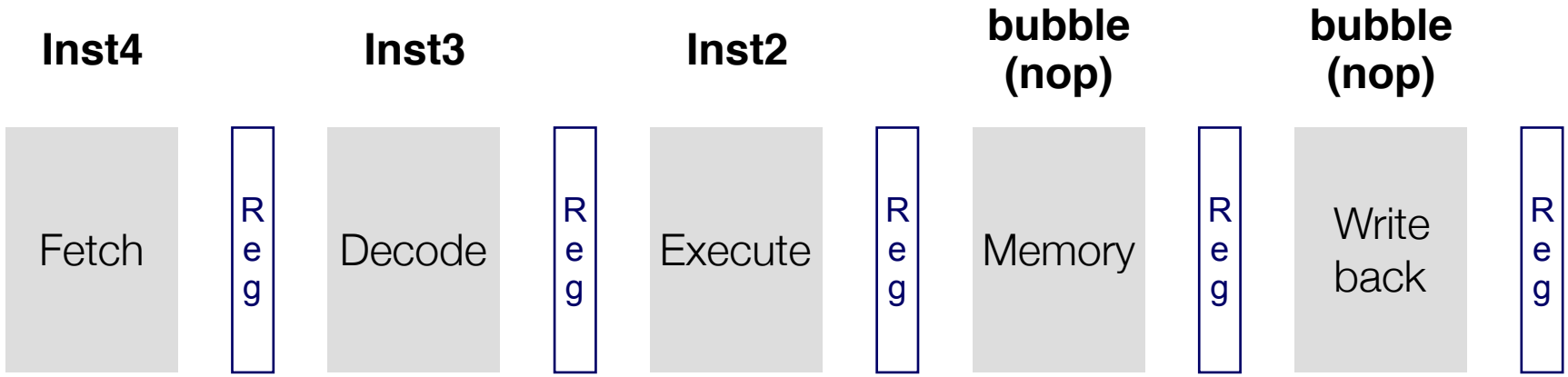
Stalling Illustration



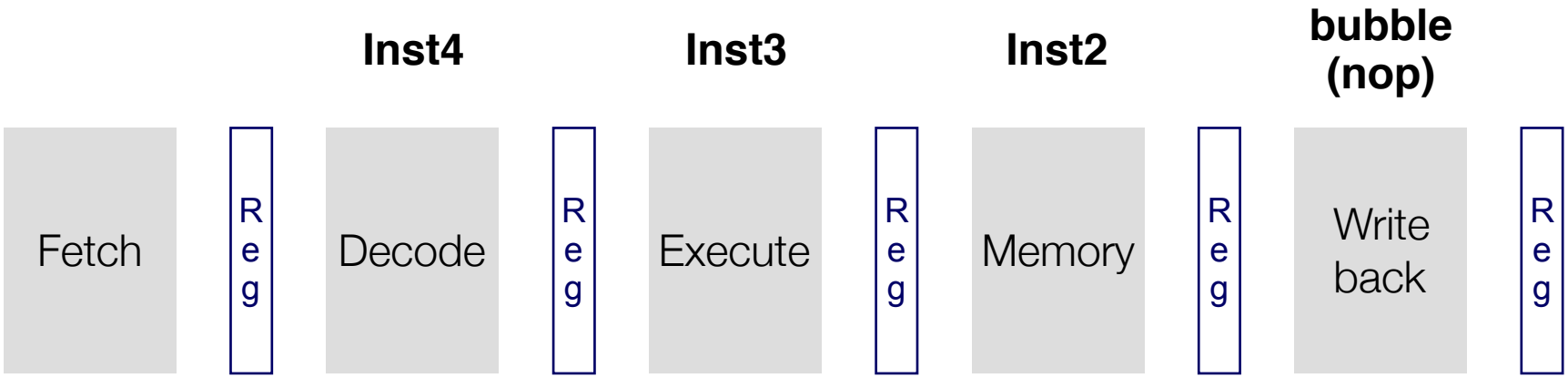
Stalling Illustration



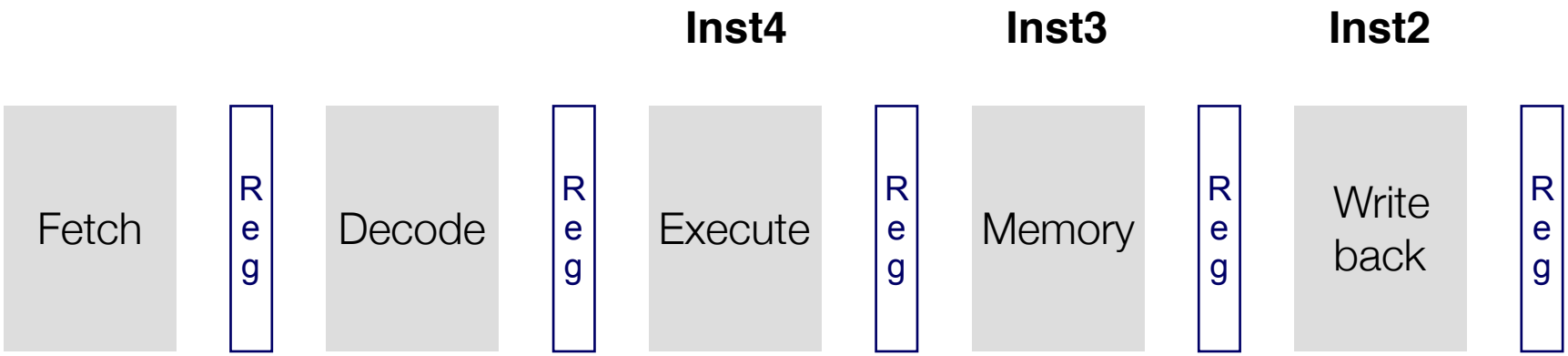
Stalling Illustration



Stalling Illustration



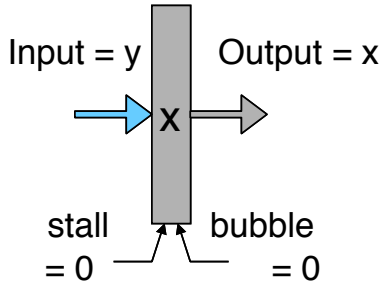
Stalling Illustration



Pipeline Register Mode to Support Stall and Bubble

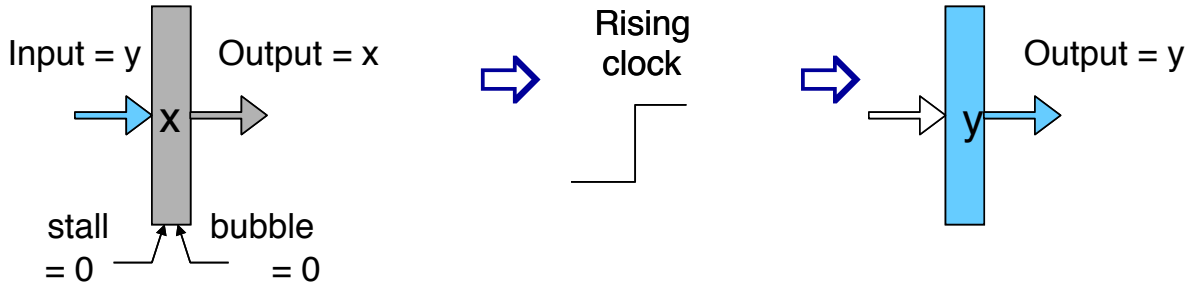
Pipeline Register Mode to Support Stall and Bubble

Normal



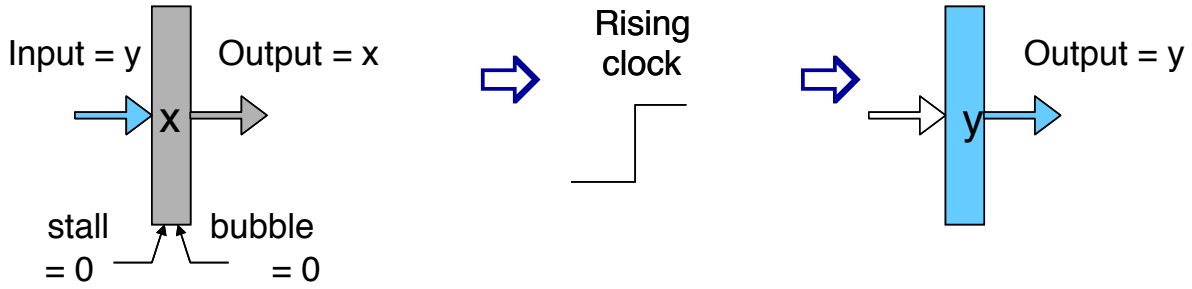
Pipeline Register Mode to Support Stall and Bubble

Normal

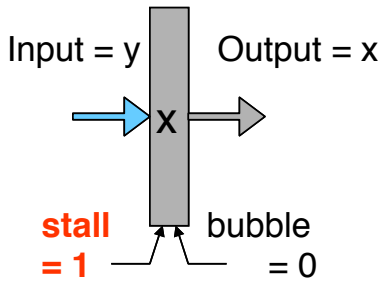


Pipeline Register Mode to Support Stall and Bubble

Normal

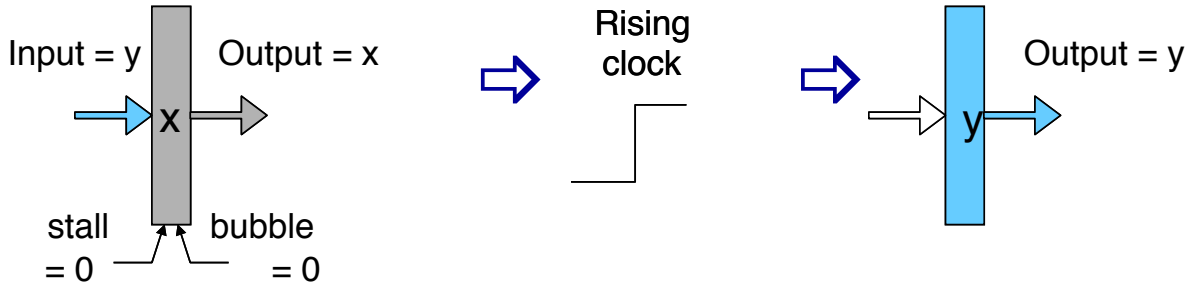


Stall

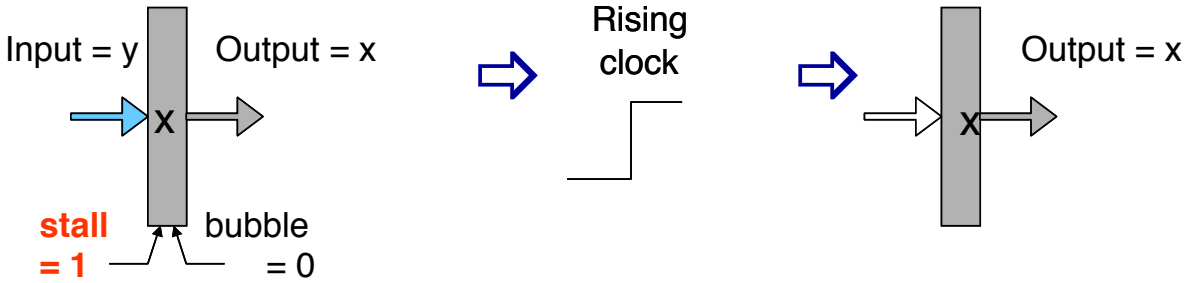


Pipeline Register Mode to Support Stall and Bubble

Normal

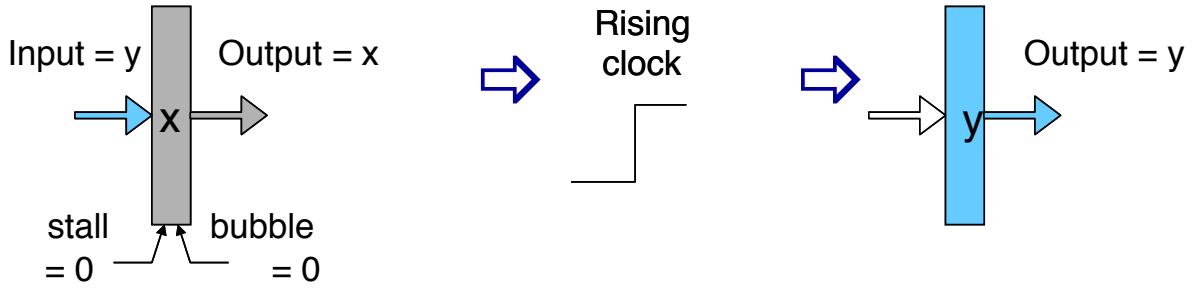


Stall

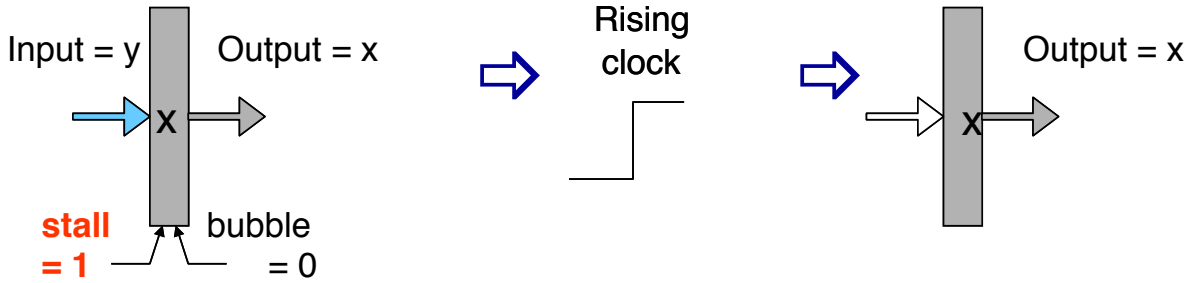


Pipeline Register Mode to Support Stall and Bubble

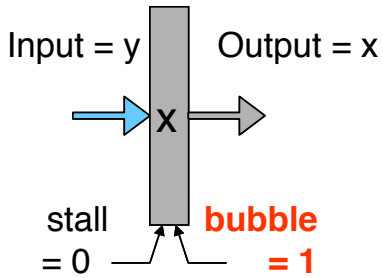
Normal



Stall

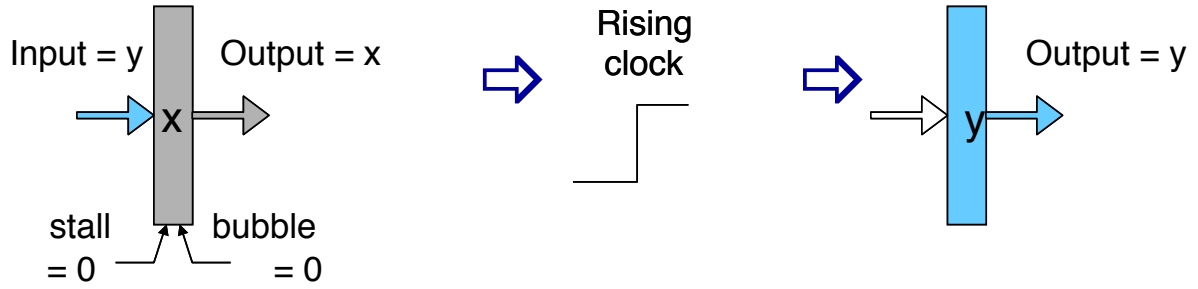


Bubble

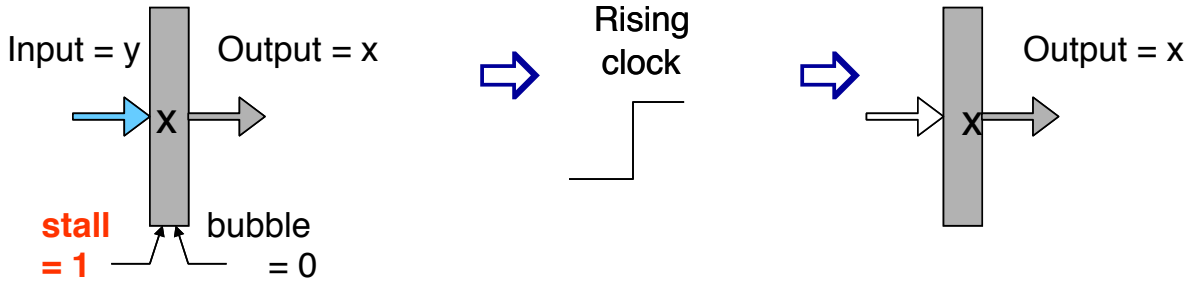


Pipeline Register Mode to Support Stall and Bubble

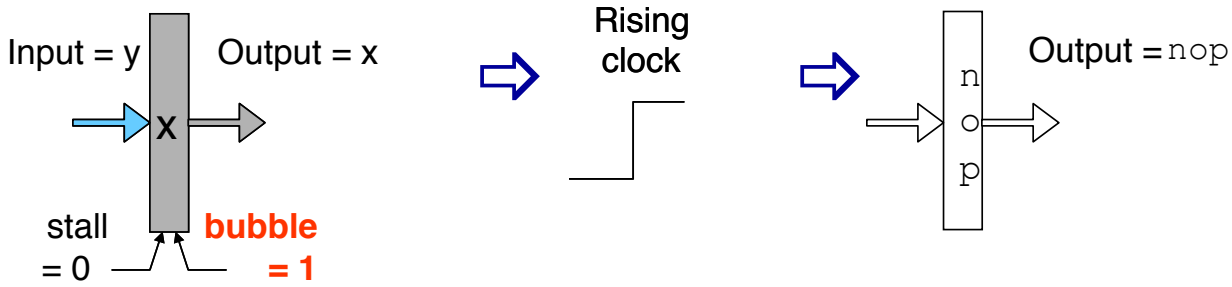
Normal



Stall

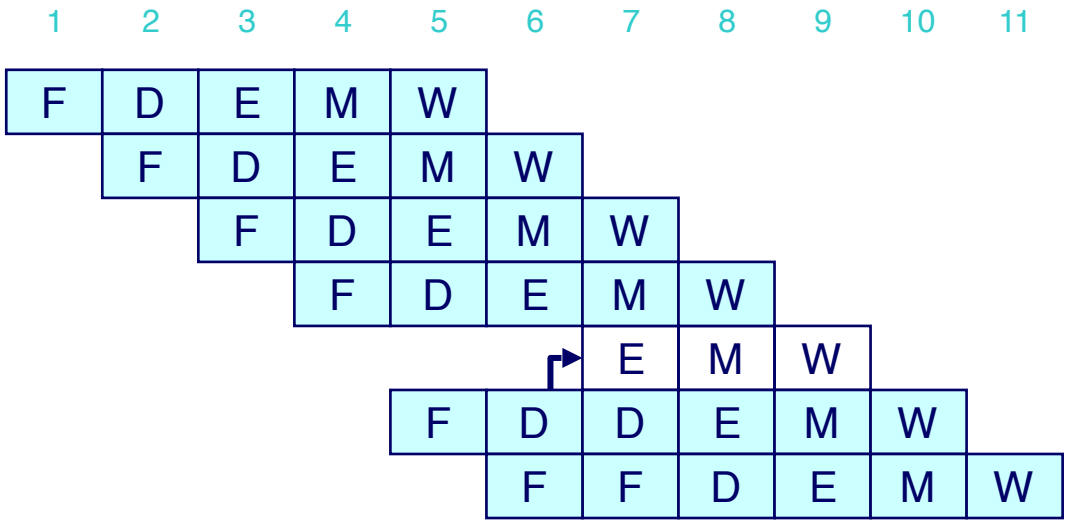


Bubble



Stalling for Data Dependencies

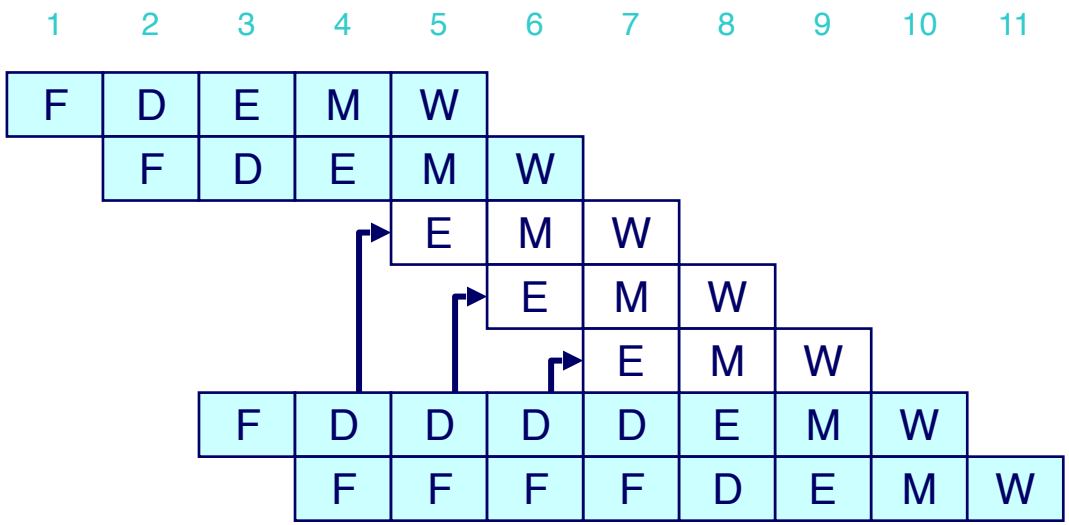
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Think of it as dynamically injecting nops into execute stage

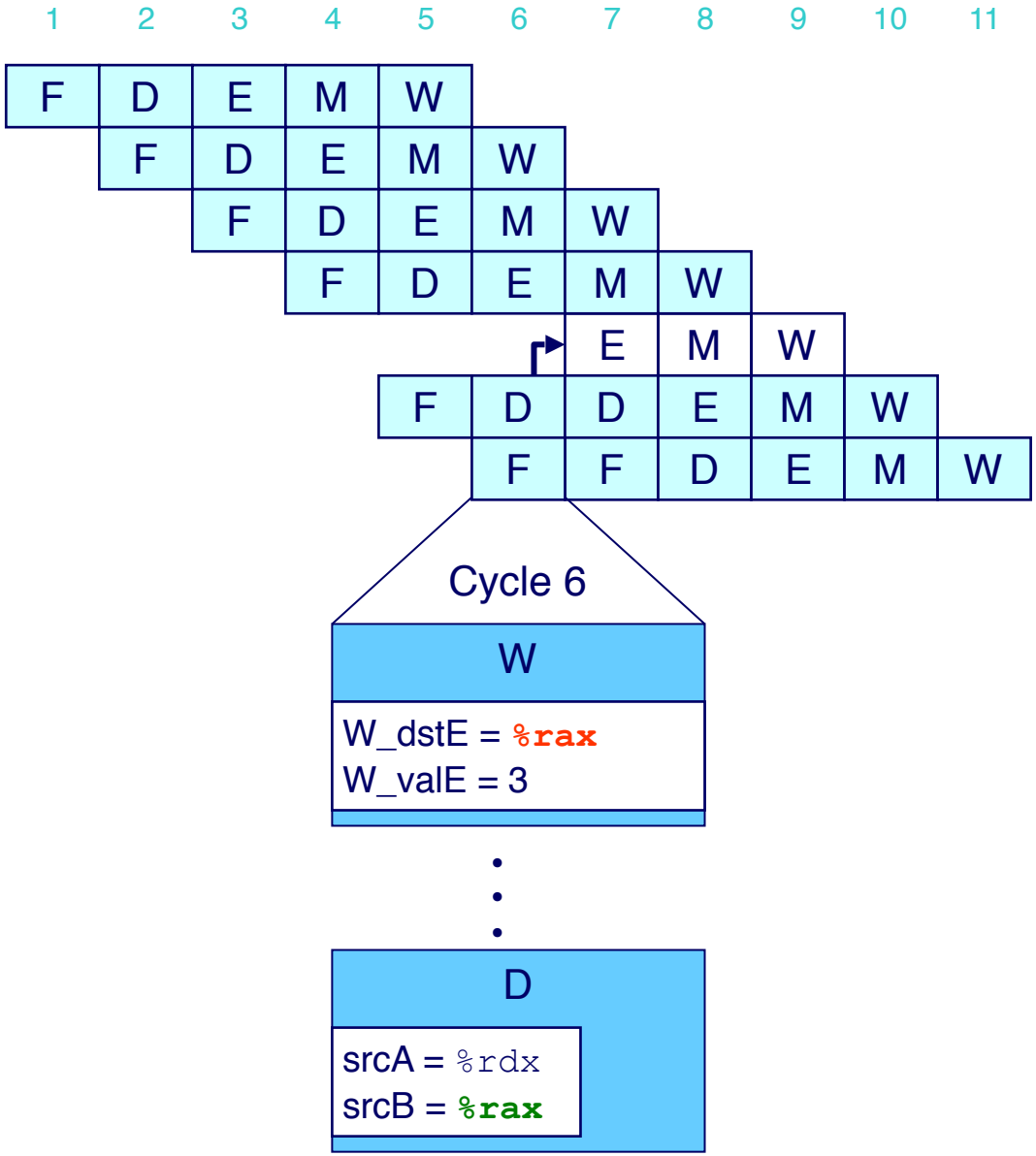
Stalling X3

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



Detecting Stall Condition

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

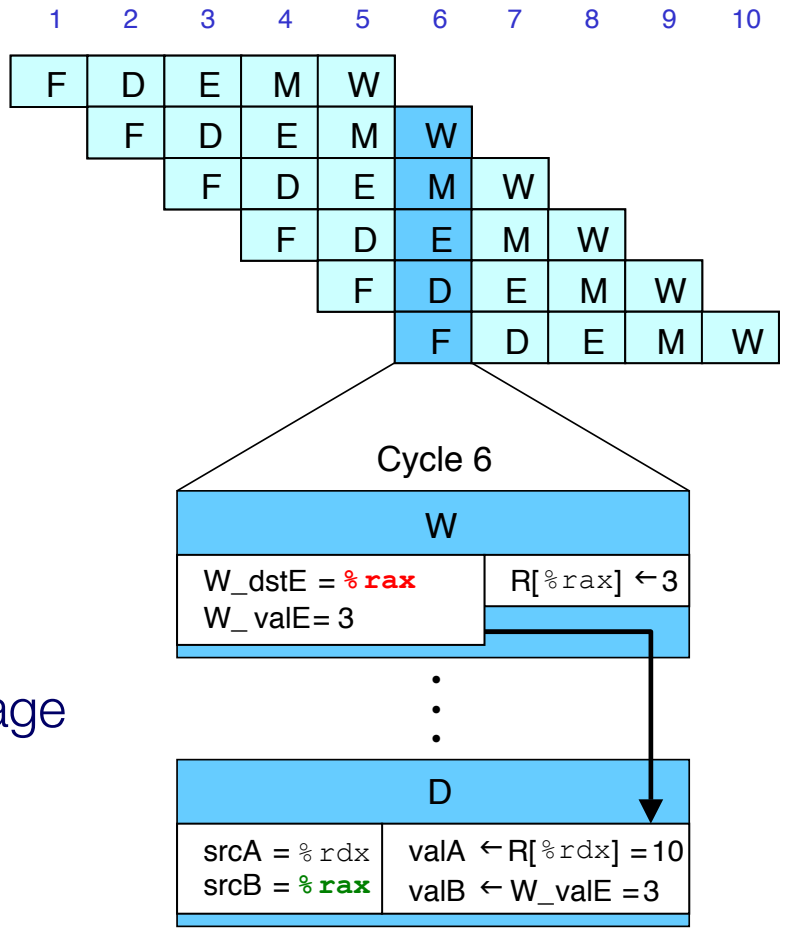
- Value generated in execute or memory stage

Trick

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

Data Forwarding Example

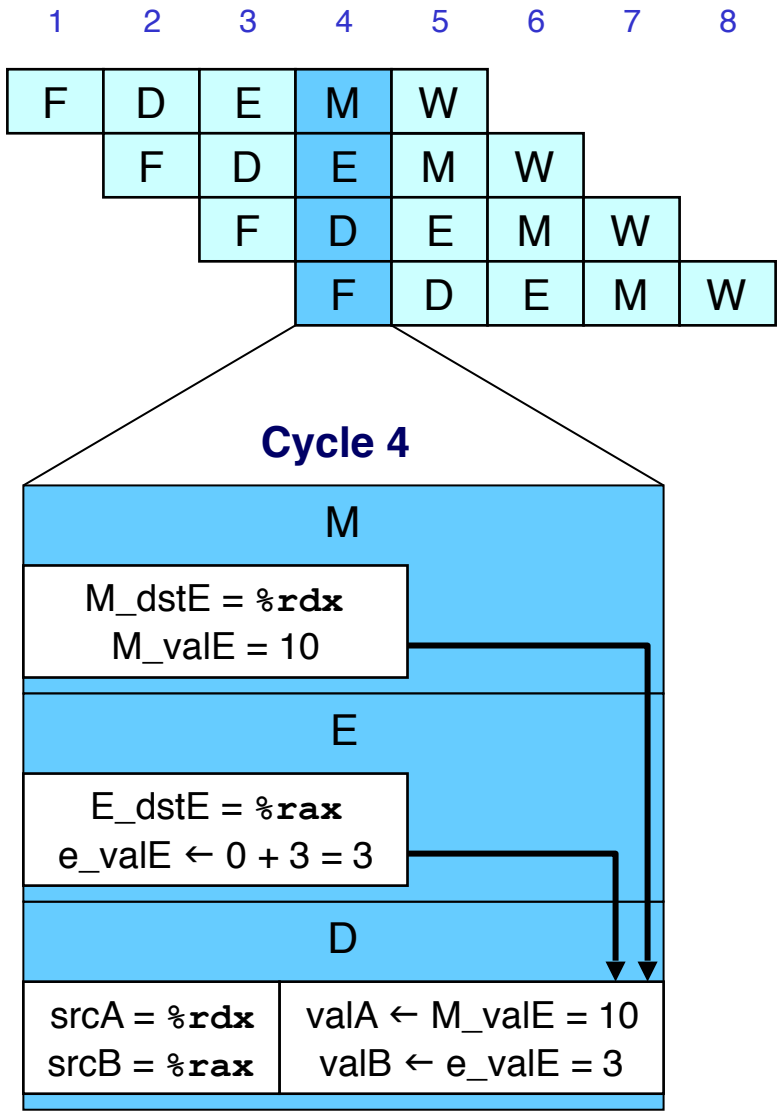
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

Data Forwarding Example #2

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



Register `%rdx`

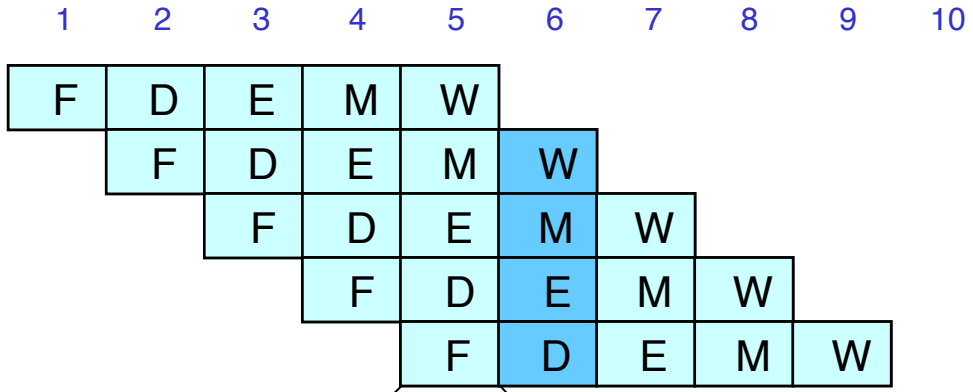
- Generated by ALU during previous cycle
- Forward from memory as valA

Register `%rax`

- Value just generated by ALU
- Forward from execute as valB

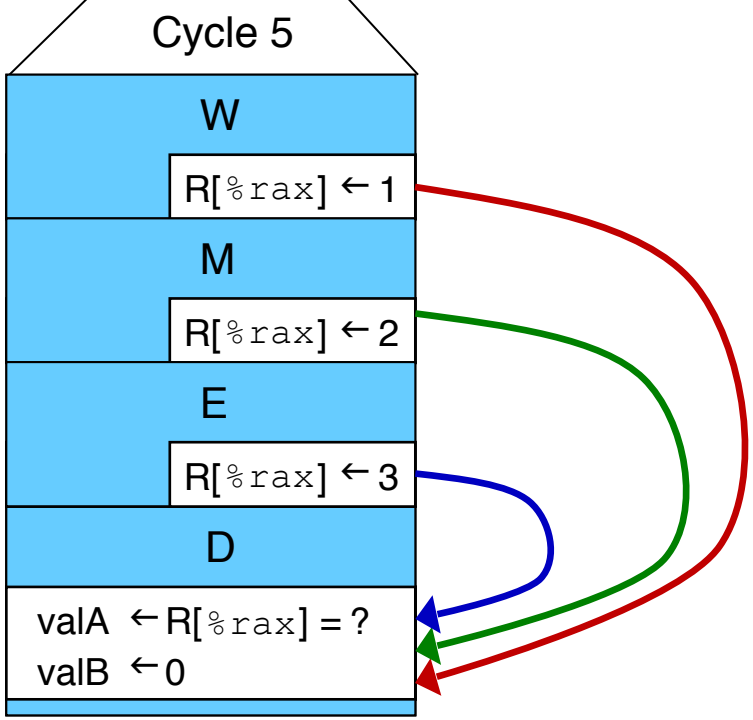
Forwarding Priority

```
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



Multiple Forwarding Choices

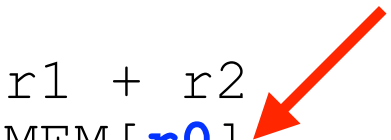
- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage



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

```
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
```

If you are interested, Google “**Tomasolu Algorithm**.” It is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.