CSC 252: Computer Organization Spring 2019: Lecture 14

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Mid-term on Thursday, this room, 75 mins**

Announcements

- Lab 2 grades are out. Talk to Yu if you saw issues.
- Thursday office hours canceled. Traveling to meetings.

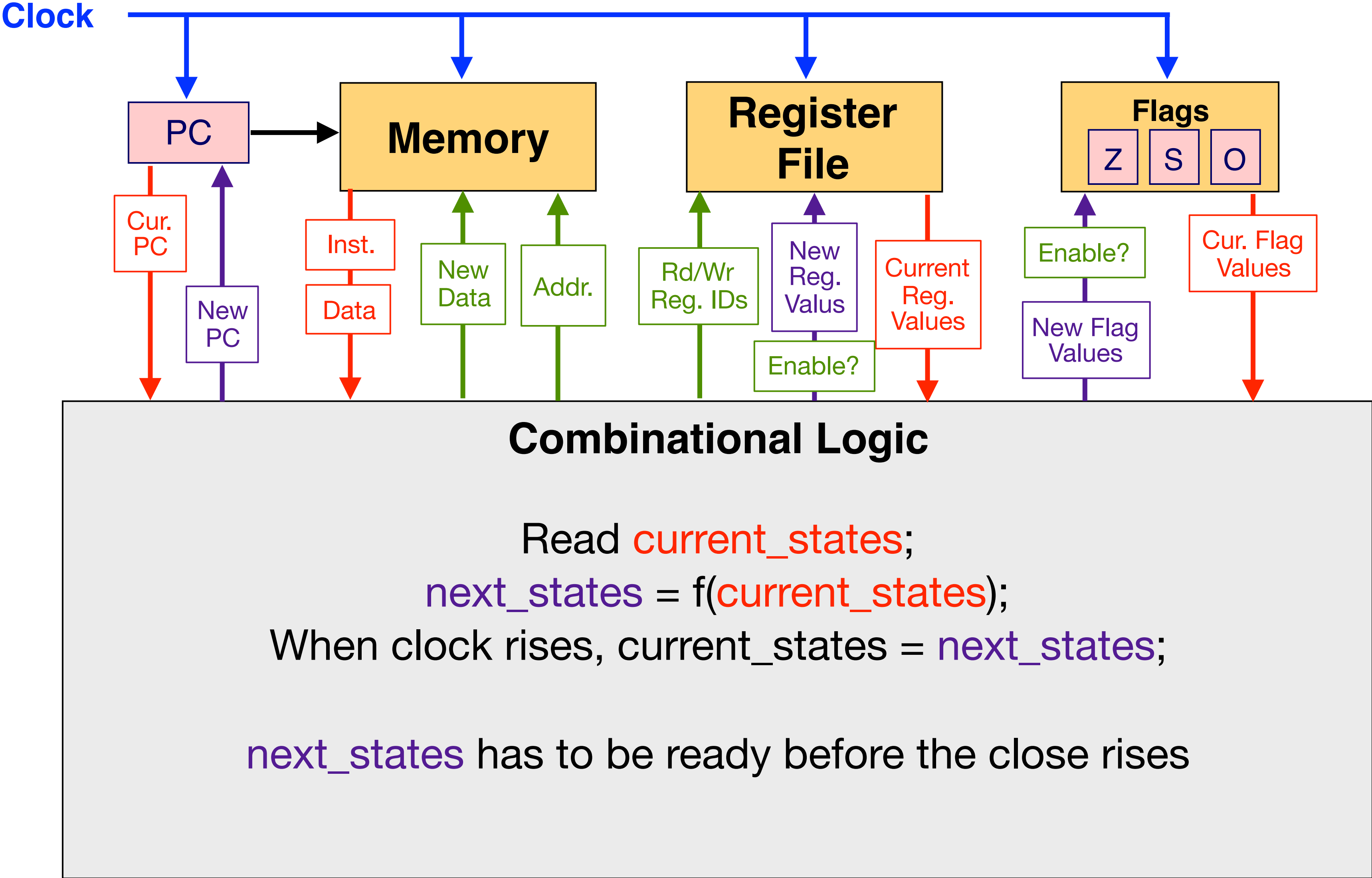
Announcements

- Lab 2 grades are out. Talk to Yu if you saw issues.
- Thursday office hours canceled. Traveling to meetings.
- Mid-term exam: March 7; in class.
- Past exam and problem set: http://www.cs.rochester.edu/courses/252/spring2019/handouts.html

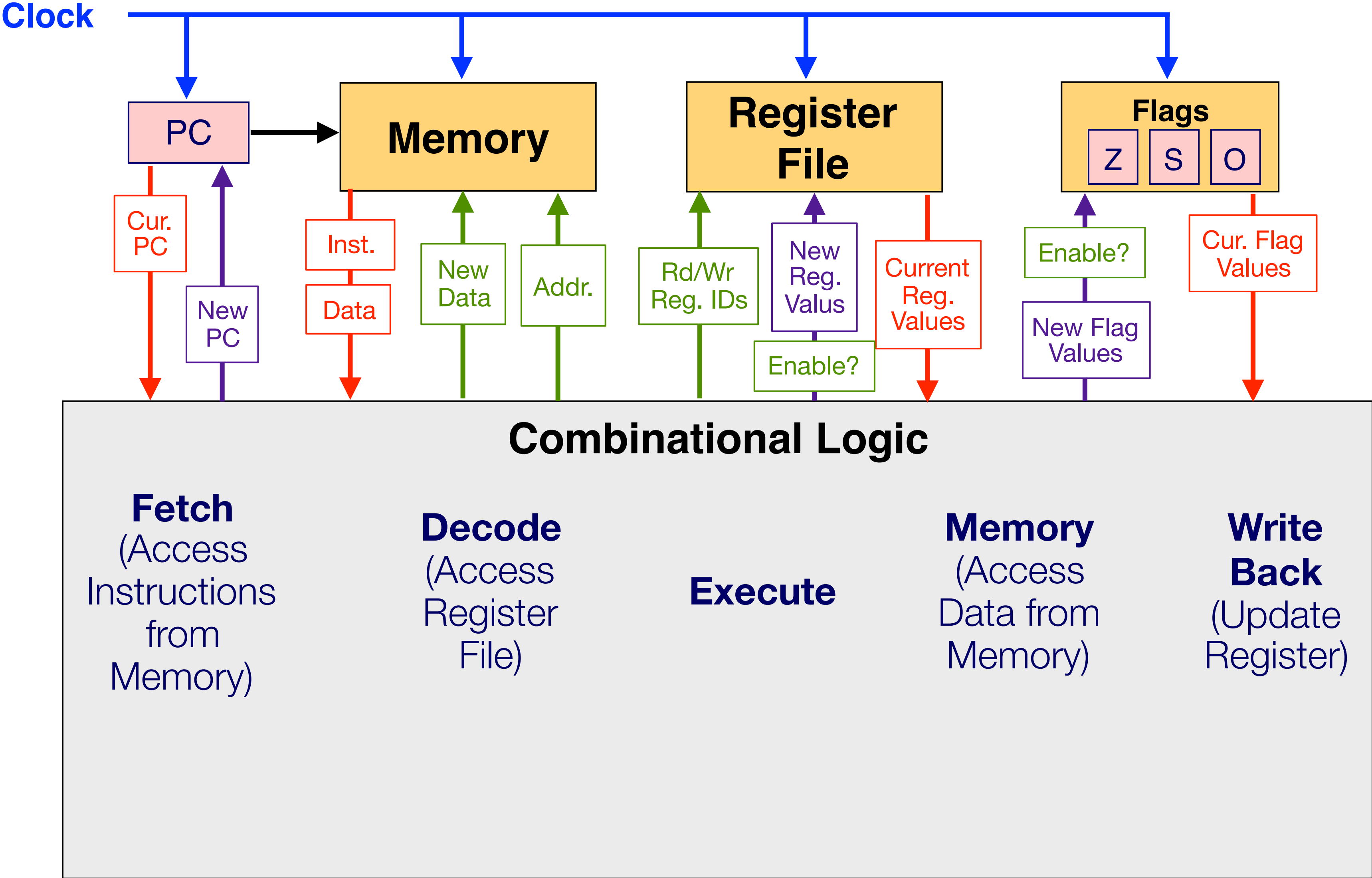
Announcements

- Lab 2 grades are out. Talk to Yu if you saw issues.
- Thursday office hours canceled. Traveling to meetings.
- Mid-term exam: March 7; in class.
- Past exam and problem set: http://www.cs.rochester.edu/courses/252/spring2019/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.

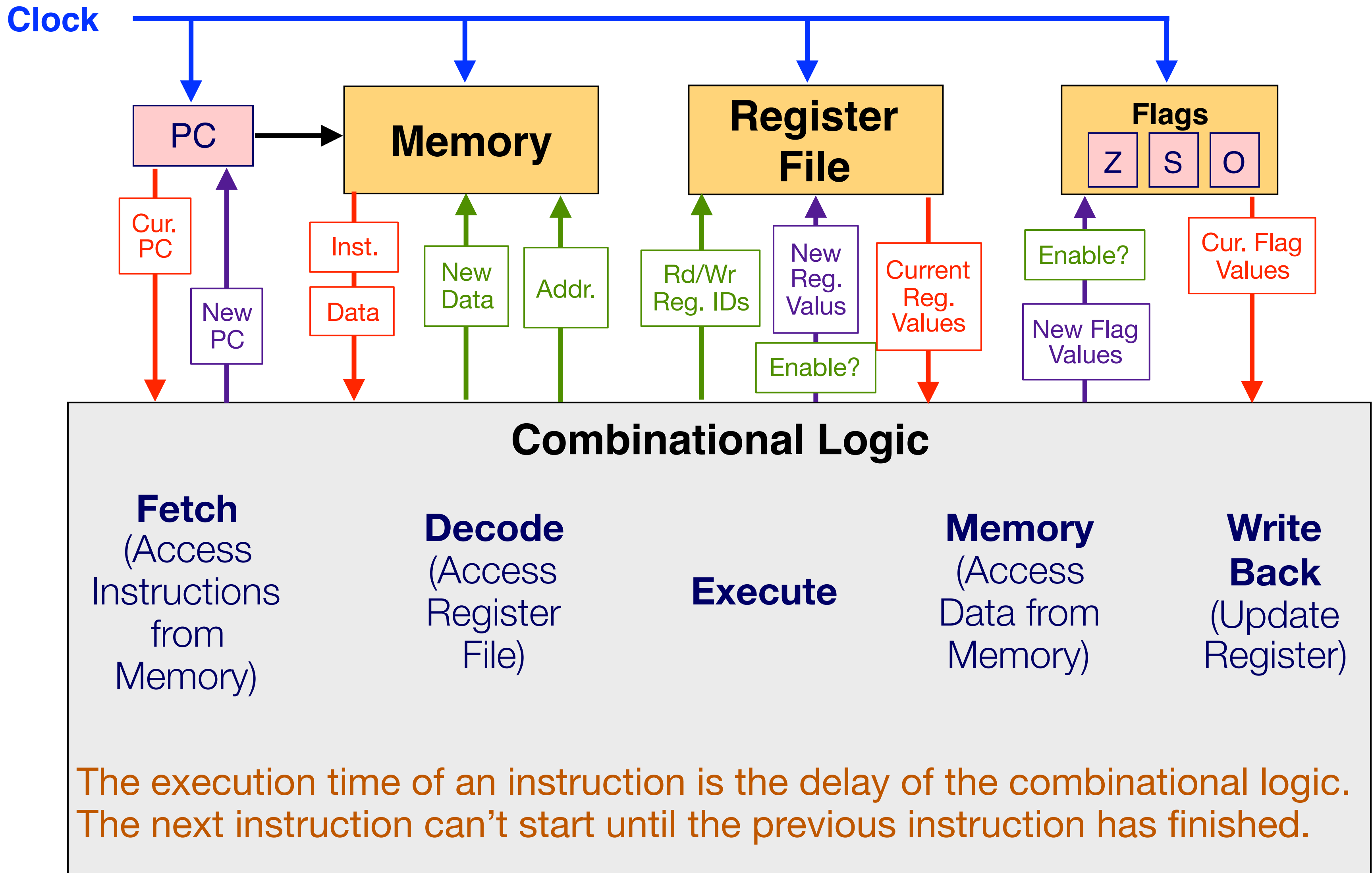
Single-Cycle Microarchitecture



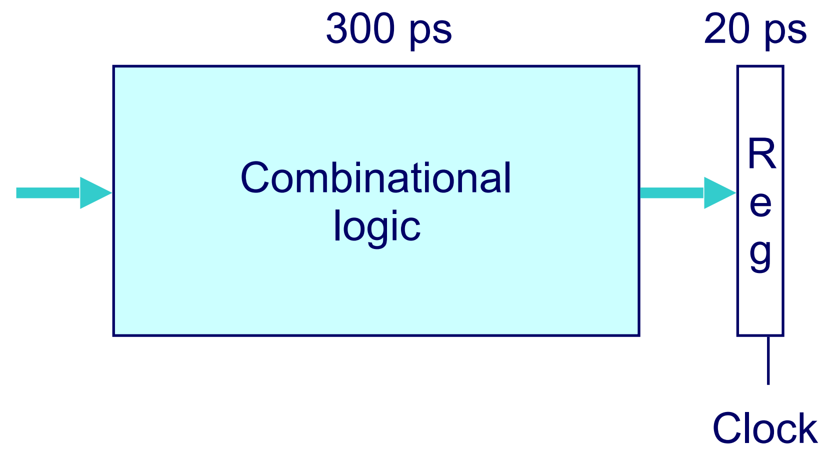
Single-Cycle Microarchitecture



Single-Cycle Microarchitecture

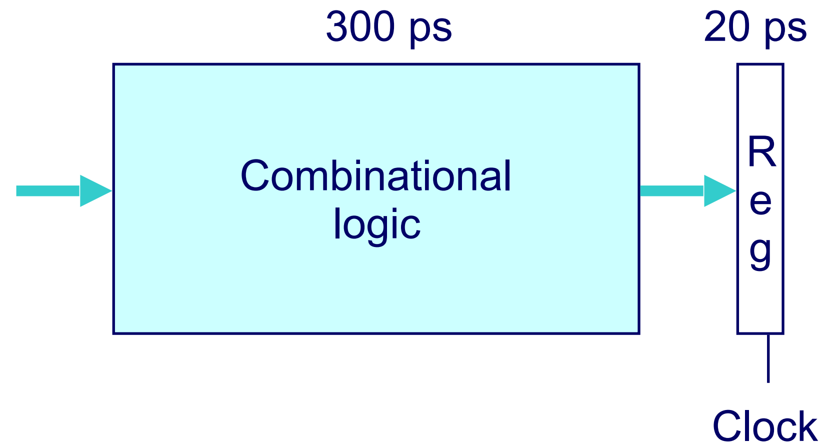


Pipeline



System Characteristics

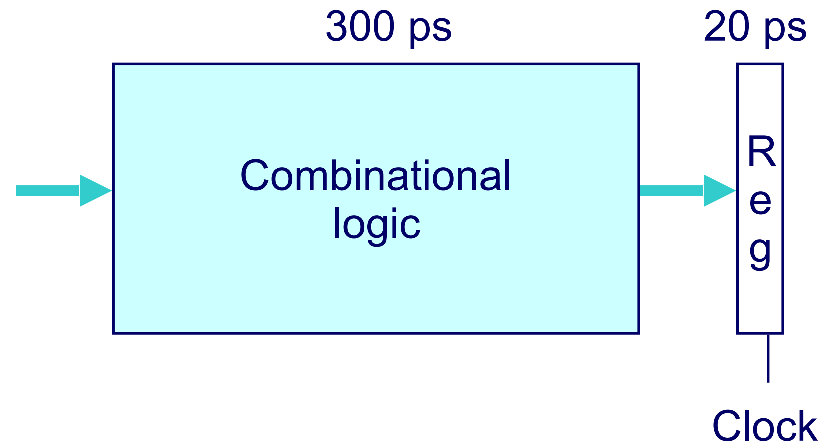
Pipeline



System Characteristics

- Computation requires total of 300 picoseconds

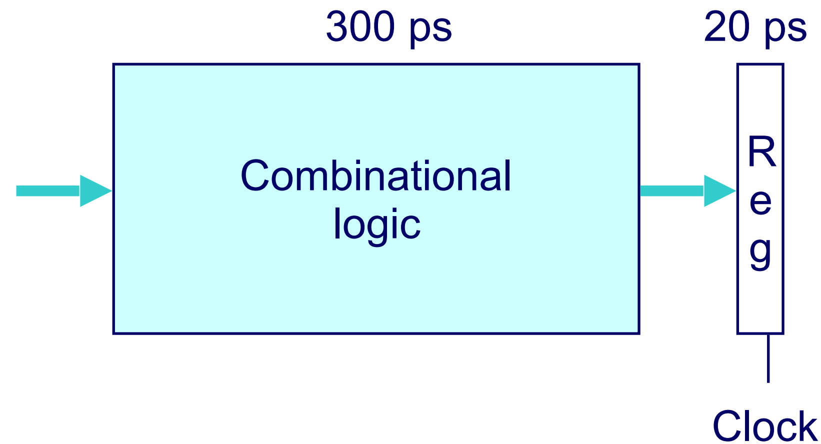
Pipeline



System Characteristics

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register

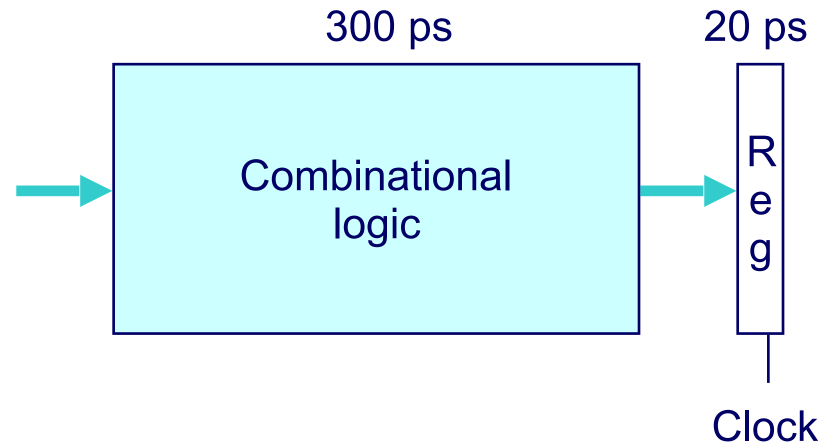
Pipeline



System Characteristics

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Delay for each instruction: 320 ps

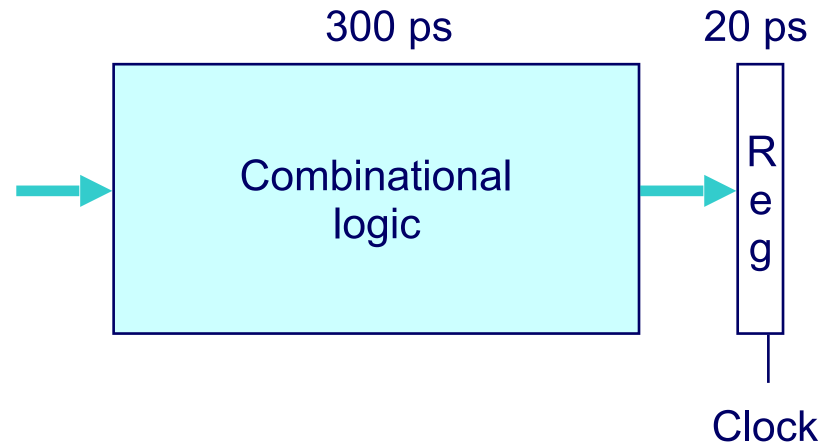
Pipeline



System Characteristics

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Delay for each instruction: 320 ps
- The cycle time of the clock has to be at least 320 ps

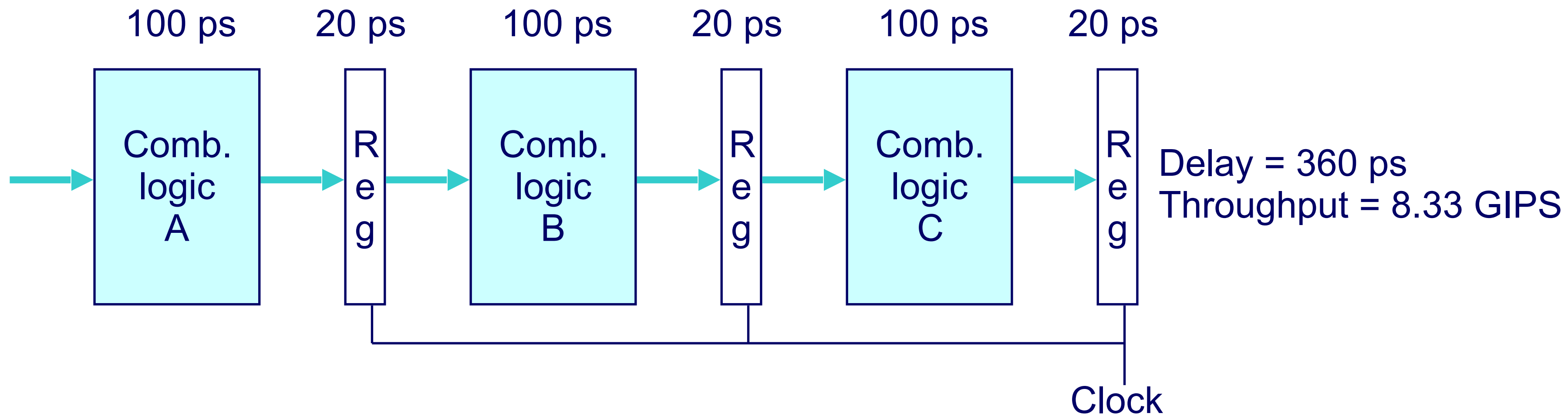
Pipeline



System Characteristics

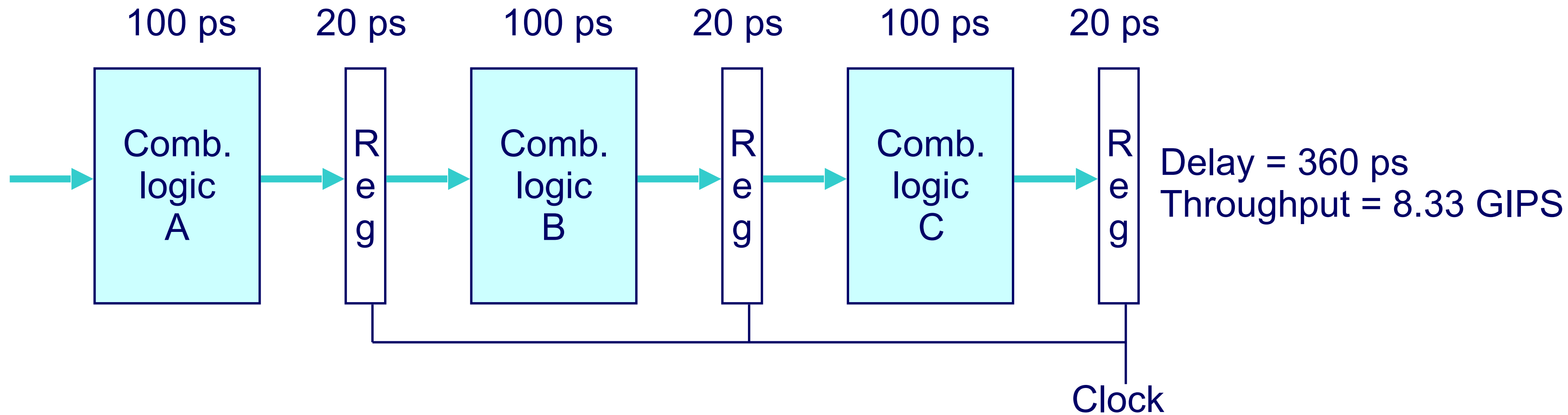
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Delay for each instruction: 320 ps
- The cycle time of the clock has to be at least 320 ps
- Throughput (how many operations can the system handle in a second): 3.12 Giga Instructions Per Second (GIPS)

3-Stage Pipelined Version



System Characteristics

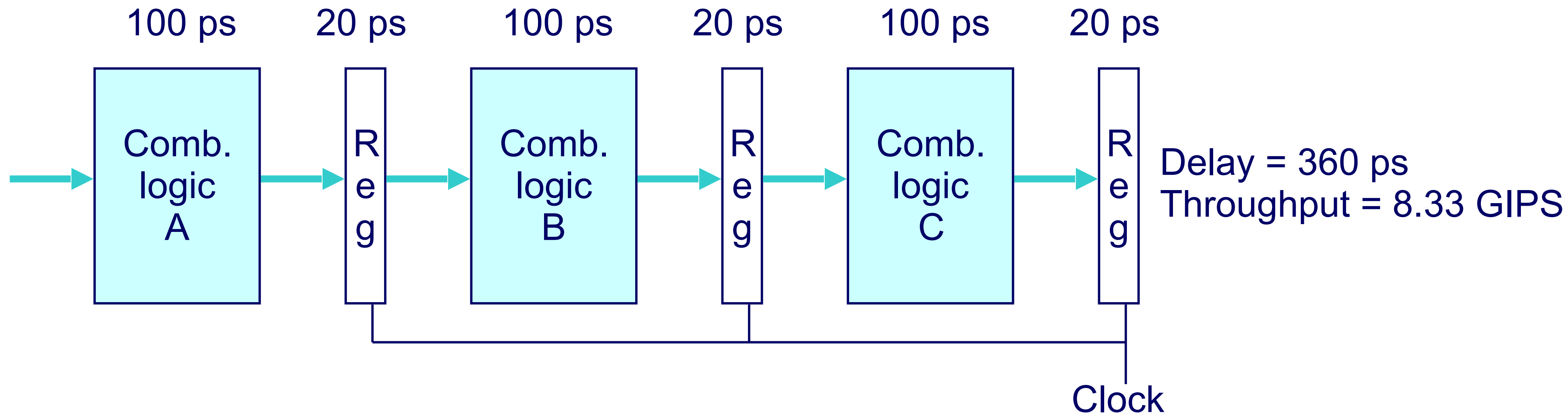
3-Stage Pipelined Version



System Characteristics

- Can push a new instruction every 120 ps

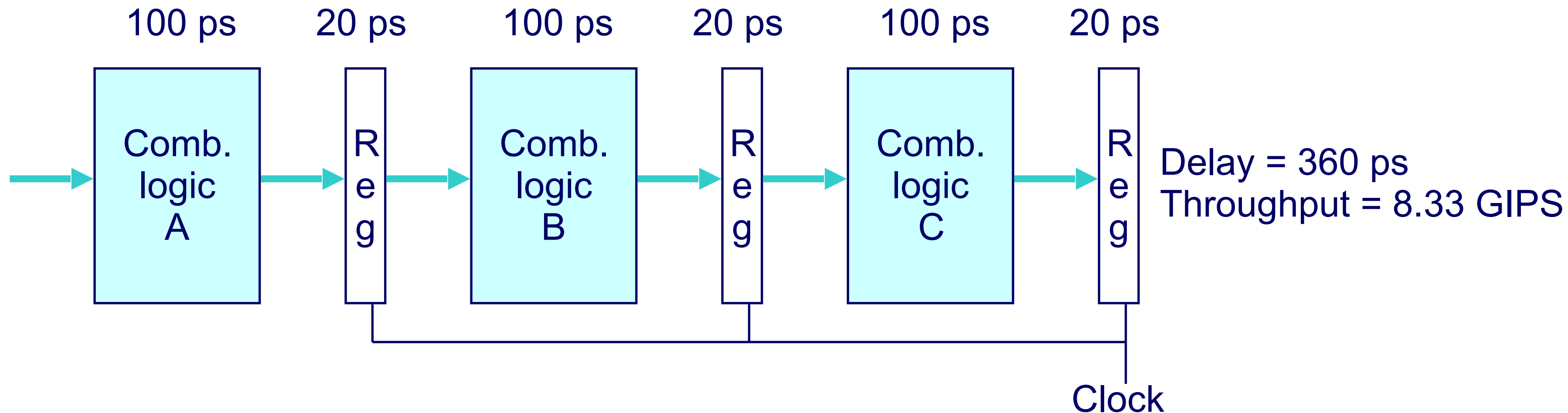
3-Stage Pipelined Version



System Characteristics

- Can push a new instruction every 120 ps
- The cycle time could be reduced to 120 ps

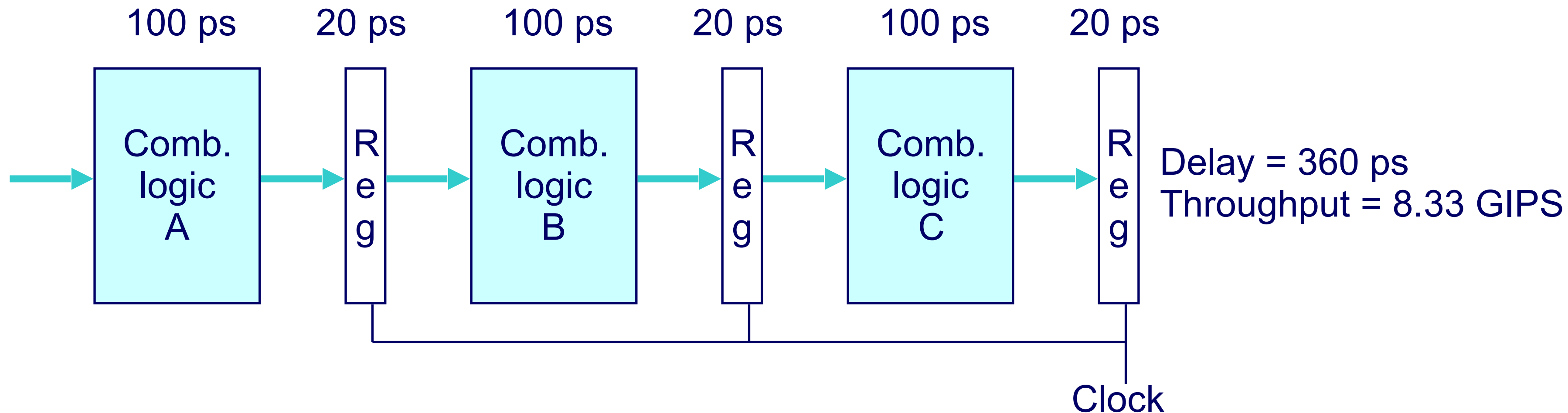
3-Stage Pipelined Version



System Characteristics

- Can push a new instruction every 120 ps
- The cycle time could be reduced to 120 ps
- Delay for each instruction: 360 ps (60 ps in loading registers)

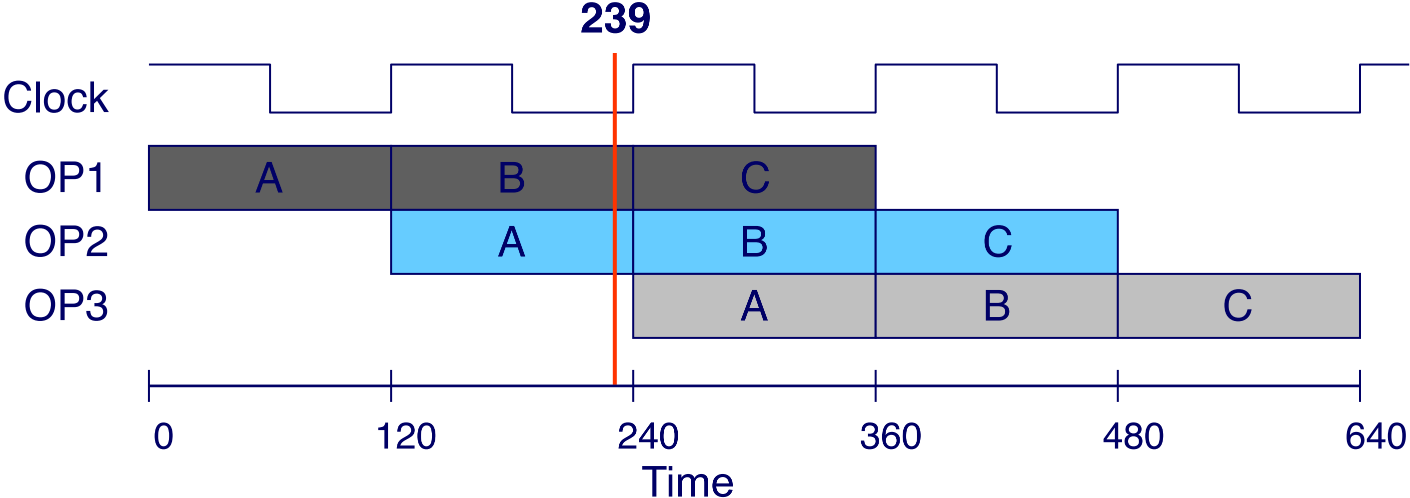
3-Stage Pipelined Version

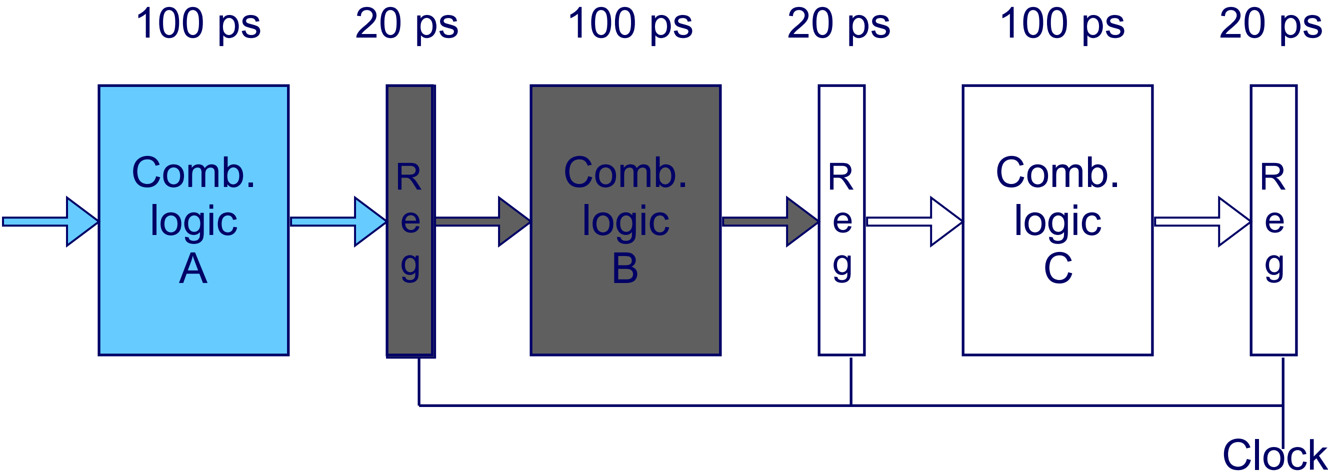


System Characteristics

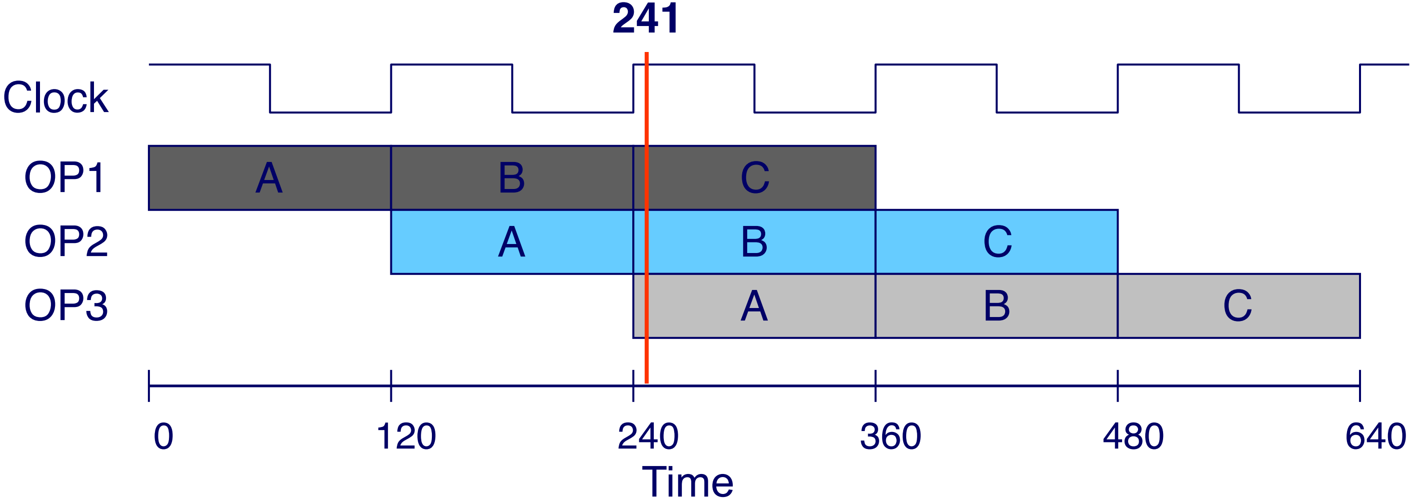
- Can push a new instruction every 120 ps
- The cycle time could be reduced to 120 ps
- Delay for each instruction: 360 ps (60 ps in loading registers)
- Throughput (how many operations can the system handle in a second): 8.33 Giga Instructions Per Second (GIPS)

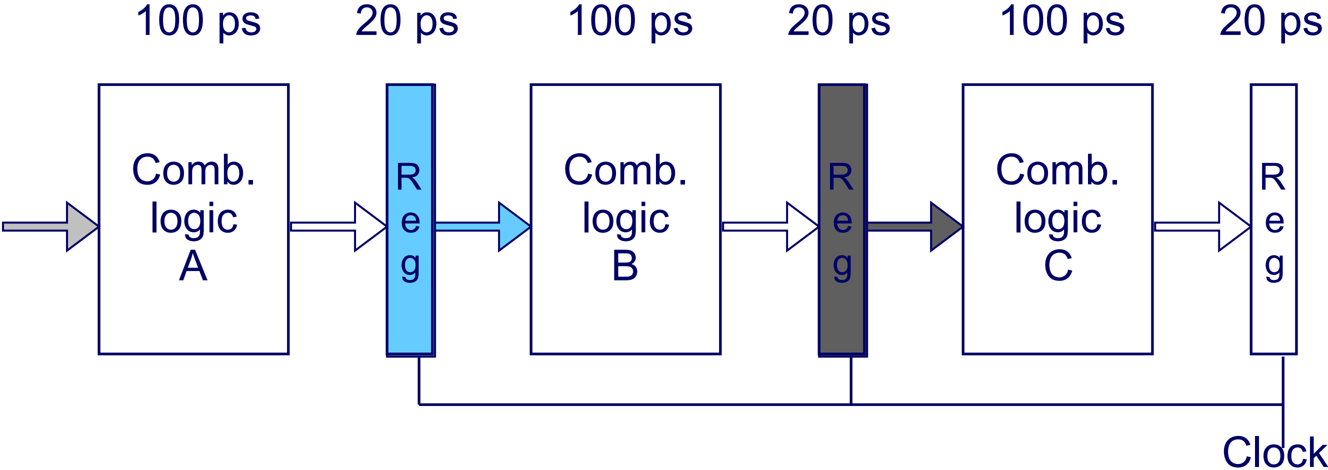
Operating a Pipeline



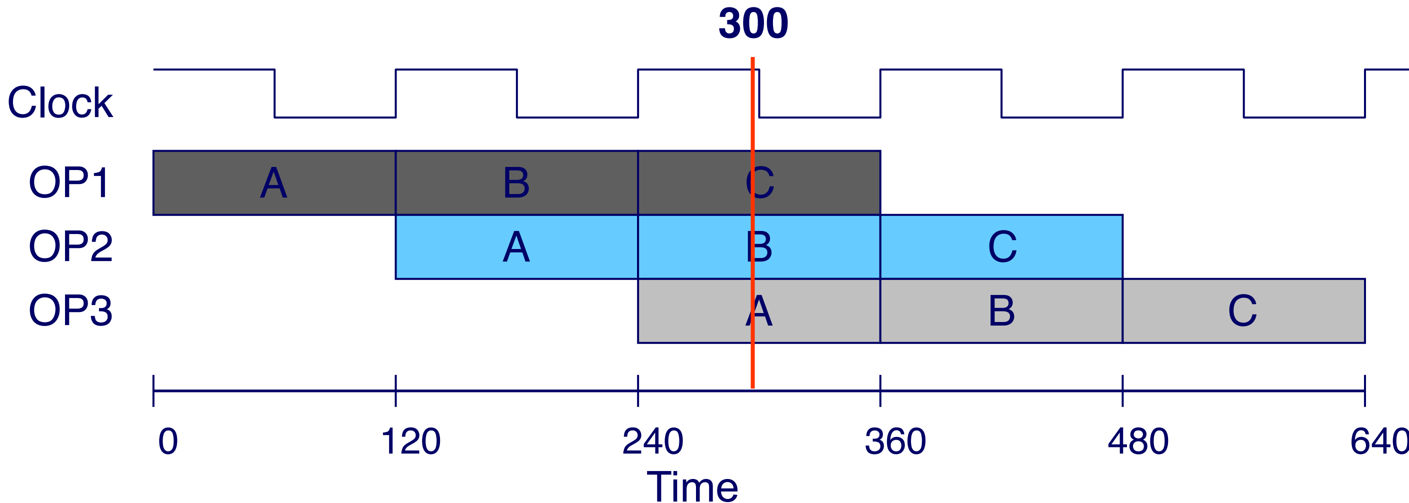


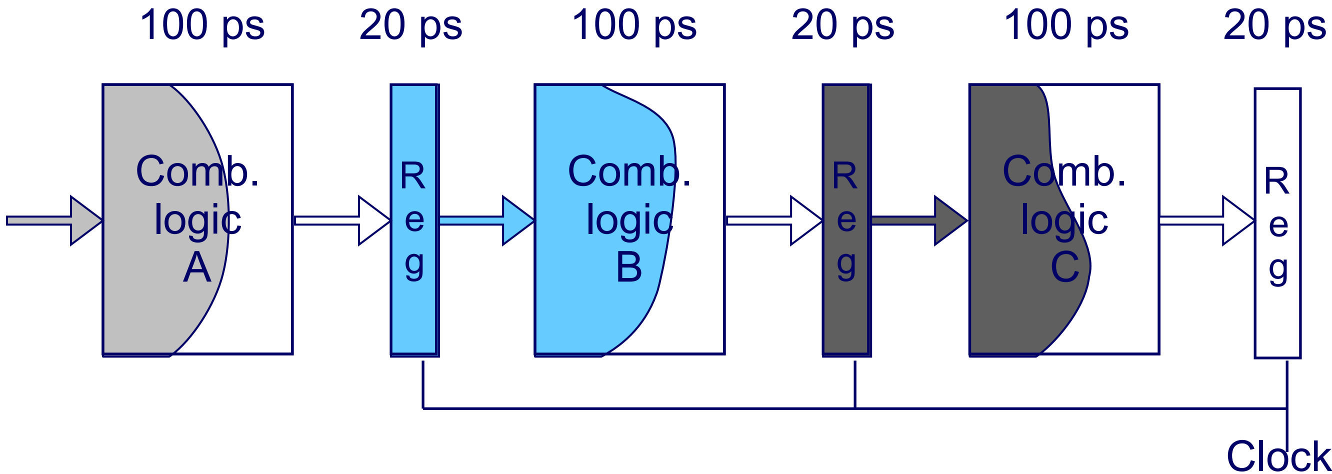
Operating a Pipeline



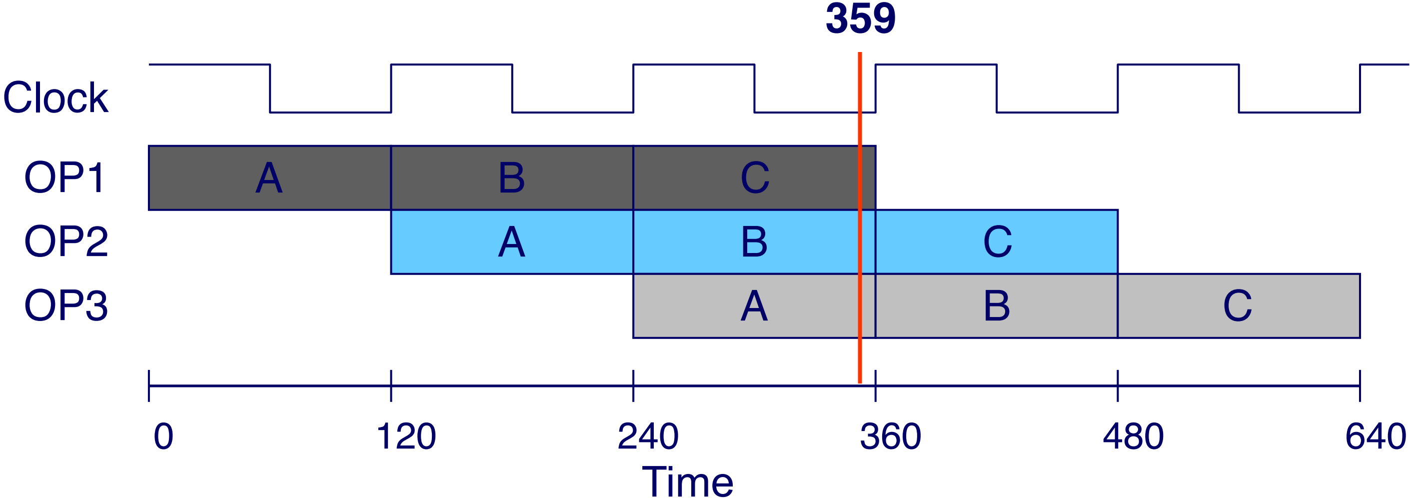


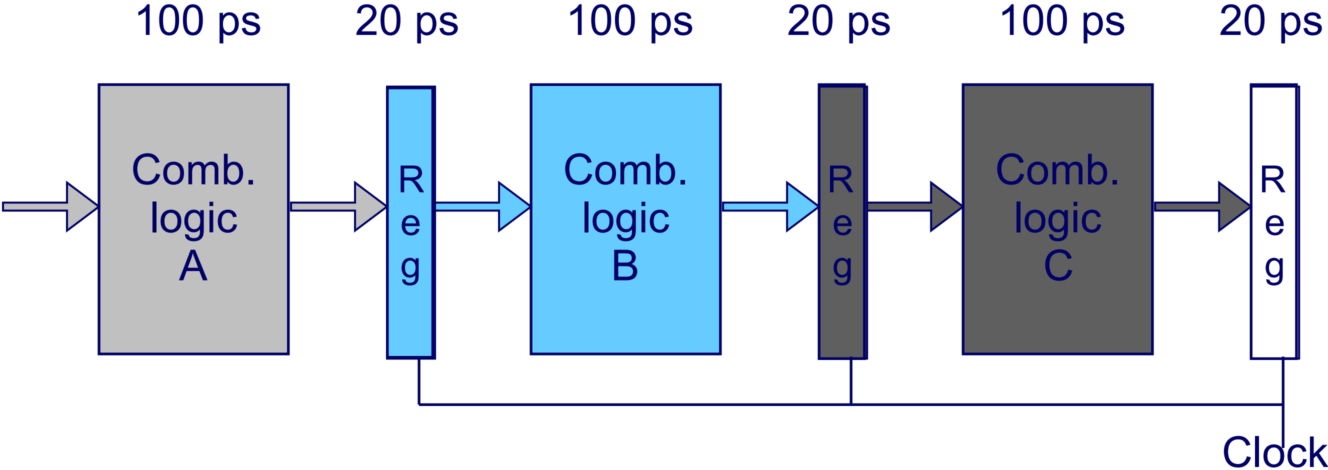
Operating a Pipeline





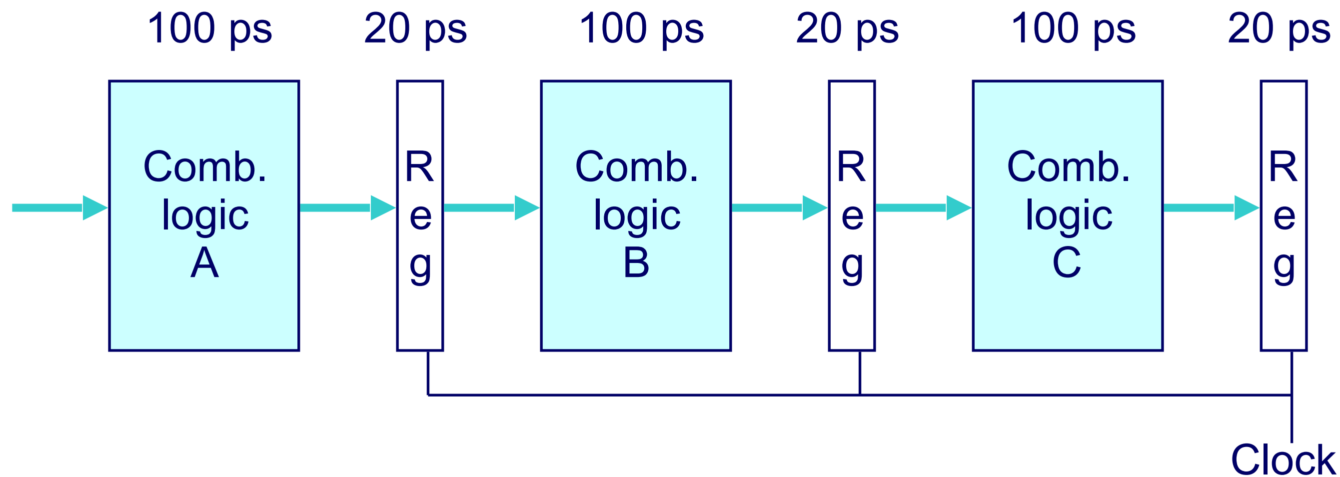
Operating a Pipeline

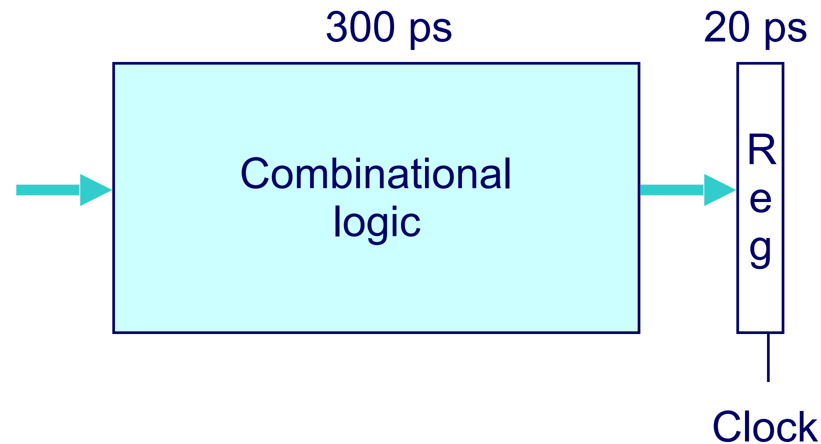




Pipeline Trade-offs

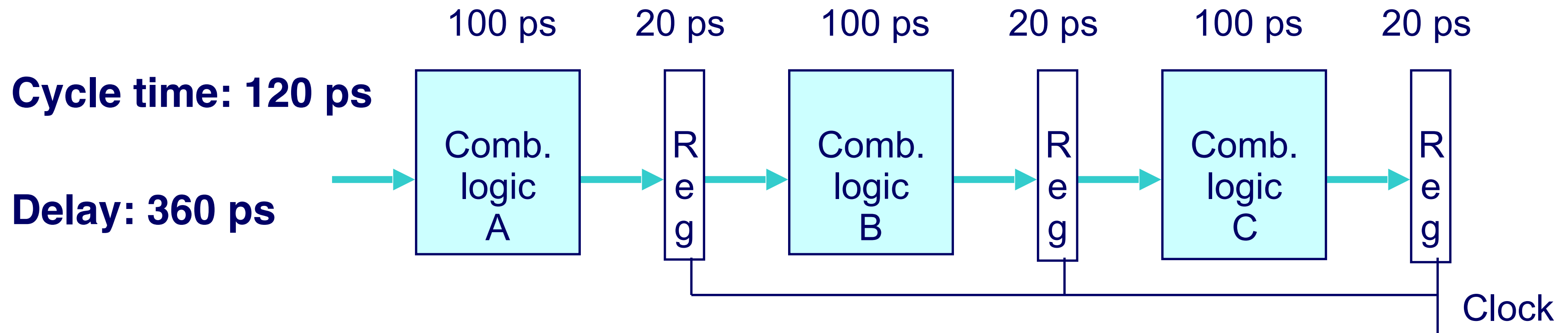
- Pros: Increase throughput. Can process more instructions in a given time span.
- Cons: Increase latency as new registers are needed between pipeline stages.





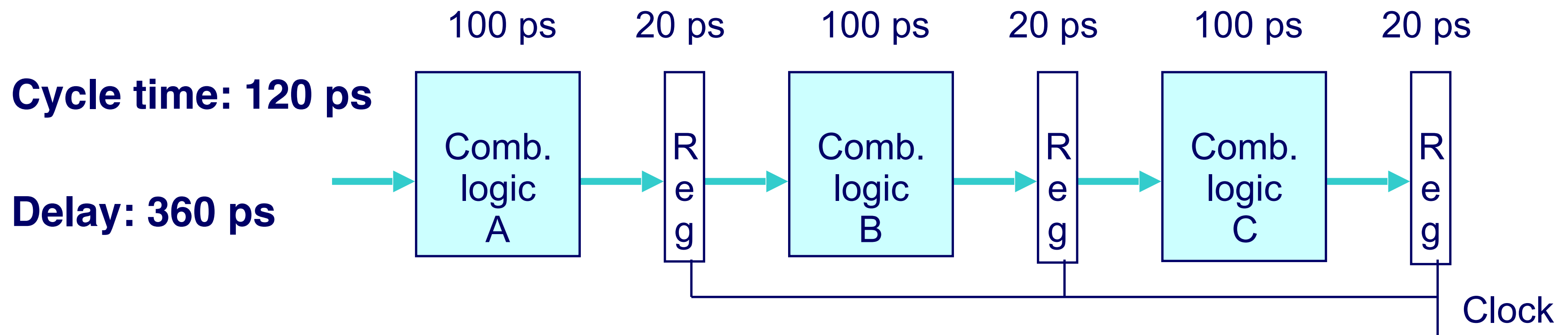
Unbalanced Pipeline

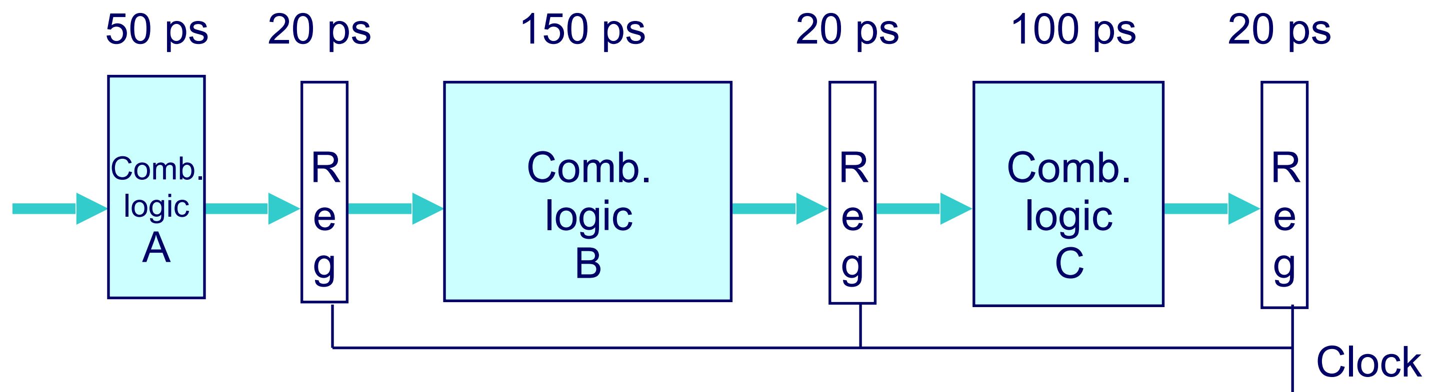
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput



Unbalanced Pipeline

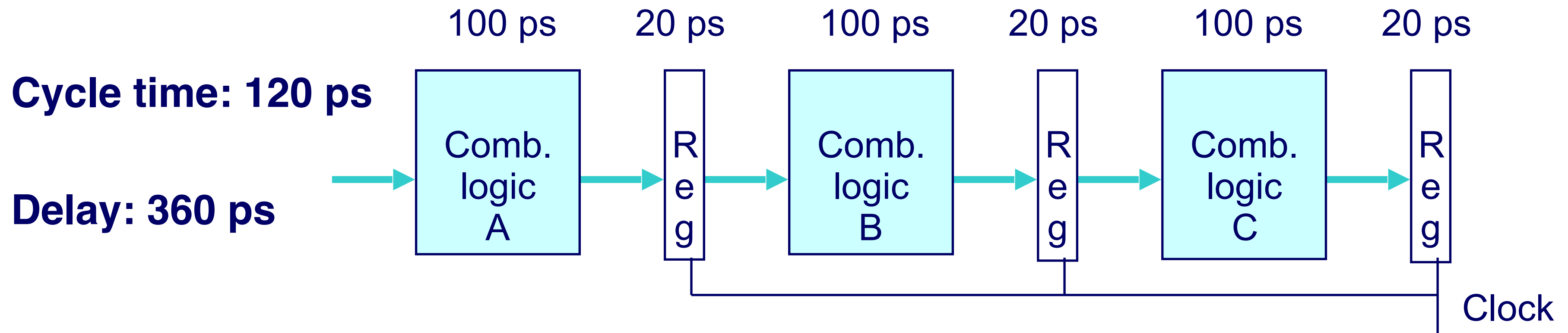
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

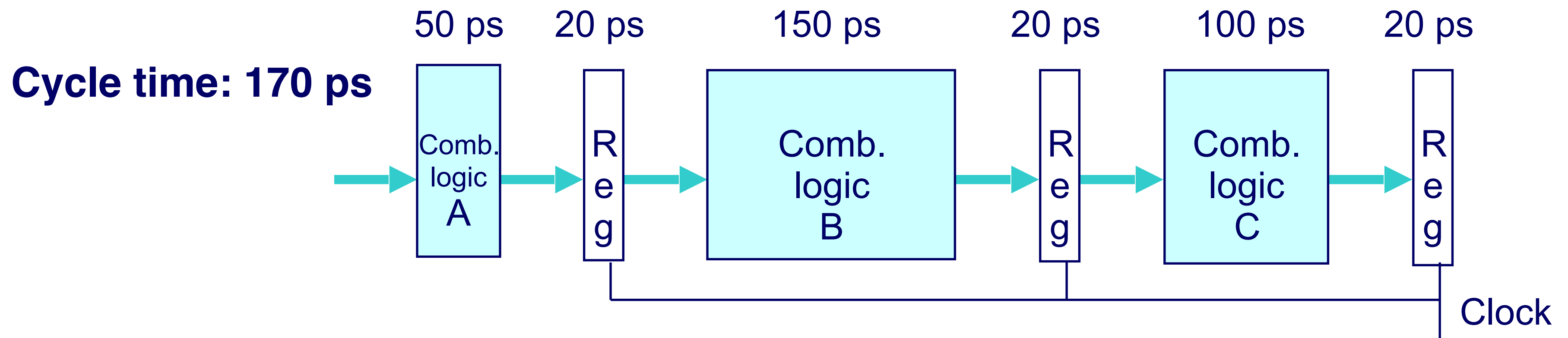




Unbalanced Pipeline

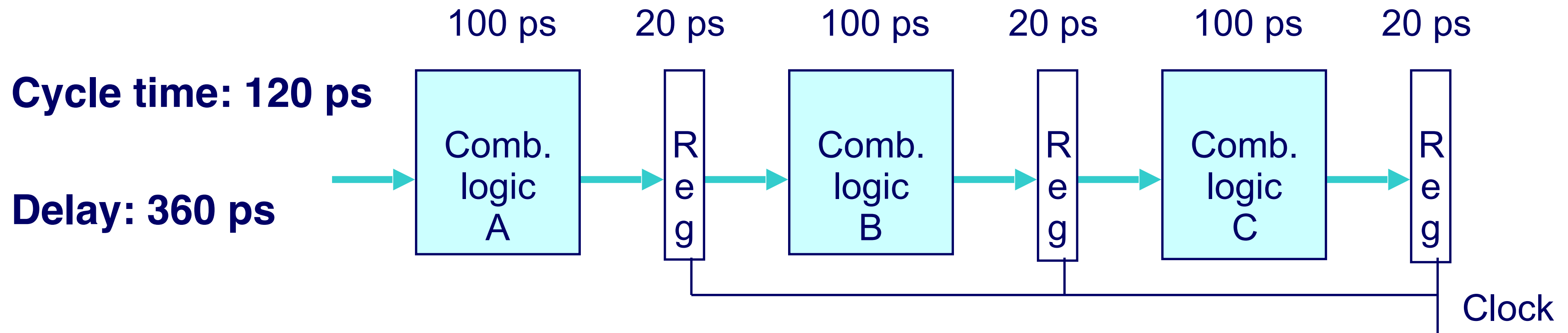
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

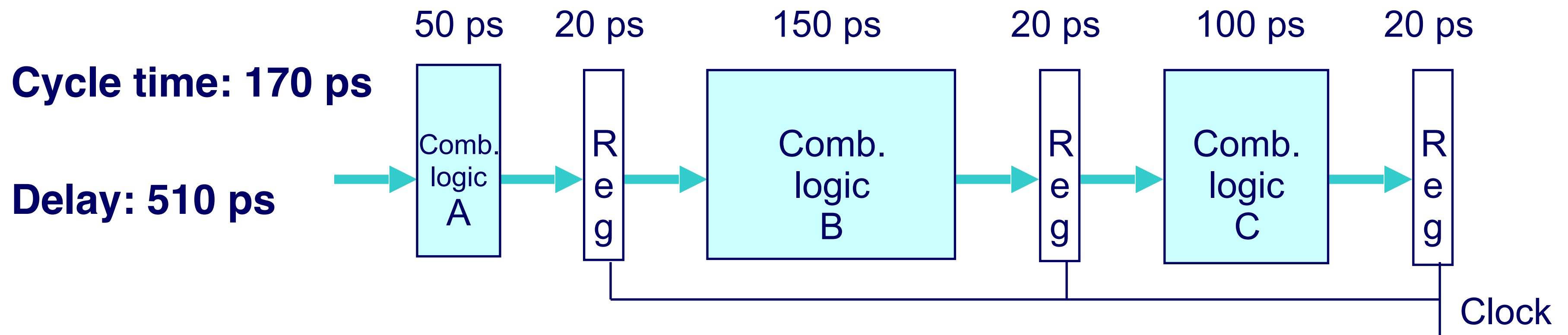




Unbalanced Pipeline

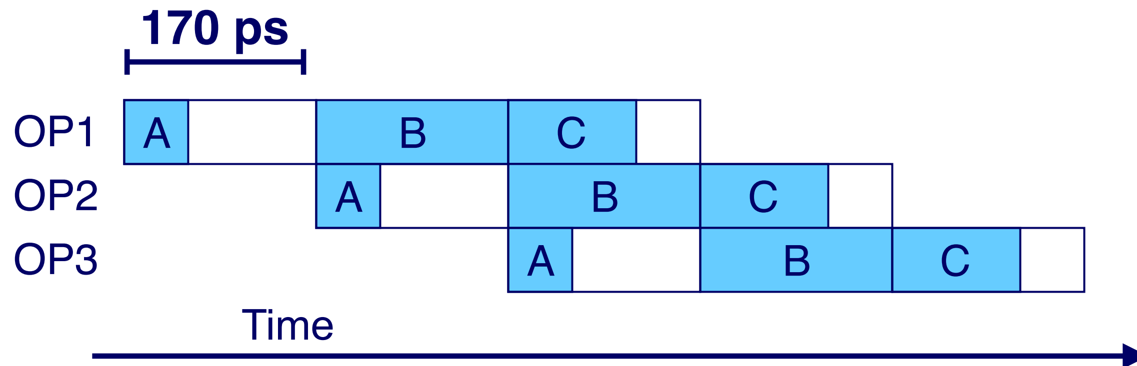
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

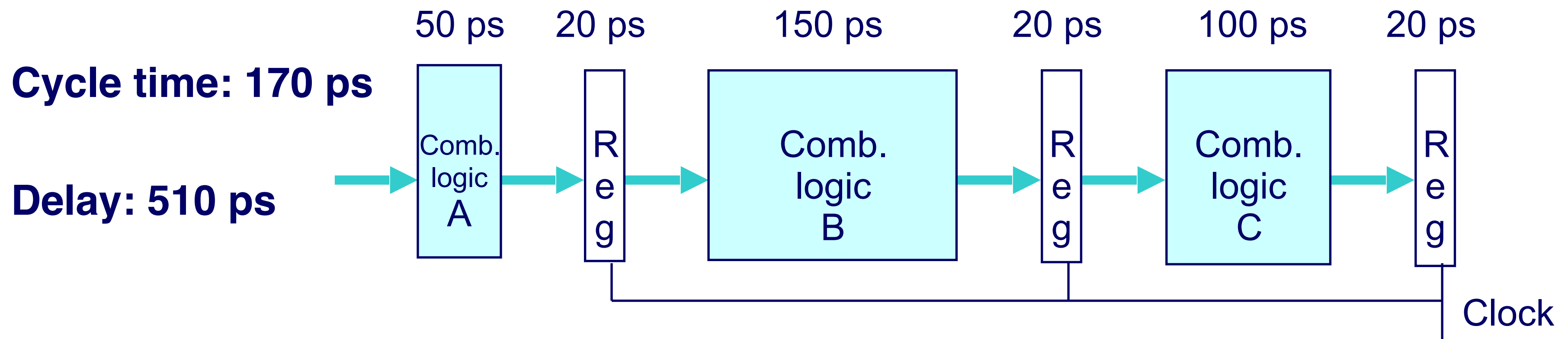




Unbalanced Pipeline

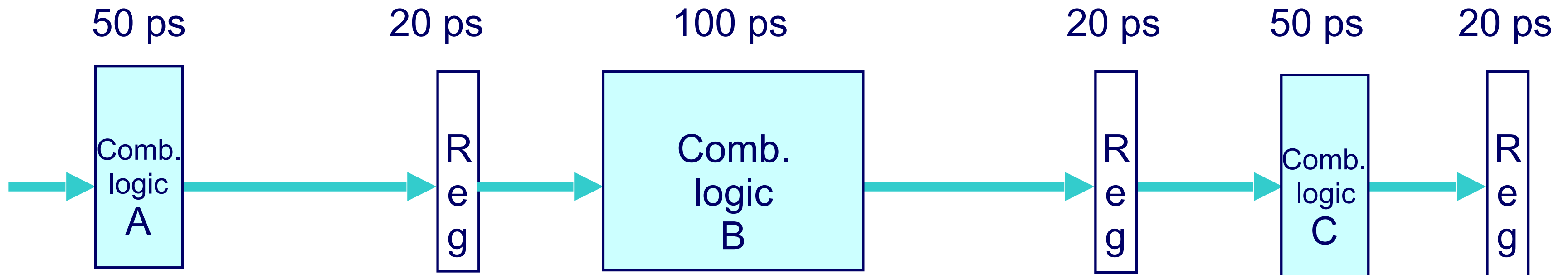
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput





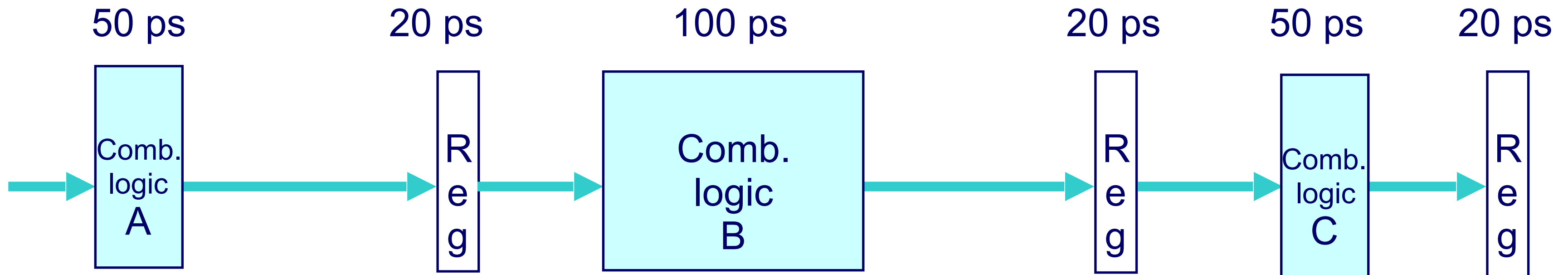
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages



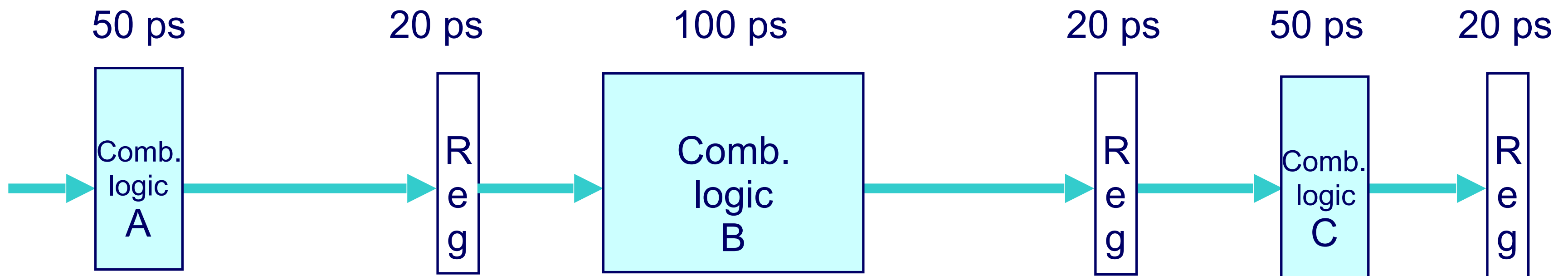
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?



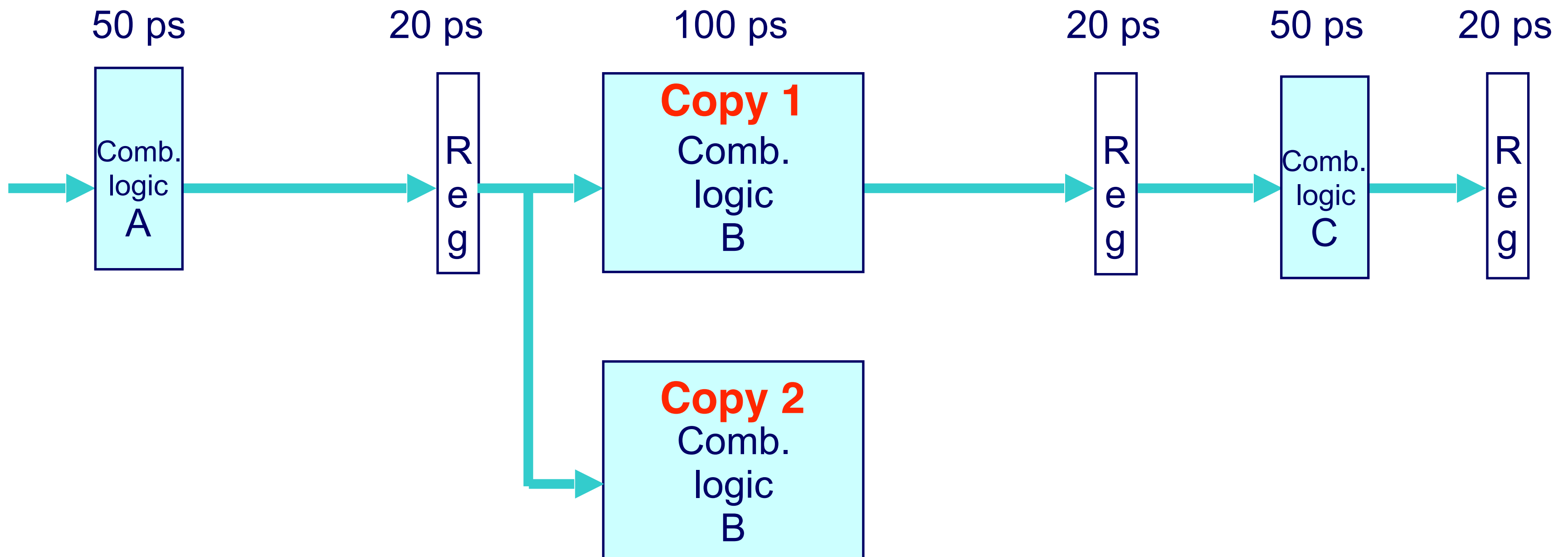
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component



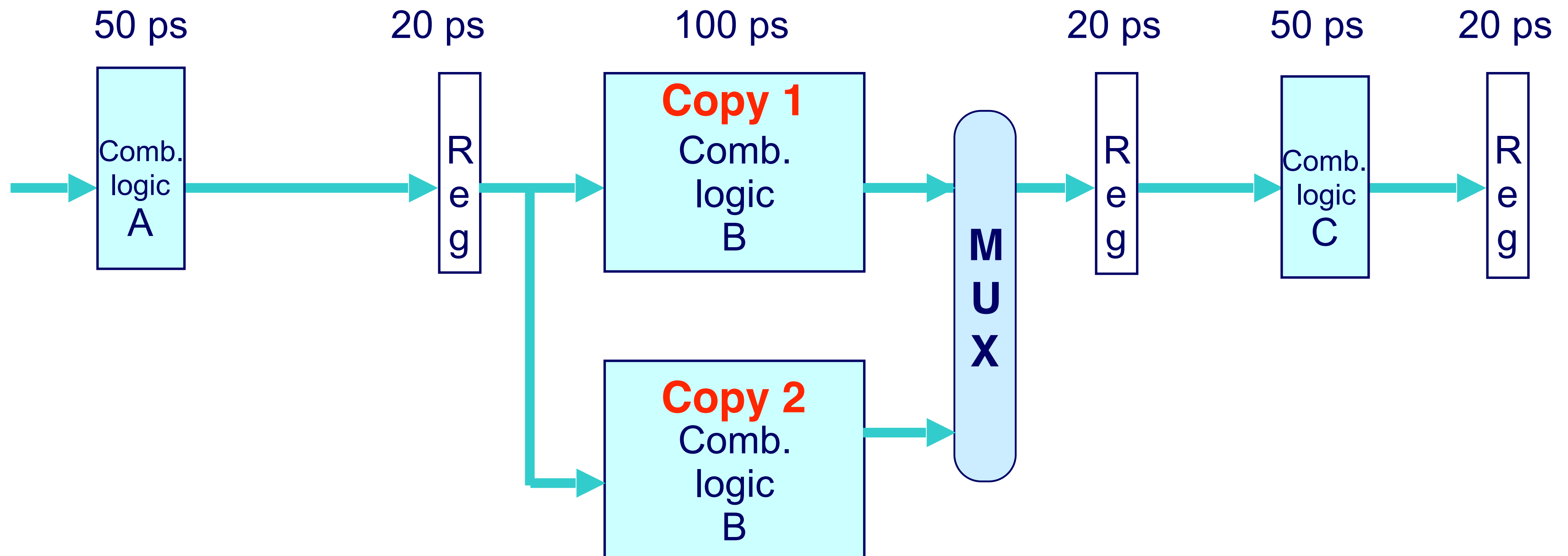
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component



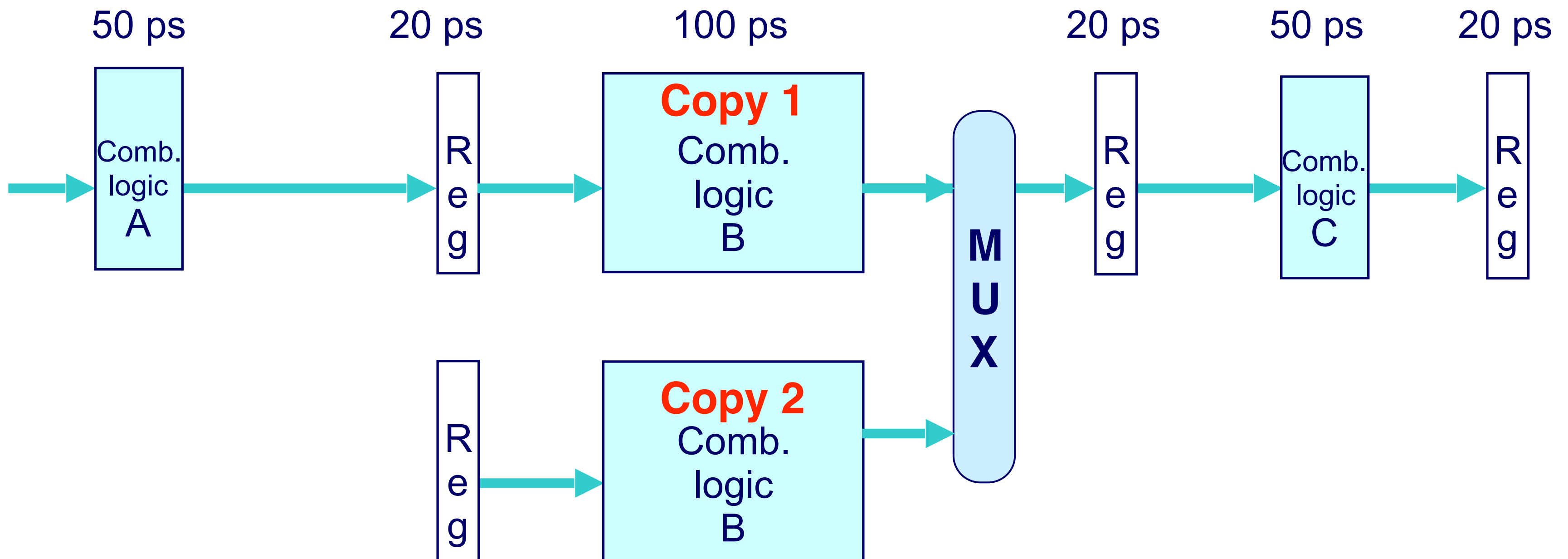
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component



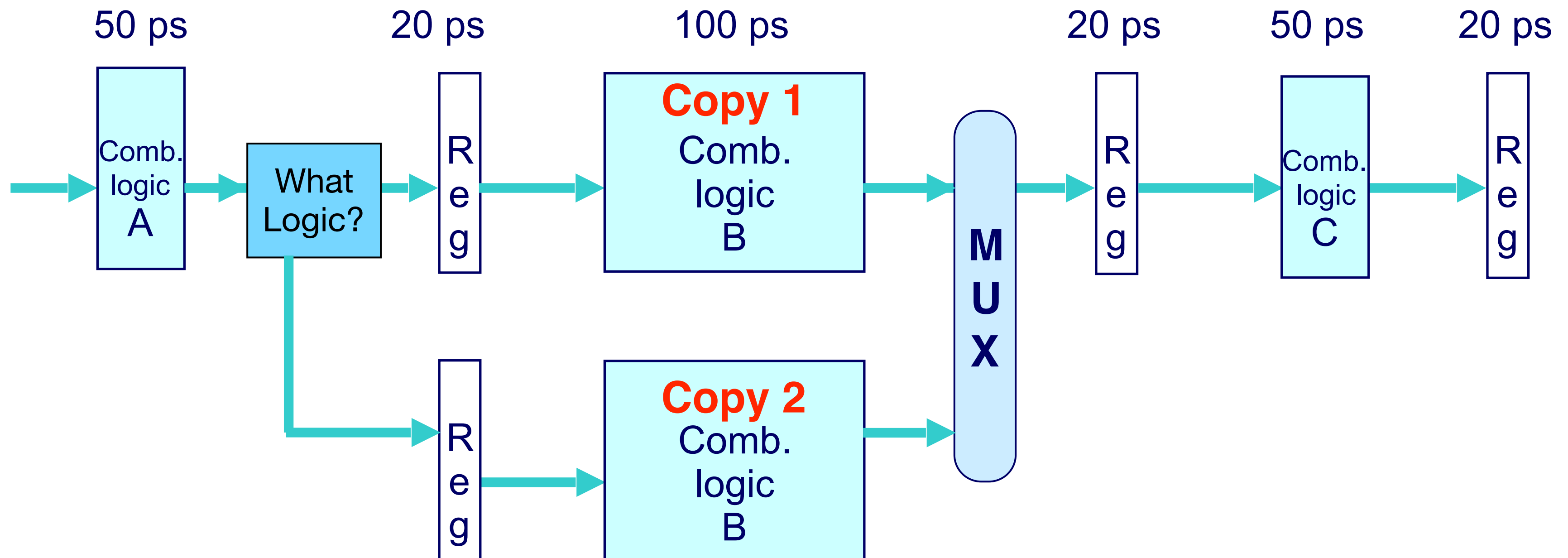
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component



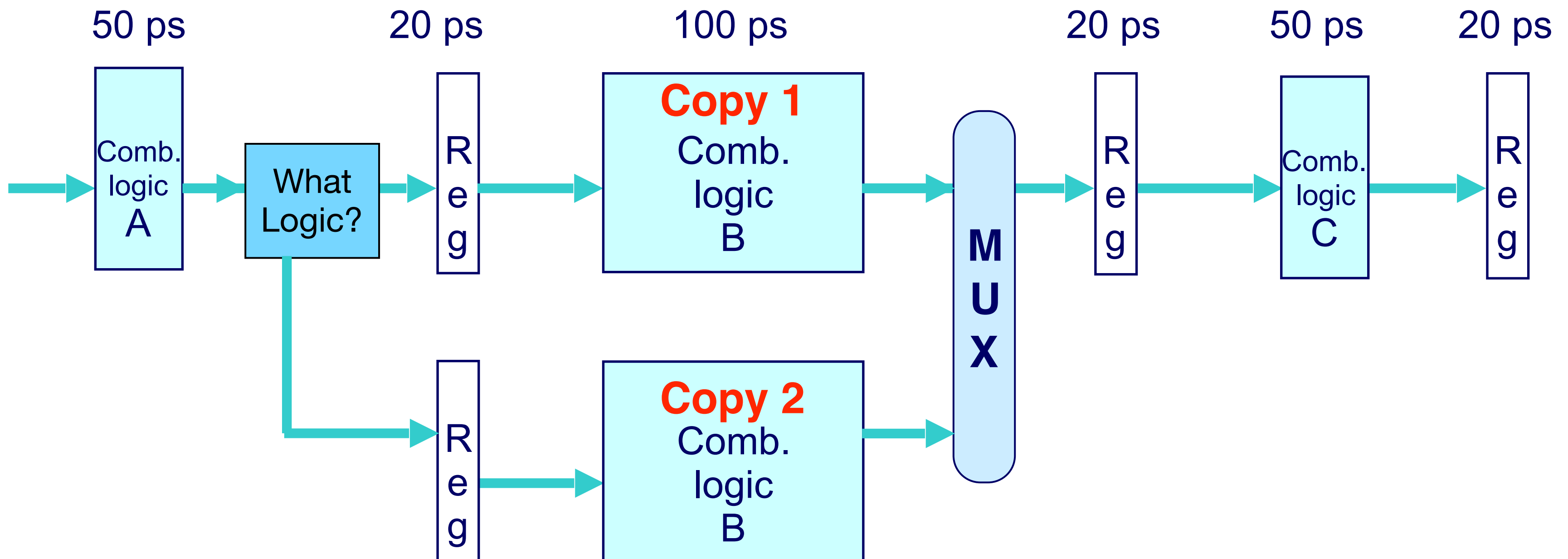
Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component



Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can't further pipeline a stage?
- Solution 2: Use multiple copies of the slow component

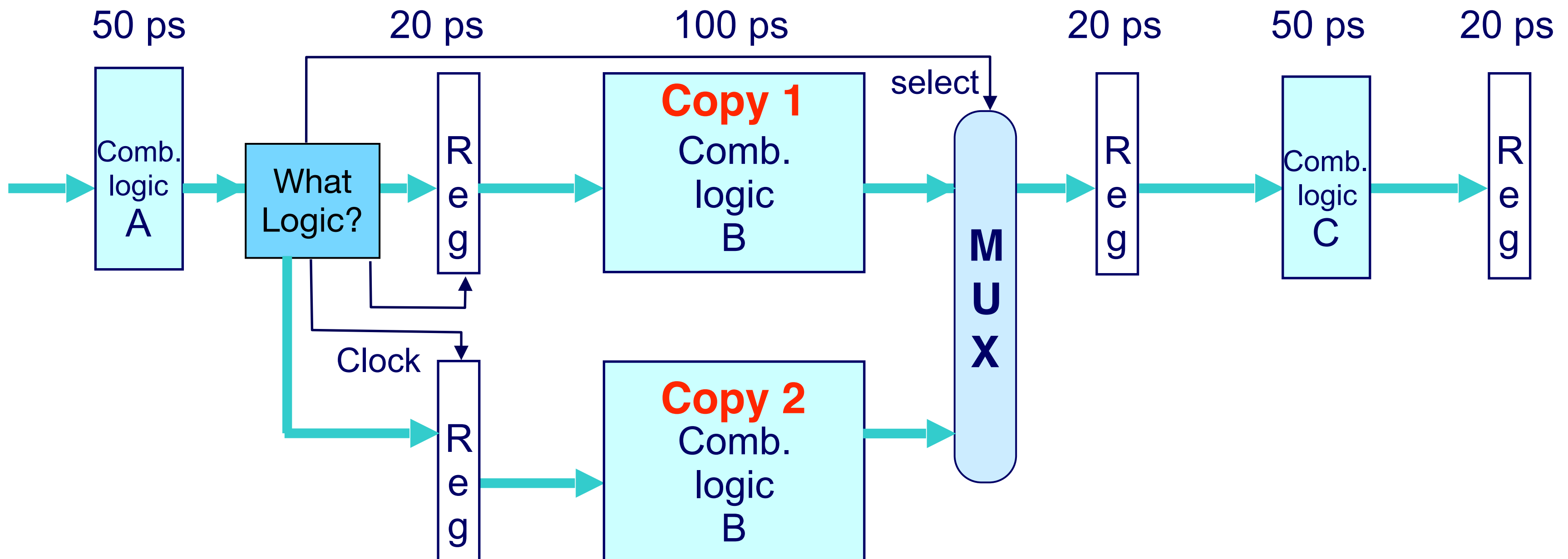


What logic do you need there?

Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.

Mitigating Unbalanced Pipeline

- Solution 1: Further pipeline the slow stages
  - Not always possible. What to do if we can’t further pipeline a stage?
- Solution 2: Use multiple copies of the slow component

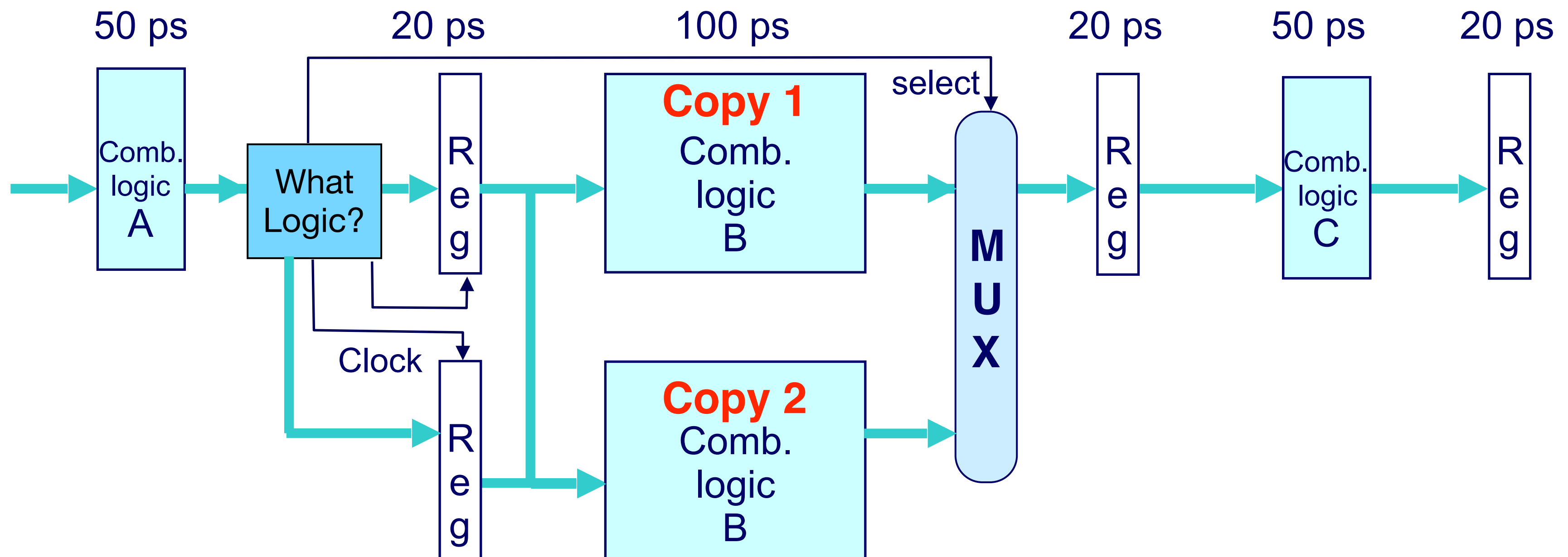


What logic do you need there?

Hint: it needs to control the clock signals of the two registers and the select signal of the MUX.

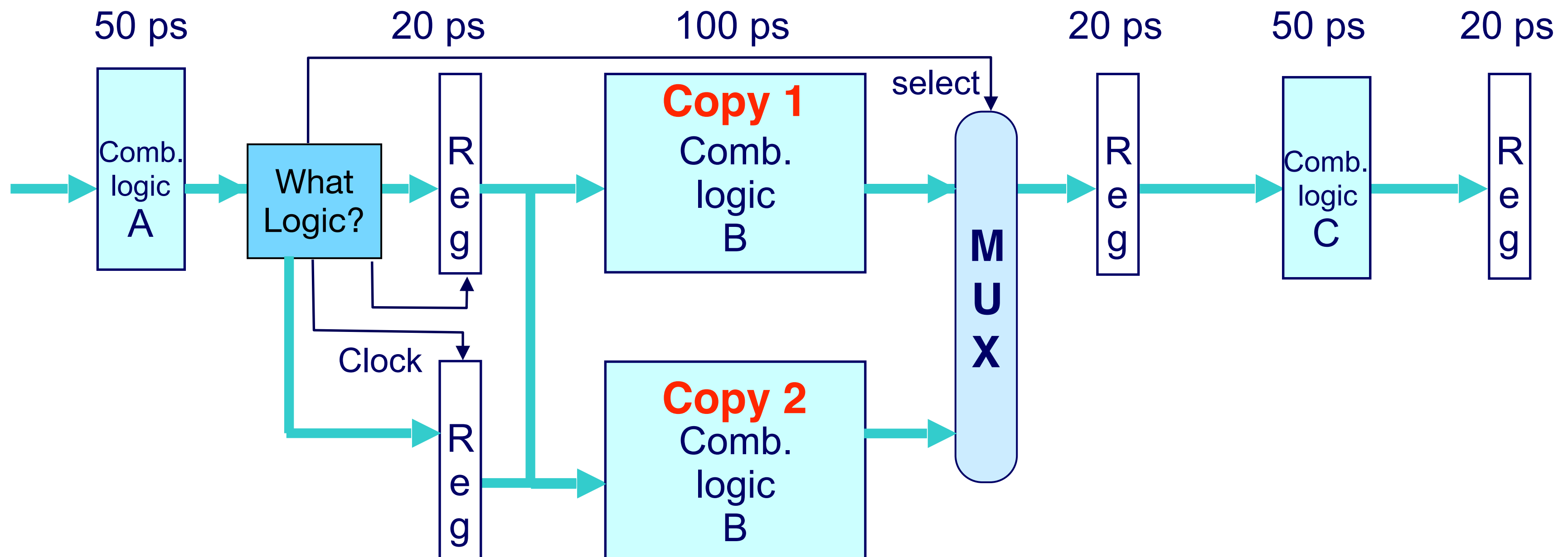
Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.



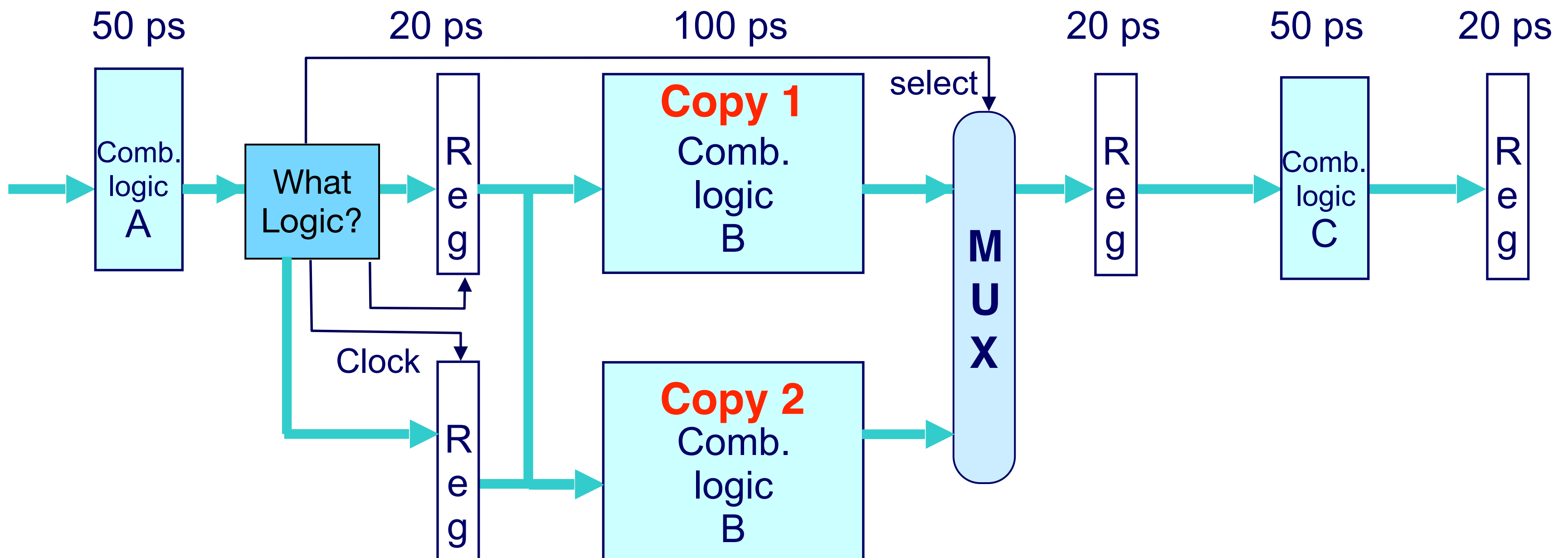
Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
- This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.



Mitigating Unbalanced Pipeline

- Data sent to copy 1 in odd cycles and to copy 2 in even cycles.
- This is called 2-way interleaving. Effectively the same as pipelining Comb. logic B into two sub-stages.
- The cycle time is reduced to 70 ps (as opposed to 120 ps) at the cost of extra hardware.



Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

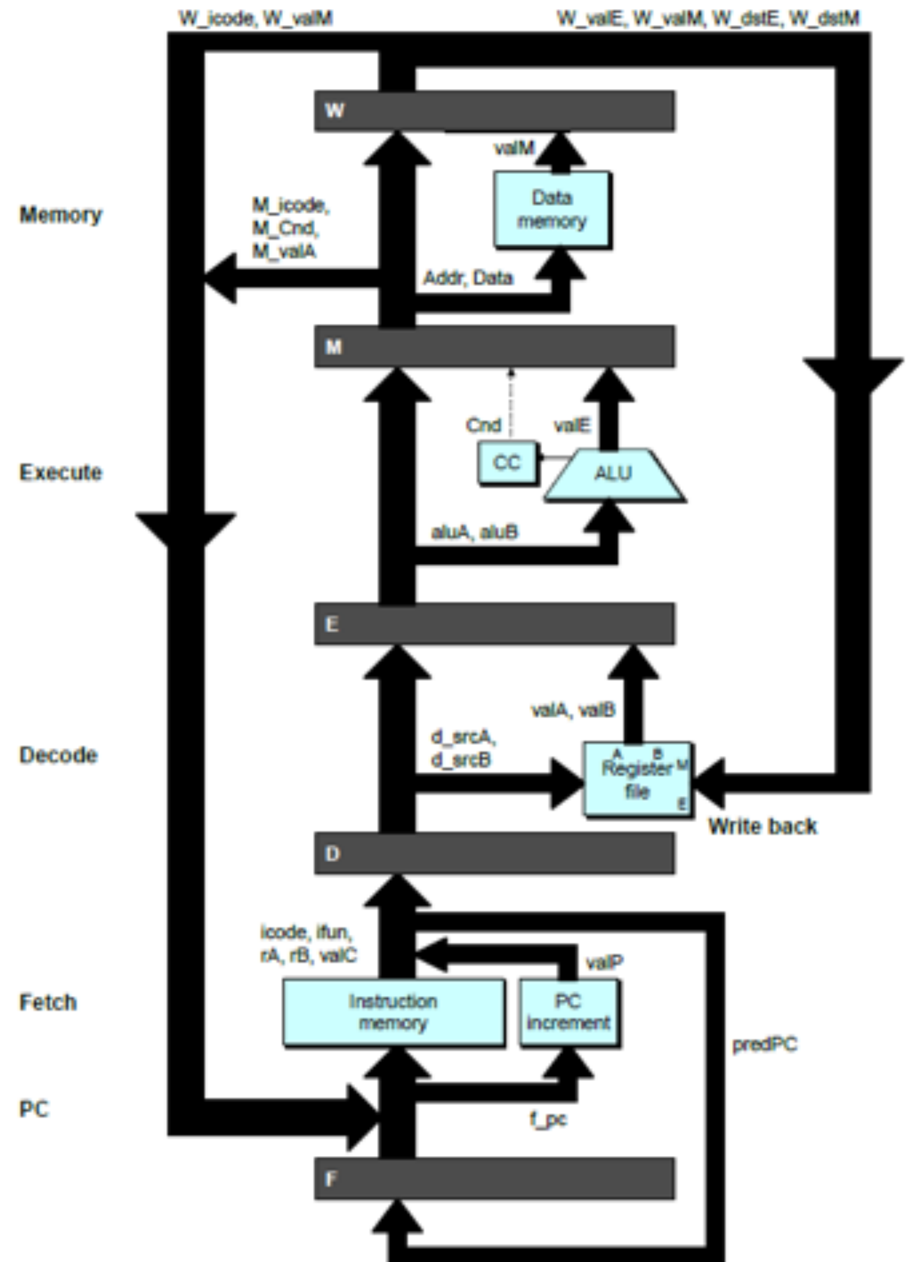
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file



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

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

1
F

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
```

	1	2
xorg %rax, %rax	F	D
jne L1		F

Control Dependency

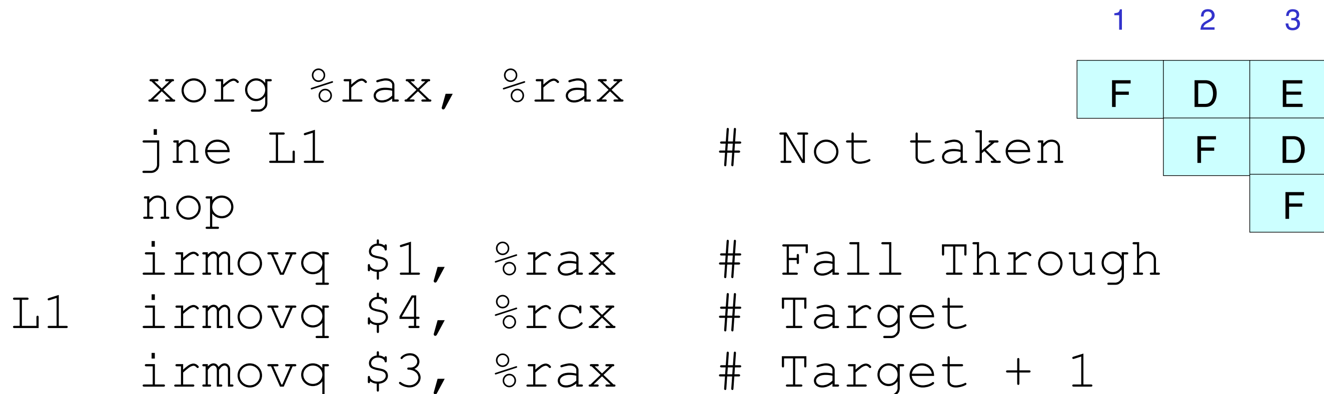
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

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

1	2	3
F	D	E
	F	D

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

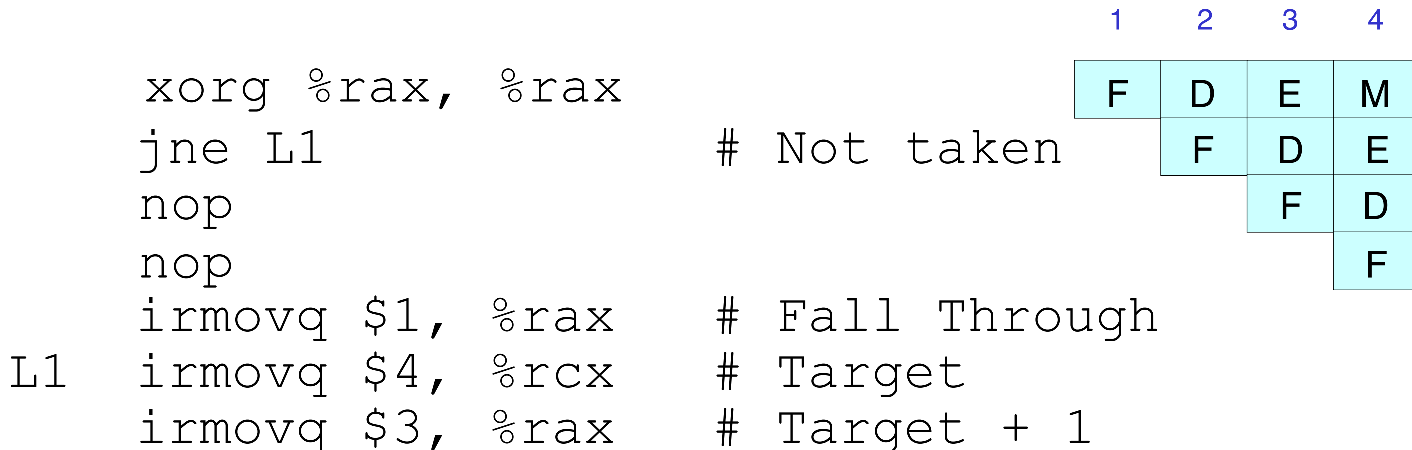
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

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

1	2	3	4
F	D	E	M
	F	D	E
		F	D

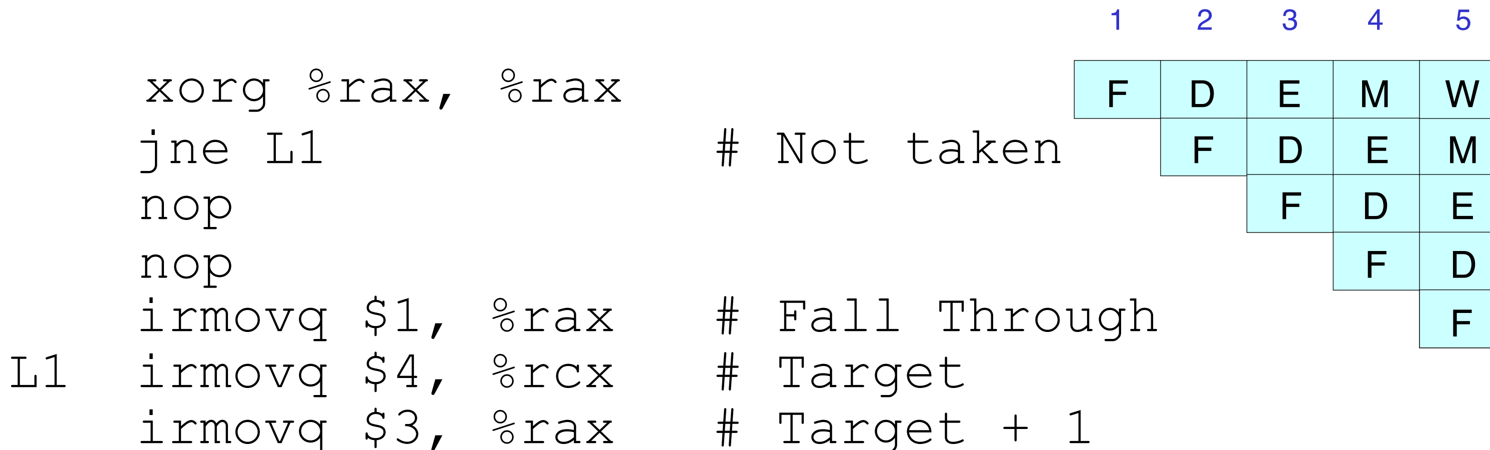
Control Dependency

- **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



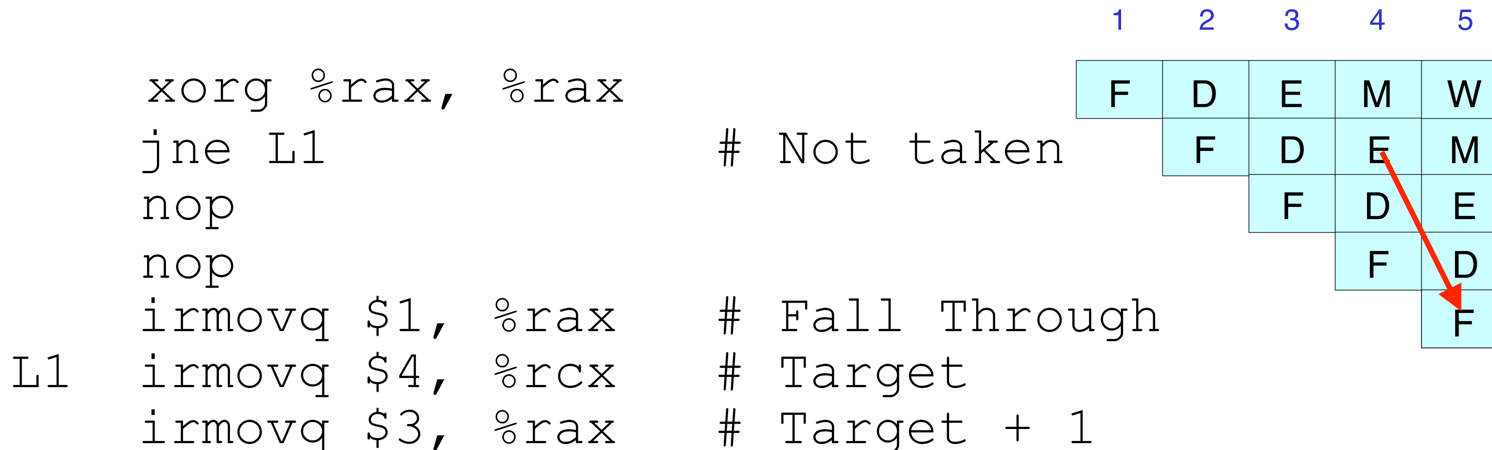
Control Dependency

- **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



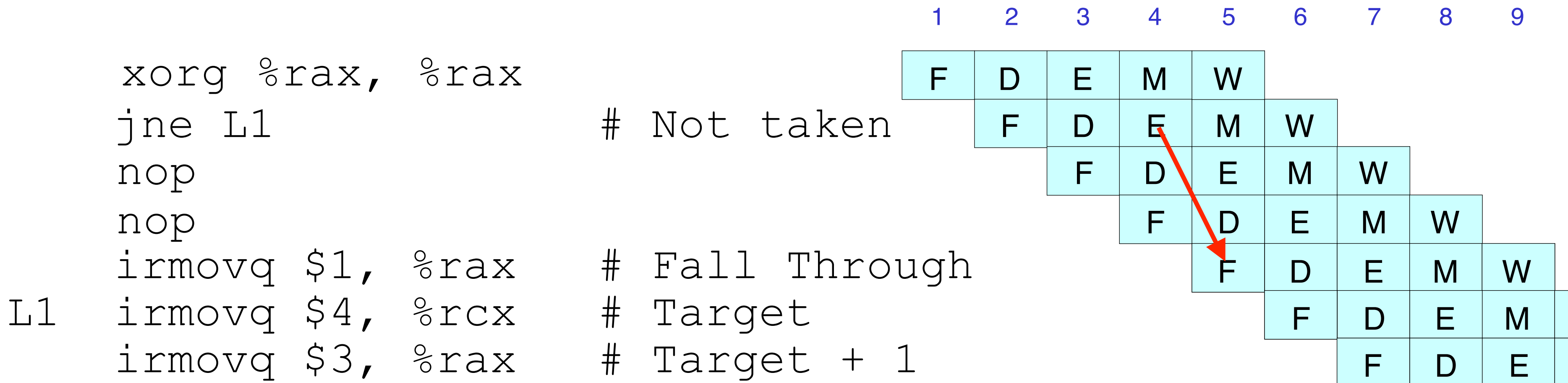
Control Dependency

- **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

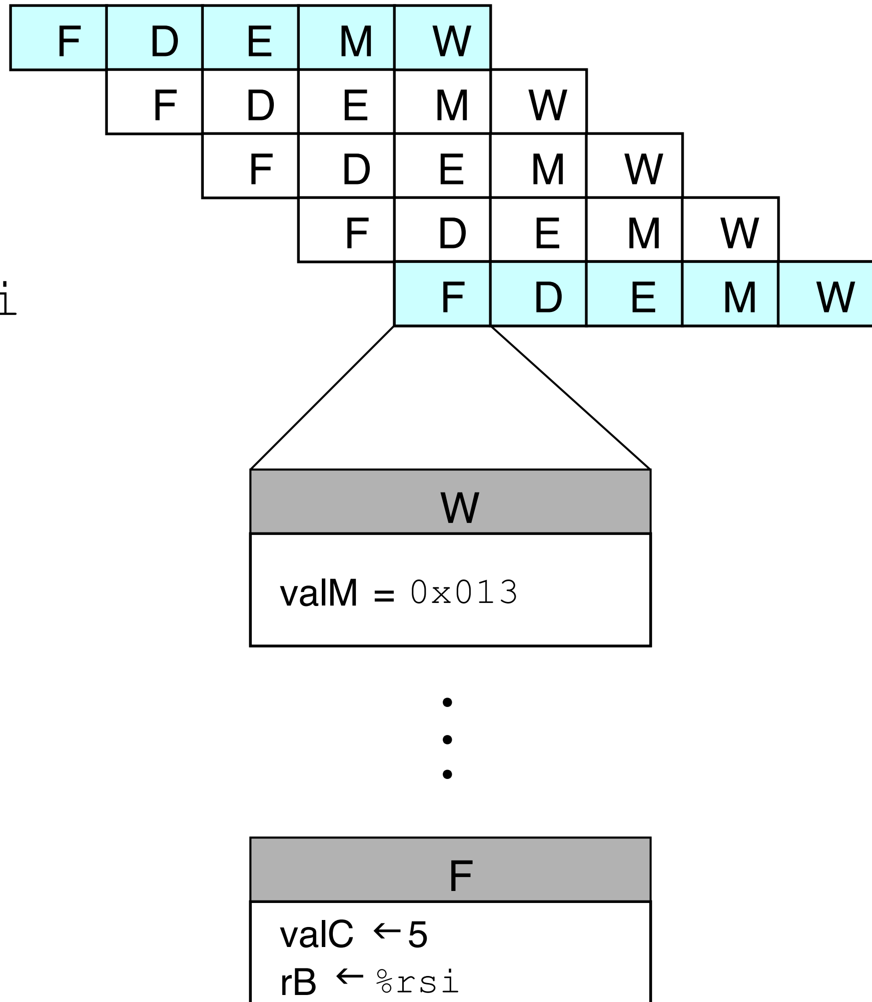


Control Dependency: Return Example

```
0x000:     irmovq Stack,%rsp   # Intialize stack pointer
0x00a:     call p              # Procedure call
0x013:     irmovq $5,%rsi      # Return point
0x01d:     halt
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: Stack:                  # Stack: Stack pointer
```

Control Dependency: Correct Return

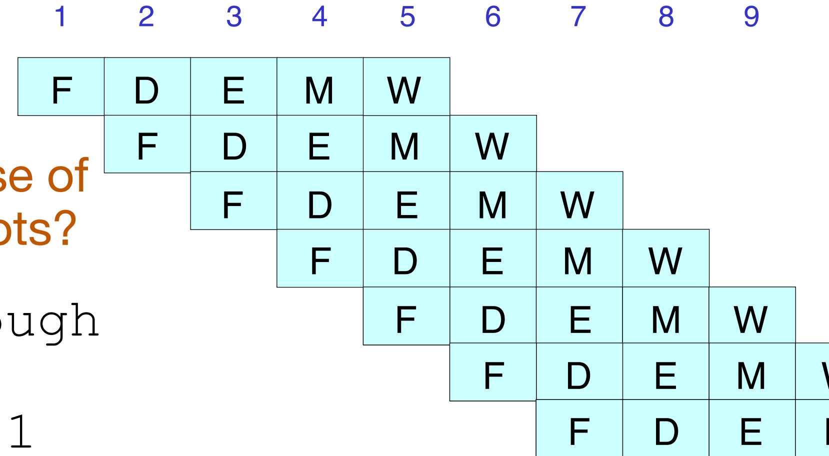
```
0x026:    ret
          nop
          nop
          nop
0x013:    irmovq $5,%rsi
```



Delay Slots

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

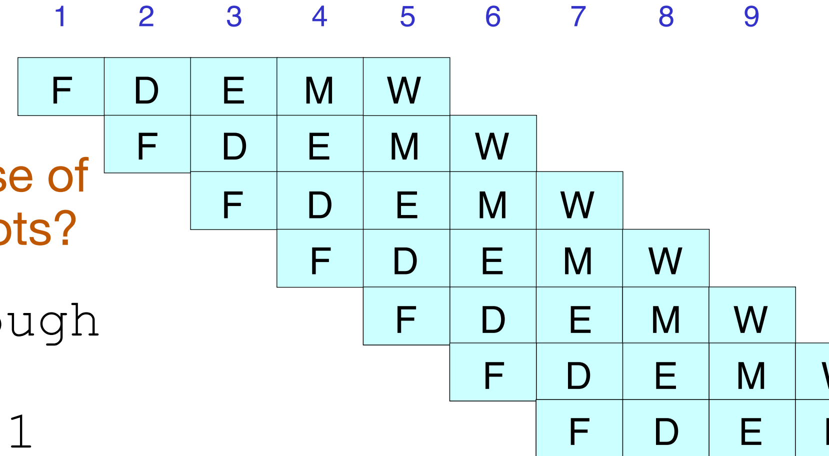
Can we make use of the 2 wasted slots?



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?

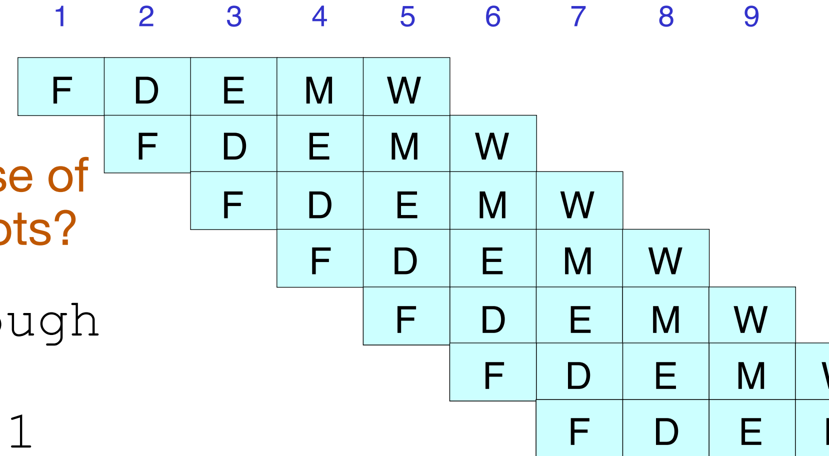



```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```

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?



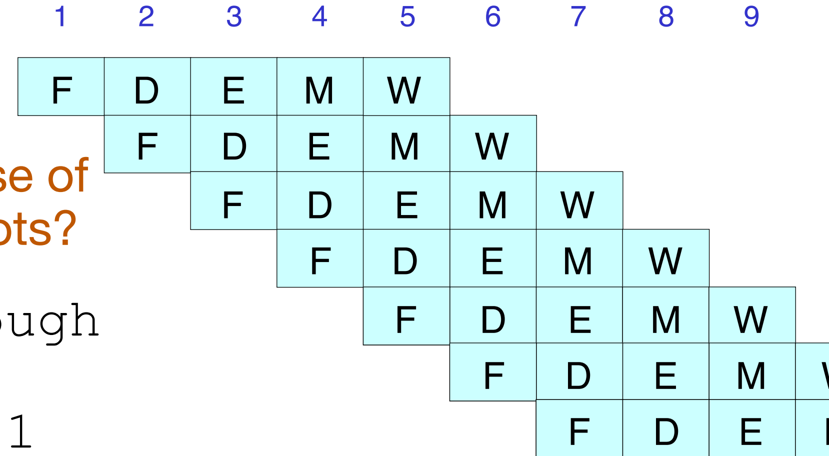

```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```

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



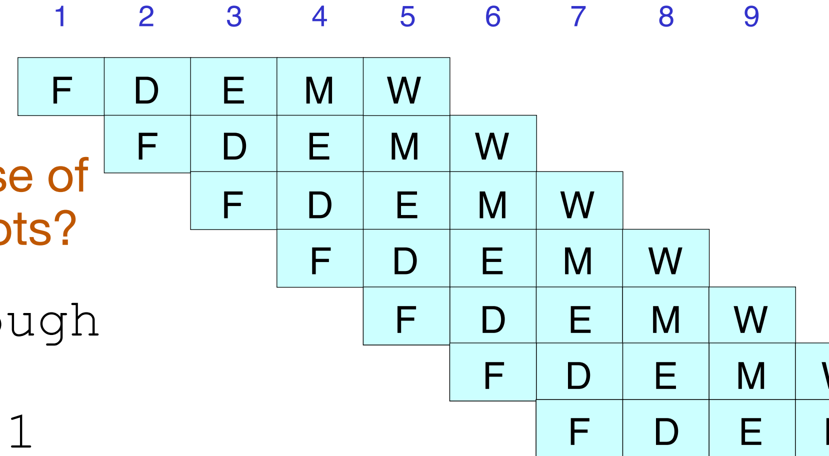
A less obvious example

```
do_C();
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

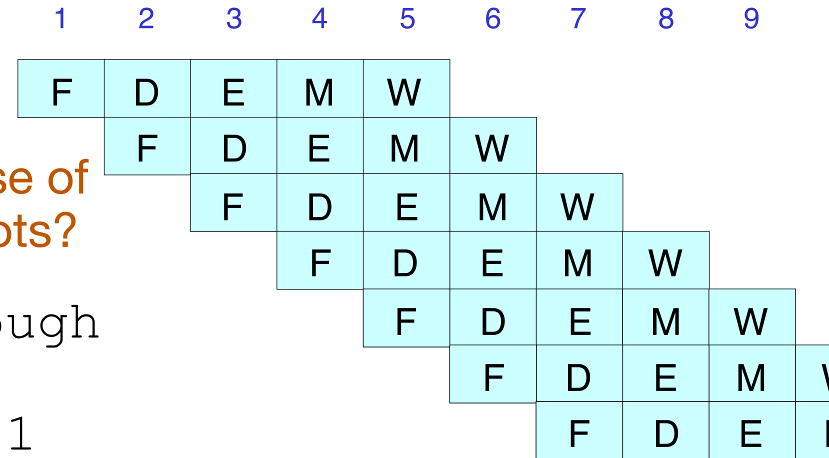
```
do_C();
if (cond) {
   do_A();
} else {
   do_B();
}
```

```
add A, B
or C, D
sub E, F
jle 0x200
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();
}
```

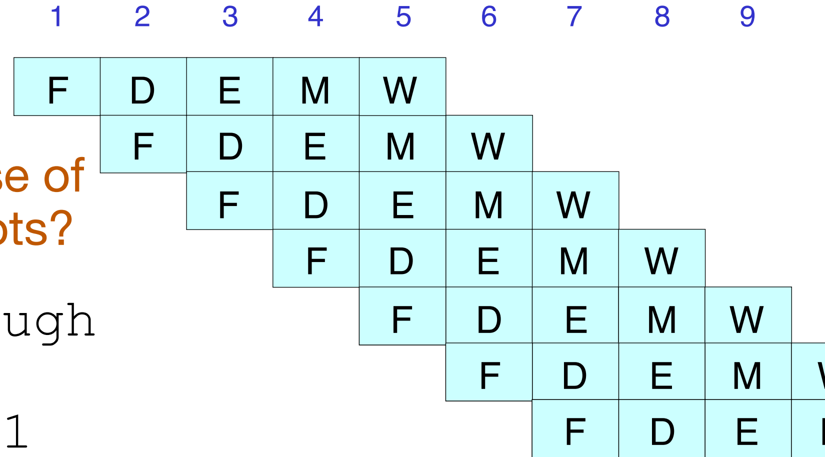
```
add A, B
or C, D
sub E, F
jle 0x200
add A, C
```

```
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?

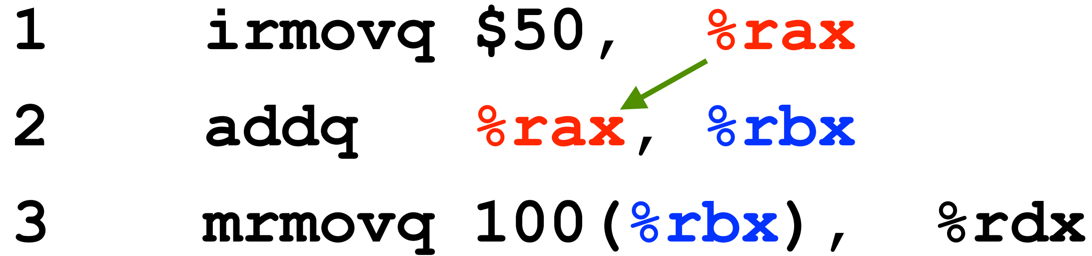
Resolving Control Dependencies

- 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
  - Branch Prediction
  - Return Address Stack
- We will discuss them more later

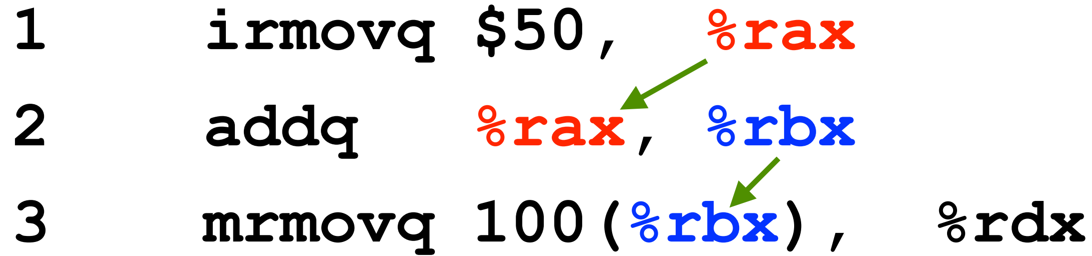
Data Dependencies

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

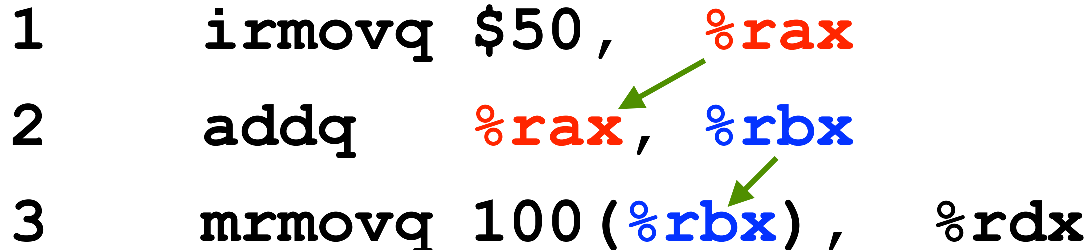
Data Dependencies



Data Dependencies

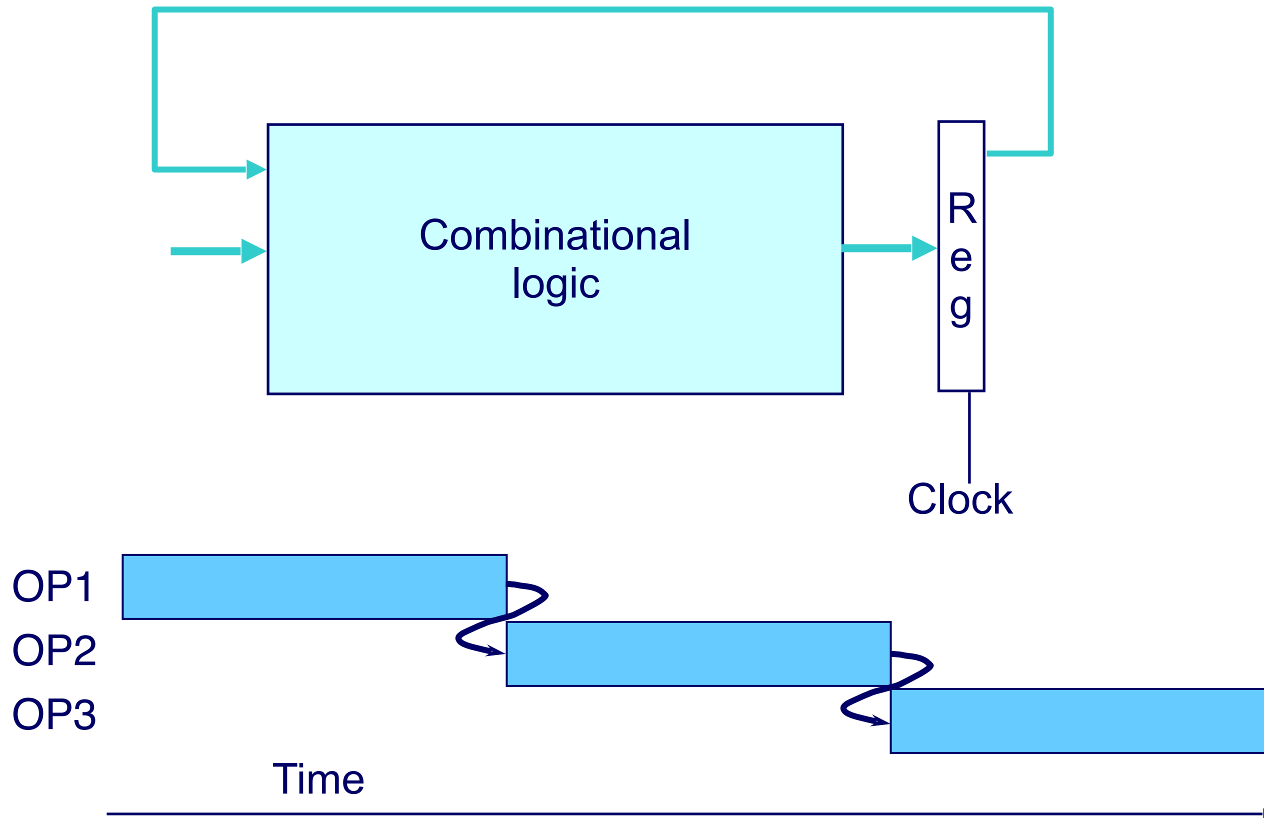


Data Dependencies



- 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

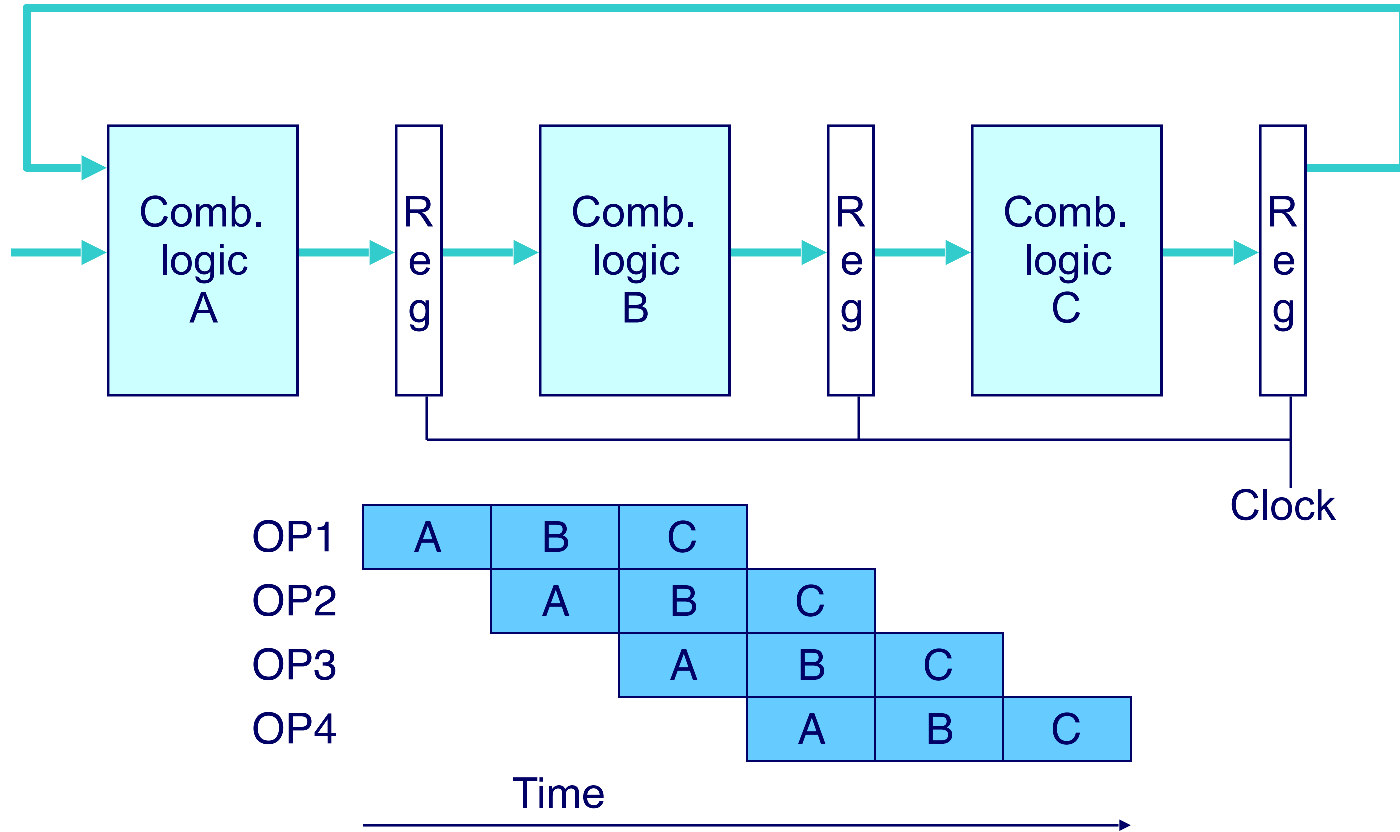
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

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

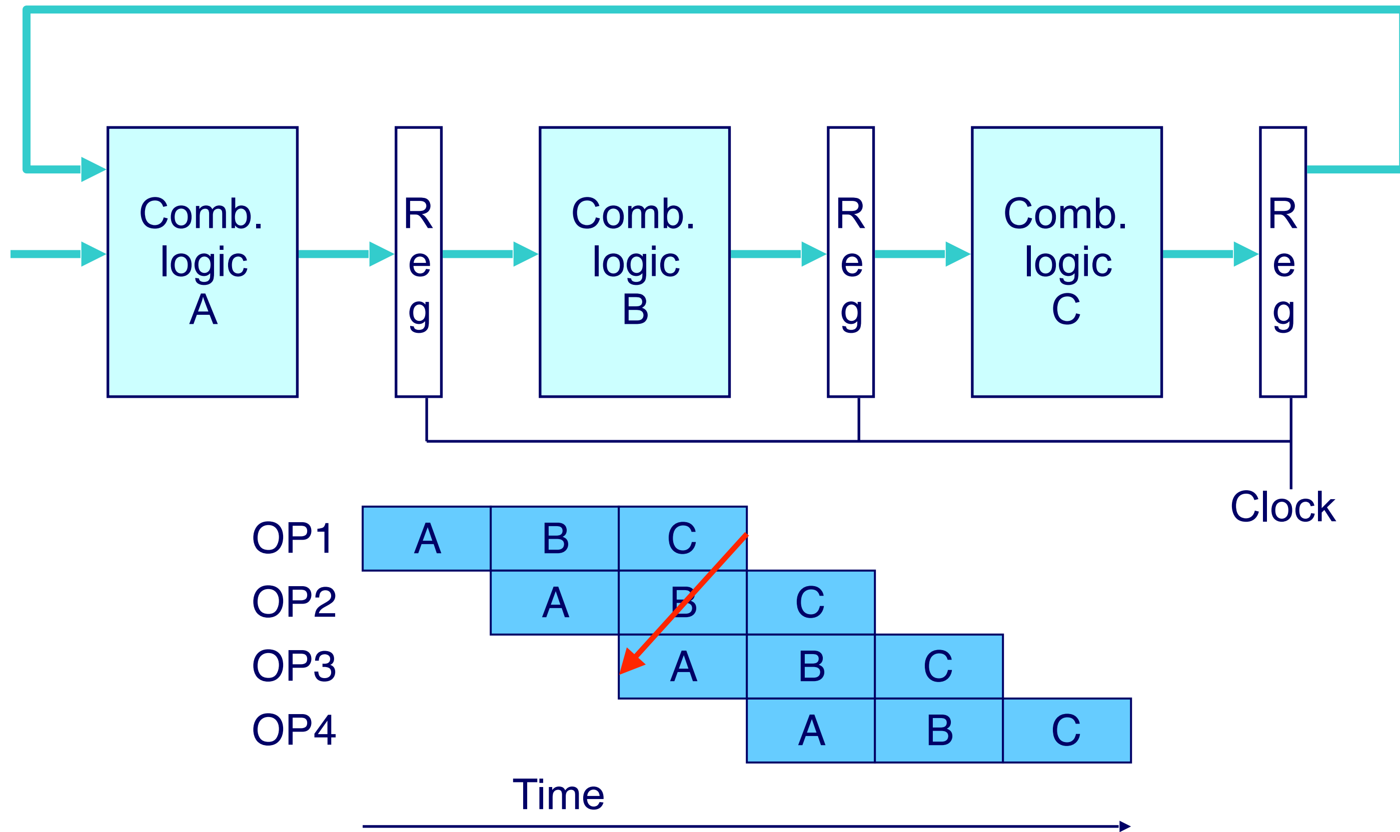
Data Dependencies in Pipeline Machines



Data Hazards happen when:

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

Data Dependencies in Pipeline Machines

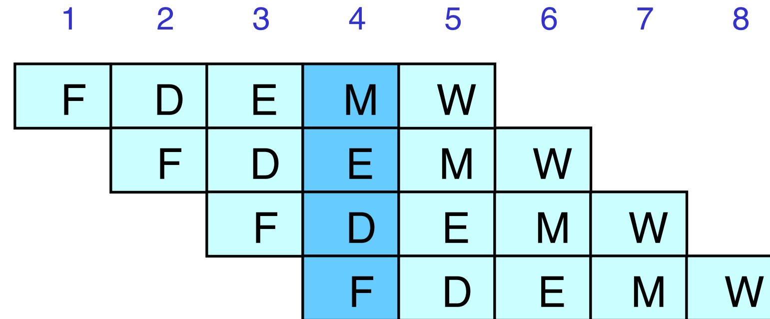


Data Hazards happen when:

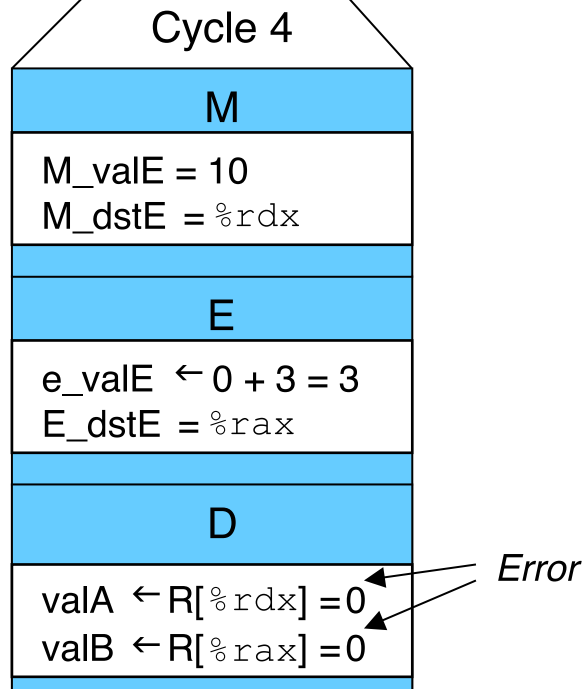
- Result does not feed back around in time for next operation

Data Dependencies: No Nop

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

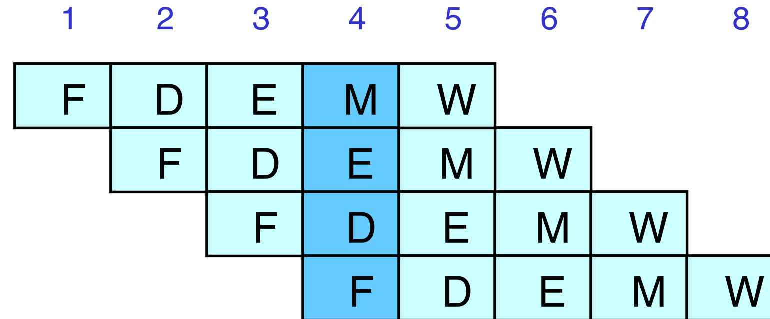


Remember registers get updated in the Write-back stage



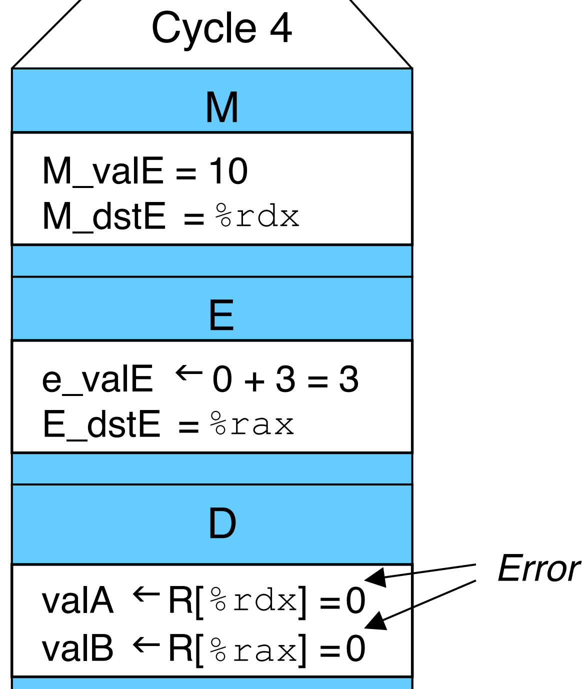
Data Dependencies: No Nop

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



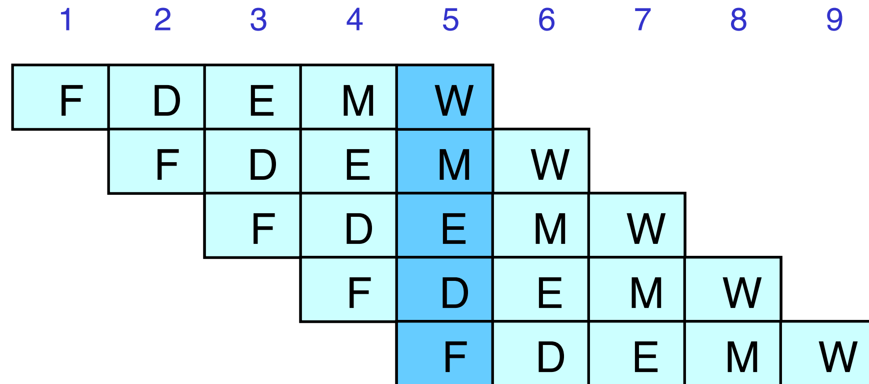
Remember registers get updated in the Write-back stage

addq reads wrong %rdx and %rax

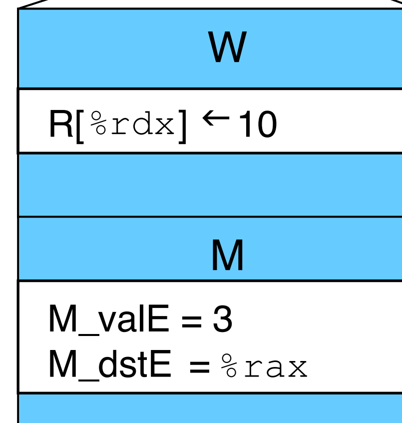


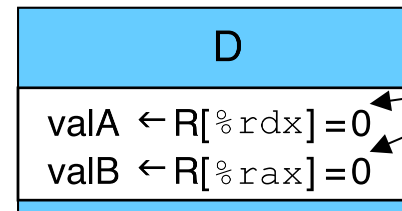
Data Dependencies: 1 Nop

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



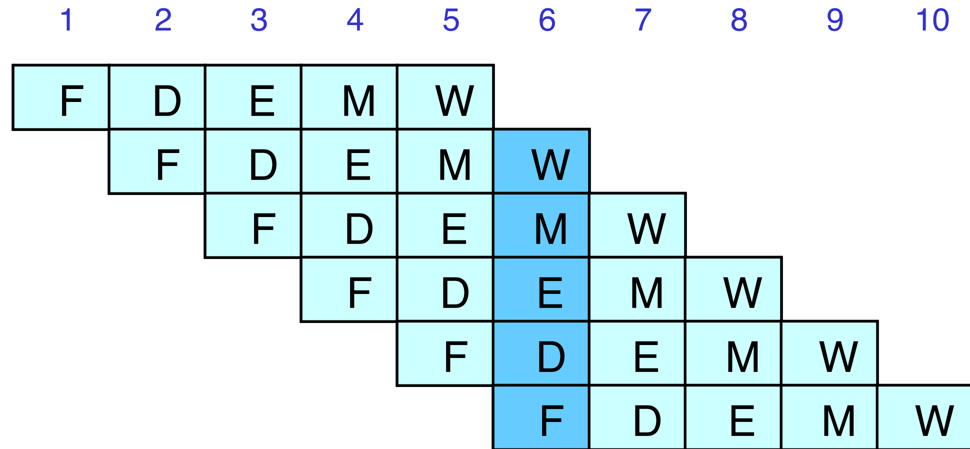
addq still reads wrong %rdx and %rax



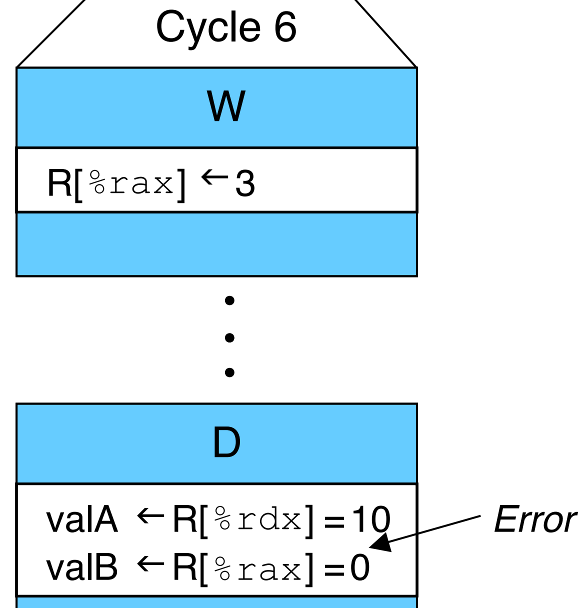


Data Dependencies: 2 Nop's

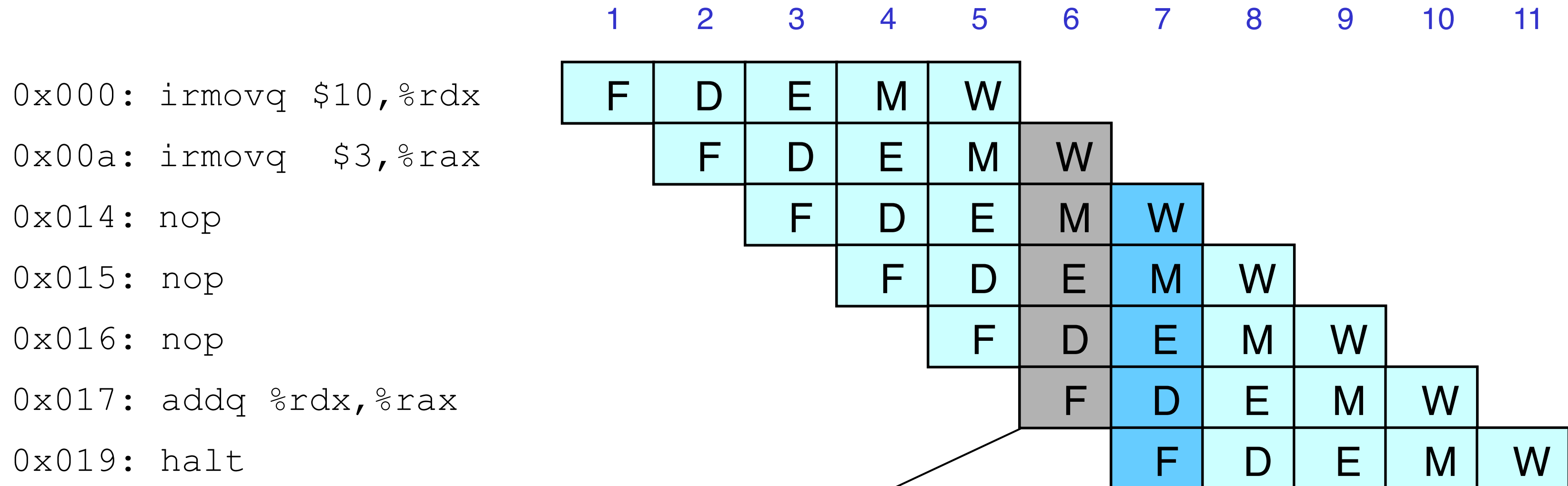
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



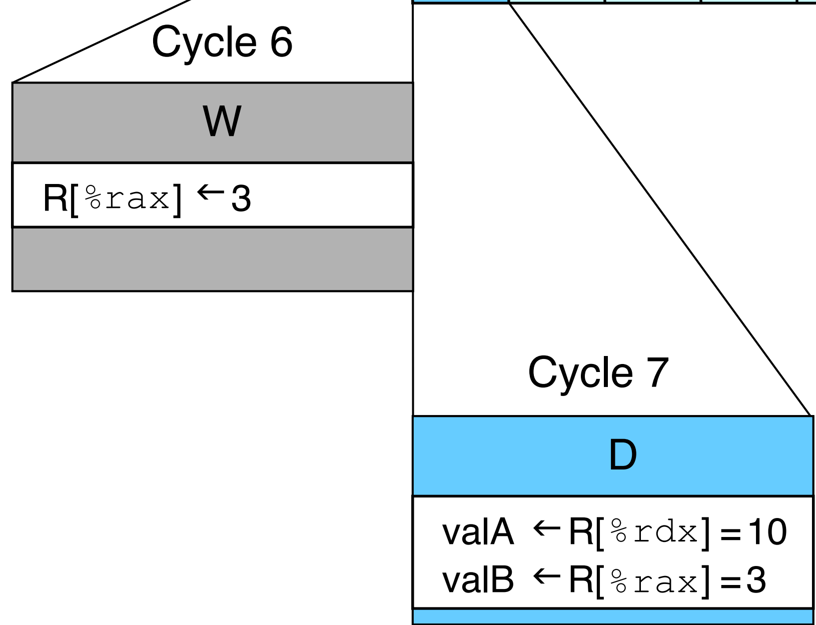
addq reads the correct %rdx, but %rax still wrong



Data Dependencies: 3 Nop's



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
- We will discuss them more later

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

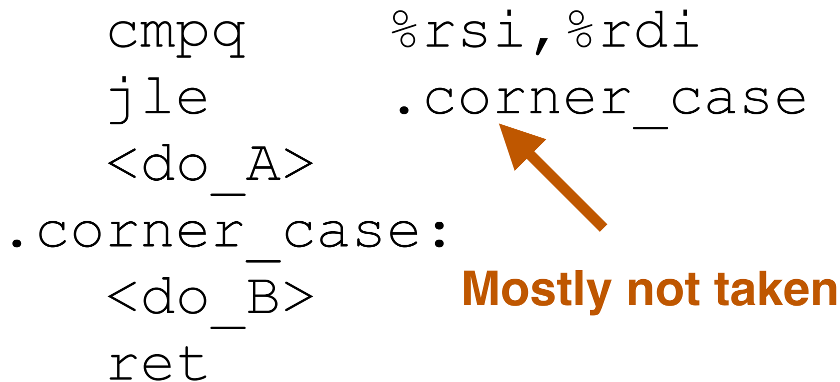
- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

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

Static Prediction

Observation: Two uses of jumps

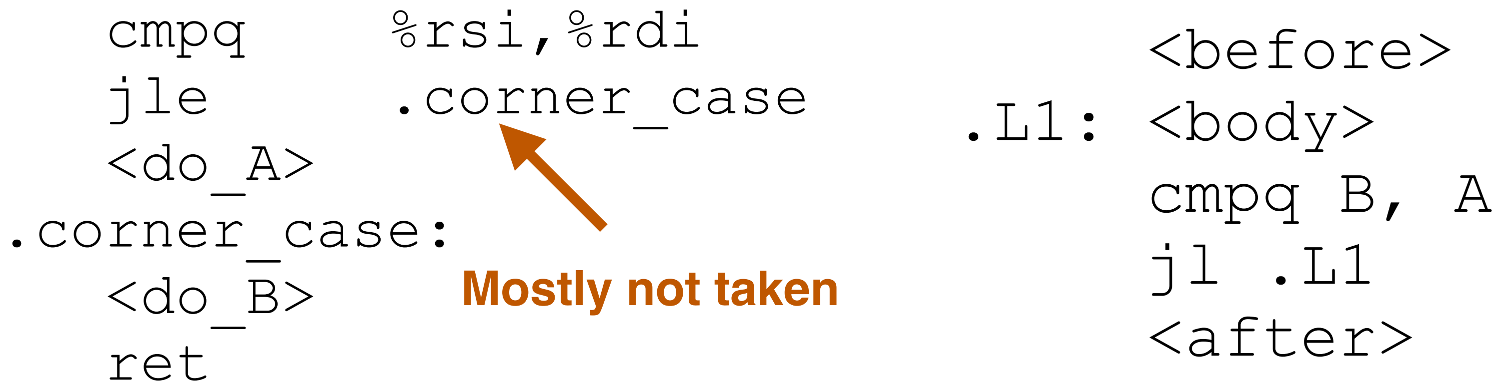
- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.



Static Prediction

Observation: Two uses of jumps

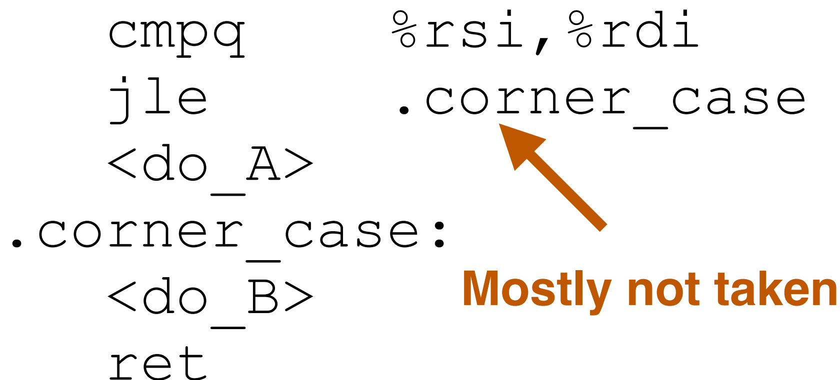
- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

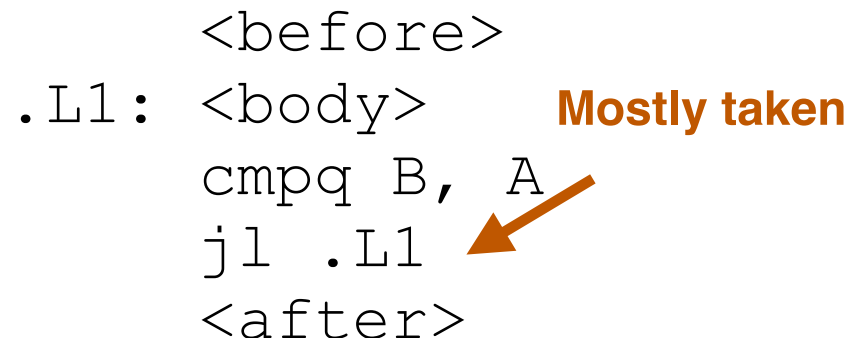


Static Prediction

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.





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

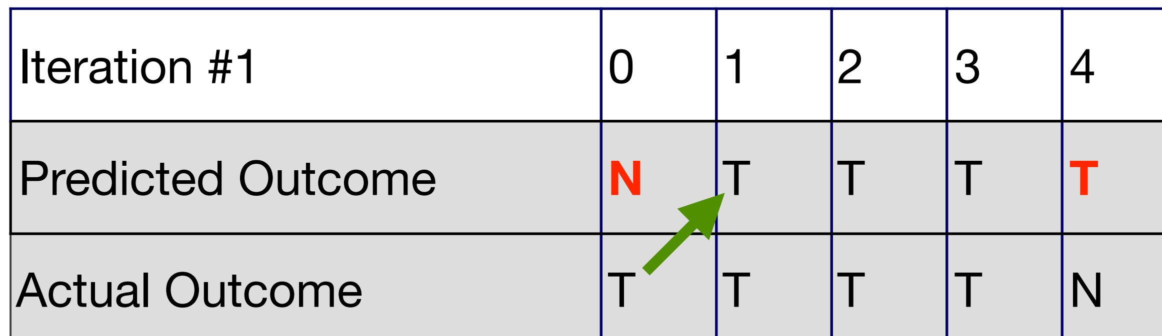
```
for (i=0; i <5; i++) {…}
```

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**