# CSC 252: Computer Organization
# Spring 2019: Lecture 15

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

**Action Items:**

- **Assignment 4 will be out soon.**
- **Mid-terms have not been graded.**

# Announcements

- Lab 3 is due, will be graded soon.
- Lab 4 will be out soon.
- Mid-term will be graded soon.

# Today: Making the Pipeline Really Work

- Control Dependencies
  - Inserting Nops
  - Delay Slots
  - Stalling
  - 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.
- 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 |
|---|---|
| F | D |
|   | F |

# Control Dependency

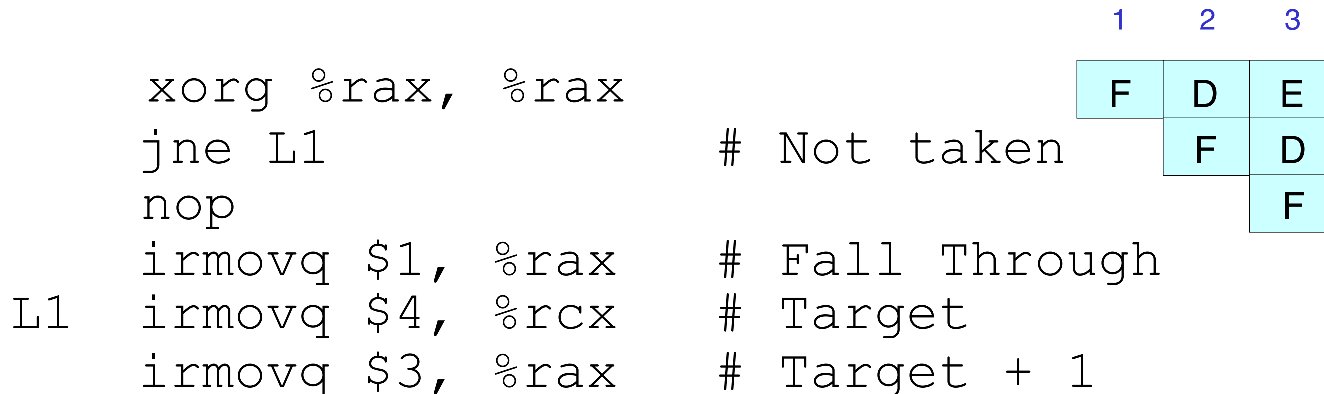
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     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 |
|---|---|---|---|
| xorg %rax, %rax | F | D | E |
| jne L1 | | 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

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

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| F | D | E | M | W |
|  | F | D | E | M |
|  |  | F | D | E |
|  |  |  | F | D |
|  |  |  |  | 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
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| F | D | E | M | W |
| | F | D | E | M |
| | | F | D | E |
| | | | F | D |
| | | | | 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
    nop
    nop
    irmovq $1, %rax      # Fall Through
L1  irmovq $4, %rcx      # Target
    irmovq $3, %rax      # Target + 1
```

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| xorg %rax, %rax | F | D | E | M | W | | | | |
| jne L1 | | F | D | E | M | W | | | |
| nop | | | F | D | E | M | W | | |
| nop | | | | F | D | E | M | W | |
| irmovq $1, %rax | | | | | F | D | E | M | W |
| L1 irmovq $4, %rcx | | | | | | F | D | E | M |
| irmovq $3, %rax | | | | | | | F | D | E |

# 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
  - Branch Prediction
  - Stalling
  - Return Address Stack
- 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

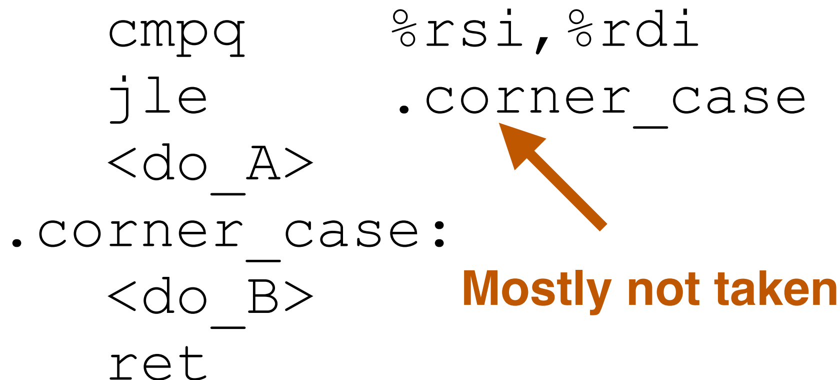
- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

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

# Static Prediction

Observation: Two uses of jumps

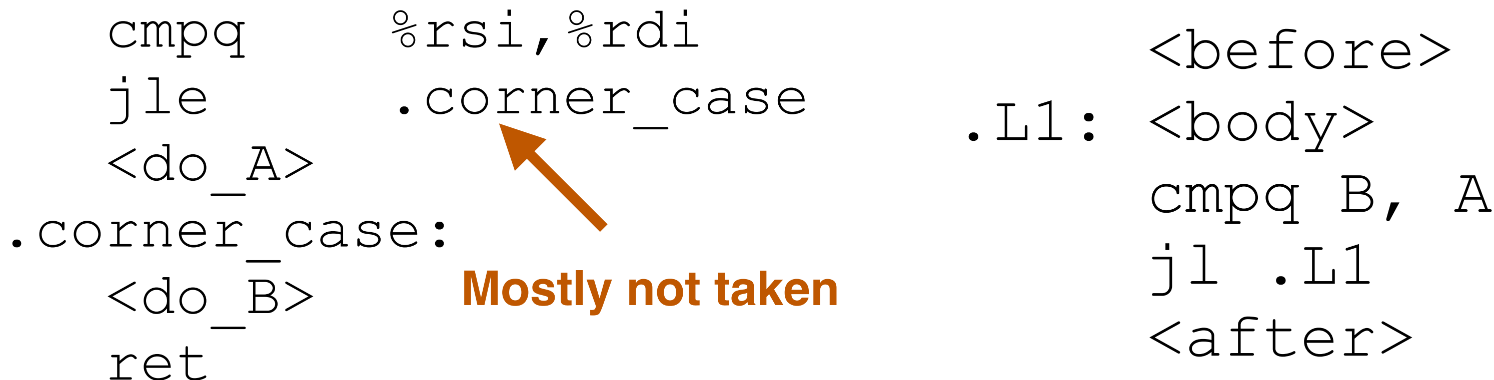
- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.



# Static Prediction

Observation: Two uses of jumps

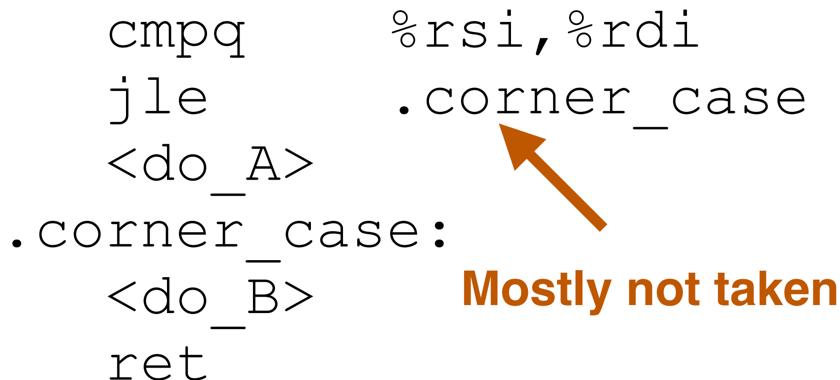
- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

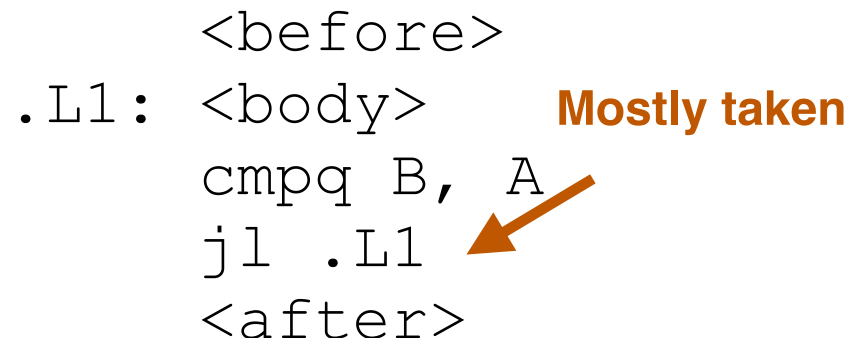


# Static Prediction

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**

# 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

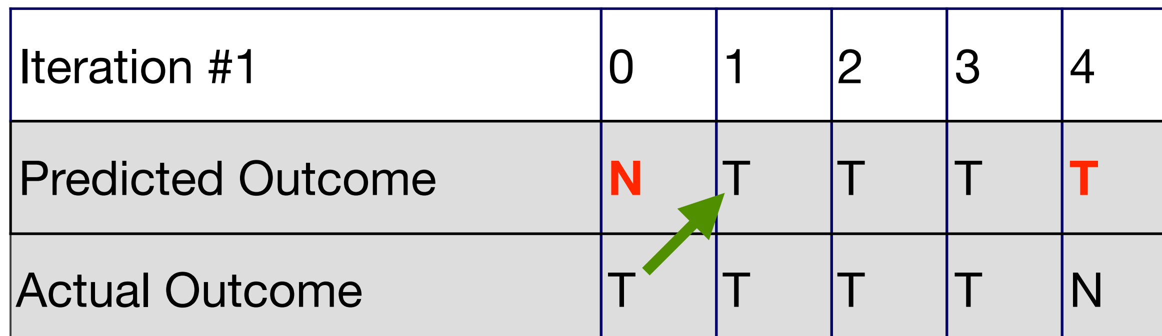
```
for (i=0; i <5; i++) {…}
```

| 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** |

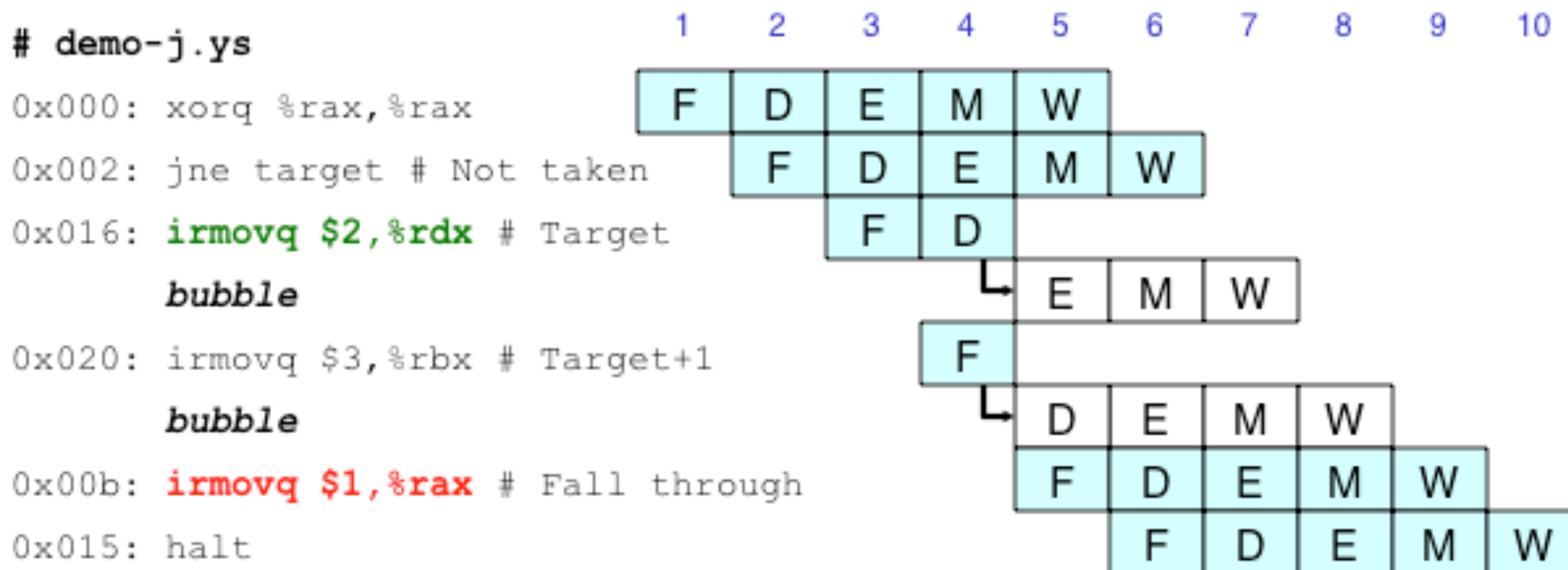
# More Advanced Dynamic Prediction

- 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?




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

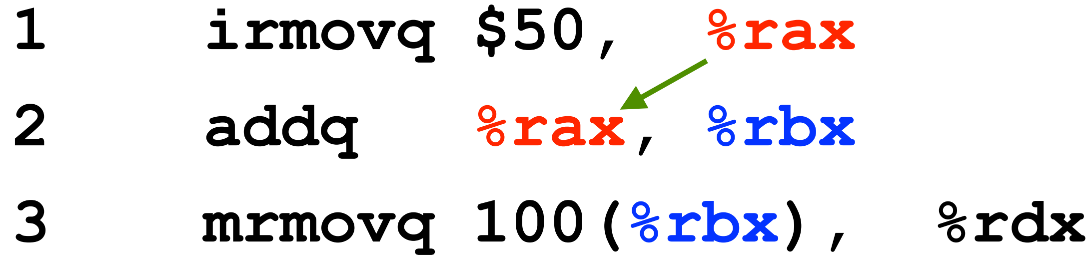
# Today: Making the Pipeline Really Work

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

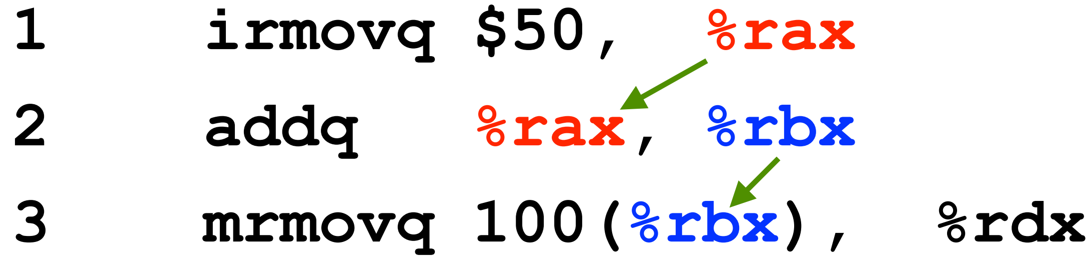
# Data Dependencies

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

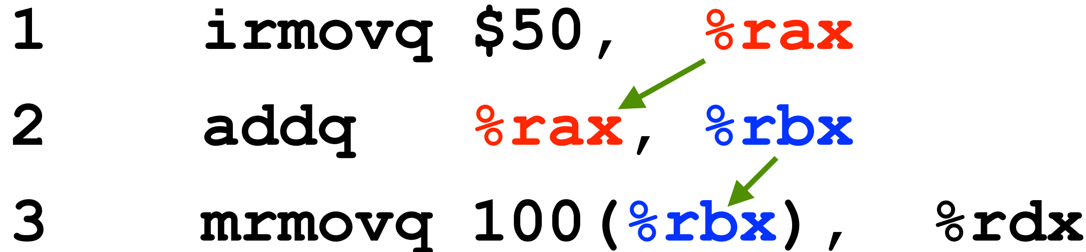
# Data Dependencies



# Data Dependencies



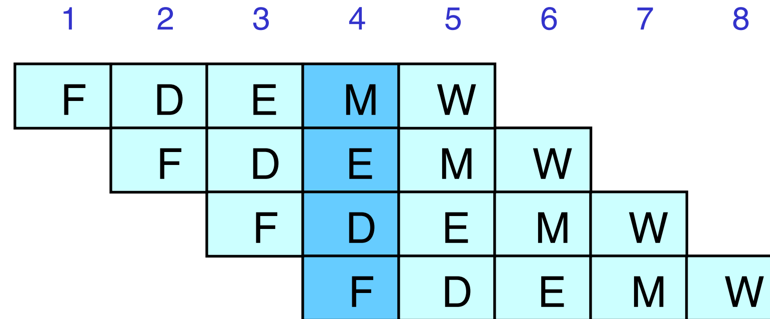
# Data Dependencies



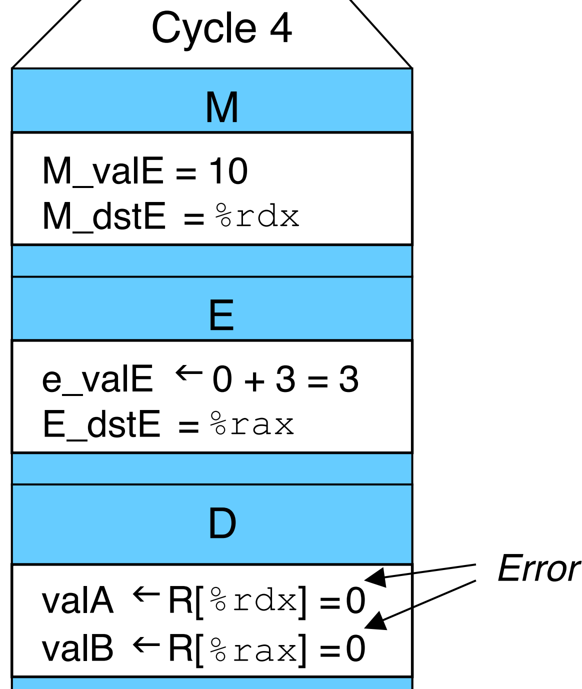

- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

# Data Dependencies: No Nop

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

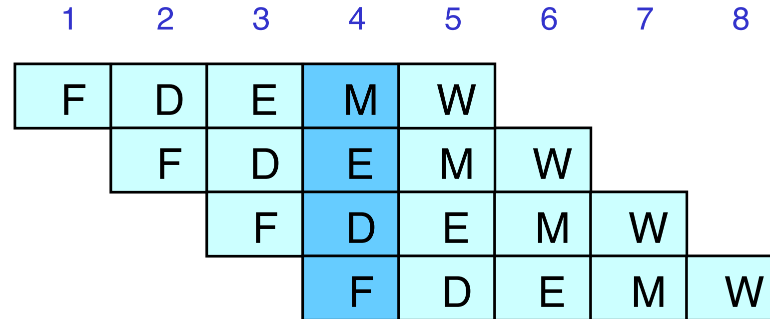



Remember registers get updated in the Write-back stage



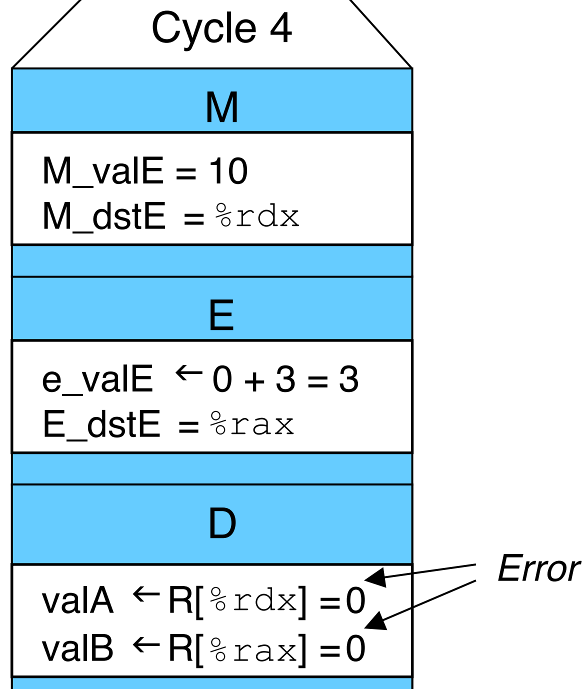
# Data Dependencies: No Nop

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



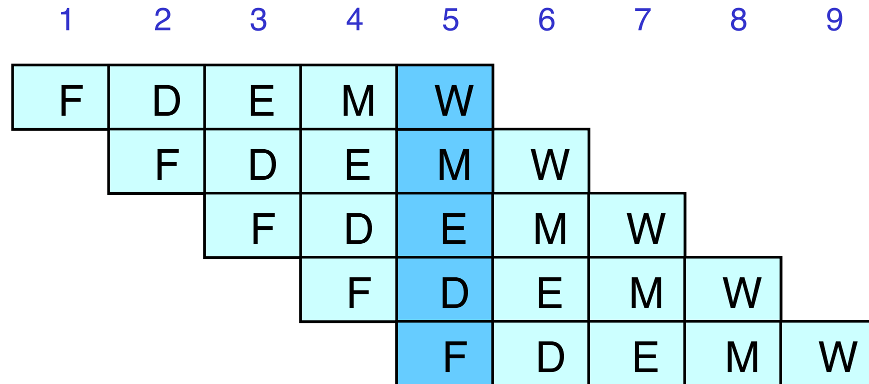

Remember registers get updated in the Write-back stage

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

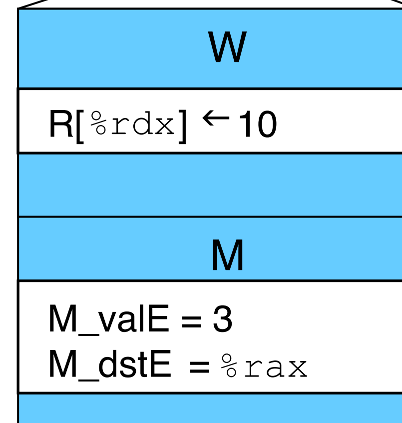


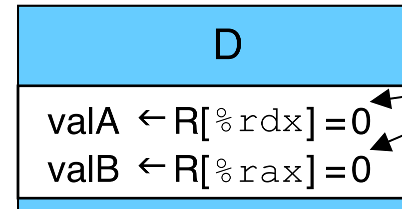
# Data Dependencies: 1 Nop

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



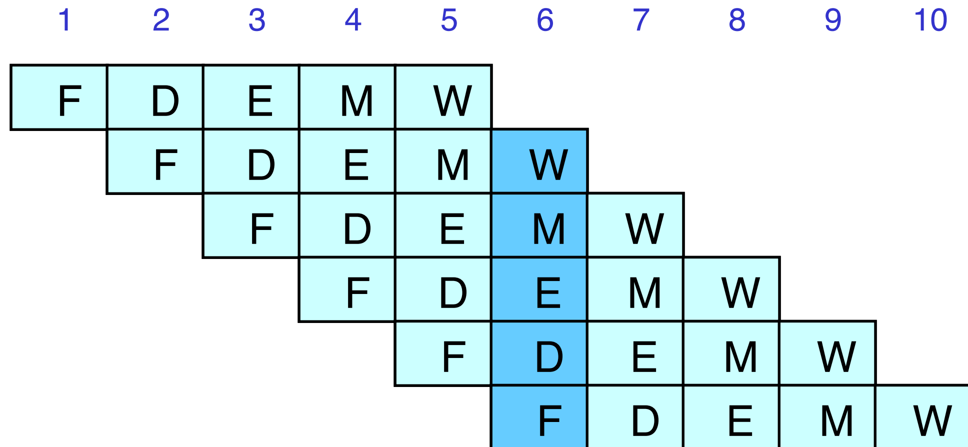

**addq still reads wrong %rdx and %rax**



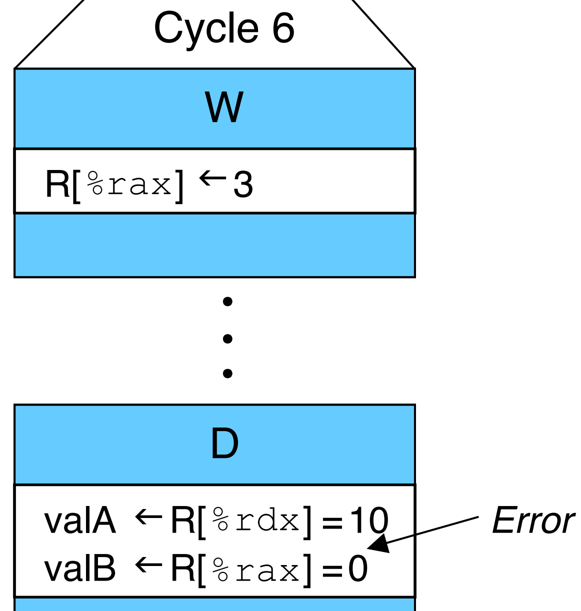



# Data Dependencies: 2 Nop's

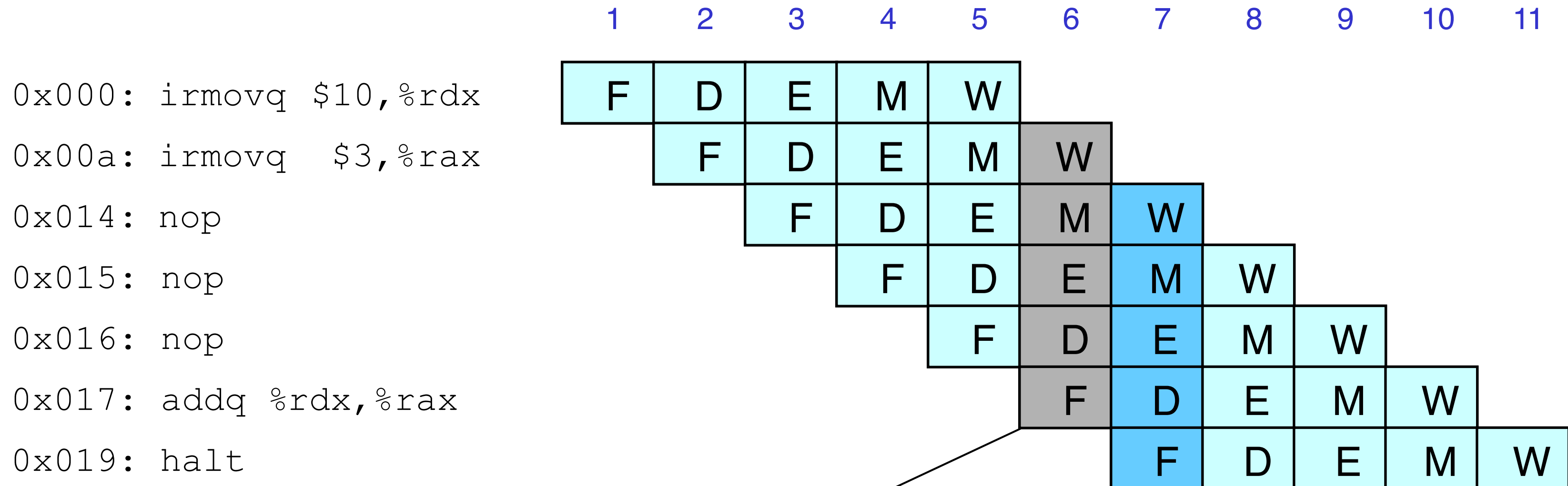
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

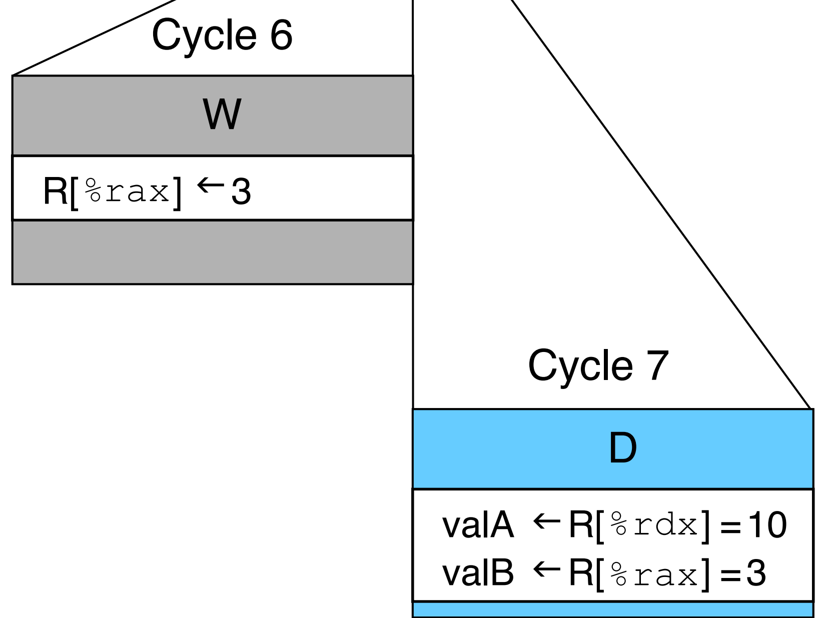



**addq reads the correct %rdx, but %rax still wrong**



# Data Dependencies: 3 Nop's



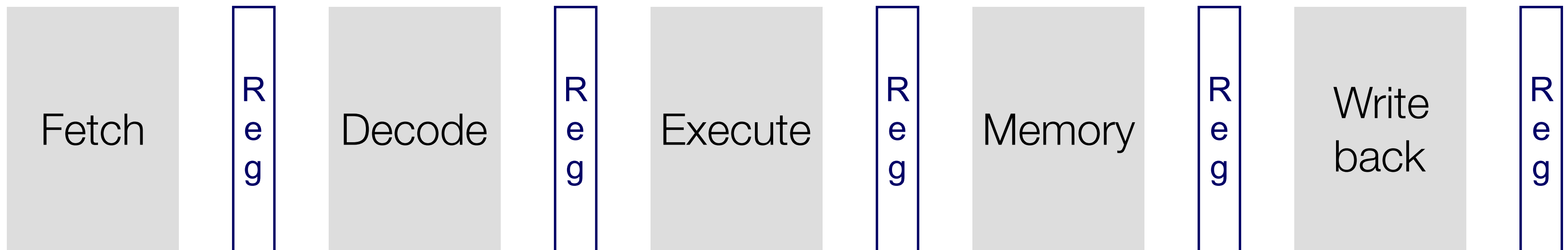




**addq reads the correct %rdx and %rax**

# 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?

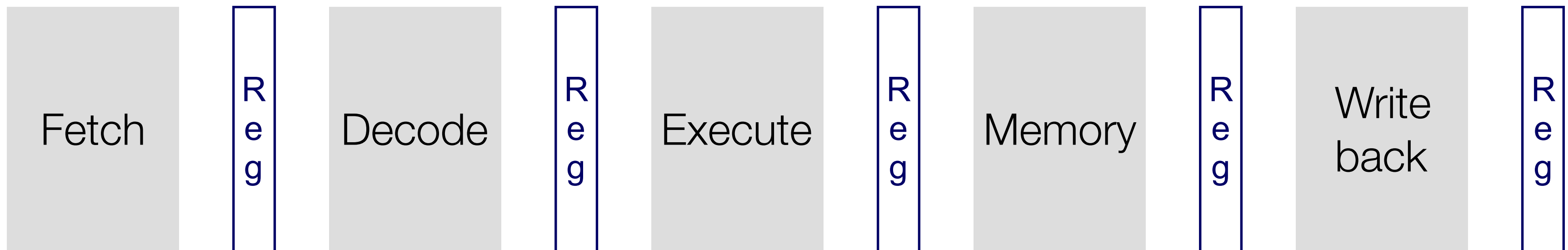


# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

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

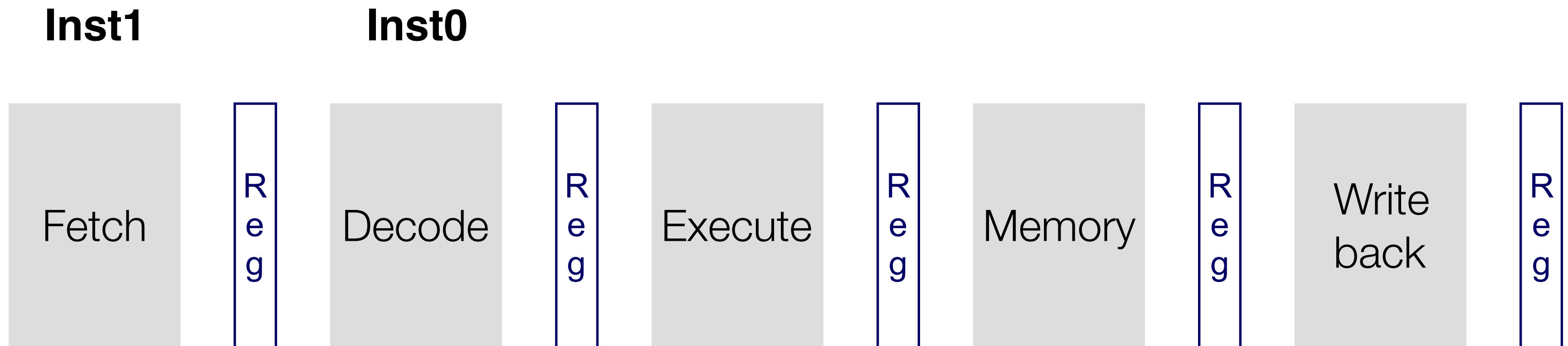
**Inst0**



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

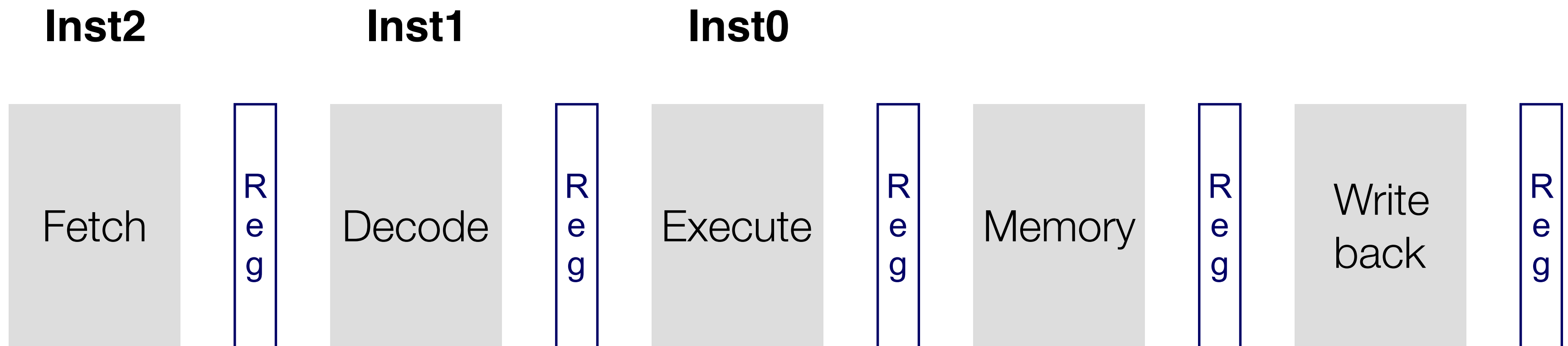
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

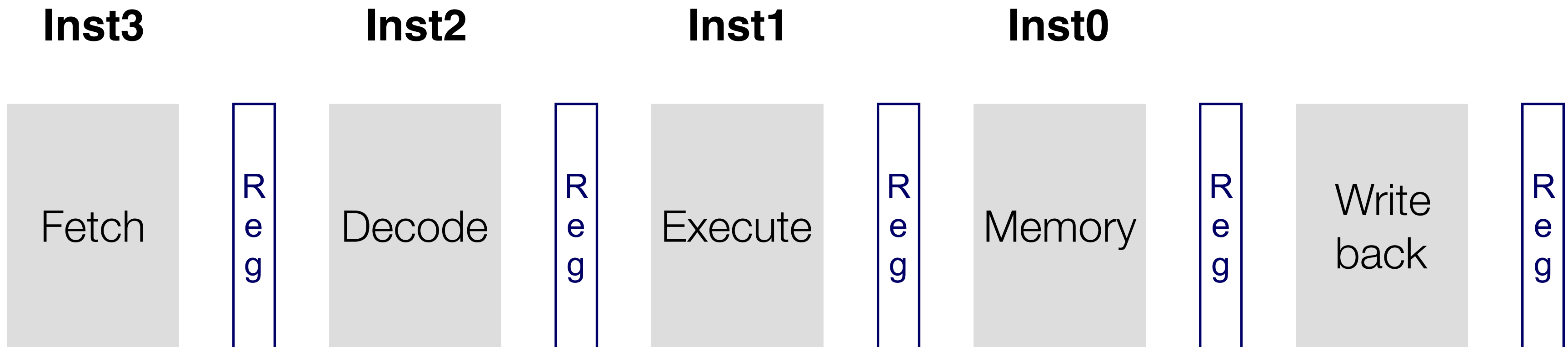
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

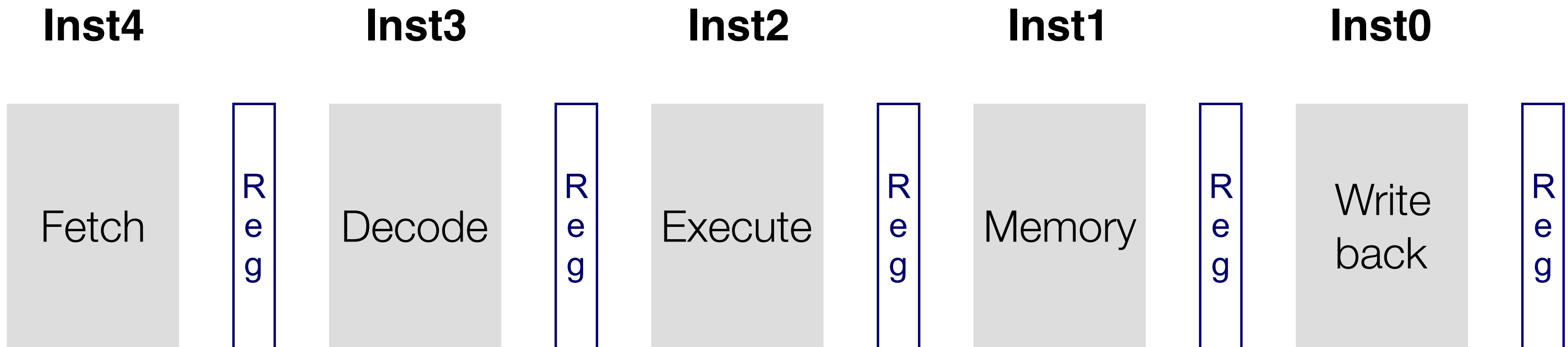
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

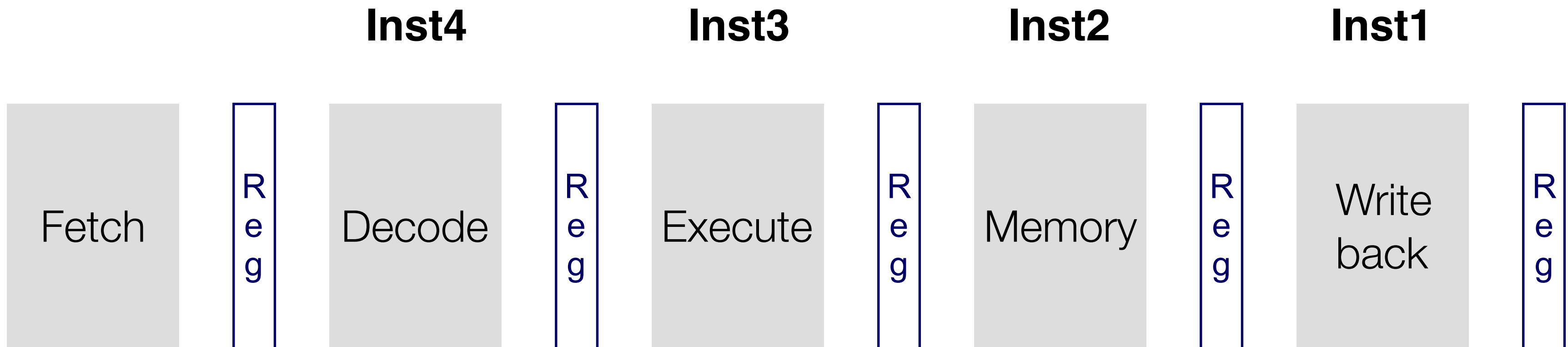
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

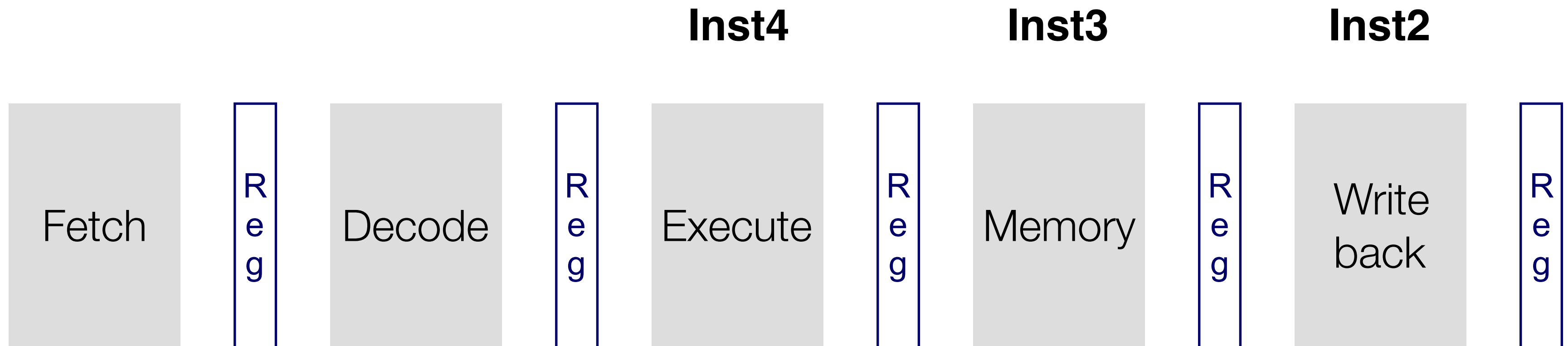
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

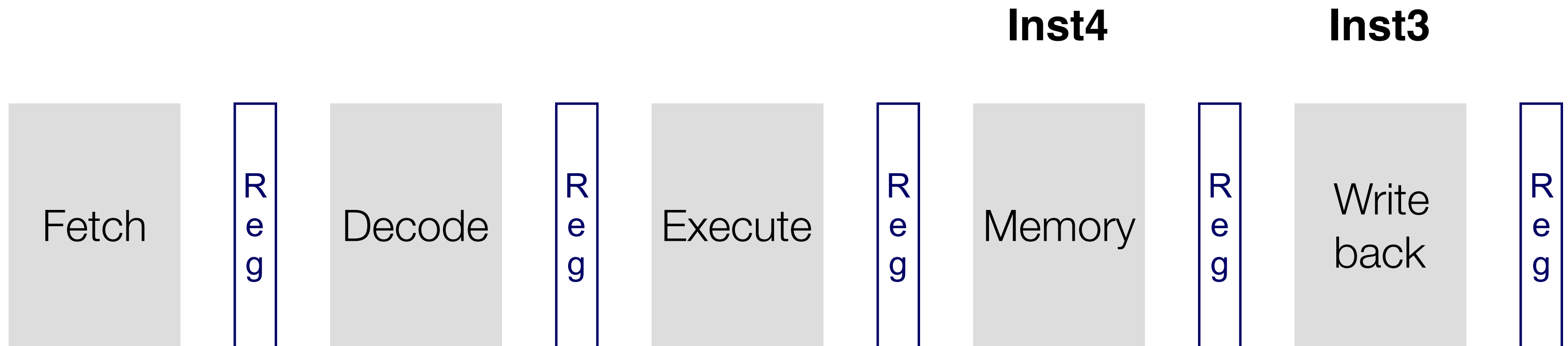
- Why is it good for the hardware to do so anyways?



# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

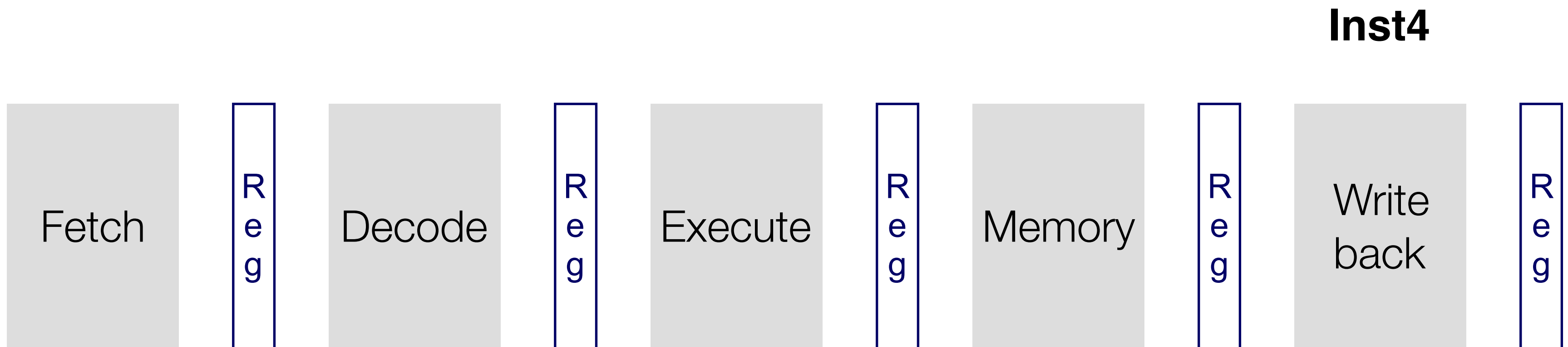
- Why is it good for the hardware to do so anyways?



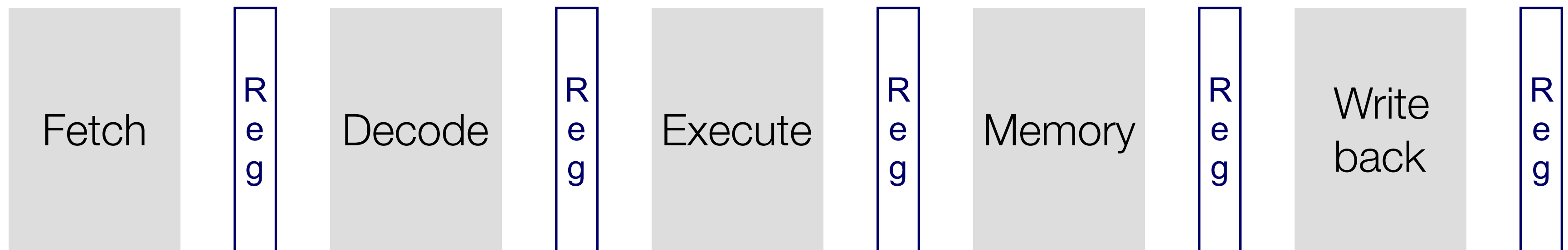
# Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

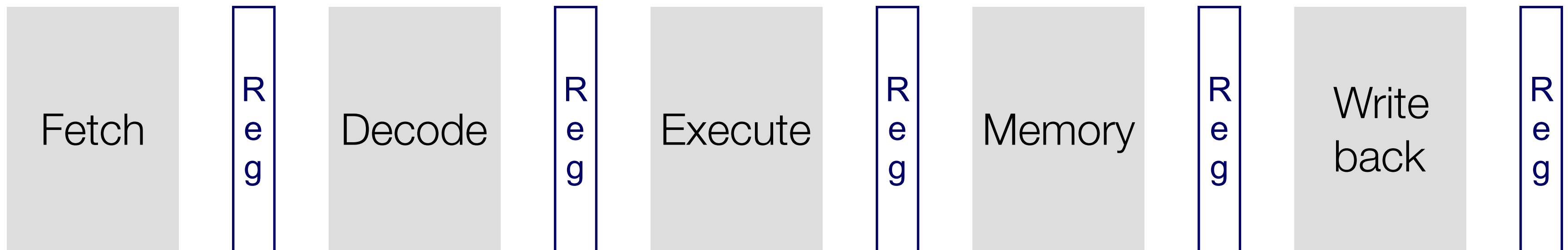
- Why is it good for the hardware to do so anyways?



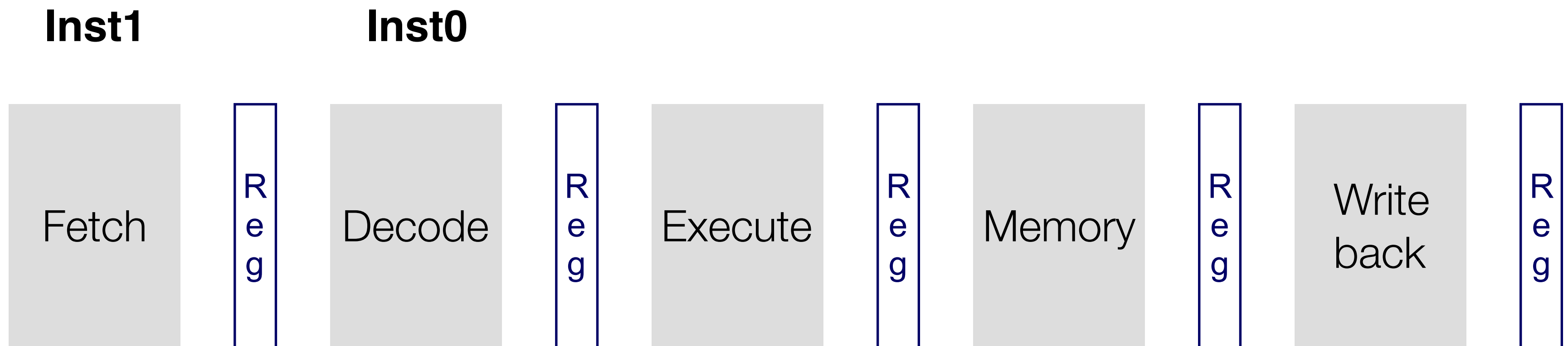
# Stalling Illustration



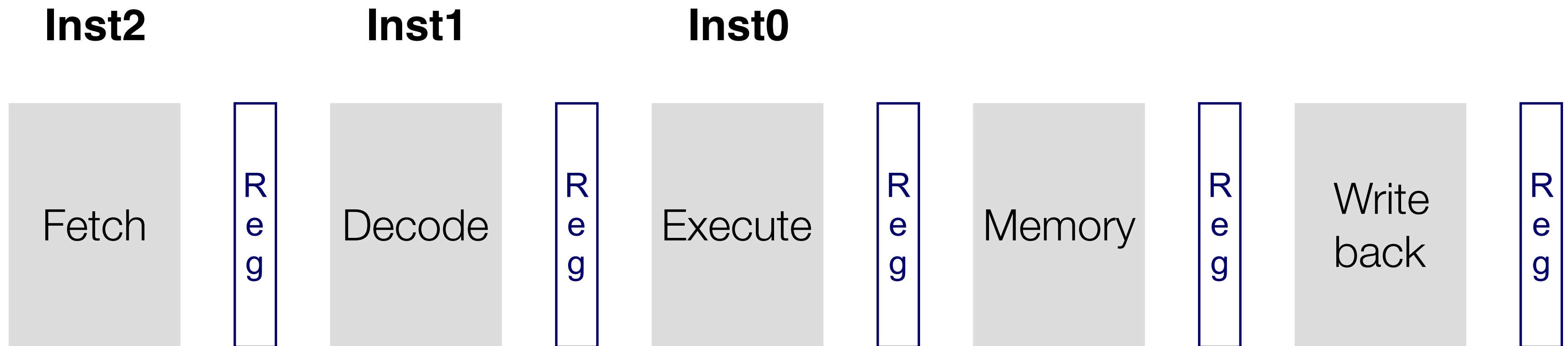
# Stalling Illustration



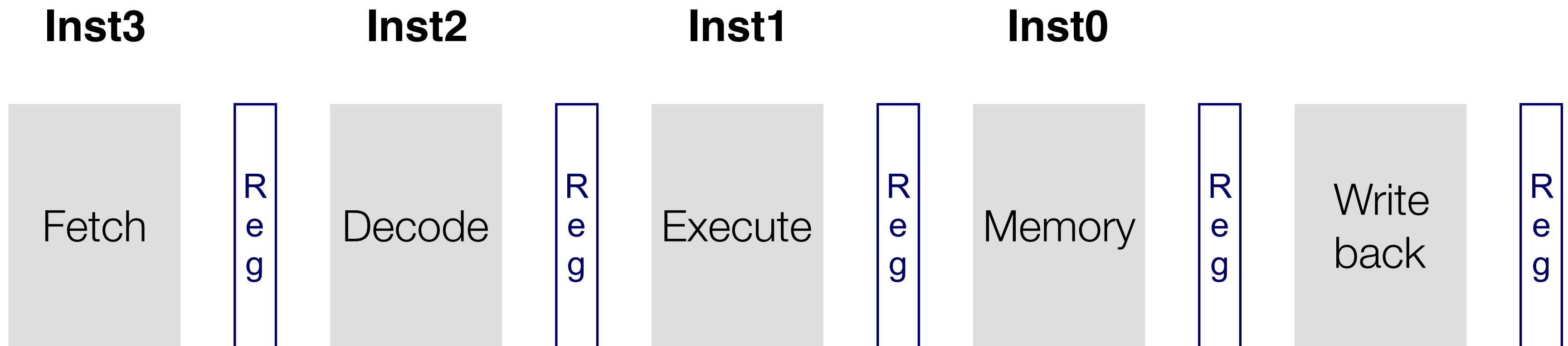
# Stalling Illustration



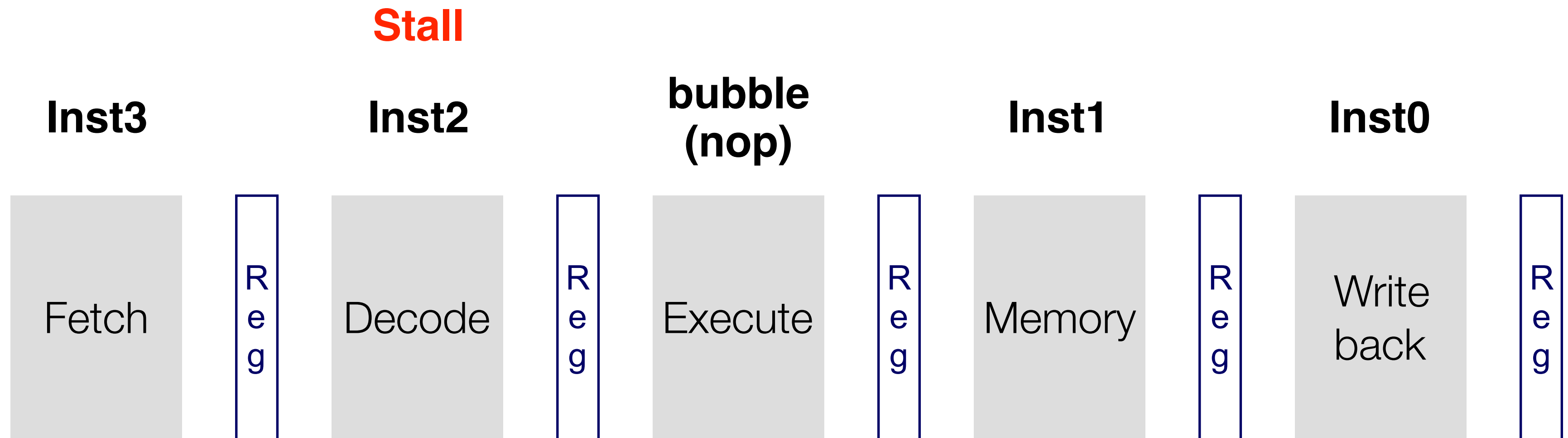
# Stalling Illustration



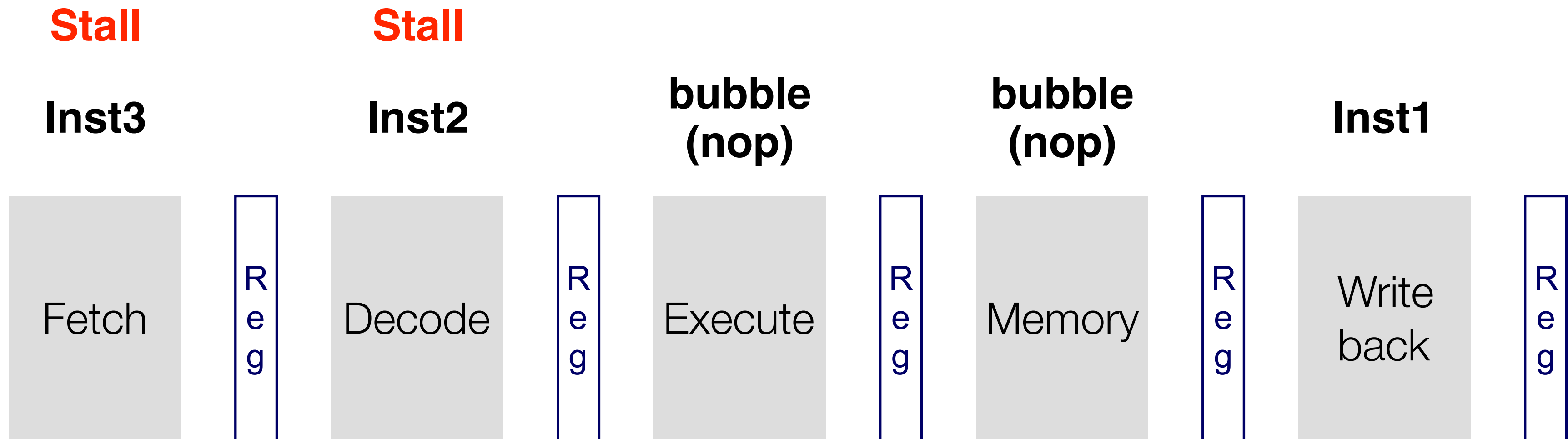
# Stalling Illustration



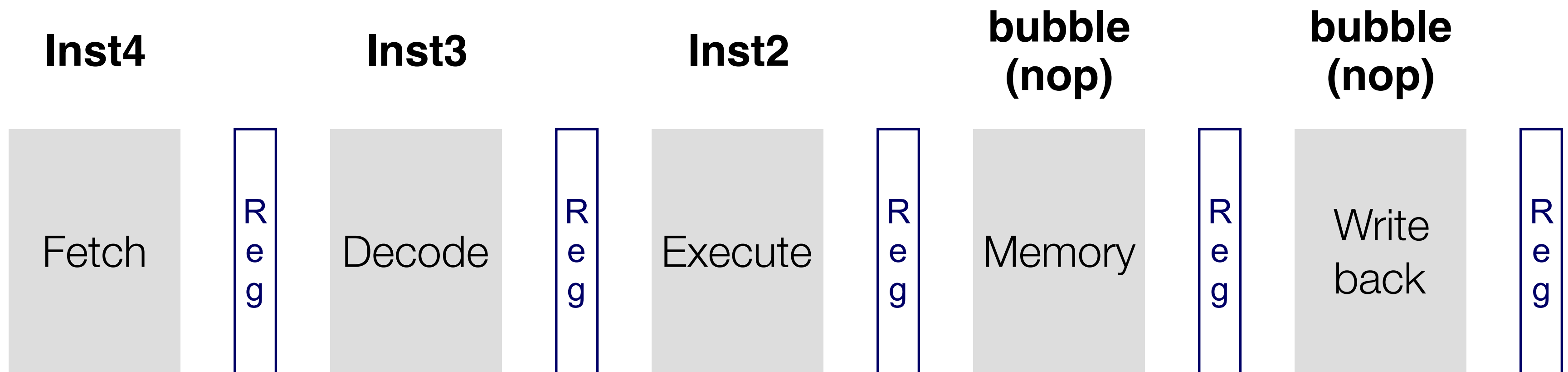
# Stalling Illustration



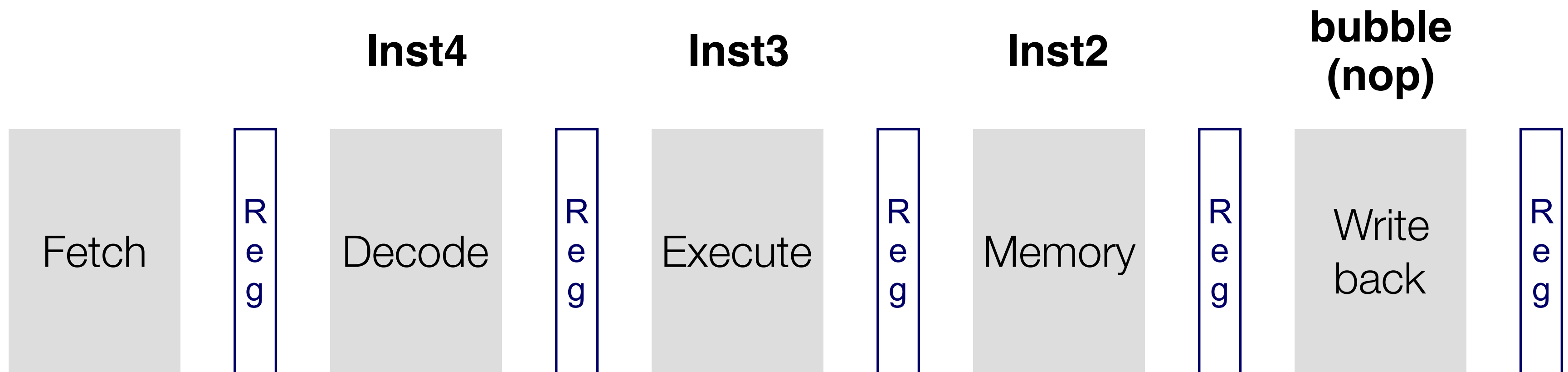
# Stalling Illustration



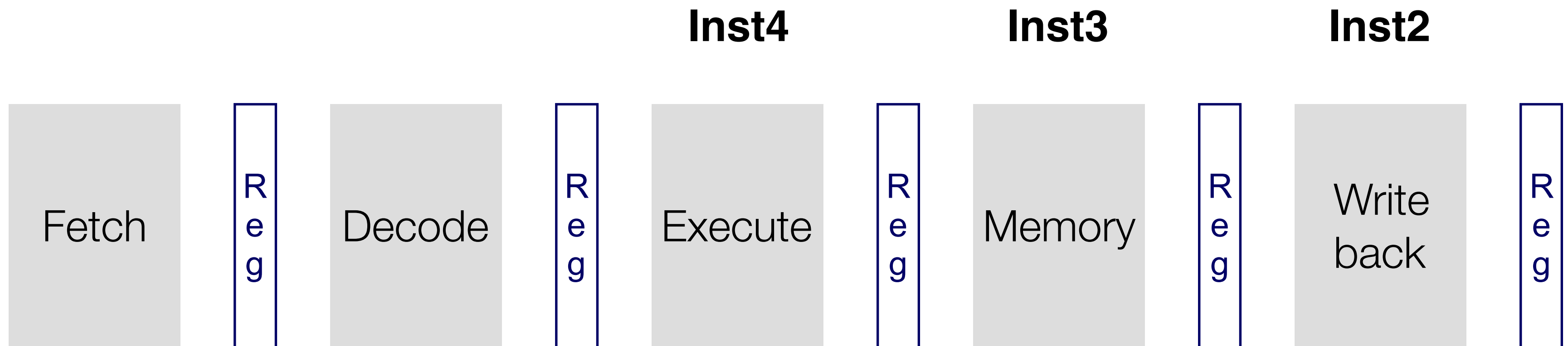
# Stalling Illustration



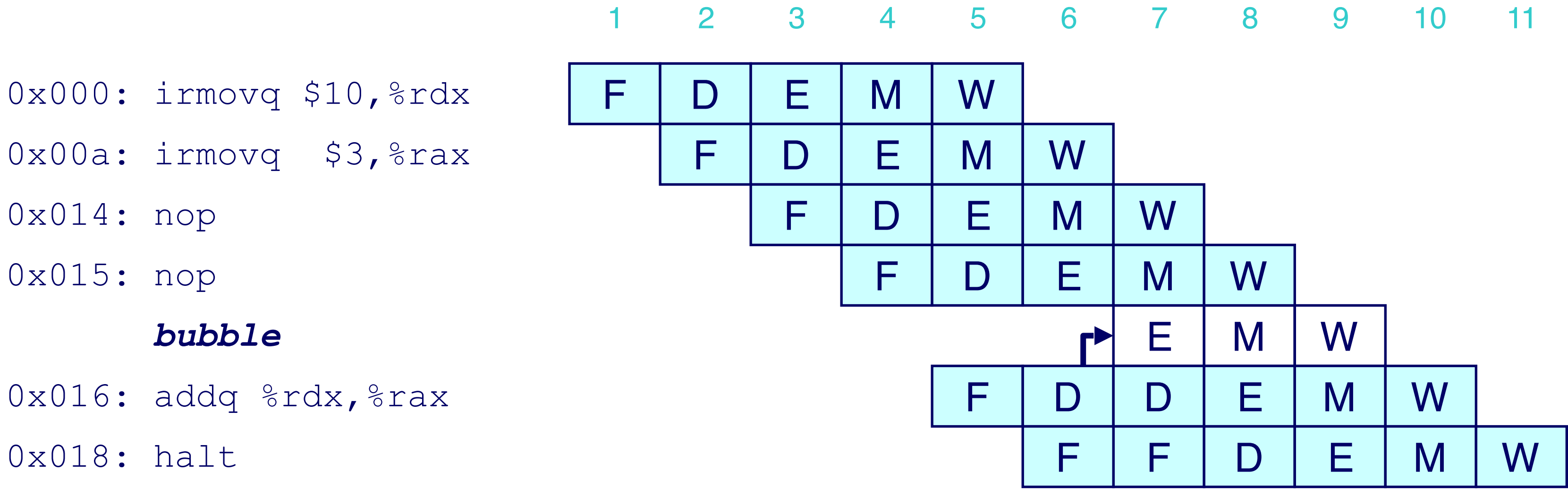
# Stalling Illustration



# Stalling Illustration

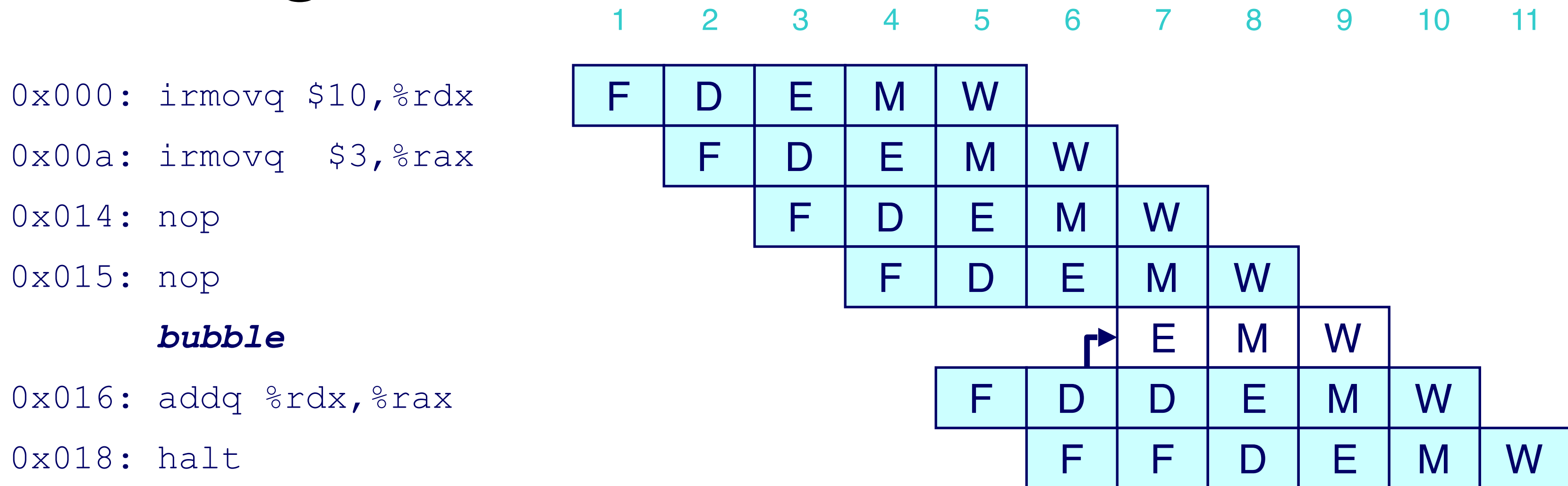


# Stalling for Data Dependencies




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

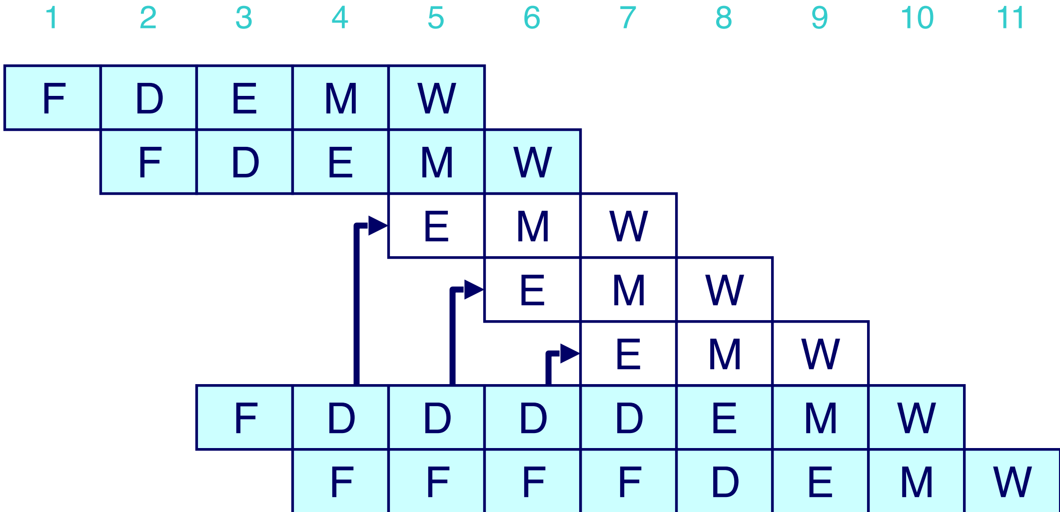
# Detecting Stall Condition




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

# Stalling X3

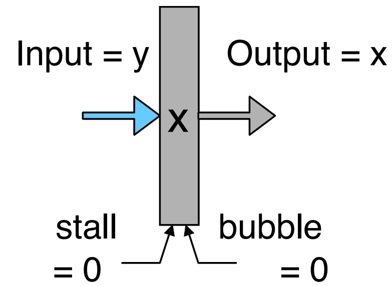
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



# How are Stall and Bubble Implemented in Hardware?

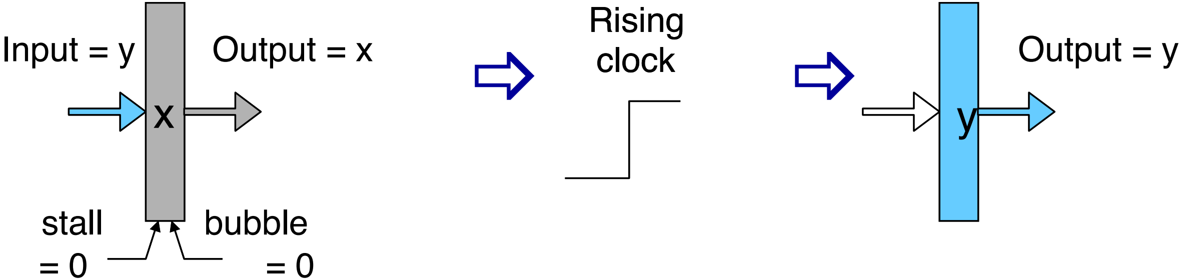
# How are Stall and Bubble Implemented in Hardware?

**Normal**



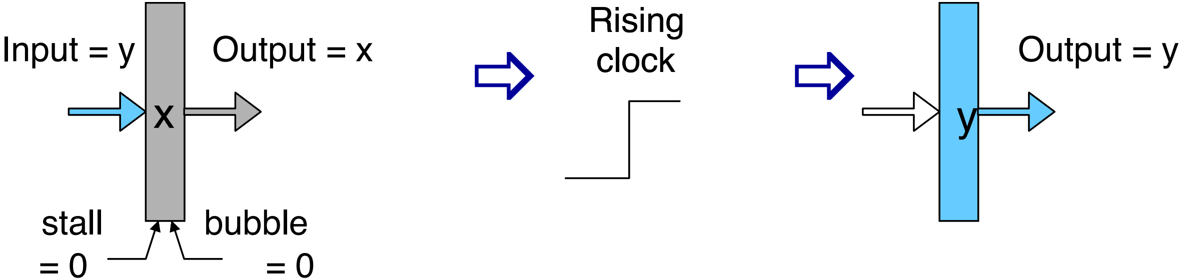
# How are Stall and Bubble Implemented in Hardware?

**Normal**

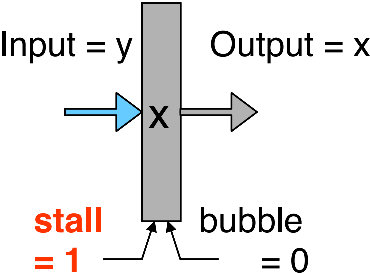


# How are Stall and Bubble Implemented in Hardware?

**Normal**

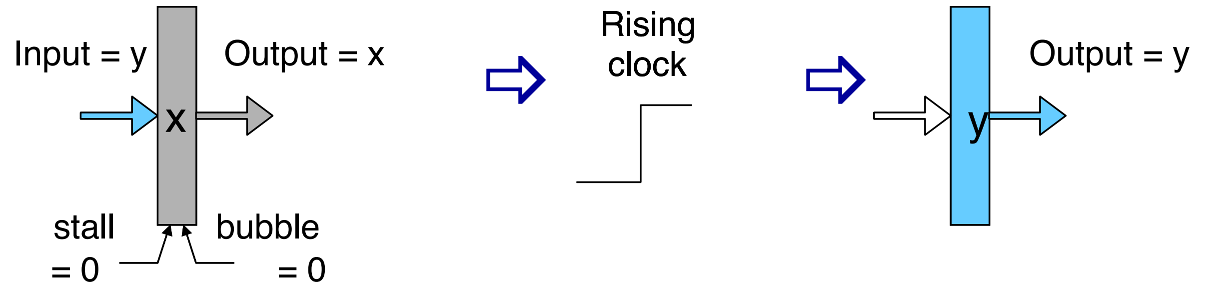



**Stall**

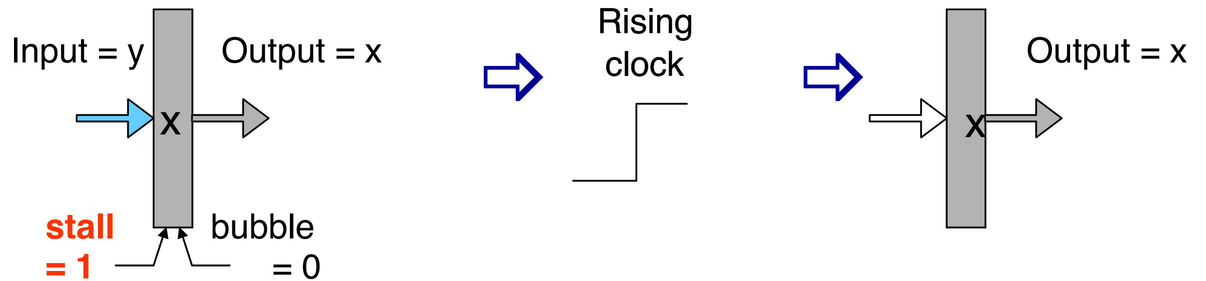


# How are Stall and Bubble Implemented in Hardware?

**Normal**

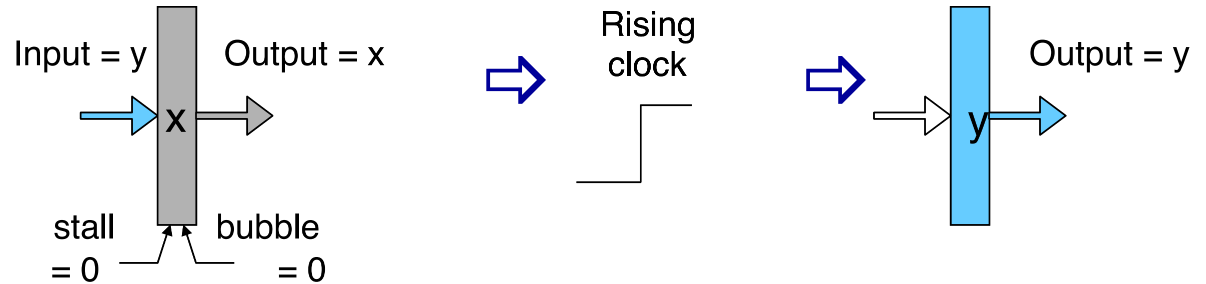



**Stall**

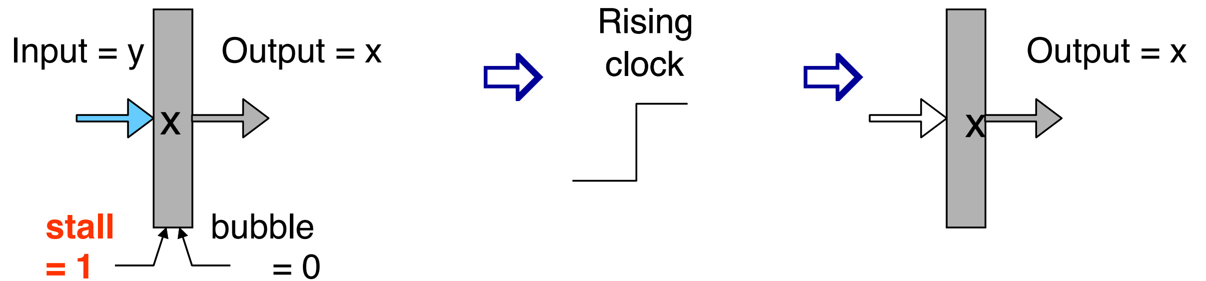


# How are Stall and Bubble Implemented in Hardware?

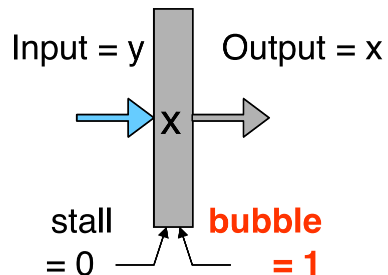
**Normal**




**Stall**

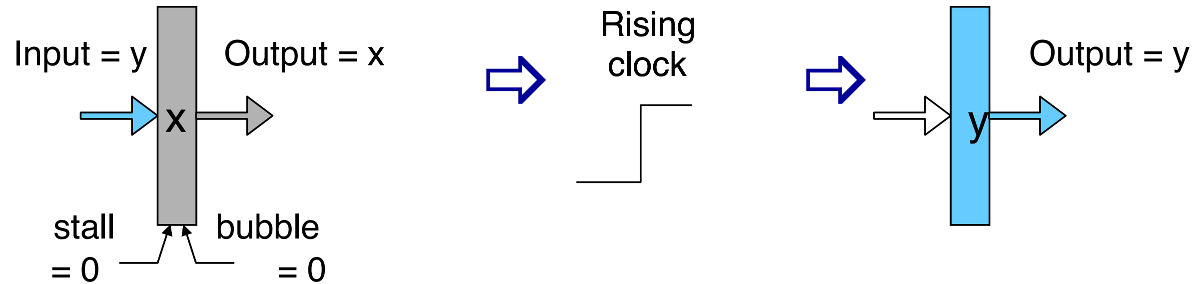



**Bubble**

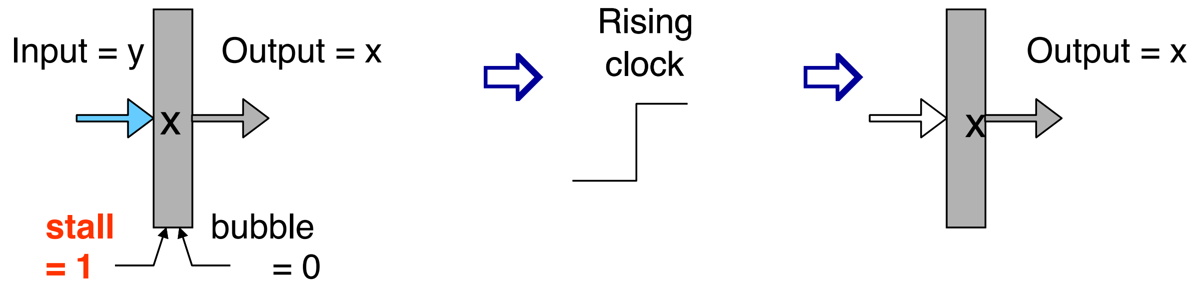


# How are Stall and Bubble Implemented in Hardware?

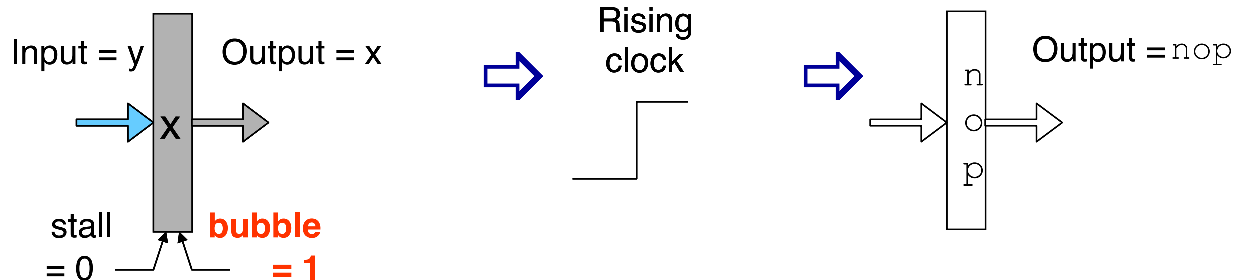
**Normal**




**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

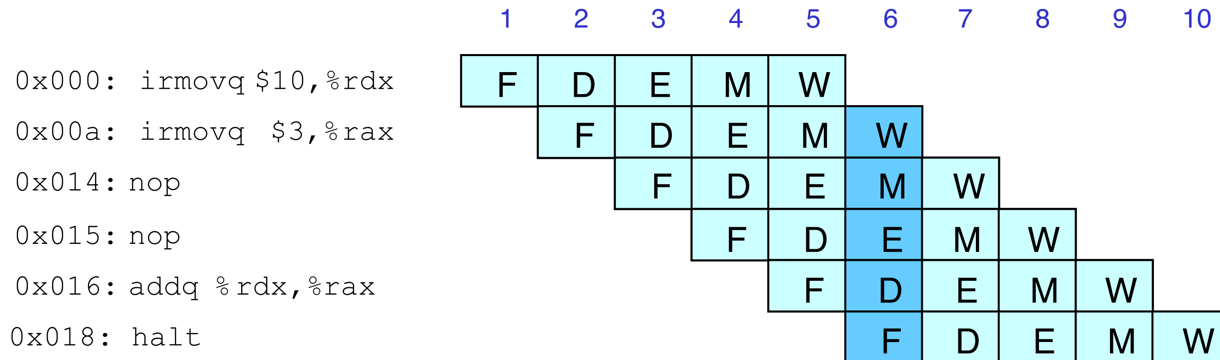
Observation

- 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

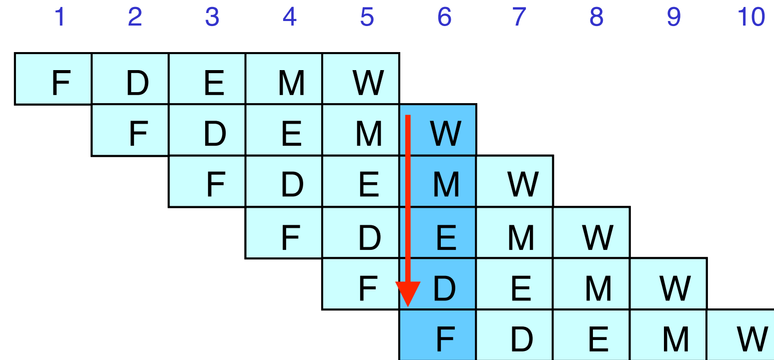
# Data Forwarding Example




- `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

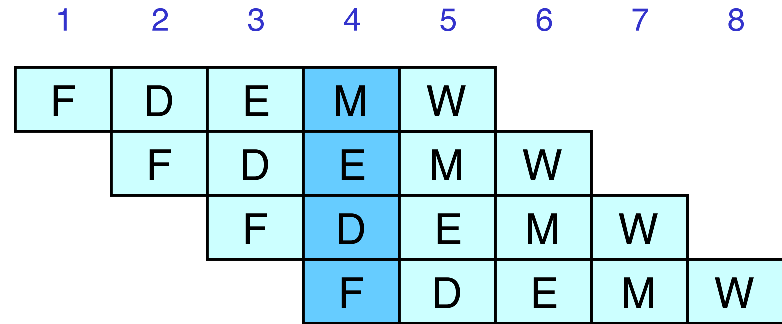
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```




- `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`

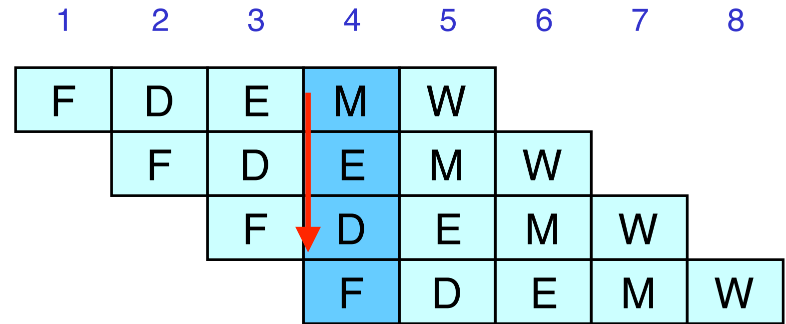
- Forward from the memory stage

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`

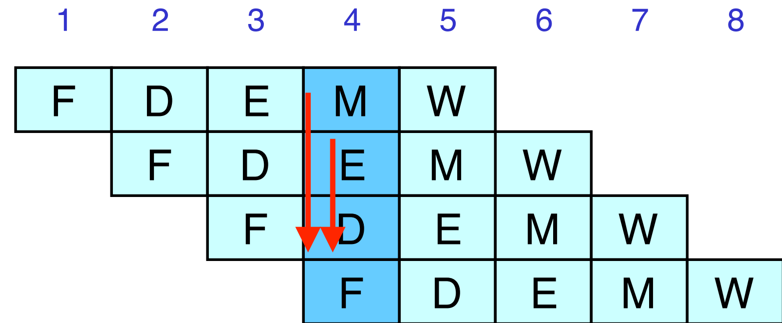
- Forward from the memory stage

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.

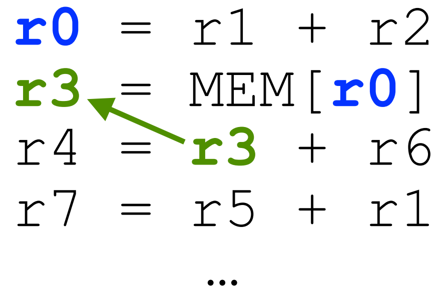
```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r7 = r5 + r1
      …
```

```
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]
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
```

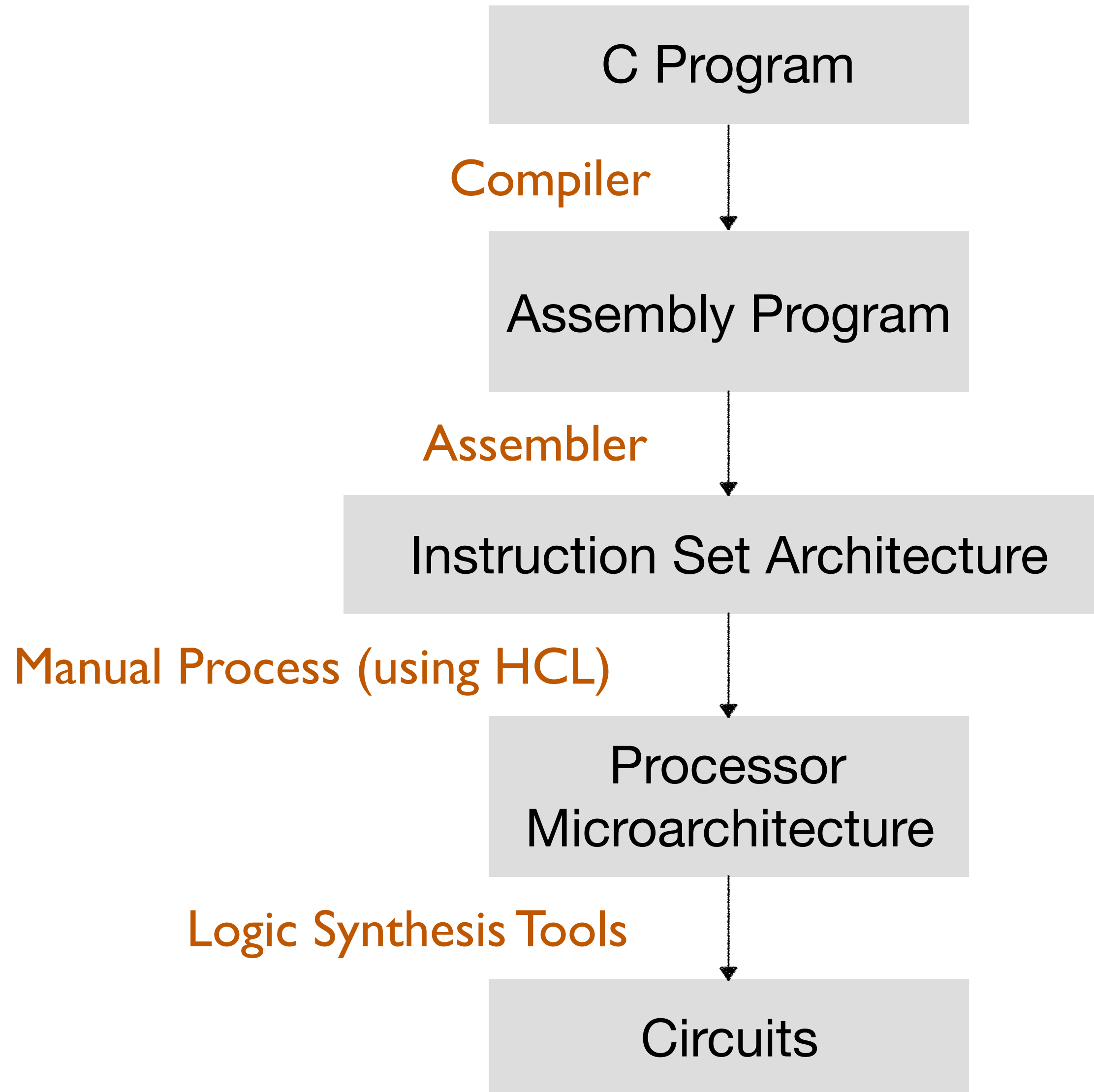
```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r4 = r5 + r1
     ...
```

Is this correct?

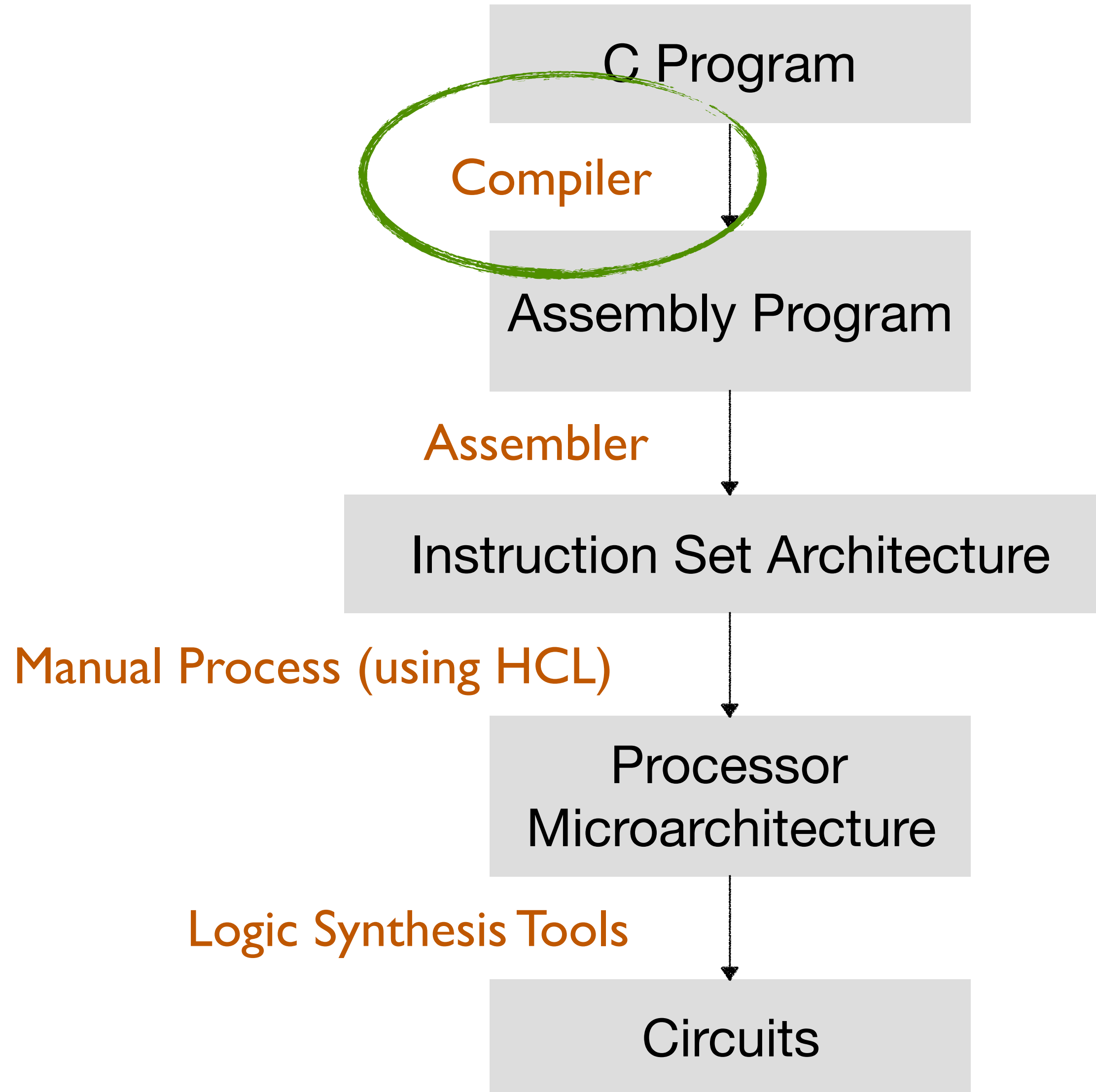
```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r5 + r1
     ...
r4 = r3 + r6
```

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.

# So far in 252…



# So far in 252…



# Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
- Exploit Hardware Microarchitecture

# Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
          a[n*i+j] = b[j];
}
```

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
          a[ni+j] = b[j];
```

# Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
          a[n*i+j] = b[j];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
          *rowp++ = b[j];
```

```
set_row:
          testq     %rcx, %rcx                      # Test n
          jle       .L1                             # If 0, goto done
          imulq     %rcx, %rdx                      # ni = n*i
          leaq      (%rdi,%rdx,8), %rdx # rowp = A + ni*8
          movl      $0, %eax                        # j = 0
.L3:                                                # loop:
          movsd     (%rsi,%rax,8), %xmm0            # t = b[j]
          movsd     %xmm0, (%rdx,%rax,8)            # M[A+ni*8 + j*8] = t
          addq      $1, %rax                        # j++
          cmpq      %rcx, %rax                      # j:n
          jne       .L3                             # if !=, goto loop
.L1:                                                # done:
          rep ; ret
```

# Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - `16*x` --> `x << 4`
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

# Common Subexpression Elimination

- Reuse portions of expressions
- GCC will do this with –O1

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

1 multiplication: i*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

```
imulq     %rcx, %rsi   # i*n
addq      %rdx, %rsi   # i*n+j
movq      %rsi, %rax   # i*n+j
subq      %rcx, %rax   # i*n+j-n
leaq      (%rsi,%rcx), %rcx # i*n+j+n
```

# Today: Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
- Exploit Hardware Microarchitecture

# Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

# Calling Strlen

```
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- Strlen performance
  - Has to scan the entire length of a string, looking for null character.
  - O(N) complexity
- Overall performance
  - N calls to strlen
  - Overall $O(N^2)$ performance

# Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

# Optimization Blocker: Procedure Calls

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}

size_t total_lencount = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    total_lencount += length;
    return length;
}
```

Why couldn't compiler move `strlen` out of loop?

- Procedure may have side effects, e.g., alters global state each time called
- Function may not return same value for given arguments

# Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
  - Use of inline functions
    - GCC does this with –O1, but only within single file
  - Do your own code motion

# Today: Optimizing Code Transformation

- Overview
- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
- Exploit Hardware Microarchitecture

# Exploiting Instruction-Level Parallelism

- Hardware can execute multiple instructions in parallel
  - Pipeline is a classic technique
- Performance limited by control/data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

# Baseline Code

```
for (i = 0; i < length; i++) {
  t = t * d[i];
  *dest = t;
}
```

```
.L519:
  imulq (%rax,%rdx,4), %ecx
  addq  $1, %rdx           # i++
  cmpq  %rdx, %rbp         # Compare length:i
  jg    .L519              # If >, goto Loop
```

← Real work

← Overhead

| Operation | Add | Mult |
|---|---|---|
| Combine4 | 1.27 | 3.01 |
| Operation Latency | 1.00 | 3.00 |

# Loop Unrolling (2x1)

```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * d[i]) * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x = x * d[i];
}
*dest = x;
```

- Perform 2x more useful work per iteration
- Reduce loop overhead (comp, jmp, index dec, etc.)

# Loop Unrolling with Separate Accumulators

```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x0 = x0 * d[i];
    x1 = x1 * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x0 = x0 * d[i];
}
*dest = x0 * x1;
```

# Separate Accumulators

```
x0 = x0 * d[i];
x1 = x1 * d[i+1];
```

x0 $d_0$ x1 $d_1$ $d_2$ $d_3$ $d_4$ $d_5$ $d_6$ $d_7$

- What changed:
  - Two independent “streams” of operations
  - Reduce data dependency