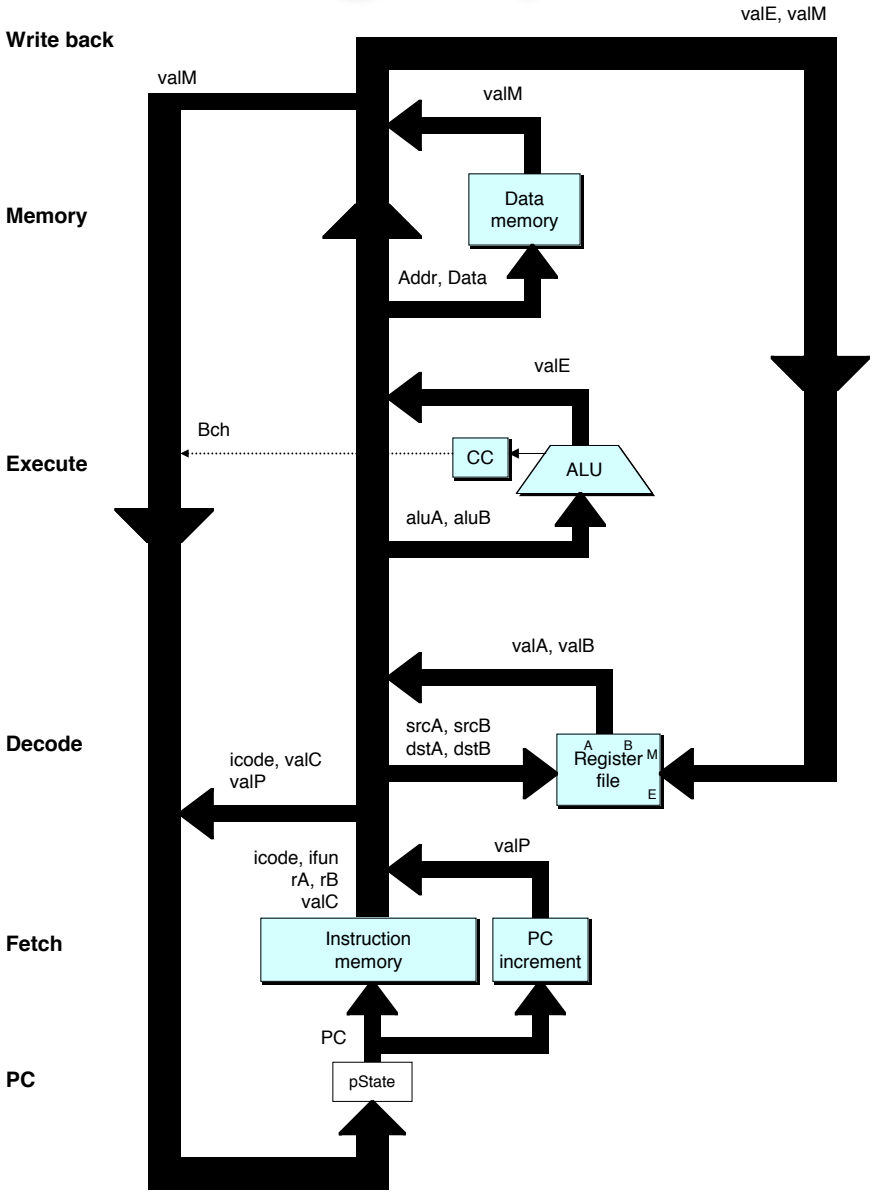
# Systems I

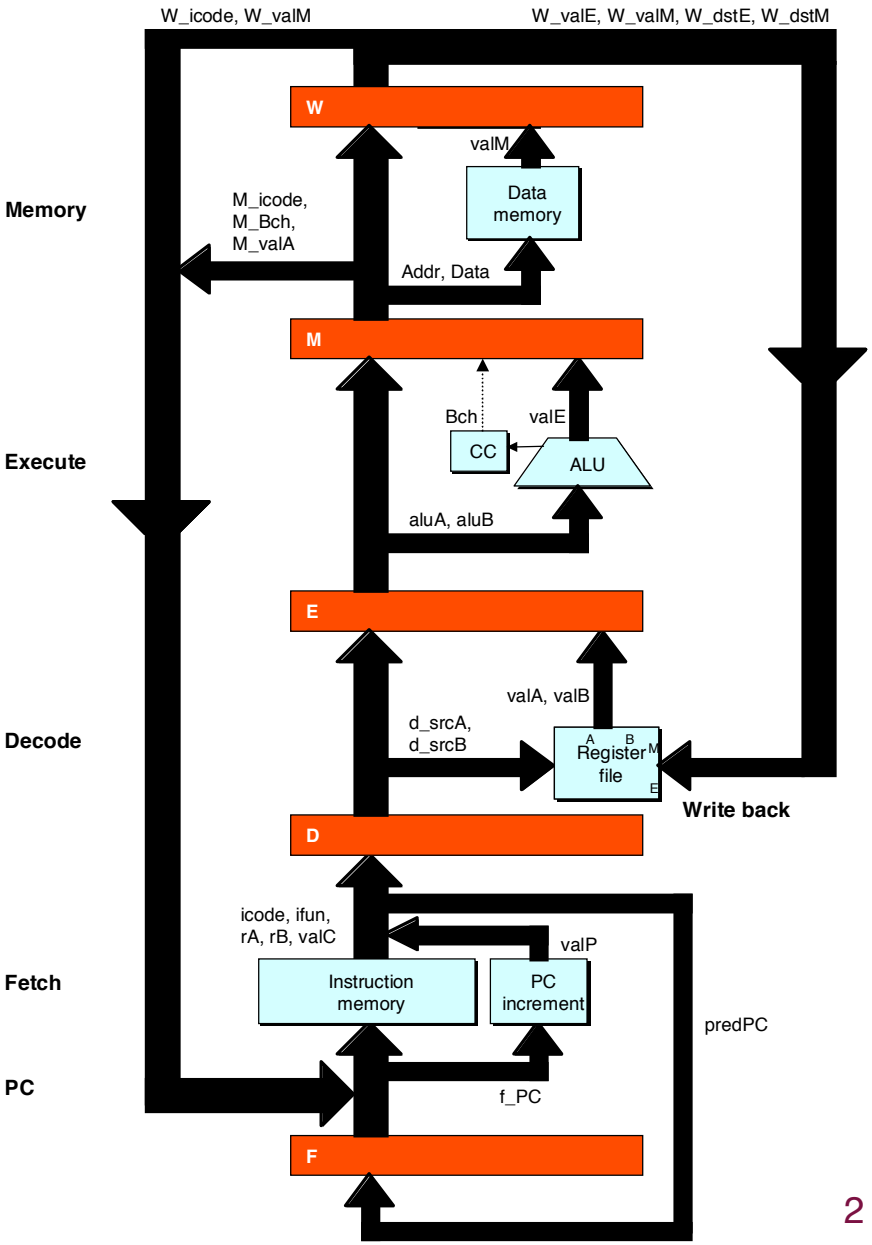
# Pipelining II

## Topics

- Pipelining hardware: registers and feedback paths
- Difficulties with pipelines: hazards
- Method of mitigating hazards

# Adding Pipeline Registers



# SEQ+ Hardware

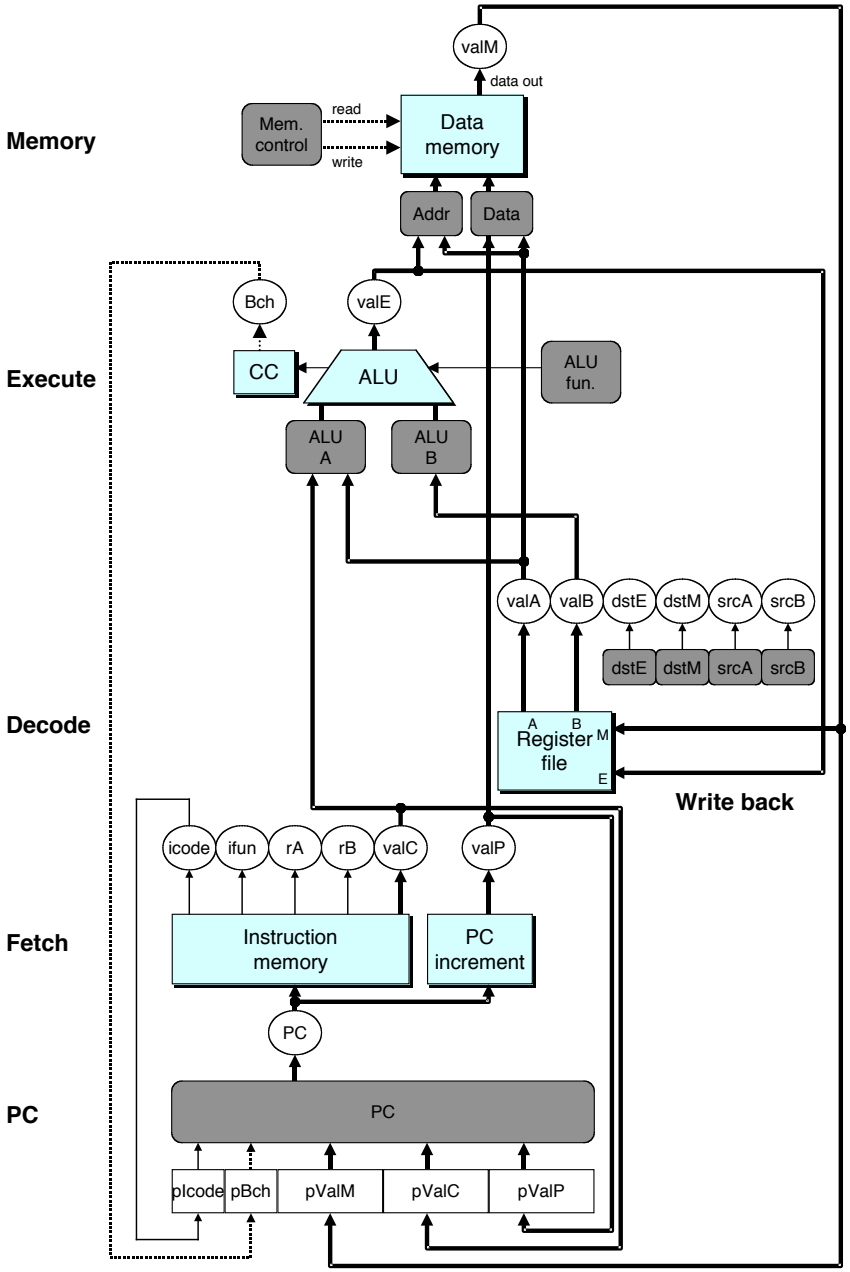
- Still sequential implementation
- Reorder PC stage to put at beginning

## PC Stage

- Task is to select PC for current instruction
- Based on results computed by previous instruction

## Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information

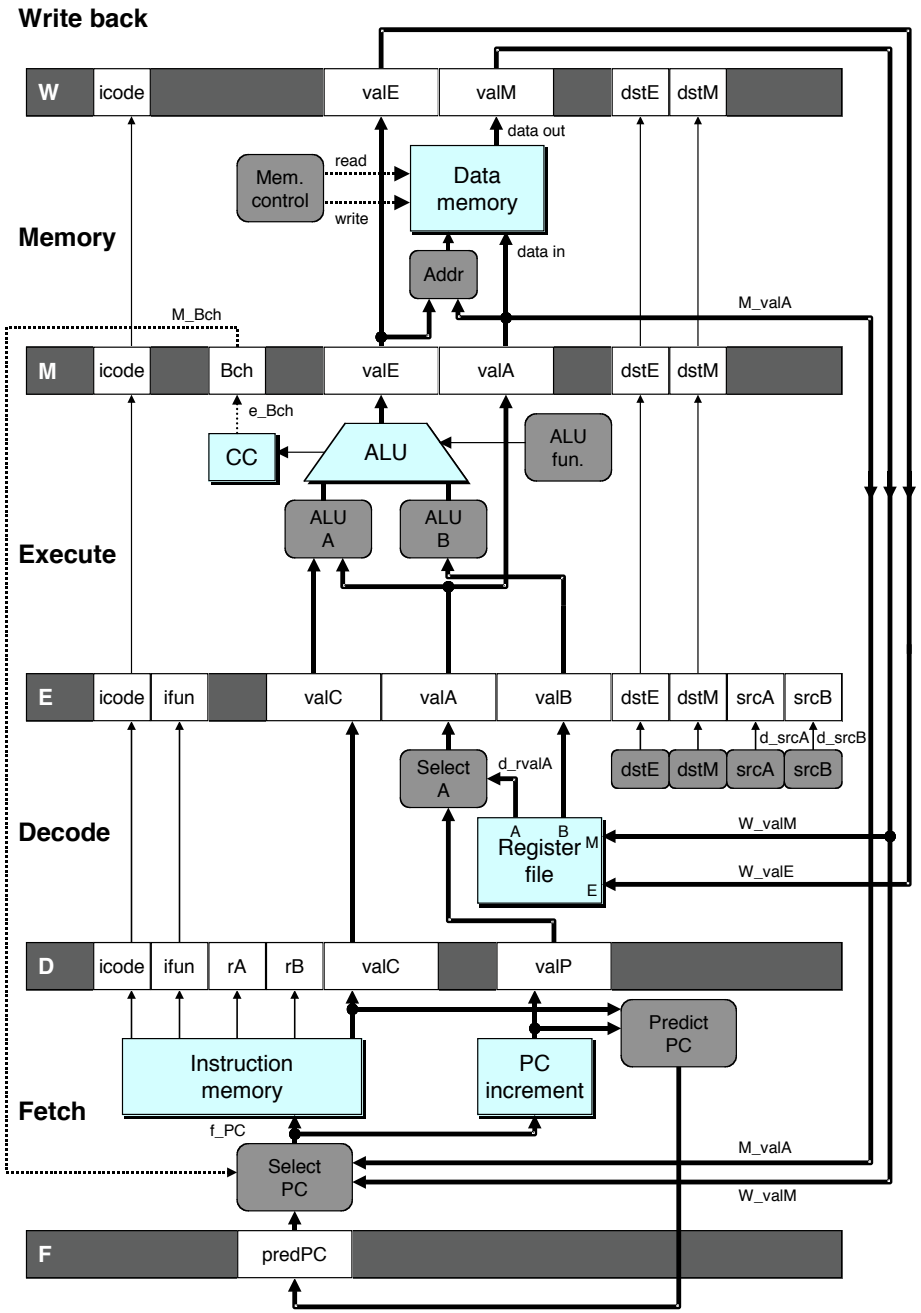


# PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution

## Forward (Upward) Paths

- Values passed from one stage to next
- Cannot jump past stages
  - e.g., valC passes through decode



# Feedback Paths

## Predicted PC

- Guess value of next PC

## Branch information

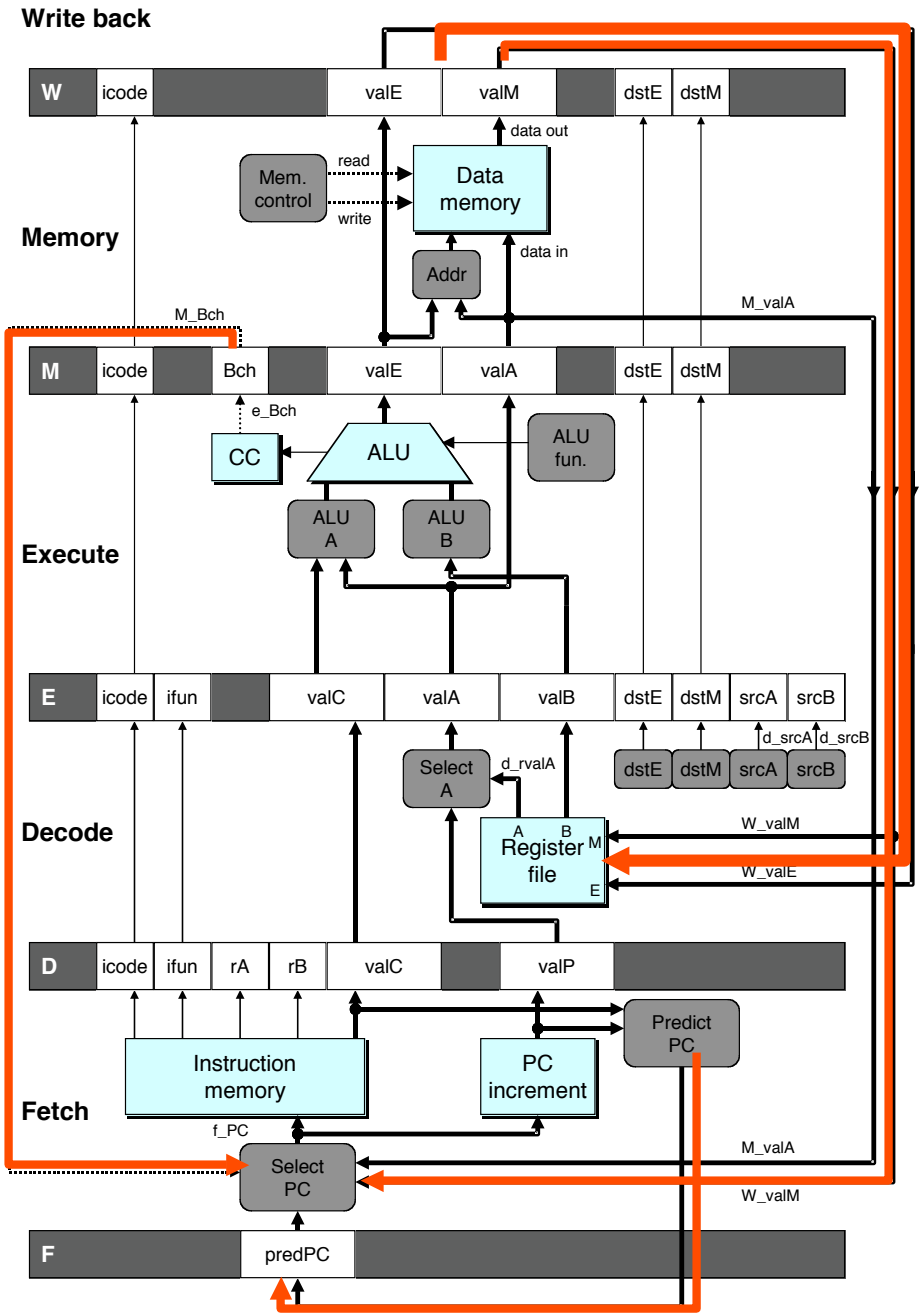
- Jump taken/not-taken
- Fall-through or target address

## Return point

- Read from memory

## Register updates

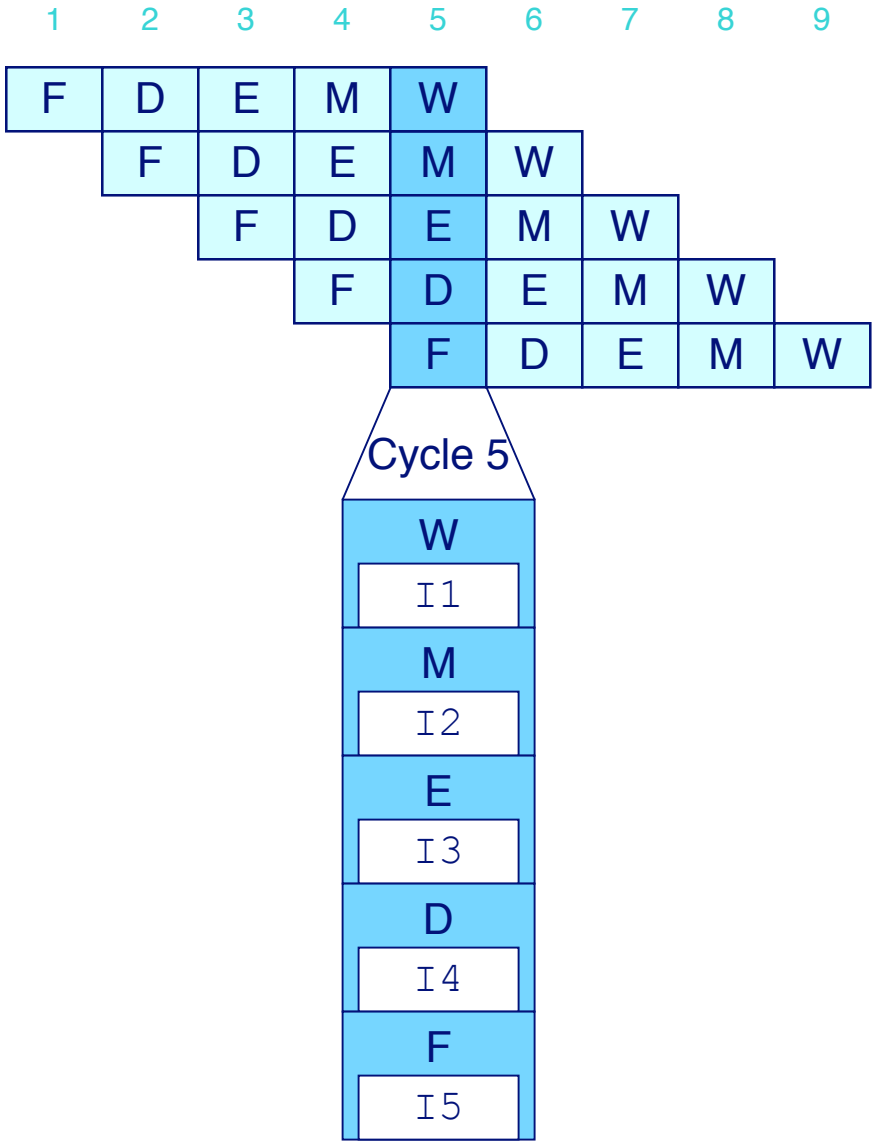
- To register file write ports



# Pipeline Demonstration

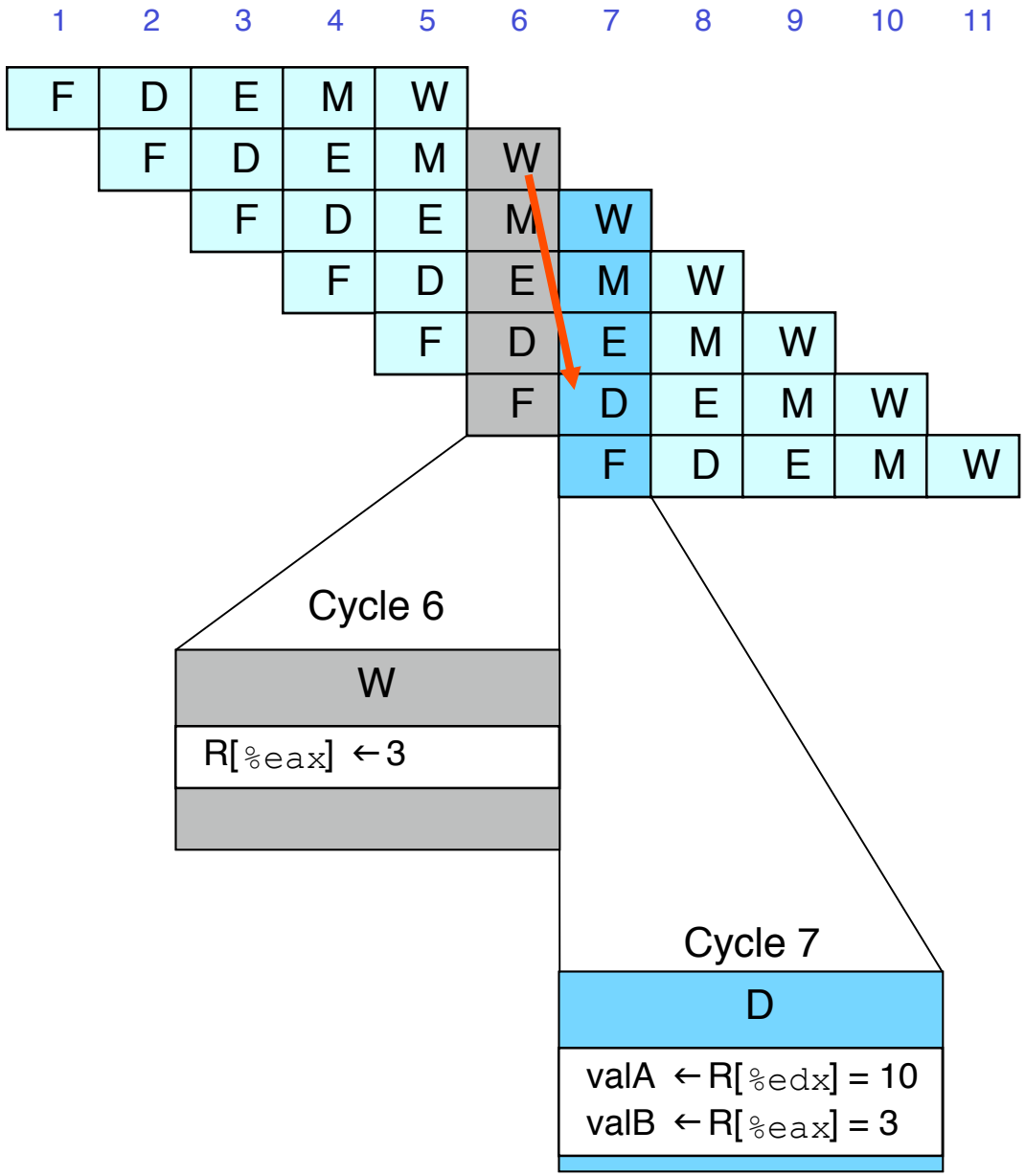
```
irmovl    $1,%eax   #I1
irmovl    $2,%ecx   #I2
irmovl    $3,%edx   #I3
irmovl    $4,%ebx   #I4
halt                #I5
```

**File: `demo-basic.ys`**



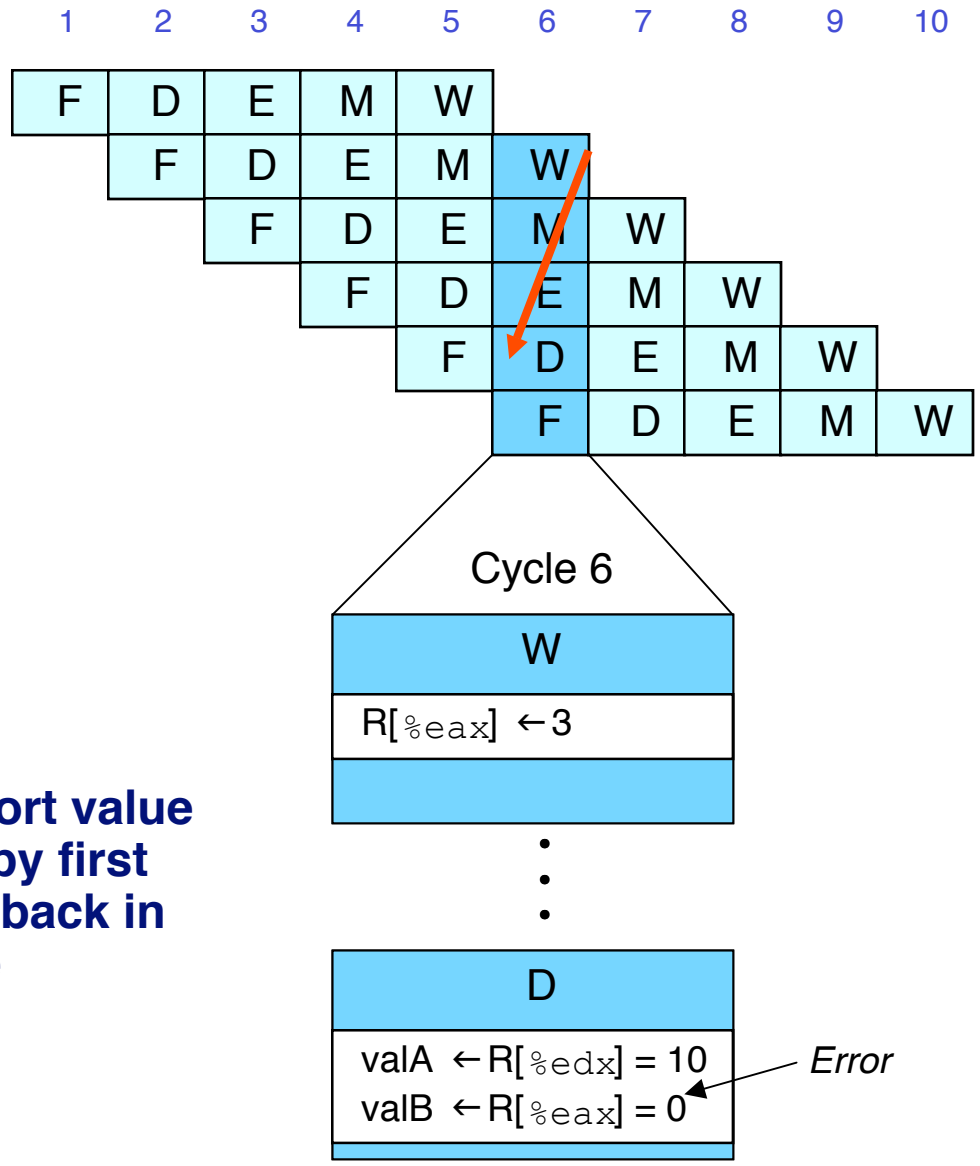
# Data Dependencies: 3 Nop's

```
# demo-h3.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```



# Data Dependencies: 2 Nop's

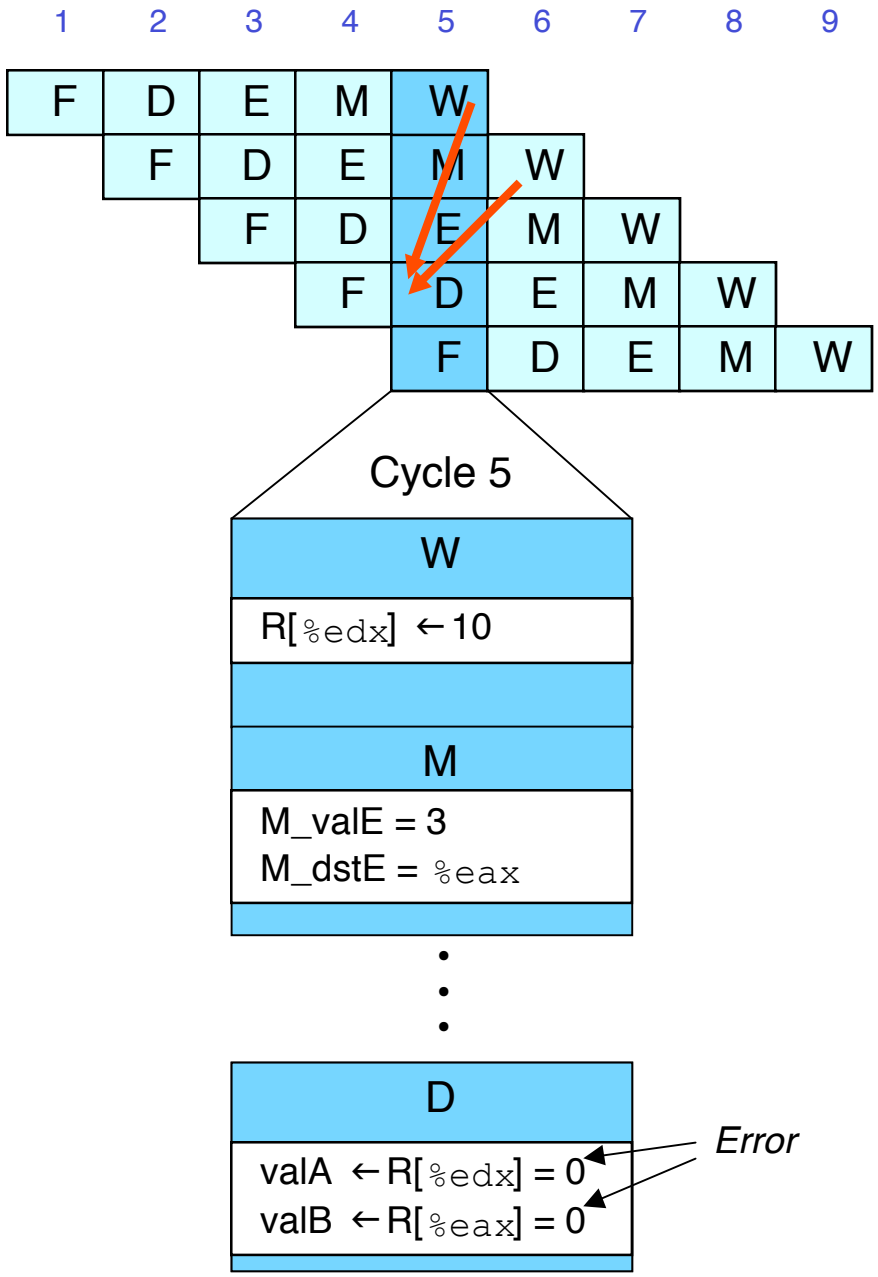
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```




**Can't transport value produced by first instruction back in time**

# Data Dependencies: 1 Nop

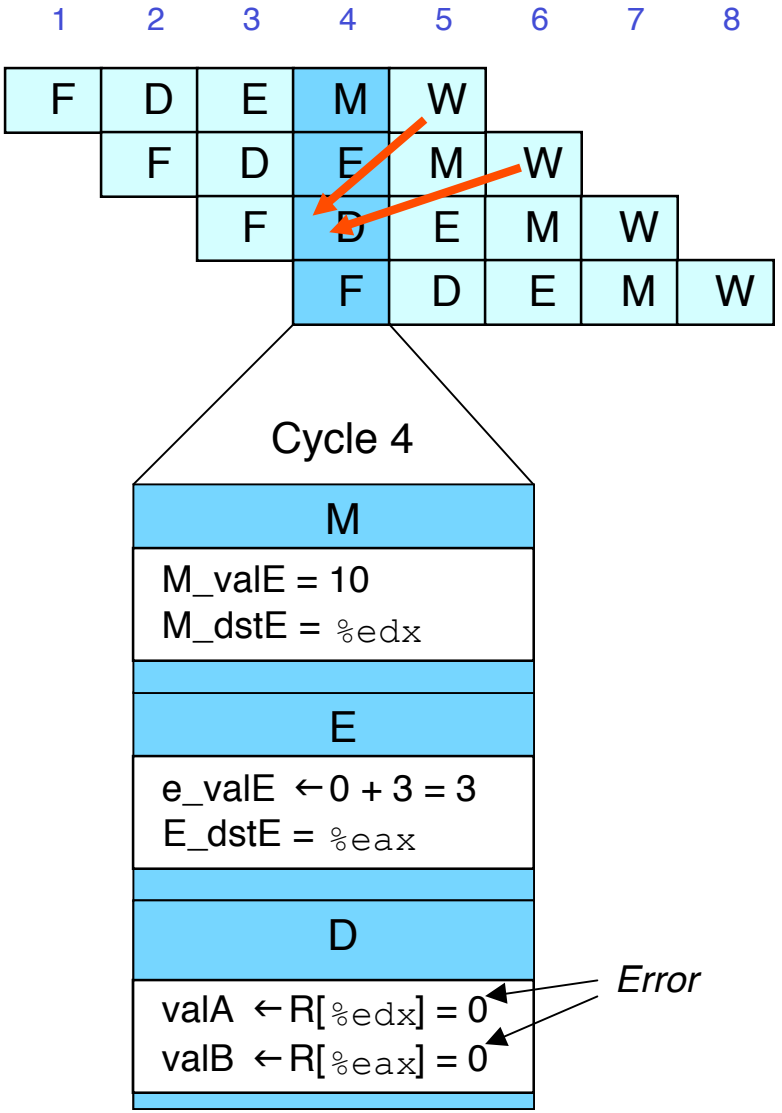
```
# demo-h1.ys

0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```




**Now a problem with both operands**

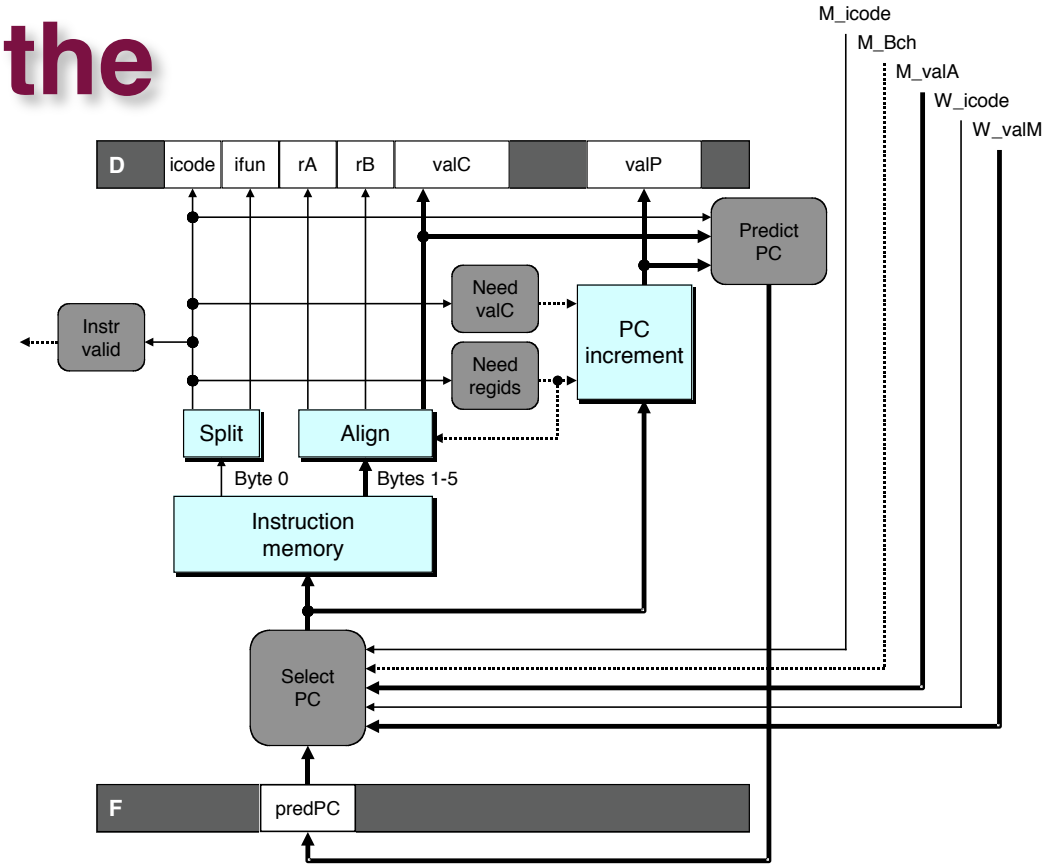
# Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```




**Wow - we really missed the boat here…**

# Predicting the PC




- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

# Our Prediction Strategy

## Instructions that Don't Transfer Control

- Predict next PC to be valP
- Always reliable

## Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

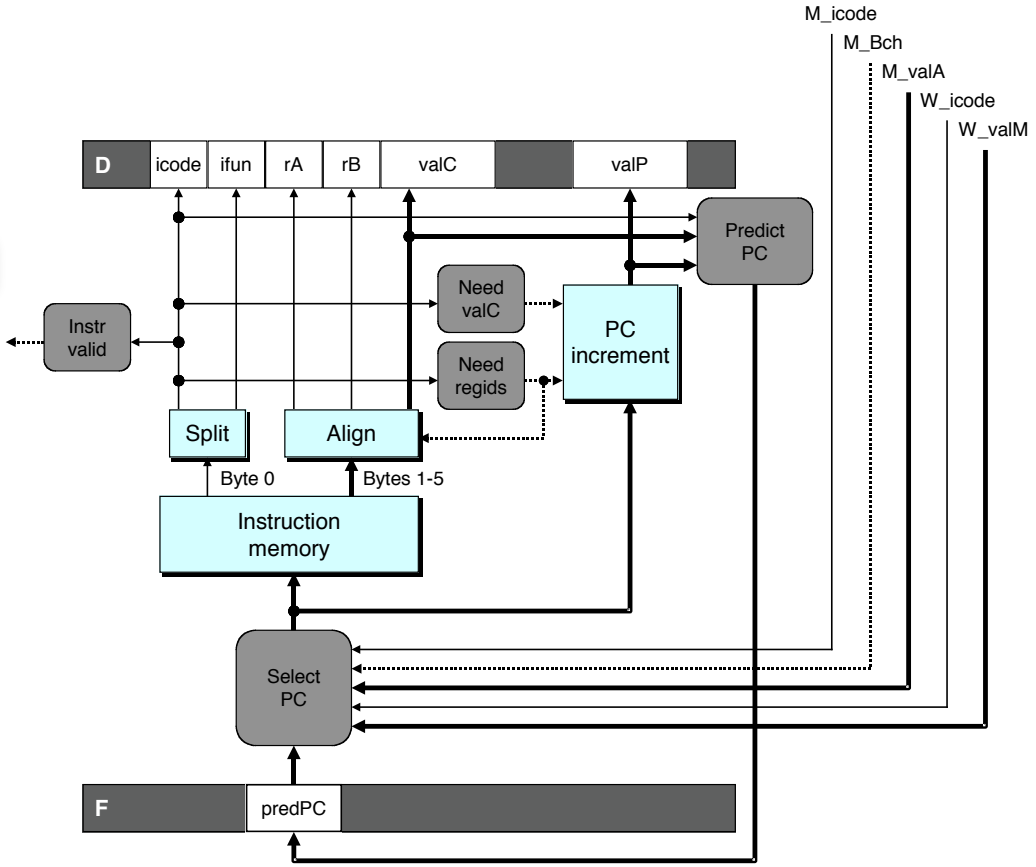
## Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

## Return Instruction

- Don't try to predict

# Recovering from PC Misprediction




- **Mispredicted Jump**
  - Will see branch flag once instruction reaches memory stage
  - Can get fall-through PC from valA
- **Return Instruction**
  - Will get return PC when `ret` reaches write-back stage
- **In both cases**
  - Need to throw away instructions fetched between prediction and resolution

# Branch Misprediction Example

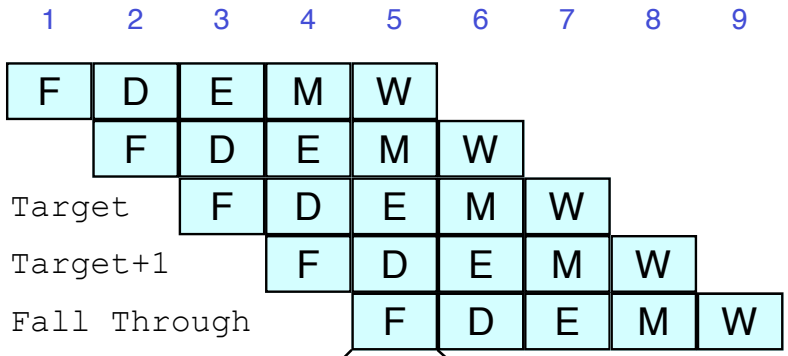
demo-j.ys

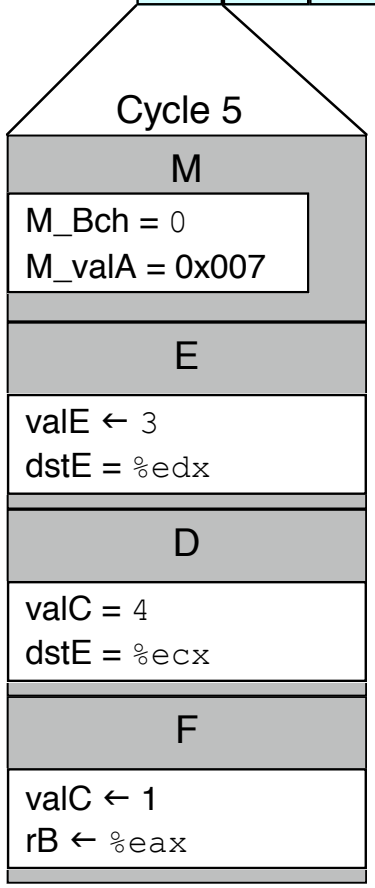
```
0x000:     xorl %eax,%eax
0x002:     jne  t               # Not taken
0x007:     irmovl $1, %eax      # Fall through
0x00d:     nop
0x00e:     nop
0x00f:     nop
0x010:     halt
0x011: t:  irmovl $3, %edx      # Target (Should not execute)
0x017:     irmovl $4, %ecx      # Should not execute
0x01d:     irmovl $5, %edx      # Should not execute
```

- Should only execute first 7 instructions

# Branch Misprediction Trace

```
# demo-j
0x000:     xorl %eax,%eax
0x002:     jne t # Not taken
0x011: t:  irmovl $3, %edx # Target
0x017:     irmovl $4, %ecx # Target+1
0x007:     irmovl $1, %eax # Fall Through
```




- **Incorrectly execute two instructions at branch target**

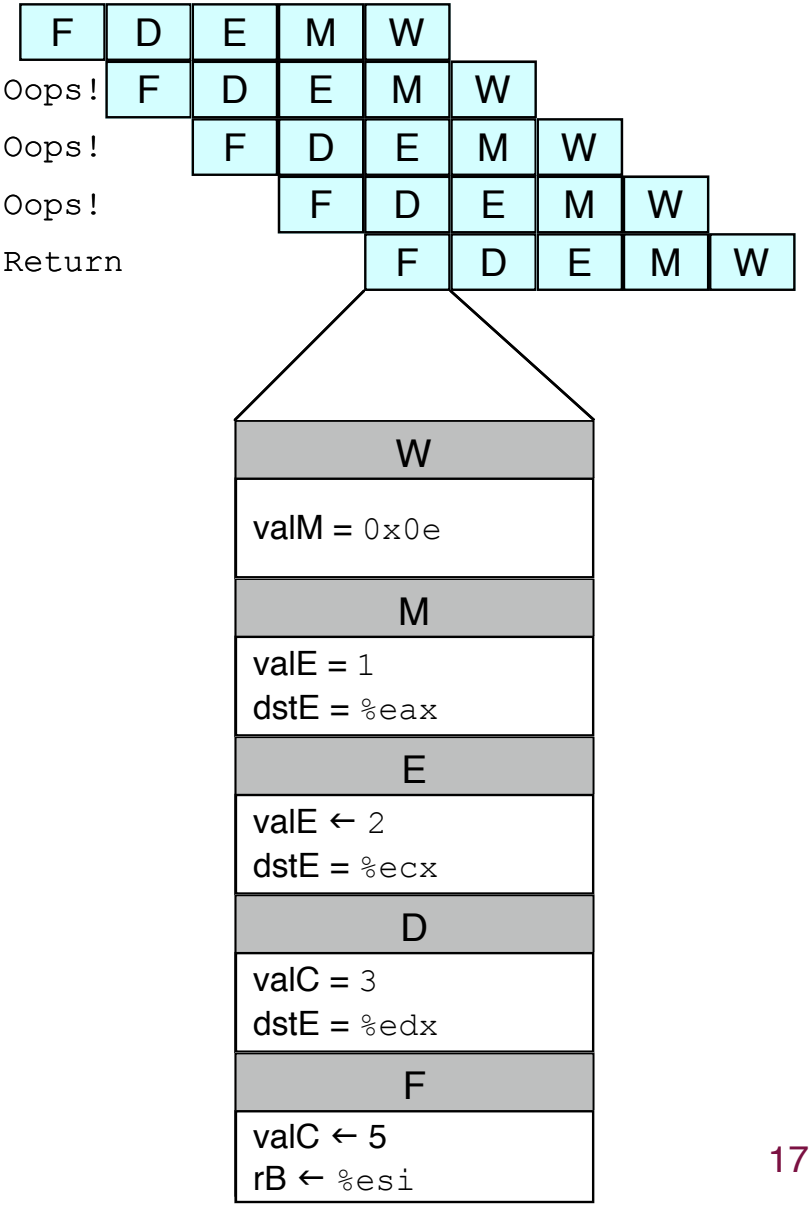
# Return Example

demo-ret.ys

```
0x000:    irmovl Stack,%esp  # Intialize stack pointer
0x006:    nop                # Avoid hazard on %esp
0x007:    nop
0x008:    nop
0x009:    call p             # Procedure call
0x00e:    irmovl $5,%esi     # Return point
0x014:    halt
0x020: .pos 0x20
0x020: p: <op>               # procedure
0x021:    <op>
0x022:    <op>
0x023:    ret
0x024:    irmovl $1,%eax     # Should not be executed
0x02a:    irmovl $2,%ecx     # Should not be executed
0x030:    irmovl $3,%edx     # Should not be executed
0x036:    irmovl $4,%ebx     # Should not be executed
0x100: .pos 0x100
0x100: Stack:                # Stack: Stack pointer
```

# Incorrect Return Example

```
# demo-ret

0x023:     ret
0x024:     irmovl $1,%eax # Oops!
0x02a:     irmovl $2,%ecx # Oops!
0x030:     irmovl $3,%edx # Oops!
0x00e:     irmovl $5,%esi # Return
```




- **Incorrectly execute 3 instructions following `ret`**

# Pipeline Summary

## Concept

- Break instruction execution into 5 stages
- Run instructions through in pipelined mode

## Limitations

- Can't handle dependencies between instructions when instructions follow too closely
- Data dependencies
  - One instruction writes register, later one reads it
- Control dependency
  - Instruction sets PC in way that pipeline did not predict correctly
  - Mispredicted branch and return

# The problem is hazards

*Make the pipelined processor work!*

## Data Hazards

- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

## Control Hazards

- Mispredict conditional branch
  - Our design predicts all branches as being taken
  - Naïve pipeline executes two extra instructions
- Getting return address for `ret` instruction
  - Naïve pipeline executes three extra instructions

## Making Sure It Really Works

- What if multiple special cases happen simultaneously?

# How do we fix the Pipeline?

## Pad the program with NOPs

- Yuck!

## Stall the pipeline

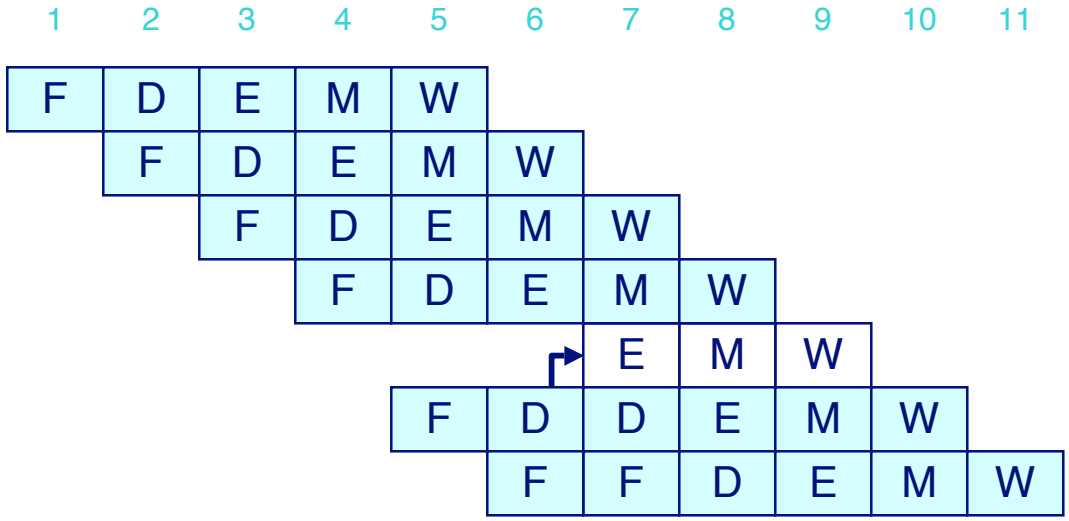
- Data hazards
  - Wait for producing instruction to complete
  - Then proceed with consuming instruction
- Control hazards
  - Wait until new PC has been determined
  - Then begin fetching

## Forward data within the pipeline

- Grab the result from somewhere in the pipe
  - After it has been computed
  - But before it has been written back

# Stalling for Data Dependencies

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```




- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

# Stall Condition

## Source Registers

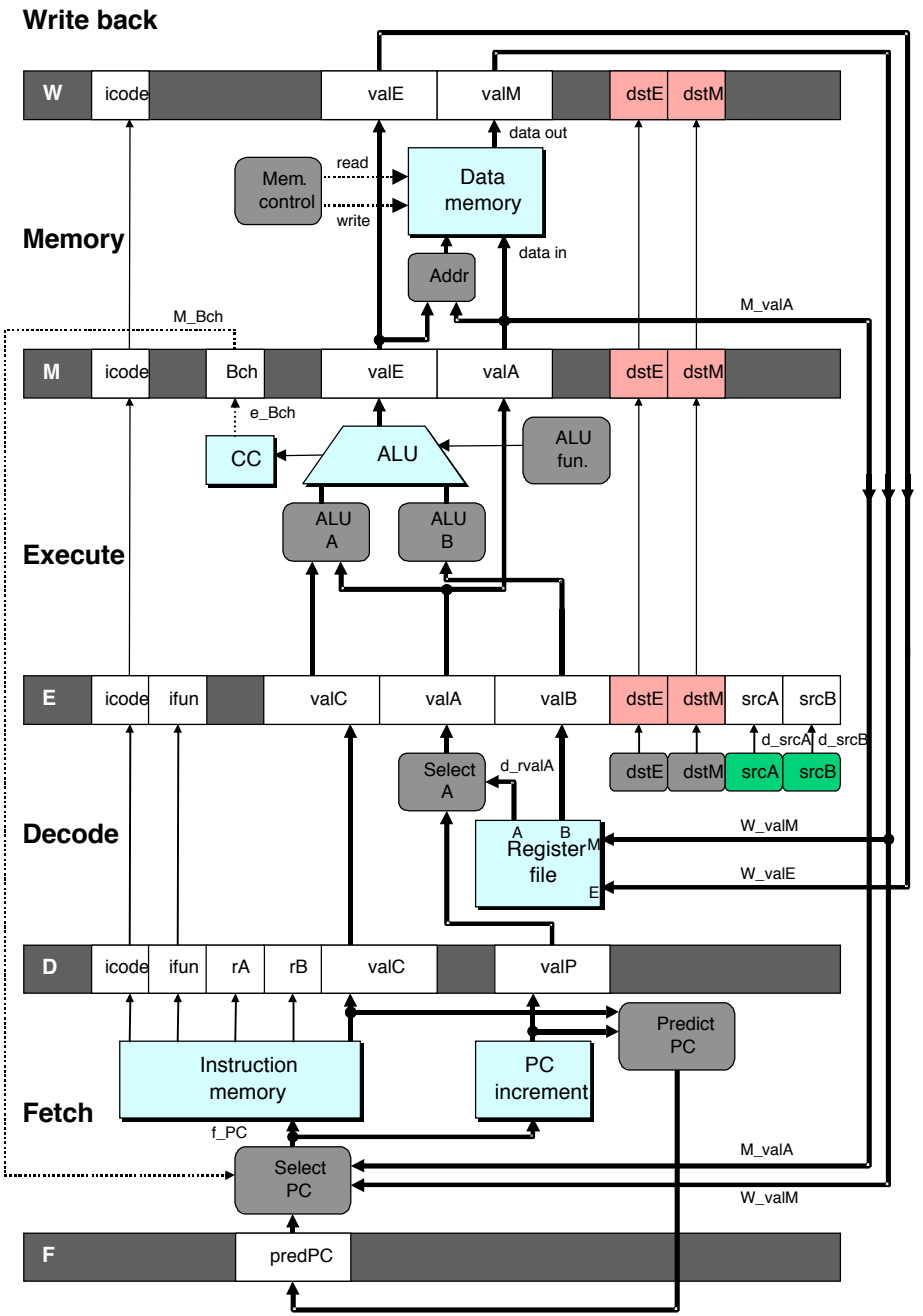
- srcA and srcB of current instruction in decode stage

## Destination Registers

- dstE and dstM fields
- Instructions in execute, memory, and write-back stages

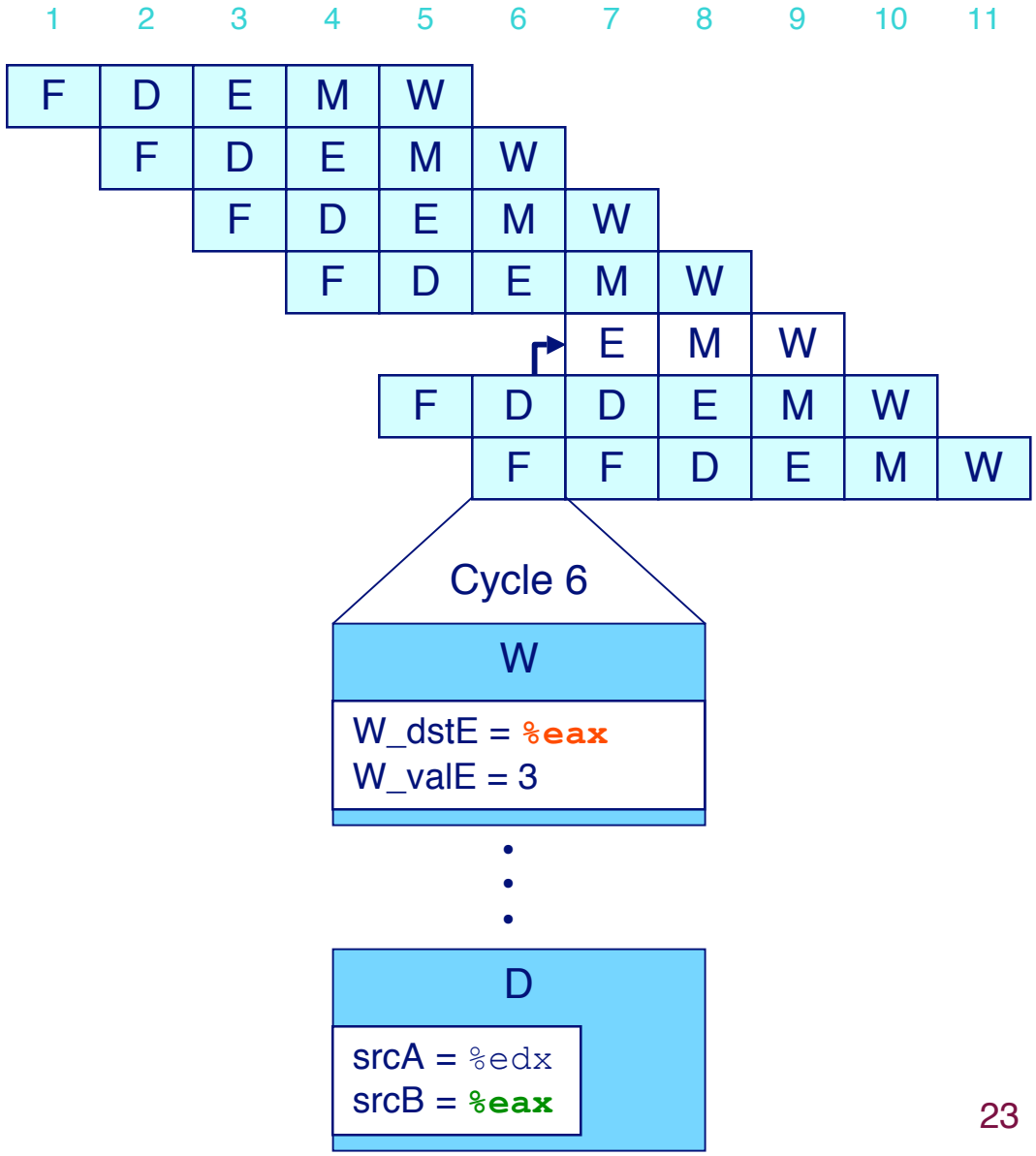
## Special Case

- Don't stall for register ID 8
  - Indicates absence of register operand



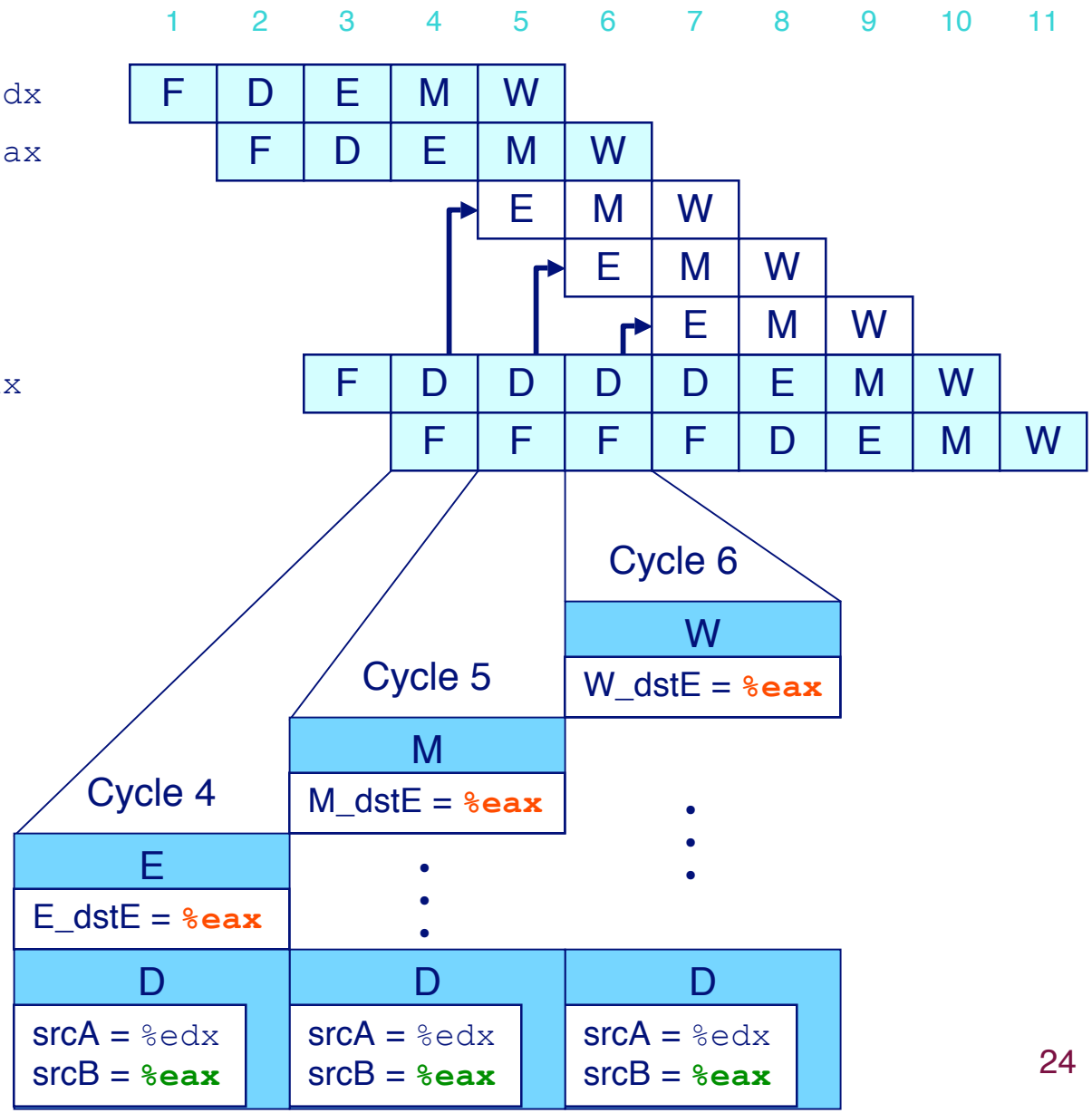
# Detecting Stall Condition

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```



# Stalling X3

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```



# What Happens When Stalling?

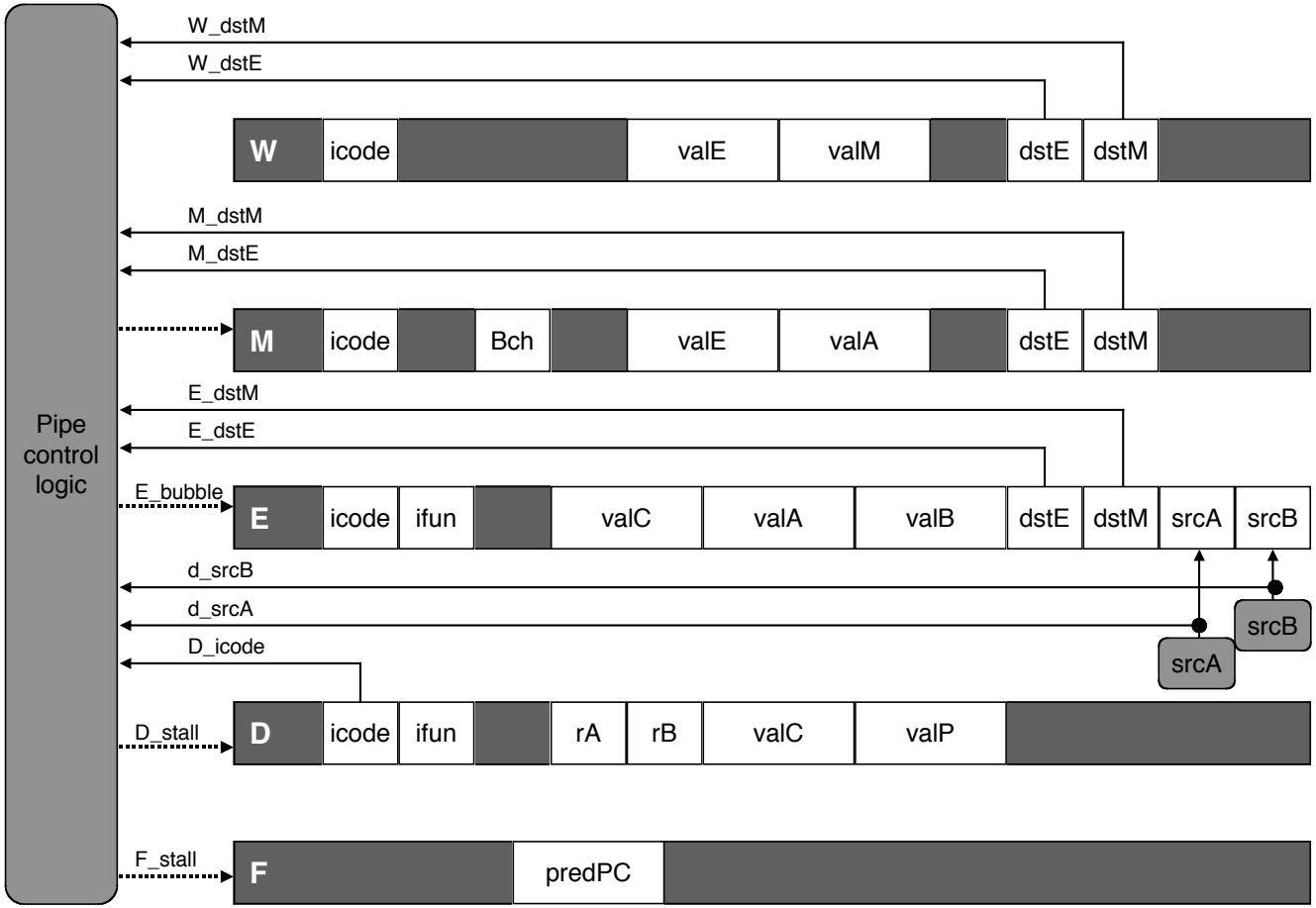
```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

Cycle 8

| | |
|---|---|
| Write Back | ***bubble*** |
| Memory | ***bubble*** |
| Execute | `0x00c: addl %edx,%eax` |
| Decode | `0x00e: halt` |
| Fetch | |

- **Stalling instruction held back in decode stage**
- **Following instruction stays in fetch stage**
- **Bubbles injected into execute stage**
  - **Like dynamically generated nop's**
  - **Move through later stages**

# Implementing Stalling

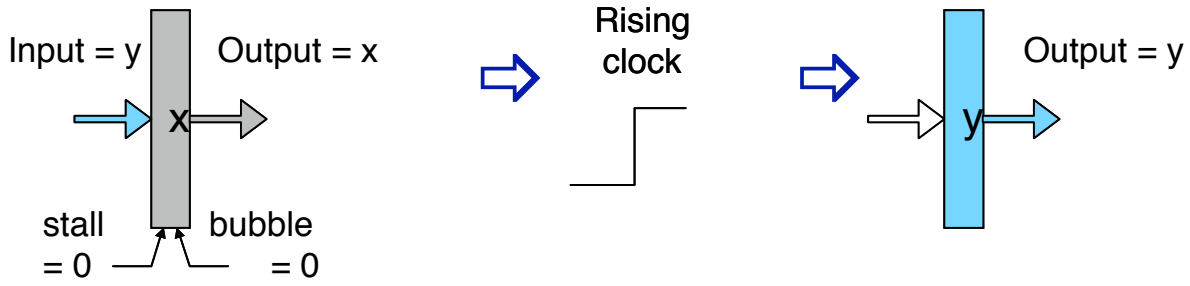



## Pipeline Control

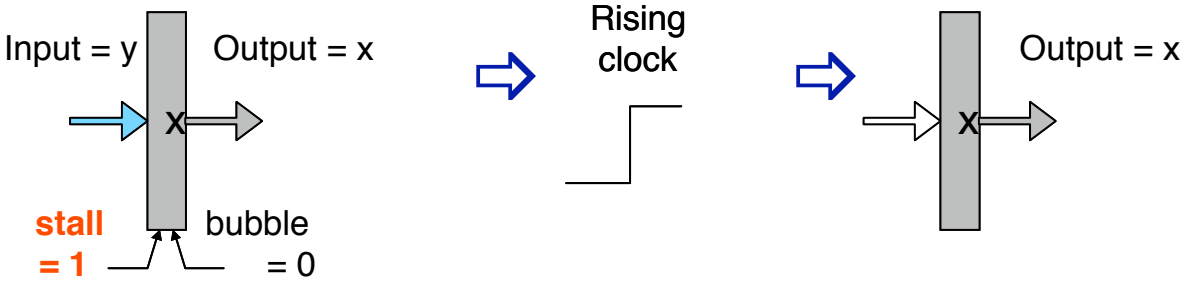
- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

# Pipeline Register Modes

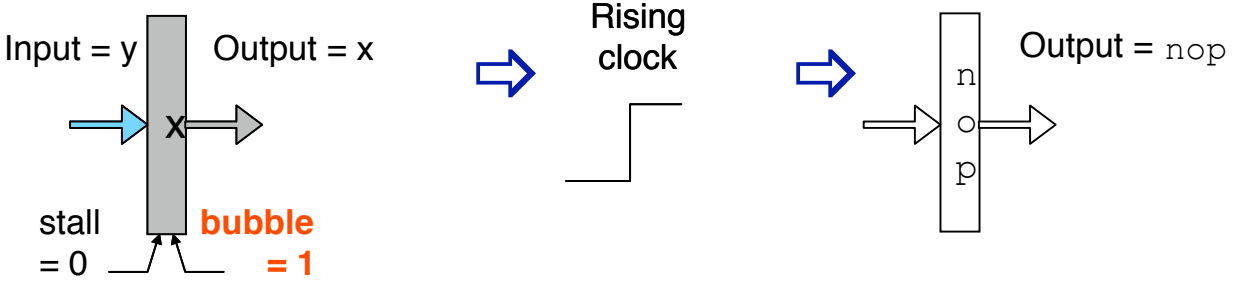
**Normal**




**Stall**




**Bubble**



# Summary

## Today

- Data hazards (read after write)
- Control hazards (branch, return)
- Mitigating hazards through stalling

## Next Time

- Hazard mitigation through pipeline forwarding
- Hardware support for forwarding
- Forwarding to mitigate control (branch) hazards