# Pipelined Implementation (2)

Lecture 12
November 9th, 2017

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: [CS:APP3e] slides from CMU; [COD5e] slides from Elsevier Inc.*

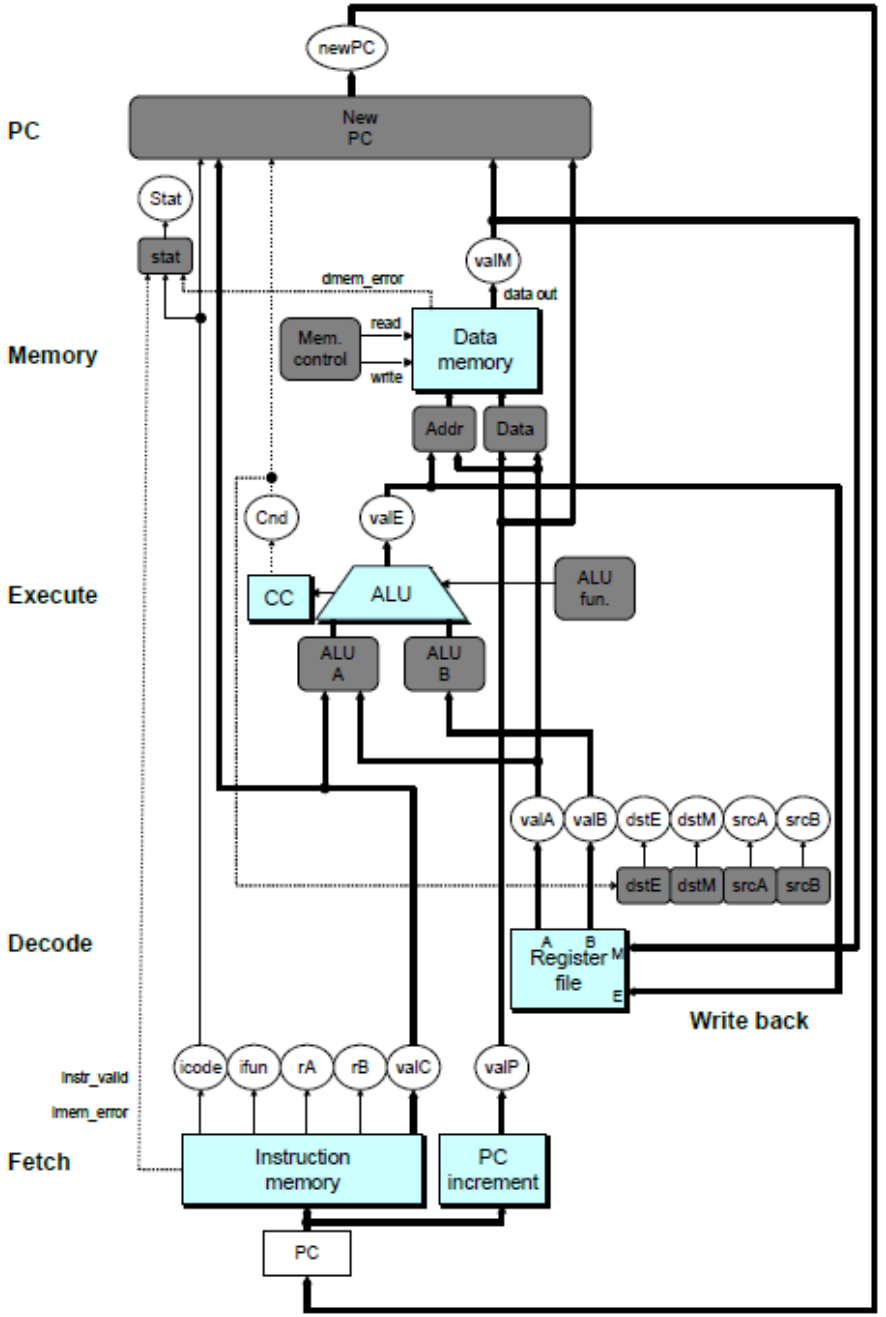
# Today

**Textbook: [CS:APP3e] 4.5.1~4.5.5**

- **SEQ+: Rearranging the Computation Stages**
- **Inserting Pipeline Registers**
- **Rearranging and Relabeling Signals**
- **Next PC Prediction**
- **Pipeline Hazards**

# SEQ Hardware

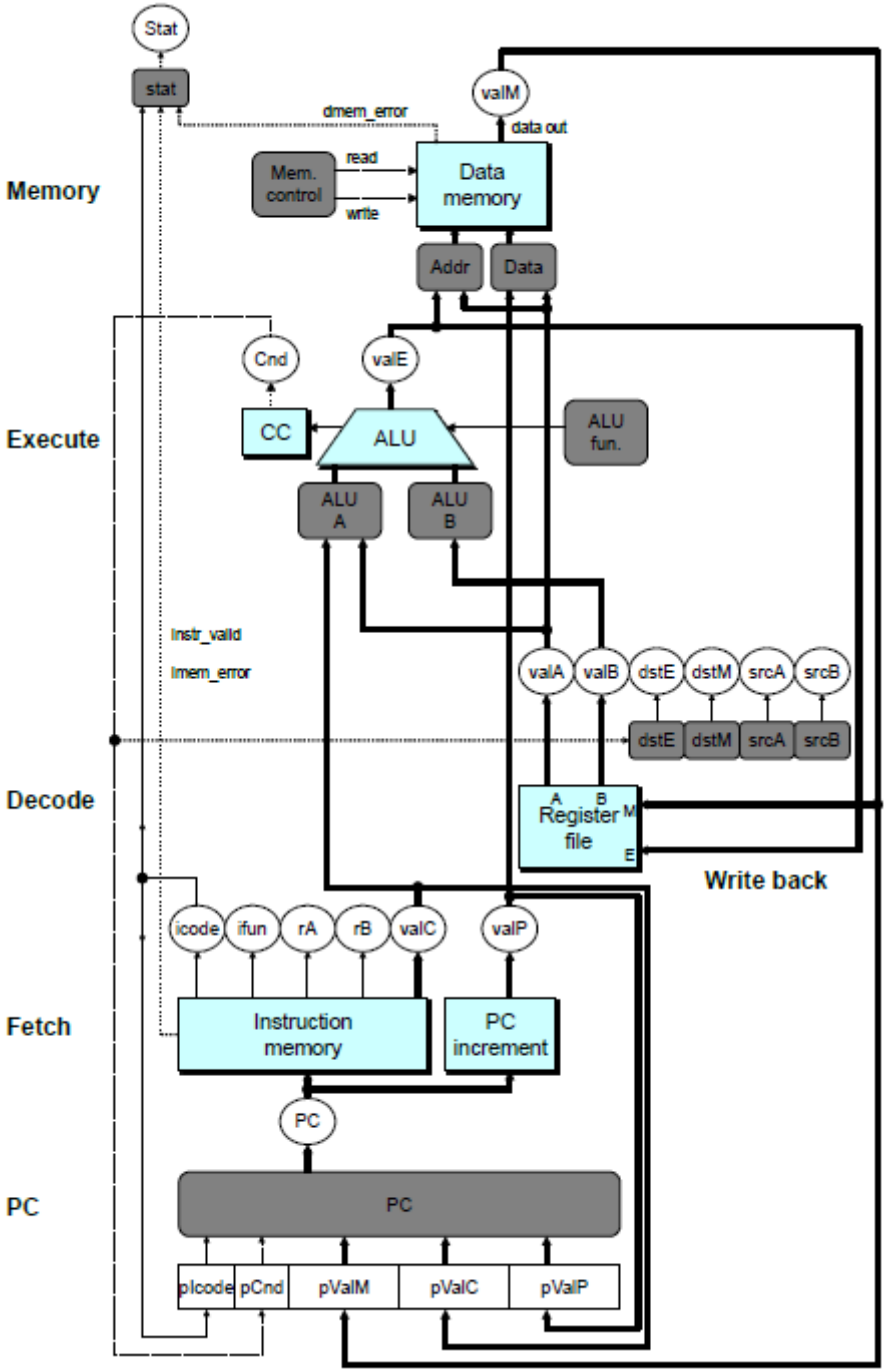
- Stages occur in sequence
- One operation in process at a time



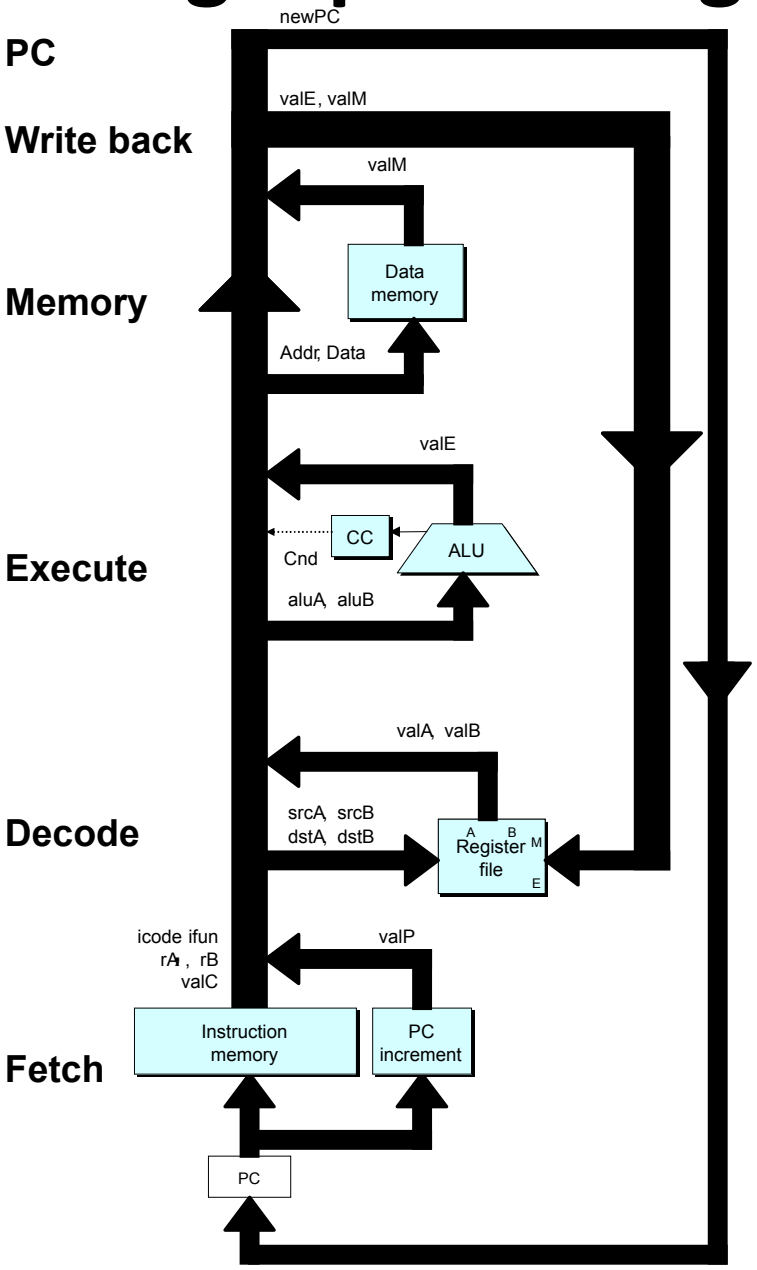
# SEQ+ Hardware

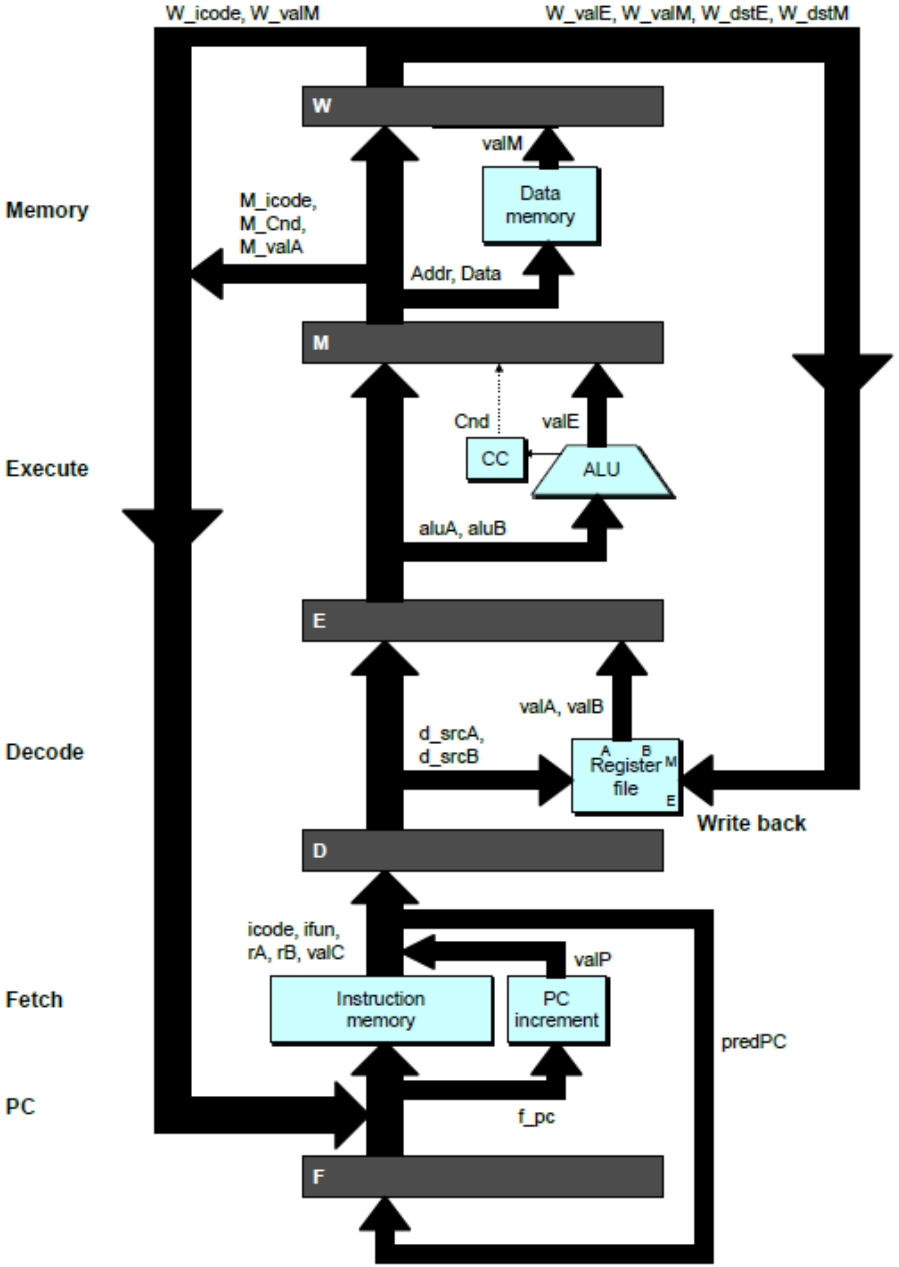
- Still sequential implementation
- Reorder PC stage to put at beginning

- **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



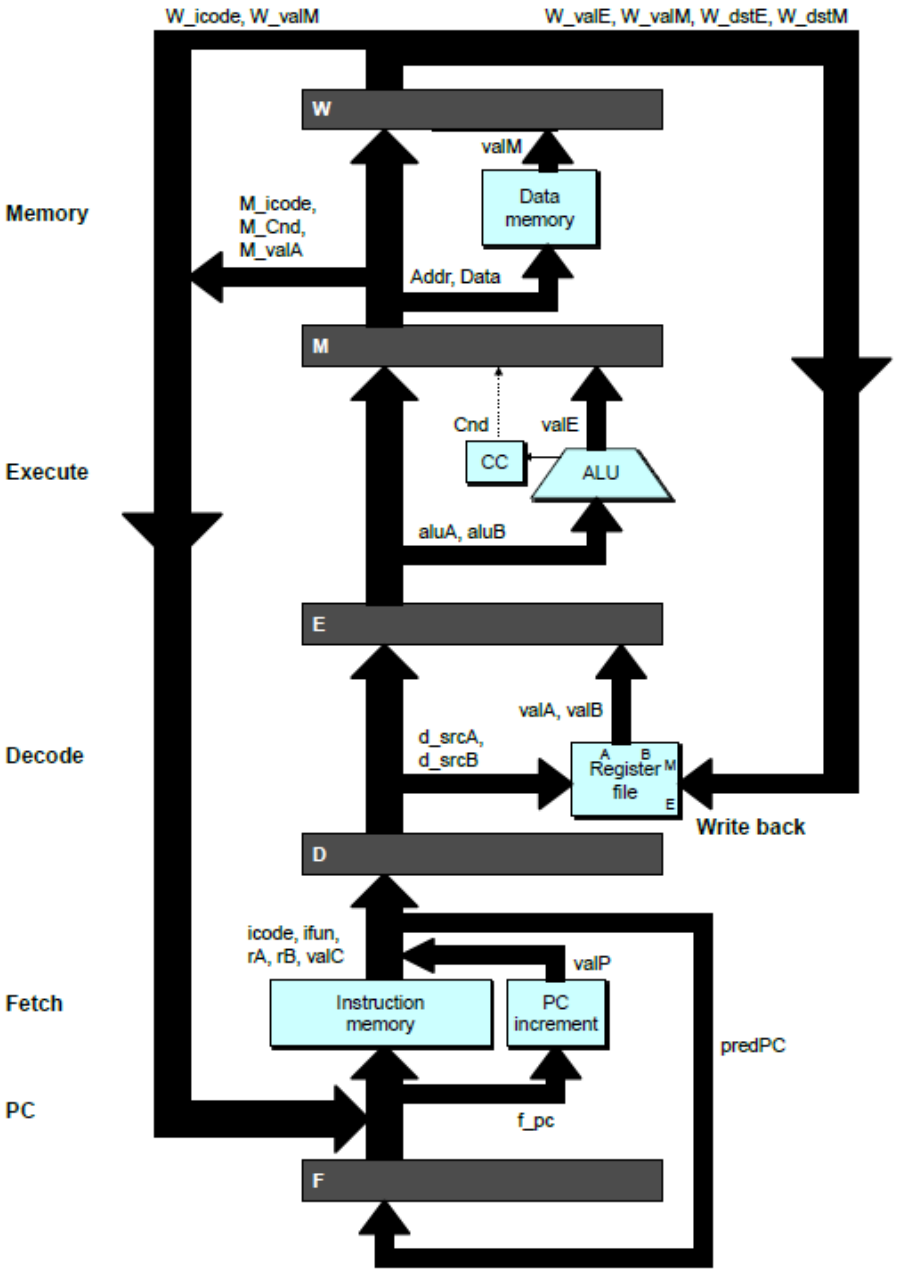
# Adding Pipeline Registers



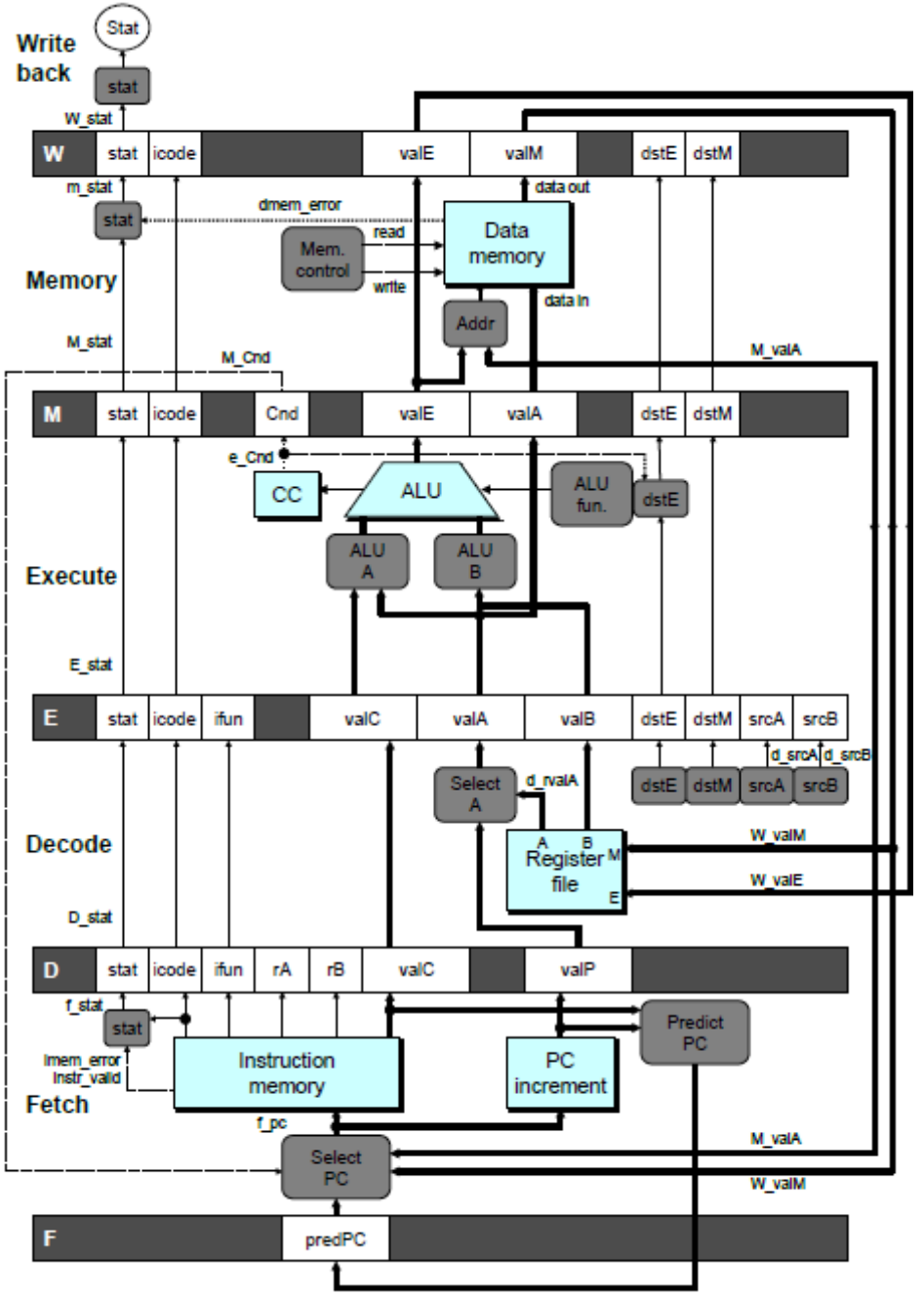
# Pipeline Stages

- **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



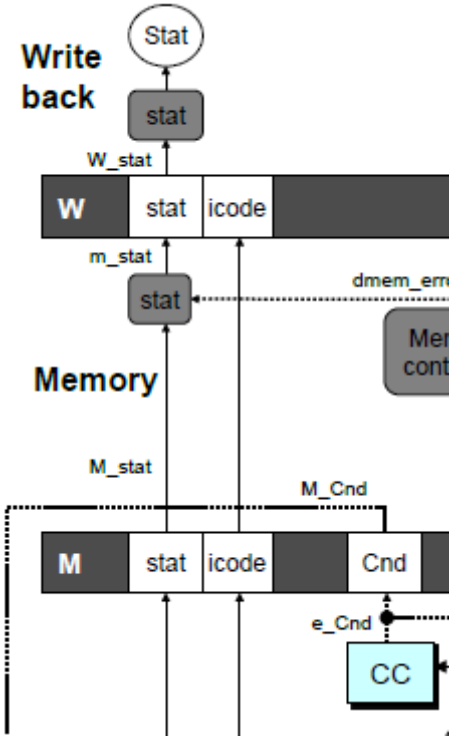
# PIPE- Hardware

- 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



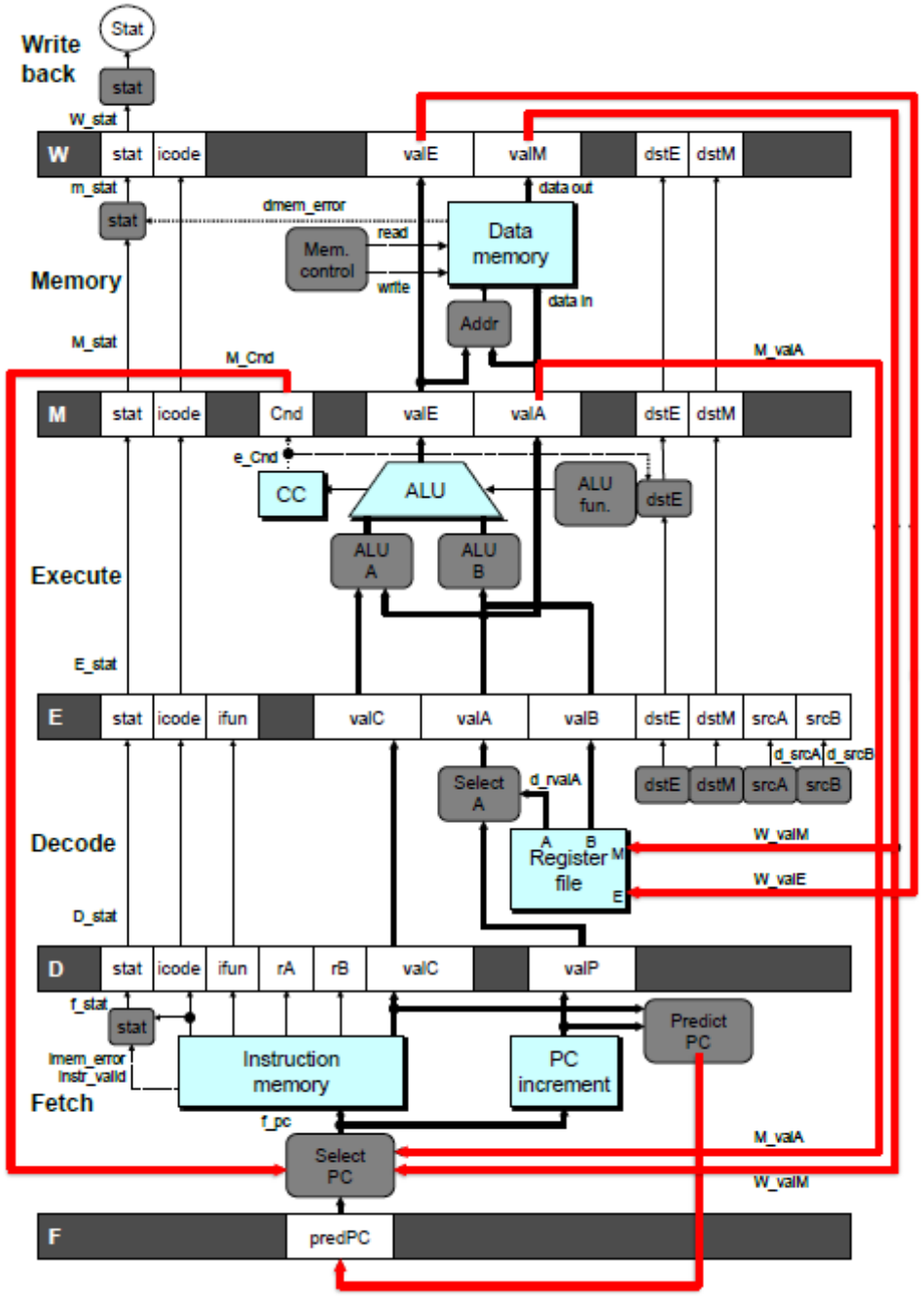
# Signal Naming Conventions

- **S_Field**
  - Value of Field held in stage S pipeline register
- **s_Field**
  - Value of Field computed in stage S

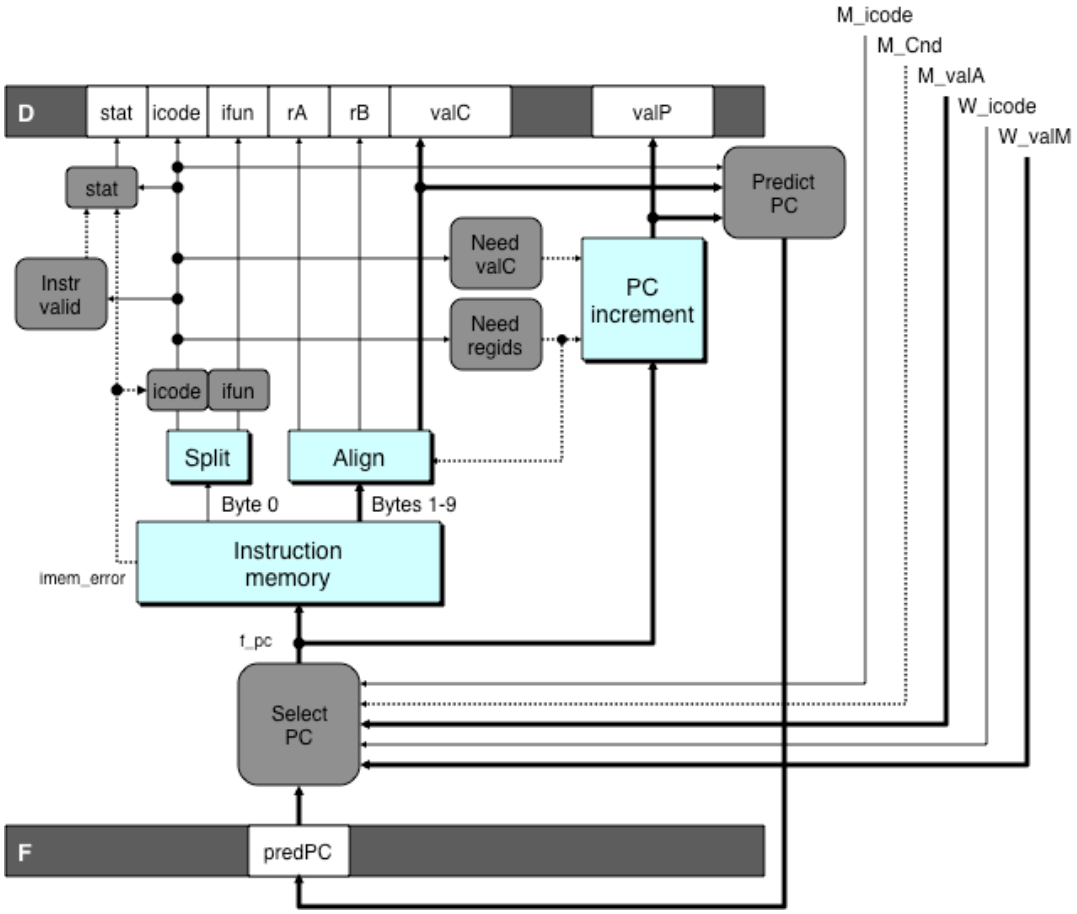


# Feedback Paths

- **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



# Predicting the PC

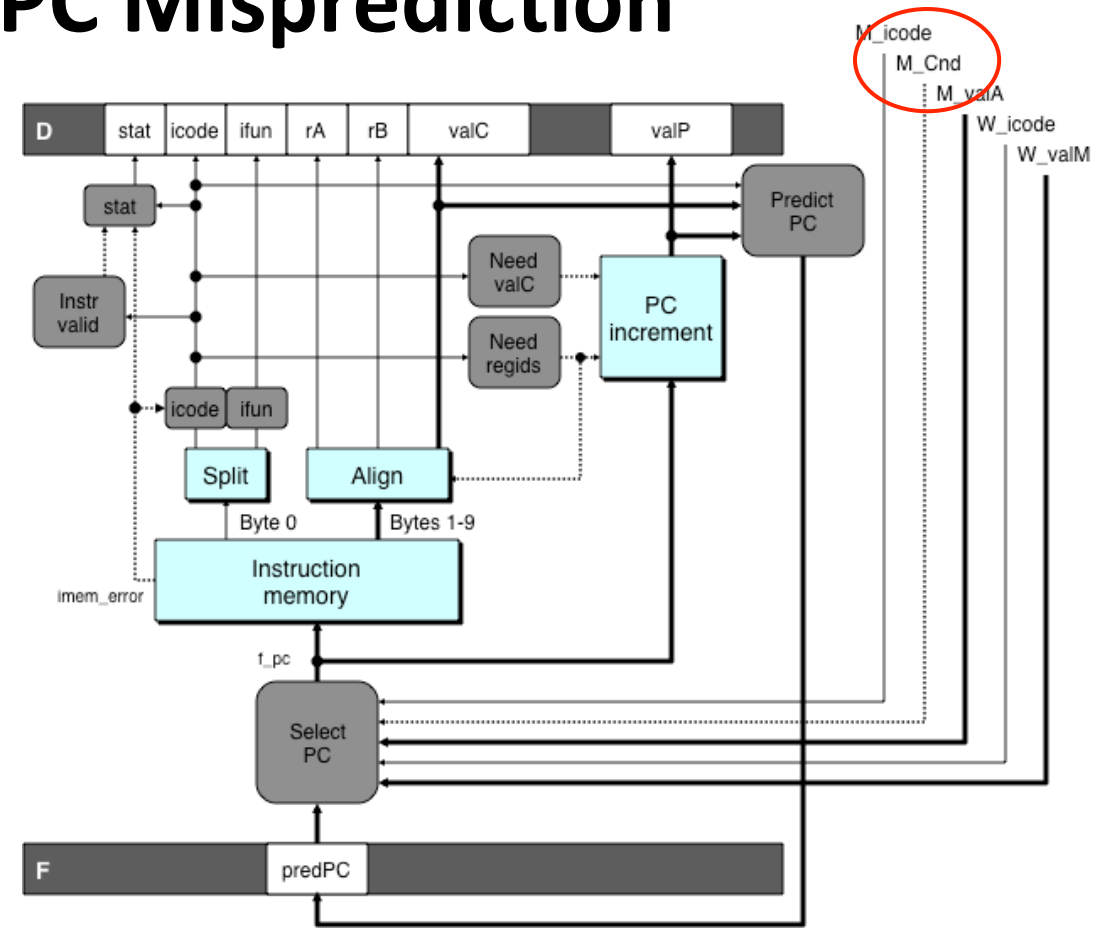



- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

# 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

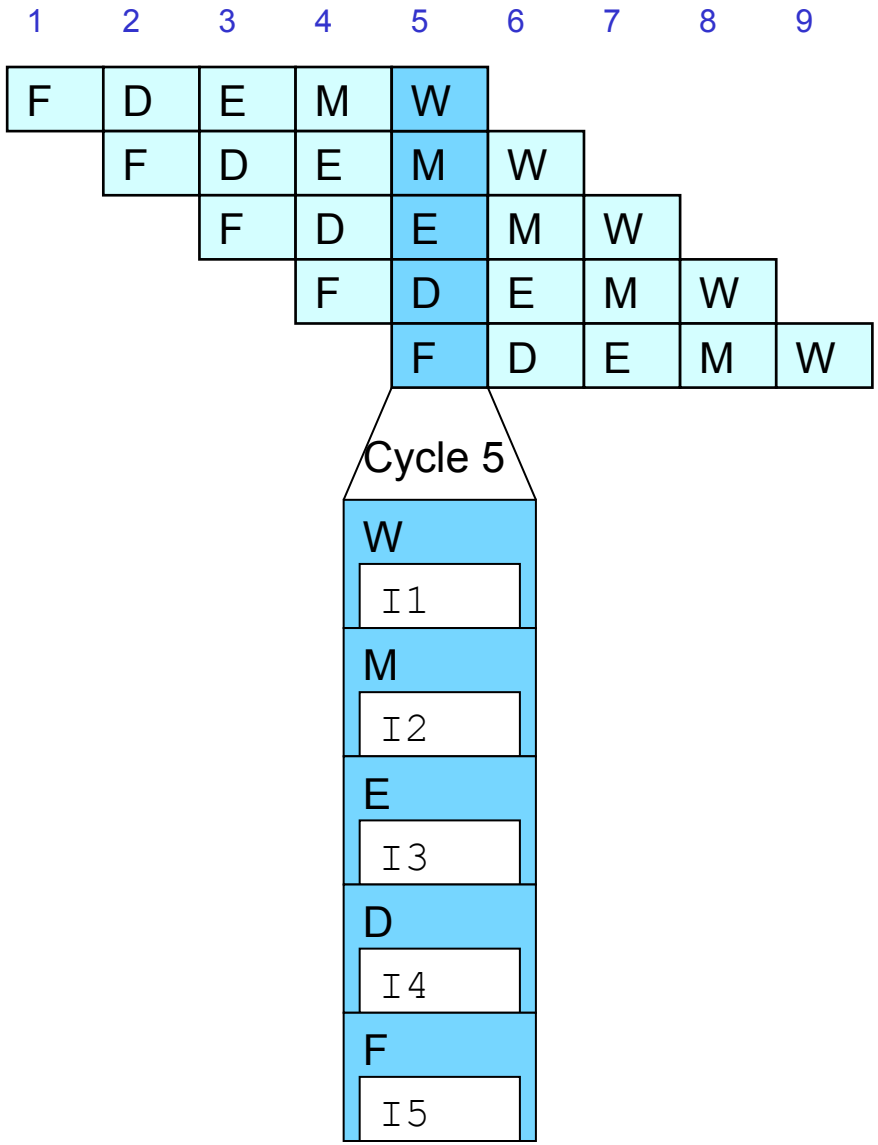
# Recovering from PC Misprediction




- Mispredicted Jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from valA (value M_valA)
- Return Instruction
  - Will get return PC when `ret` reaches write-back stage (W_valM)

# Pipeline Demonstration

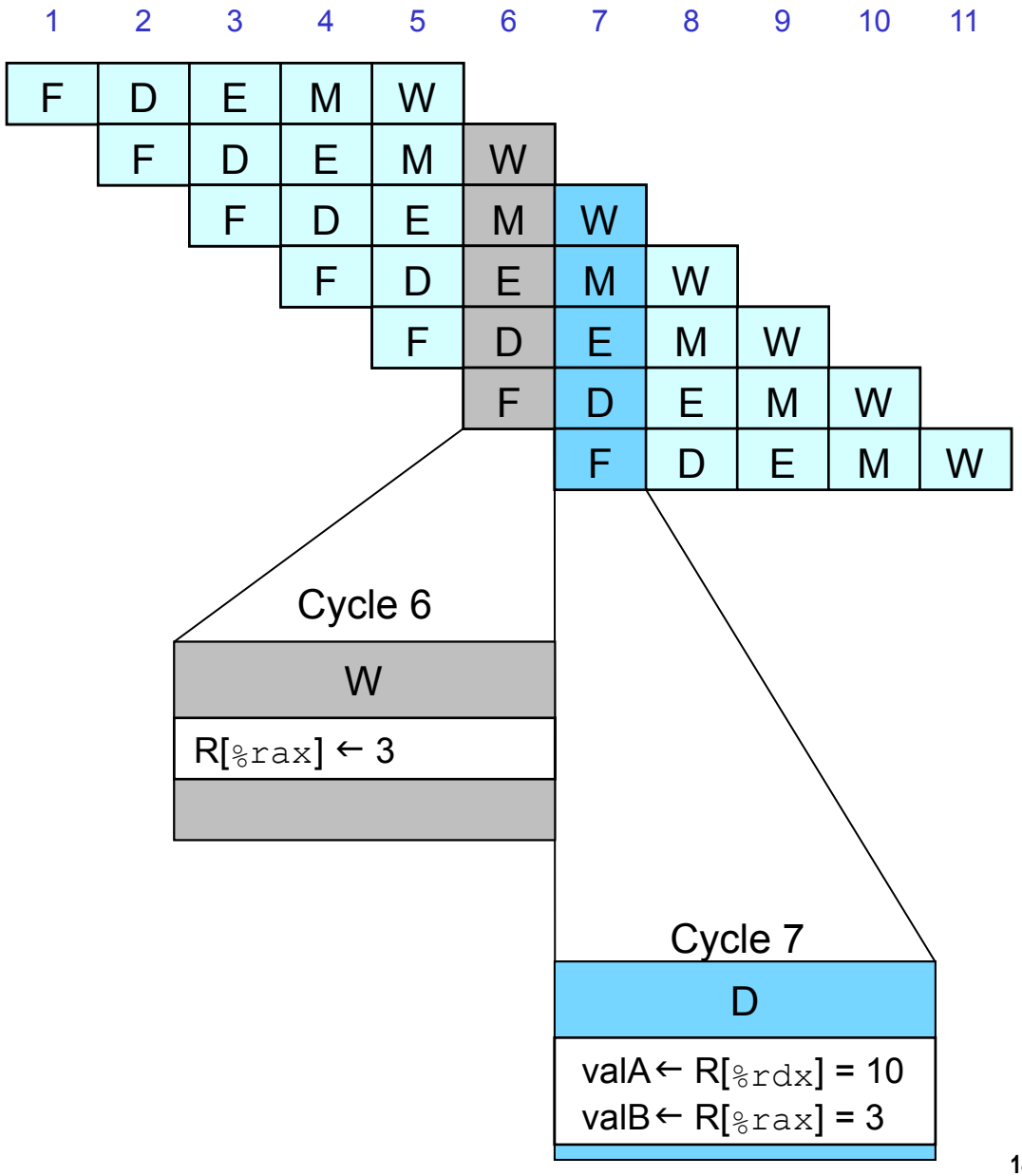
```
irmovq    $1,%rax   #I1
irmovq    $2,%rcx   #I2
irmovq    $3,%rdx   #I3
irmovq    $4,%rbx   #I4
halt                #I5
```




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

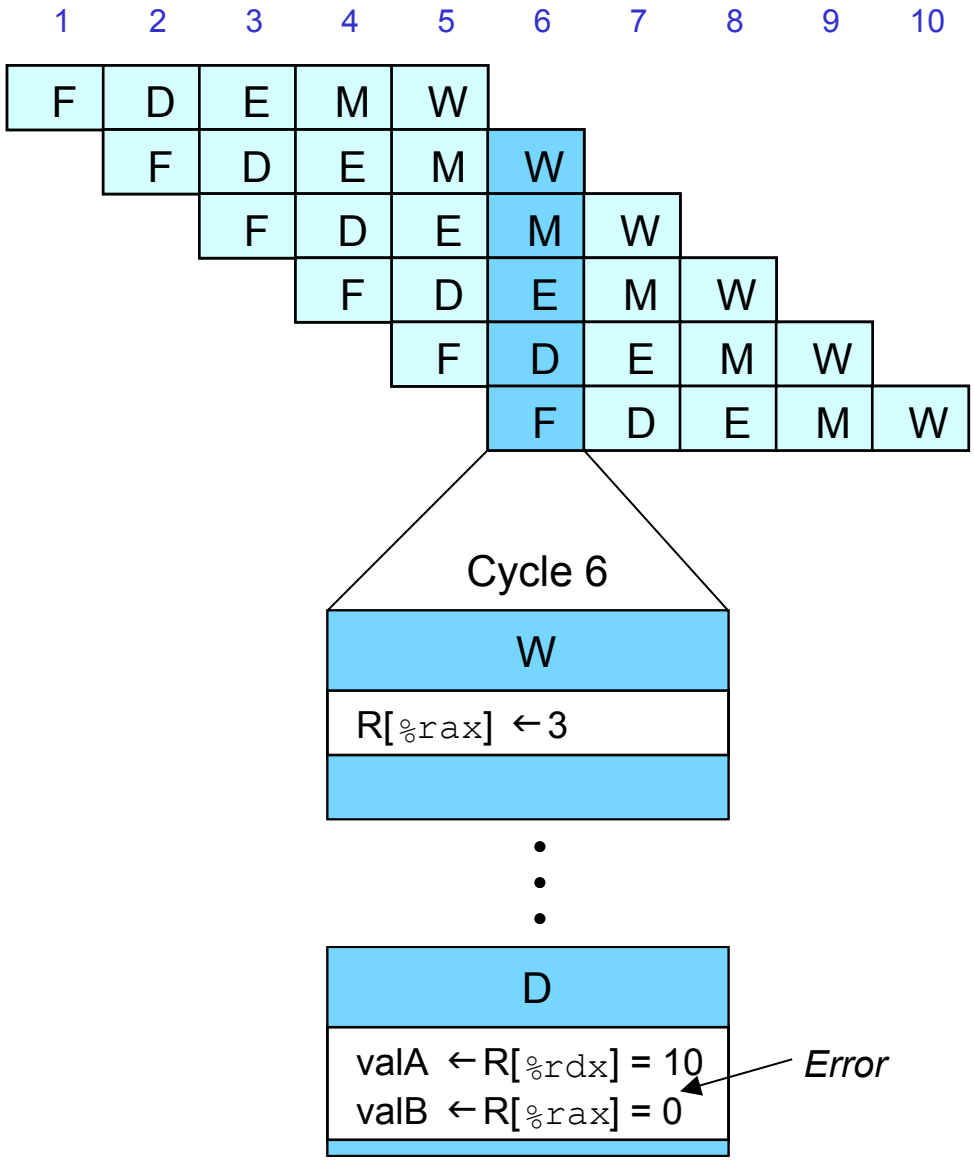
# Data Dependencies: 3 Nop's

```
# demo-h3.ys
0x000:  irmovq$10,%rdx
0x00a:  irmovq $3,%rax
0x014:  nop
0x015:  nop
0x016:  nop
0x017:  addq %rdx,%rax
0x019:  halt
```



# Data Dependencies: 2 Nop's

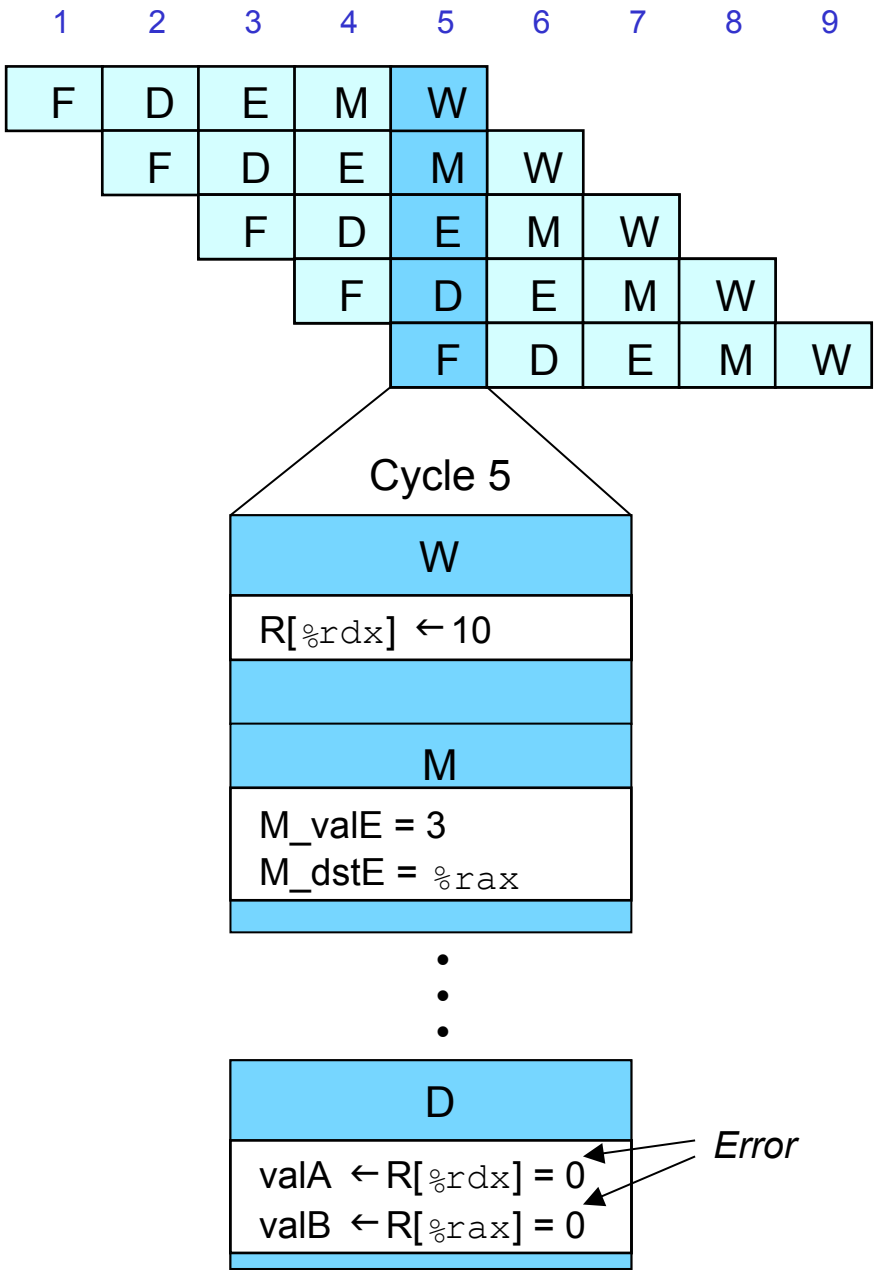
```
# demo-h2.ys
0x000:  irmovq $10,%rdx
0x00a:  irmovq  $3,%rax
0x014:  nop
0x015:  nop
0x016:  addq %rdx,%rax
0x018:  halt
```



# Data Dependencies: 1 Nop

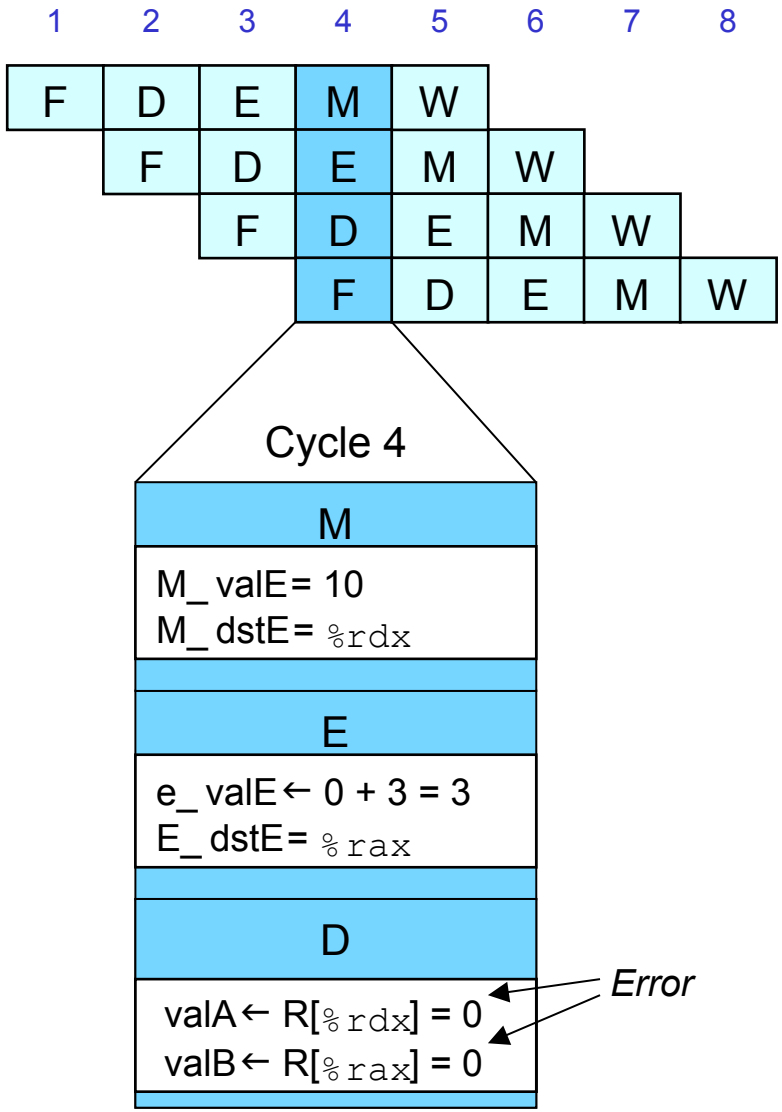
```
# demo-h1.ys

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



# Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



# Branch Misprediction Example

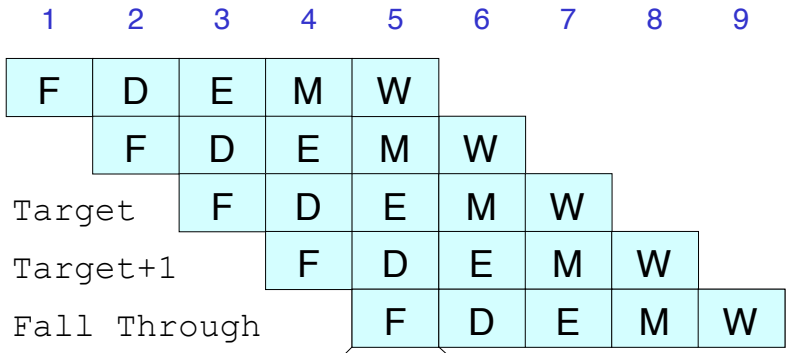
**demo-j.ys**

```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target (Should not execute)
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

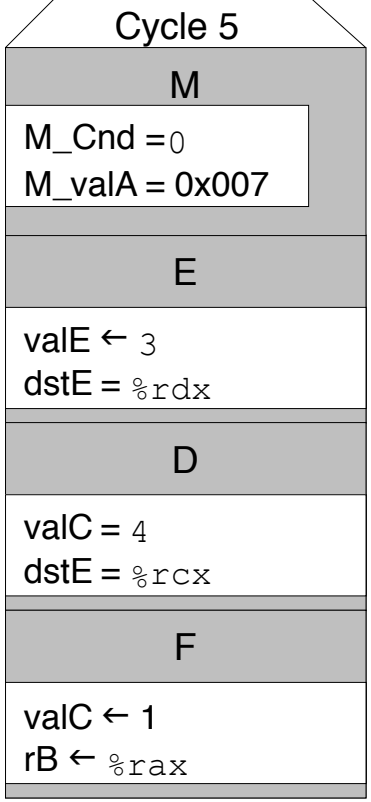
- Should only execute first 8 instructions

# Branch Misprediction Trace

```
# demo-j
0x000:    xorq %rax,%rax
0x002:    jne t # Not taken
0x019: t: irmovq $3, %rdx # Target
0x023:    irmovq $4, %rcx # Target+1
0x00b:    irmovq $1, %rax # Fall Through
```




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



# Return Example

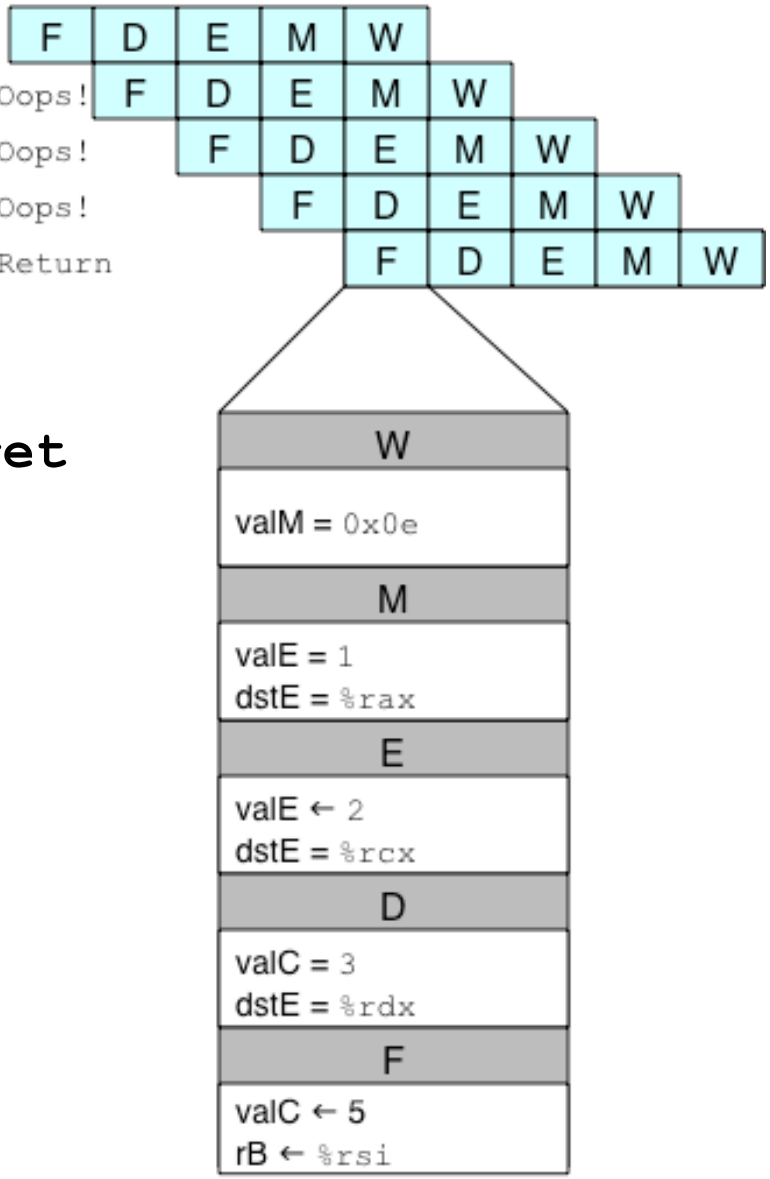
demo-ret.ys

```
0x000:     irmovq Stack,%rsp    # Intialize stack pointer
0x00a:     nop                  # Avoid hazard on %rsp
0x00b:     nop
0x00c:     nop
0x00d:     call p               # Procedure call
0x016:     irmovq $5,%rsi       # Return point
0x020:     halt
0x020:  .pos 0x20
0x020:  p: nop                   # procedure
0x021:     nop
0x022:     nop
0x023:     ret
0x024:     irmovq $1,%rax        # Should not be executed
0x02e:     irmovq $2,%rcx        # Should not be executed
0x038:     irmovq $3,%rdx        # Should not be executed
0x042:     irmovq $4,%rbx        # Should not be executed
0x100:  .pos 0x100
0x100:  Stack:                   # Initial stack pointer
```

- Require lots of nops to avoid data hazards

# Incorrect Return Example

```
# demo-ret
0x033:     ret
0x034:     irmovq $1,%rax # Oops!
0x03e:     irmovq $2,%rcx # Oops!
0x048:     irmovq $3,%rdx # Oops!
0x052:     irmovq $5,%rsi # 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
- **Fixing the Pipeline**
  - We'll do that in the next lecture