PIPE-: Basic Pipelined Y86-64 Implementation

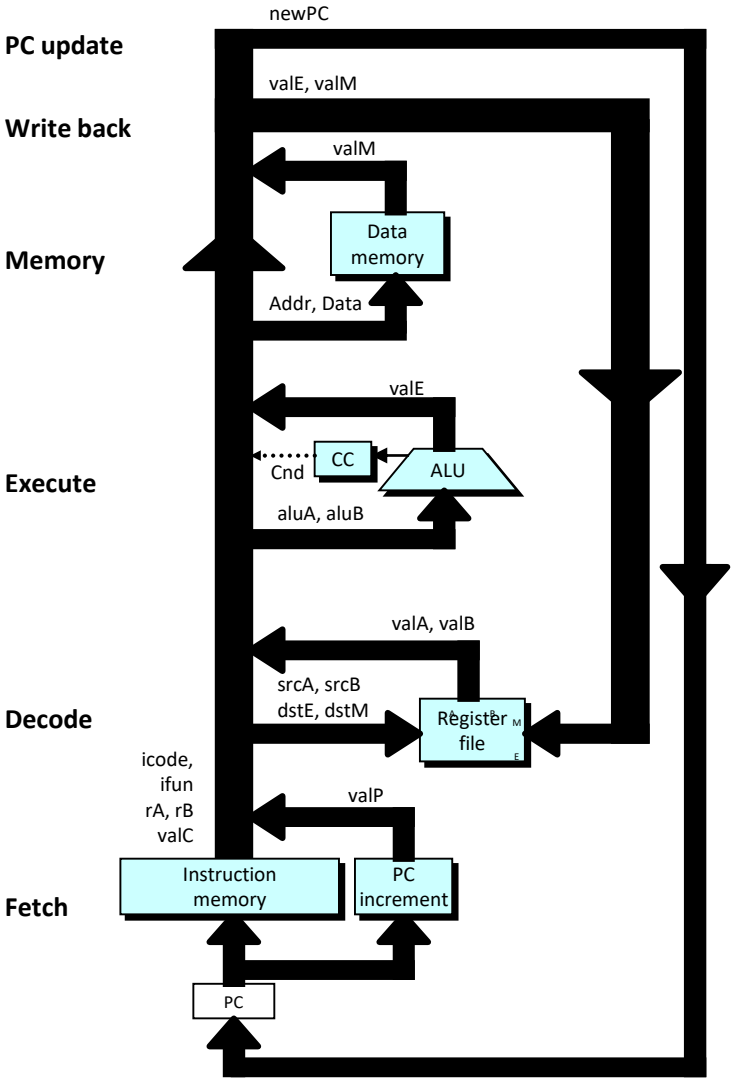
Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

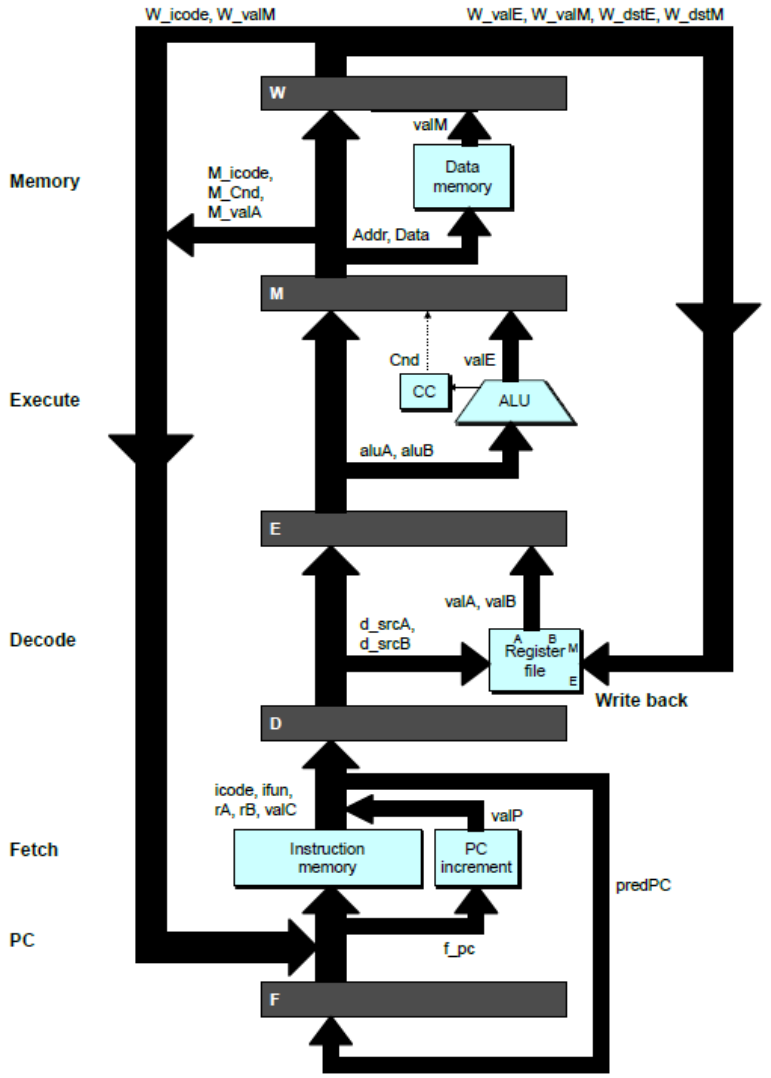
Systems Software & Architecture Lab.

Seoul National University

Fall 2018

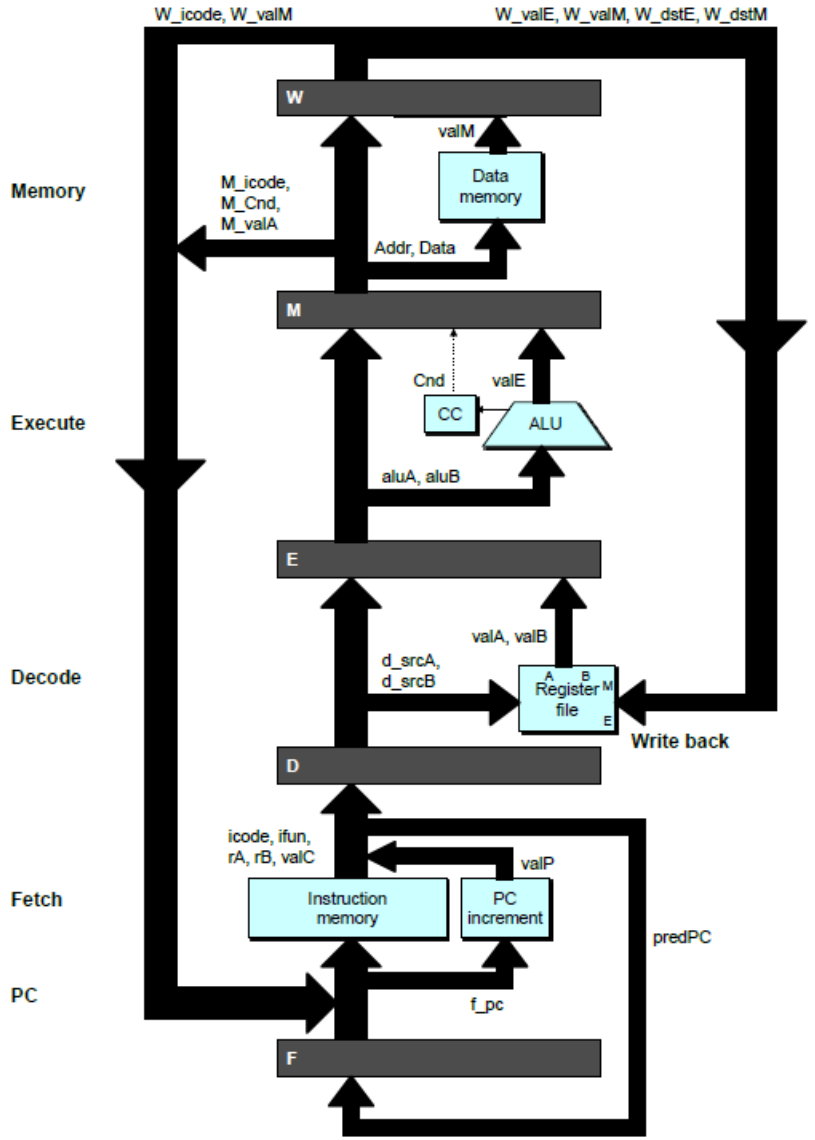
Adding Pipeline Registers





Pipeline Stages

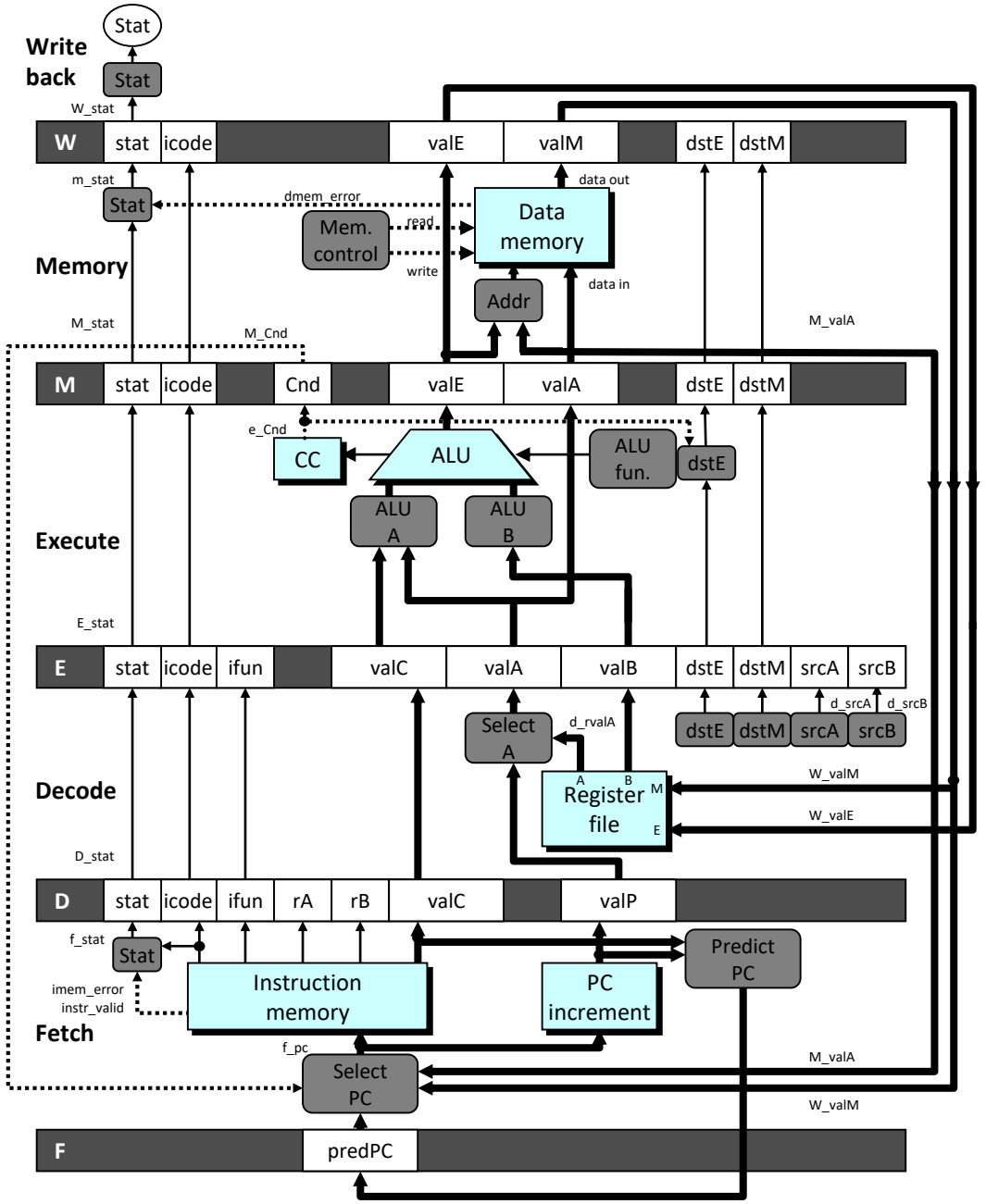
- **Fetch** — Select current PC
Read instruction
Compute incremented PC
- **Decode** — Read program registers
- **Execute** — Operate ALU
- **Memory** — Read or write data memory
- **Write Back** — Update register file



PIPE- Hardware

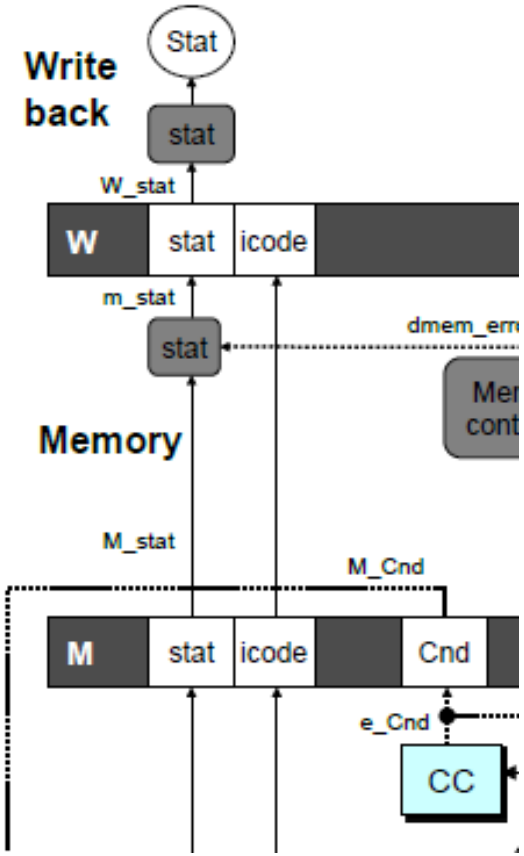
- Pipelined registers hold intermediate values from instruction execution

- Forward (Upward) paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., `valC` passes through decode



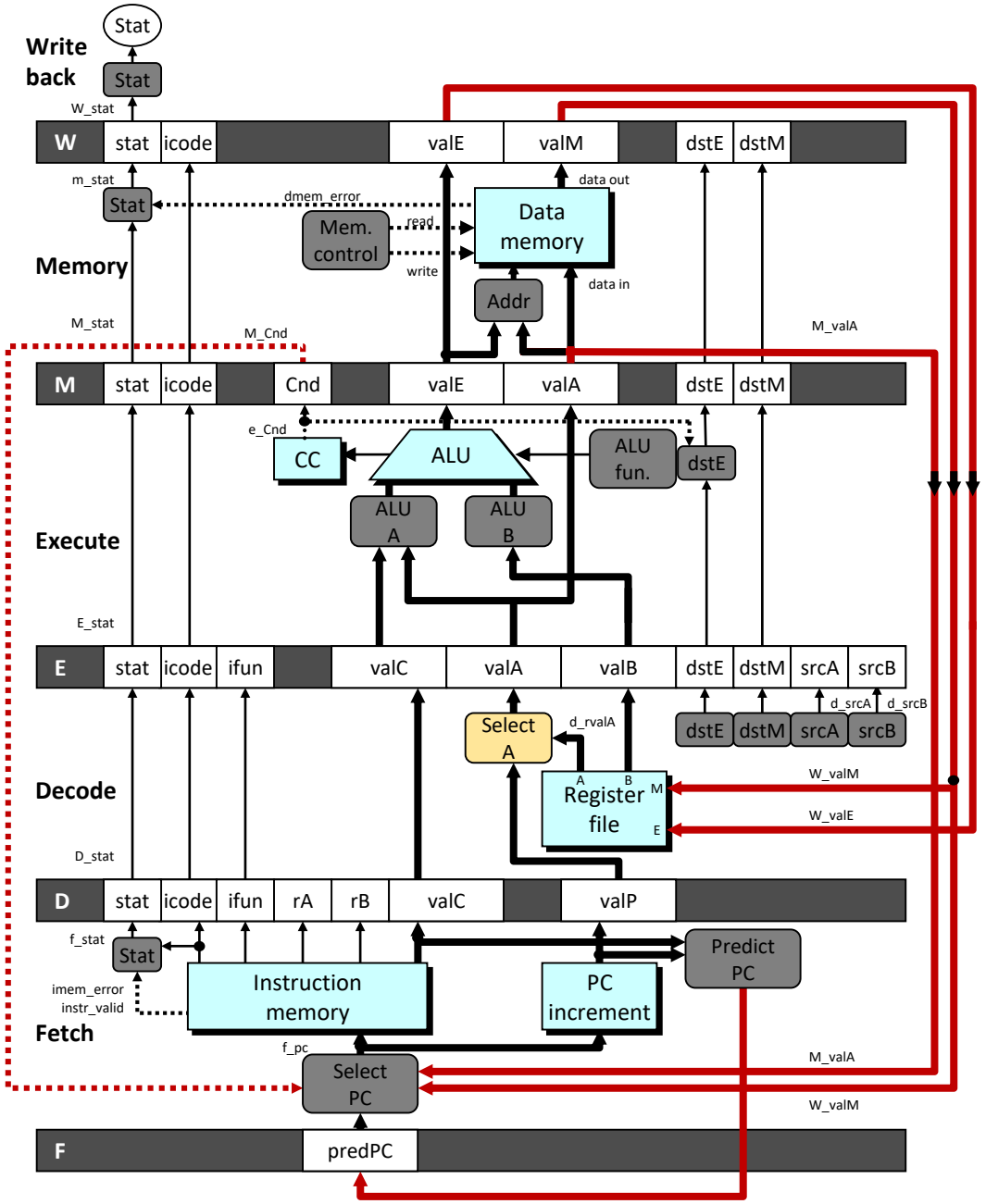
Signal Naming Conventions

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



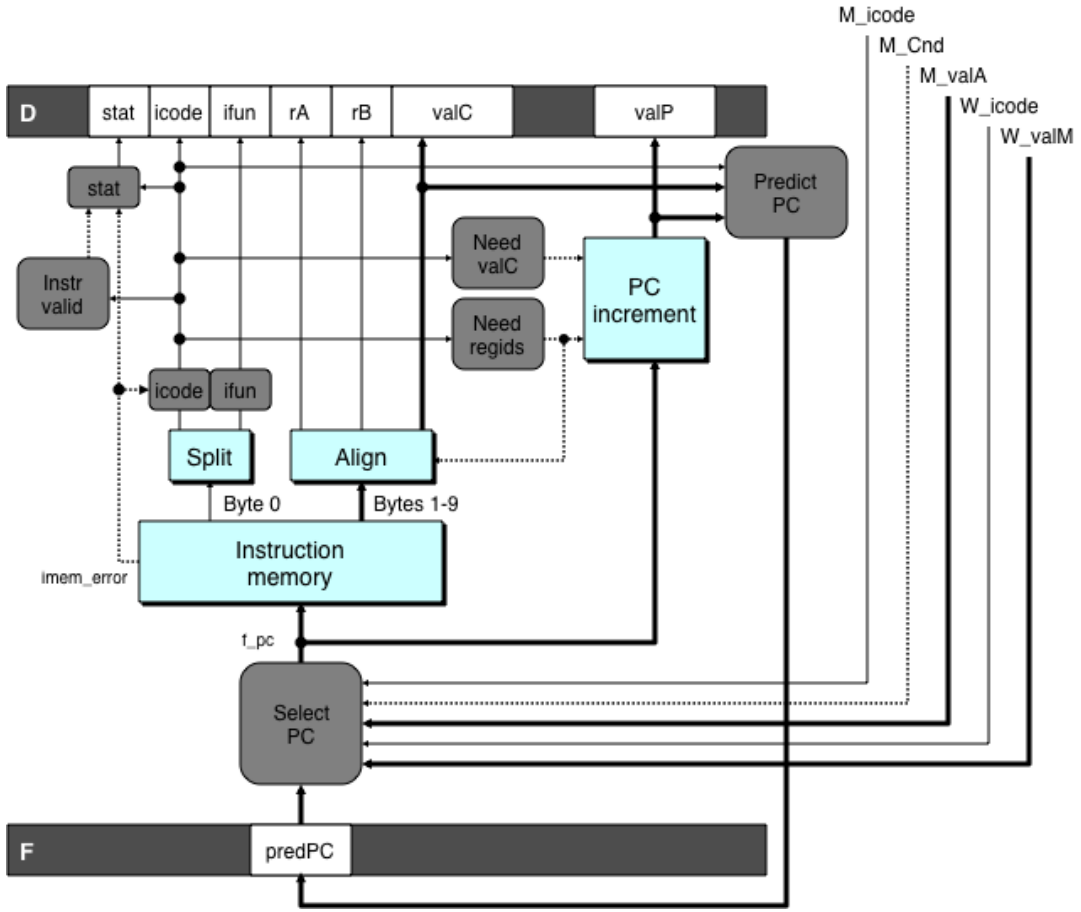
Feedback Paths

- 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

- Goal: issue a new instruction on every clock cycle
- 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



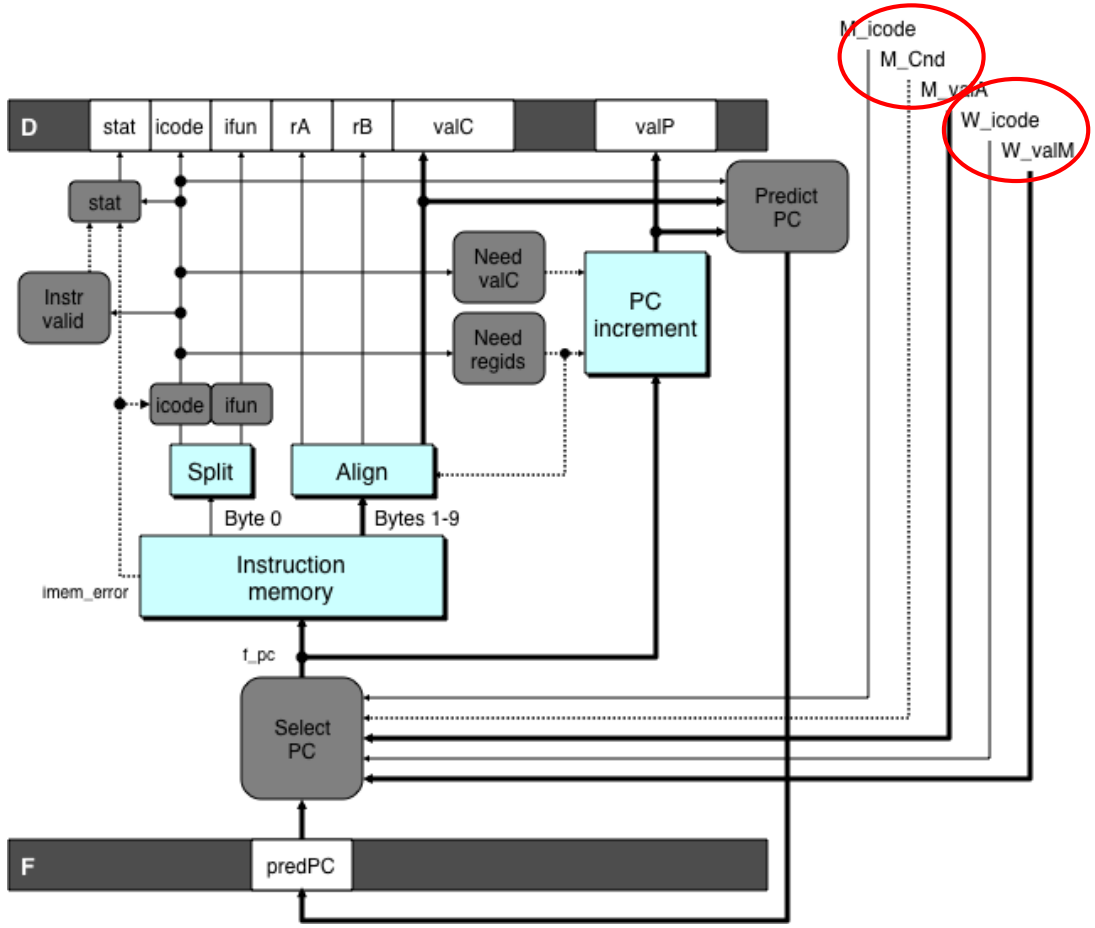
Our Prediction Strategy

- 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 (stall the pipeline)

Recovering from PC Misprediction

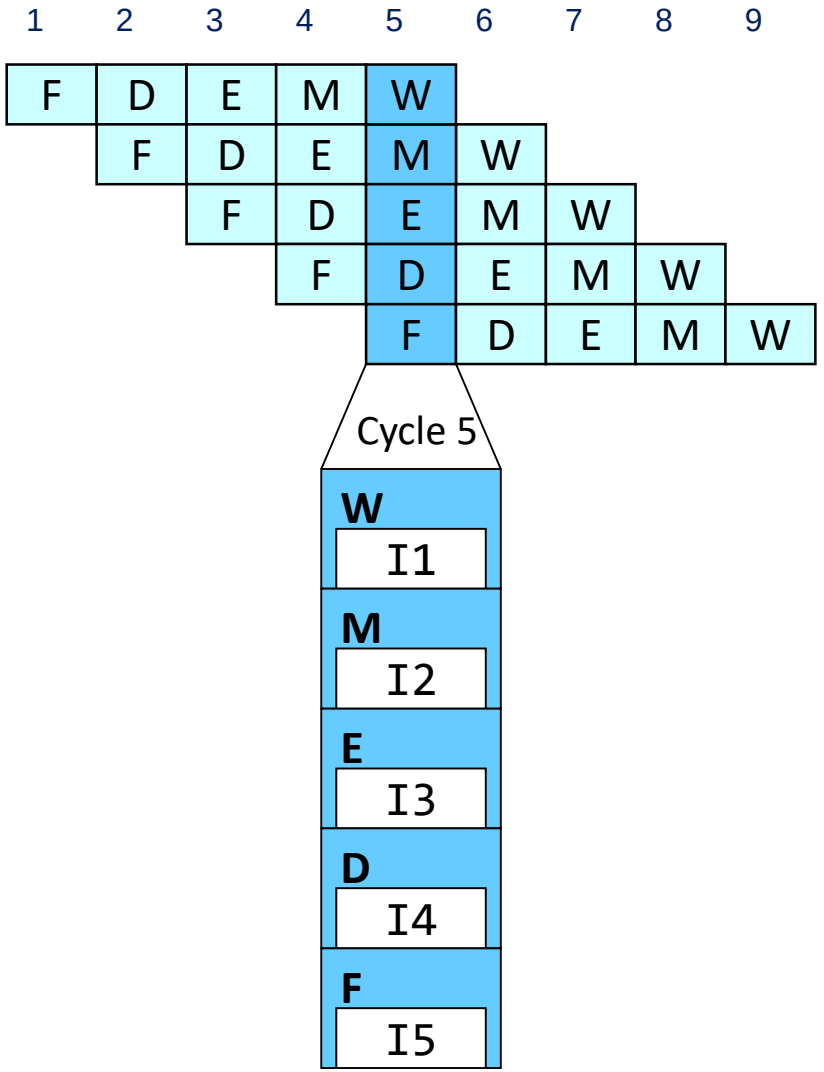
- Mispredicted jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from **`valA`** (value **`M_valA == D_valP`**)

- Return instruction
  - Will get return PC when `ret` reaches write-back stage (**`W_valM`**)



Pipeline Demonstration

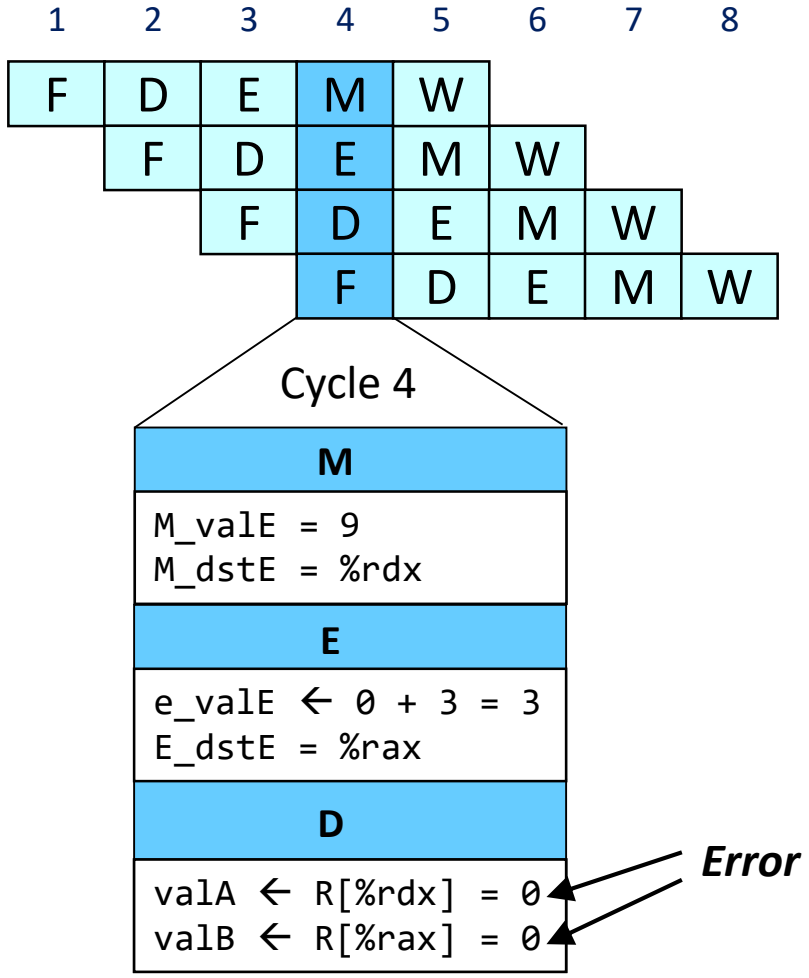
```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



Example: Data Hazard (1)

- No nop

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   addq     %rdx,%rax
0x016:   halt
```

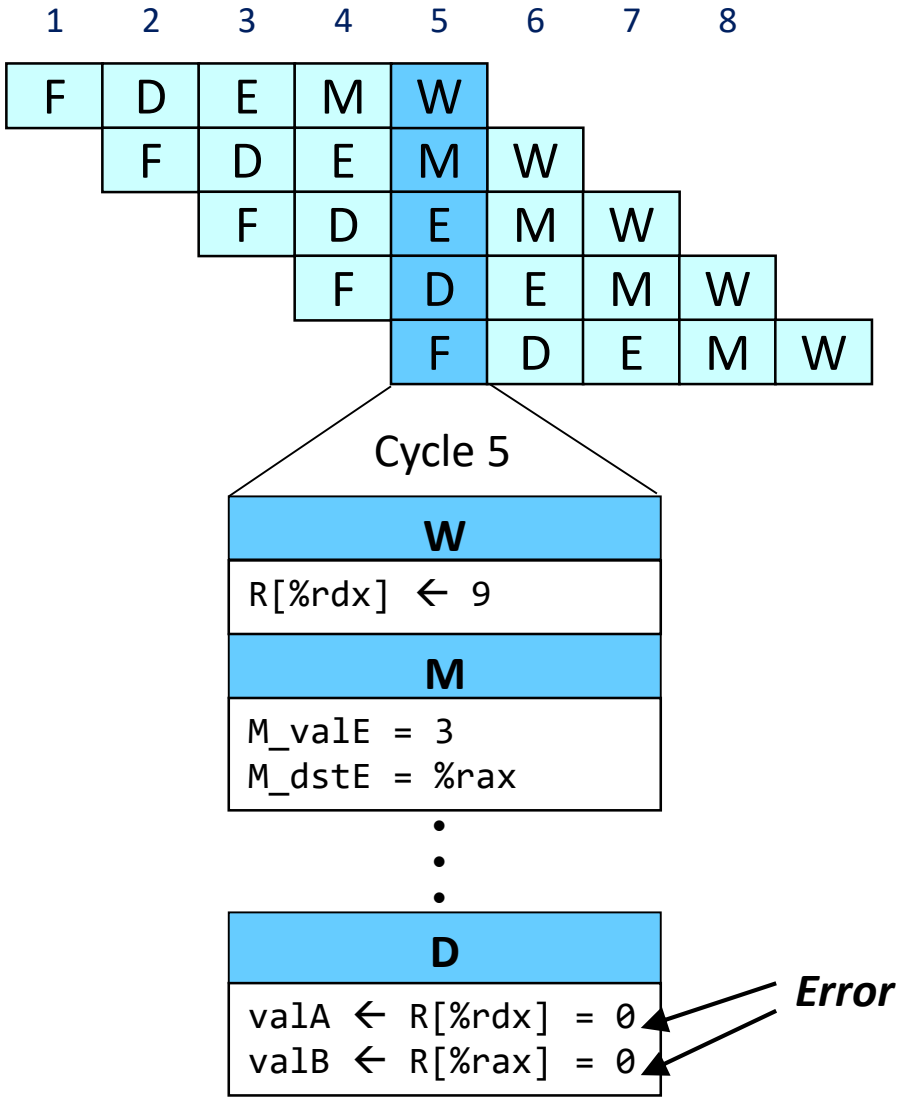


(Both %rax and %rdx are initialized to 0)

Example: Data Hazard (2)

- I nop

```
0x000:   irmovq    $9,%rdx
0x00a:   irmovq    $3,%rax
0x014:   nop
0x015:   addq      %rdx,%rax
0x017:   halt
```

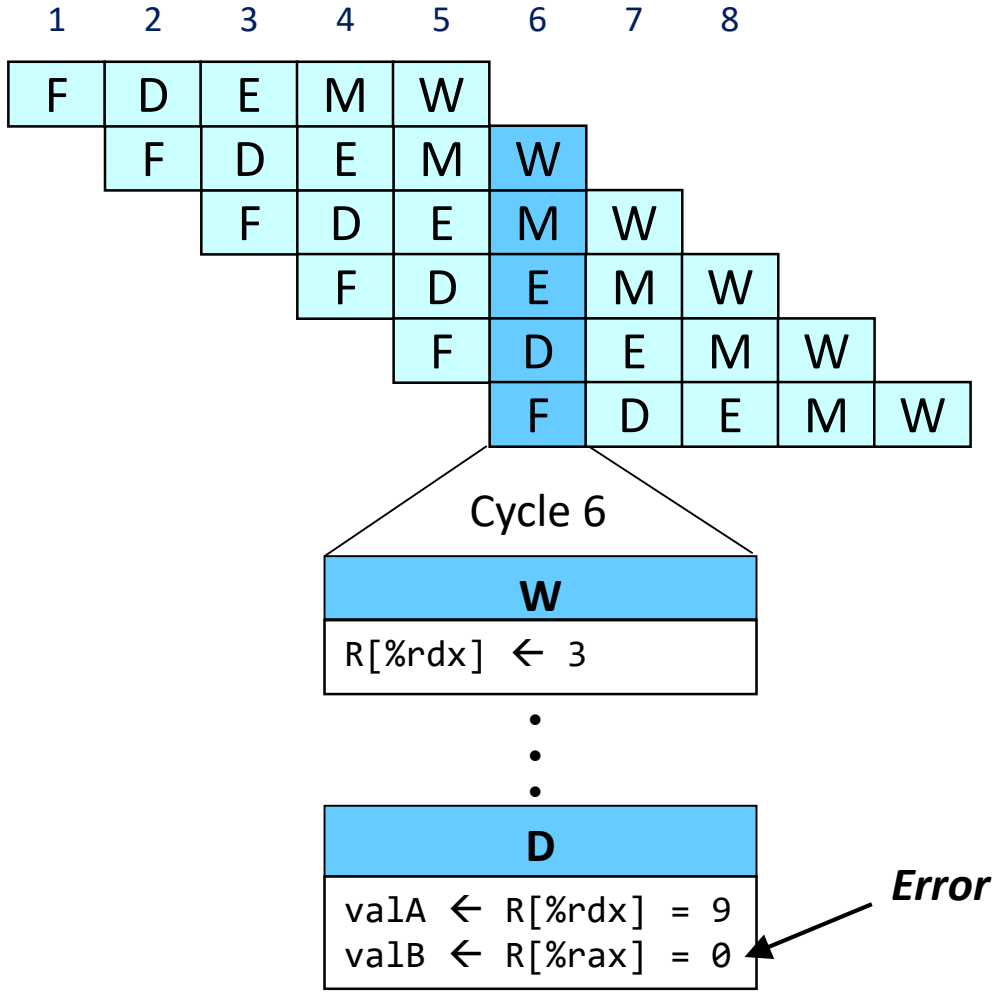


(Both %rax and %rdx are initialized to 0)

Example: Data Hazard (3)

- 2 nop's

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   nop
0x015:   nop
0x016:   addq     %rdx,%rax
0x018:   halt
```

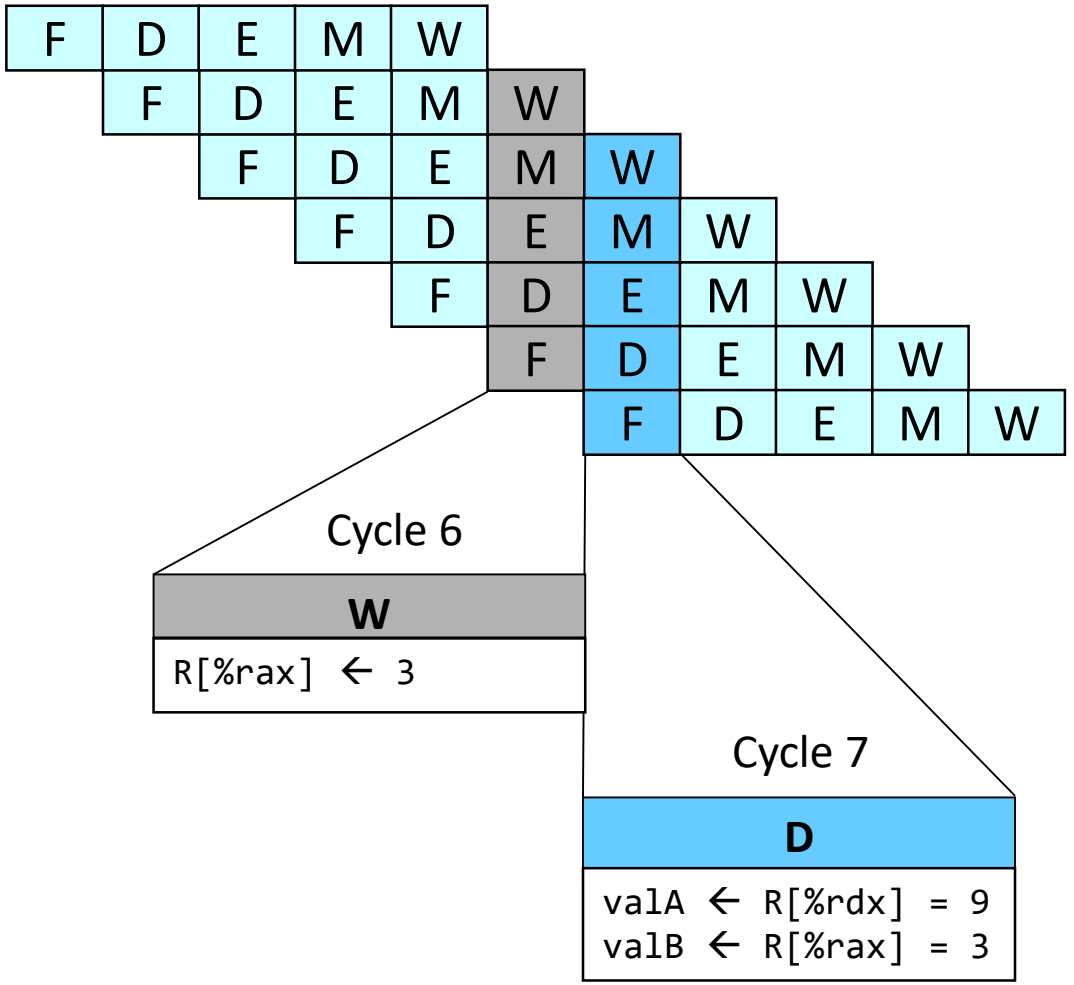


(Both %rax and %rdx are initialized to 0)

Example: Data Hazard (4)

- 3 nop's

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   nop
0x015:   nop
0x016:   nop
0x017:   addq     %rdx,%rax
0x019:   halt
```



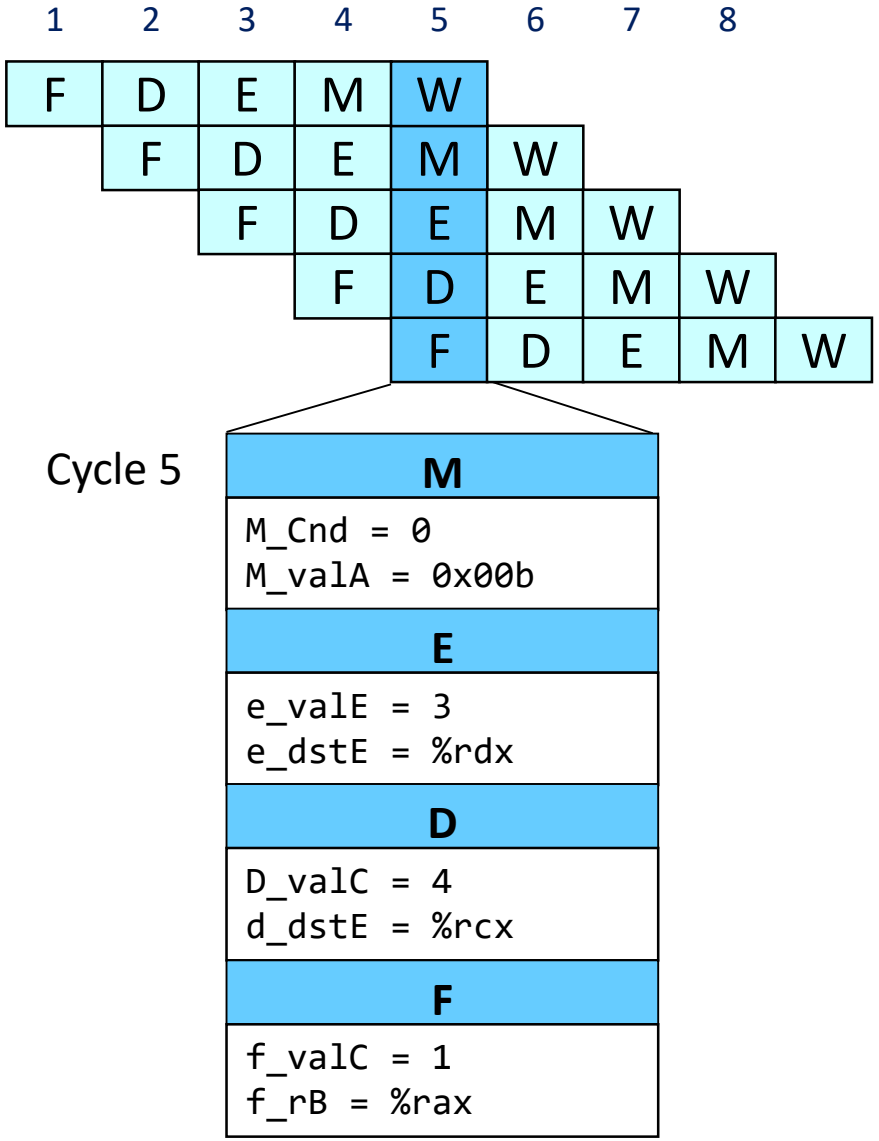
Example: Control Hazard (1)

- Branch misprediction example
  - Should only execute first 7 instructions

```
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
```

Example: Control Hazard (2)

```
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
```



- Branch misprediction trace (predict-taken)
  - Incorrectly execute two instructions at branch target

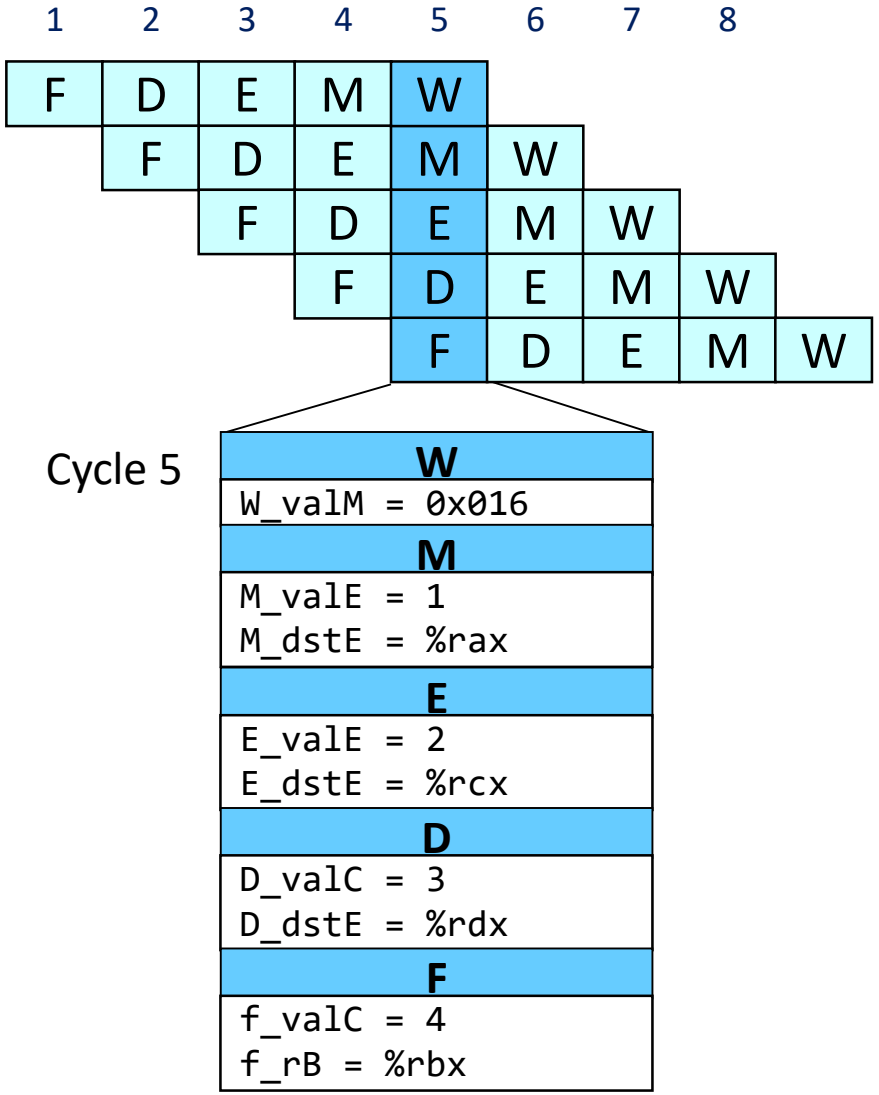
Example: Control Hazard (3)

- Return example
  - Require lots of nops to avoid data hazards

```
0x000:     irmovq  Stack,%rsp    # Initialize 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
```

Example: Control Hazard (4)

```
0x023:      ret
0x024:      irmovq  $1,%rax      # Oops!
0x02e:      irmovq  $2,%rcx      # Oops!
0x038:      irmovq  $3,%rdx      # Oops!
0x042:      irmovq  $4,%rbx      # Return
```



- Incorrect return example
  - Incorrectly execute 3 instructions following ret

PIPE- 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 dependency
    - 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
- How to fix them?