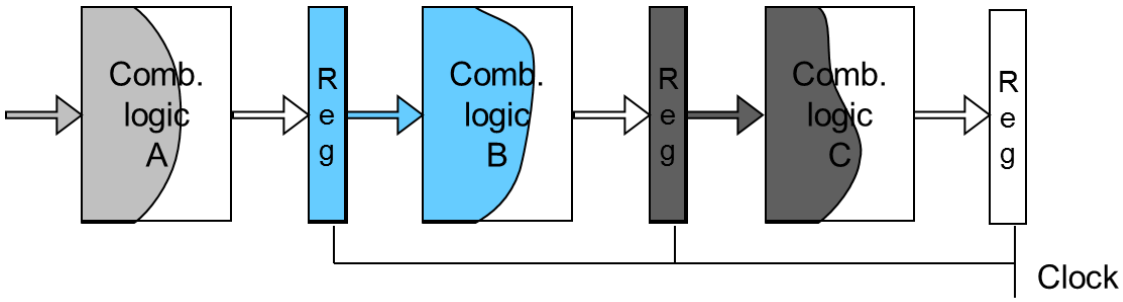
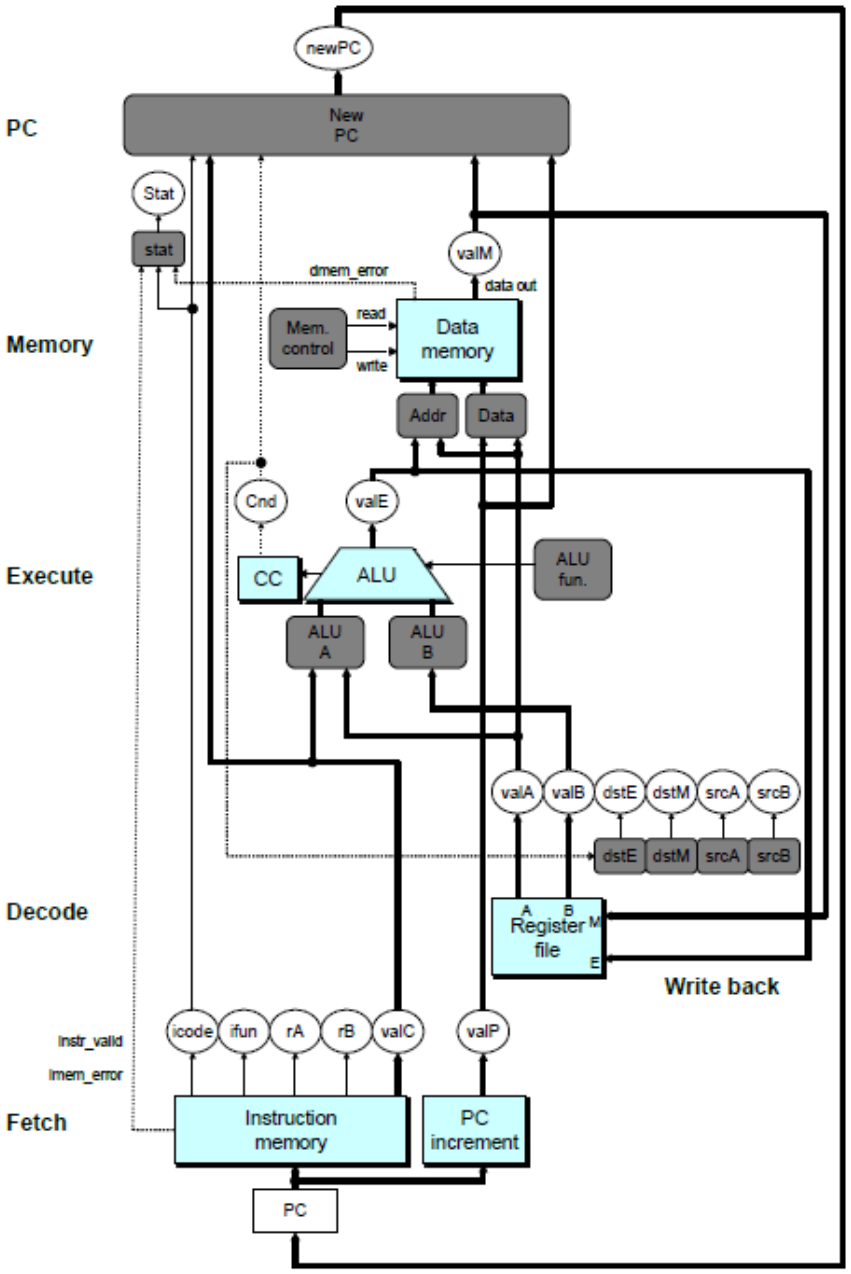
Processor Architecture

SEQ+: Y86 Pipelined Implementation (Part I)



SEQ Hardware

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

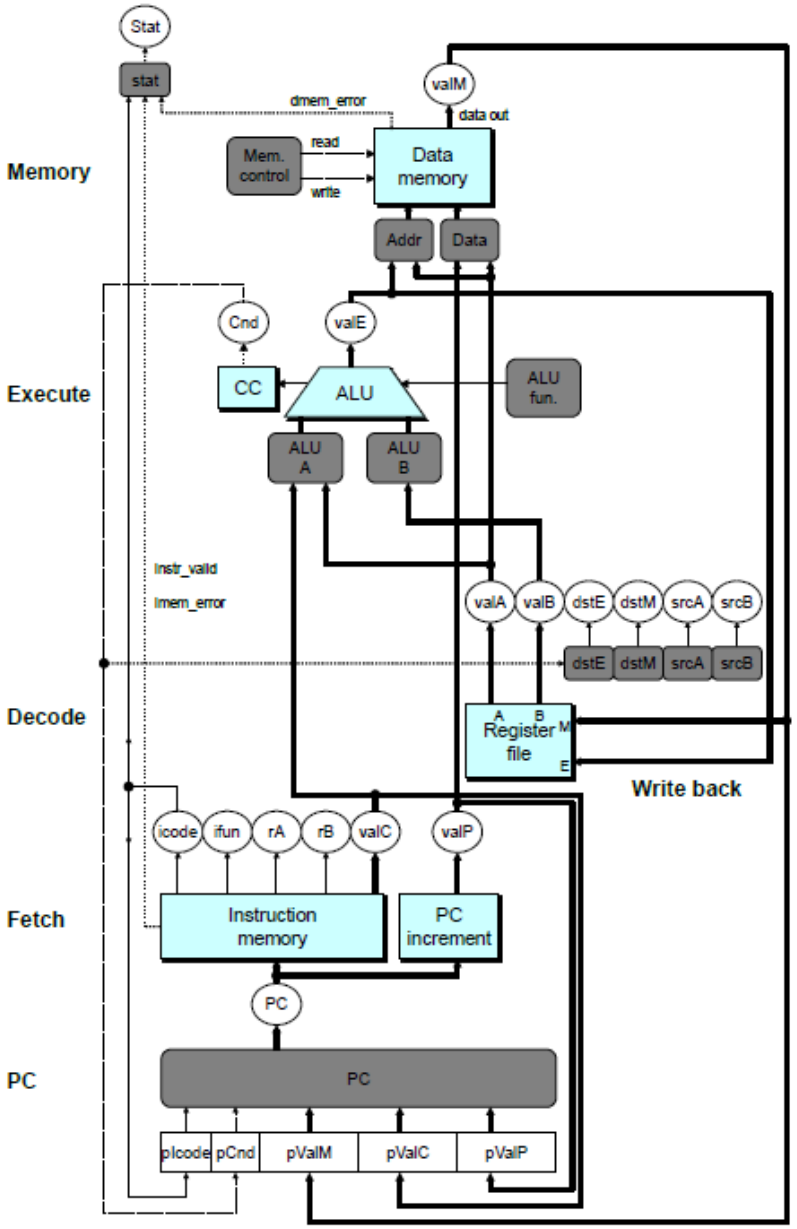


SEQ+ Hardware

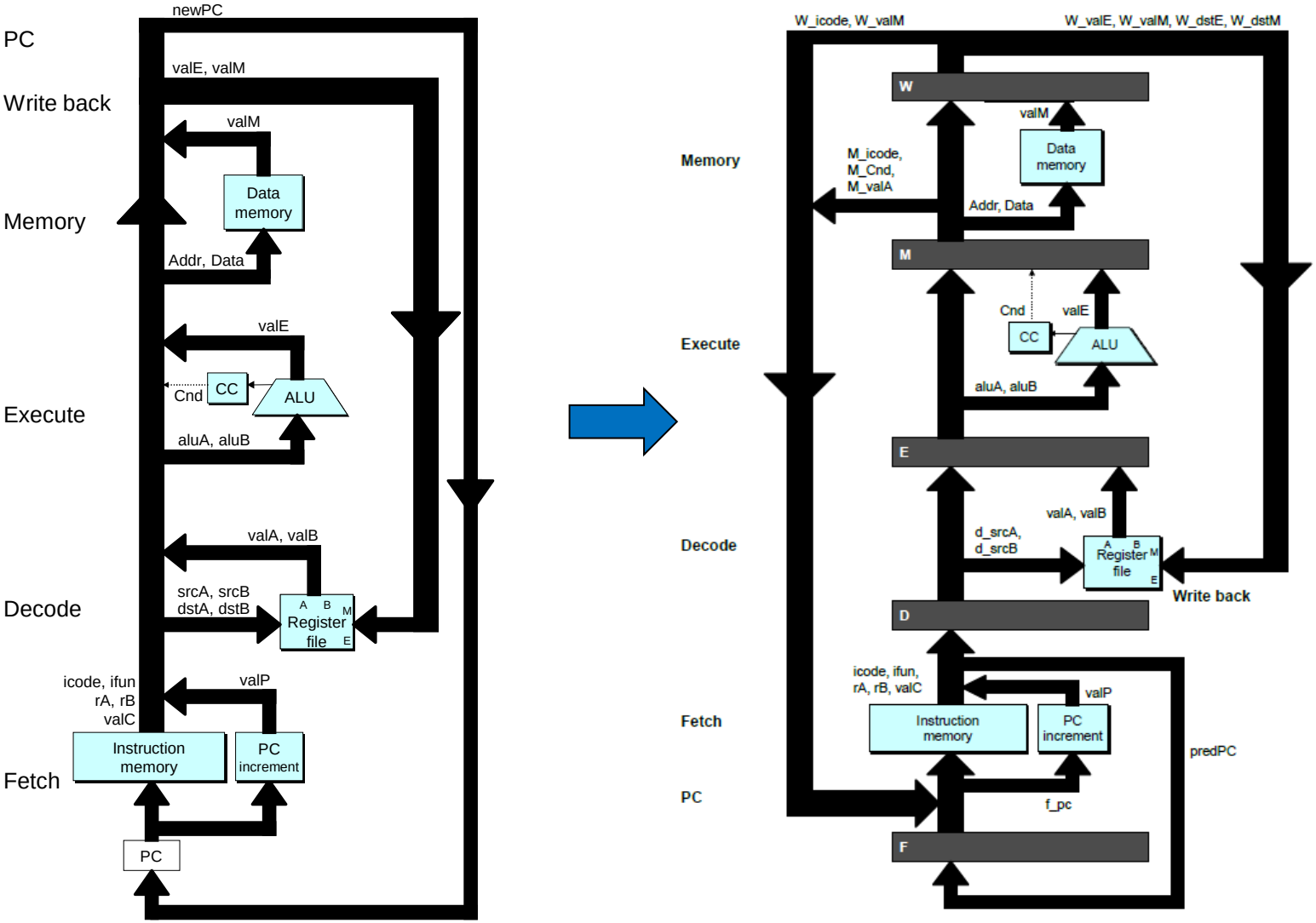
- Still sequential implementation
- Reordered PC stage to the beginning

- 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

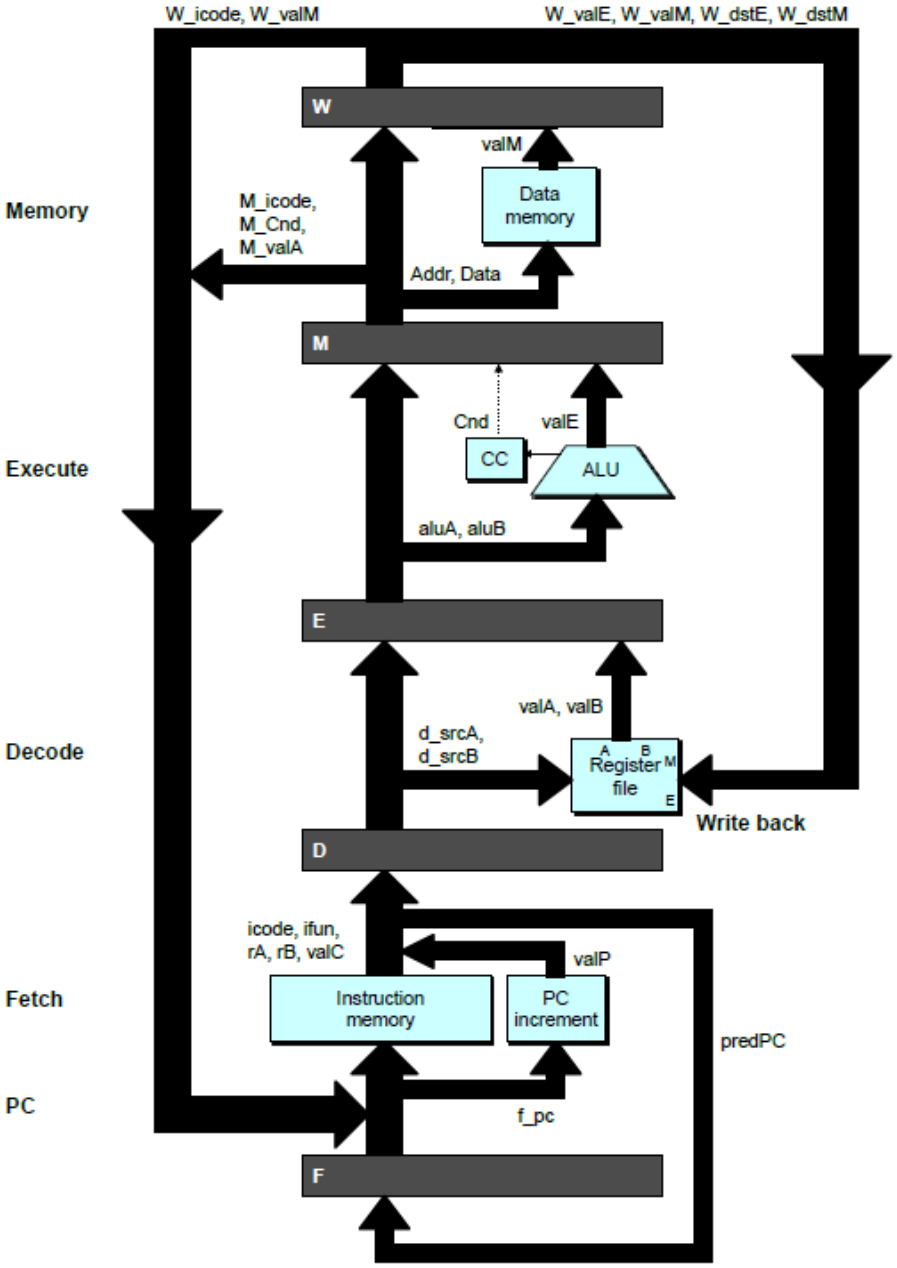


Adding Pipeline Registers



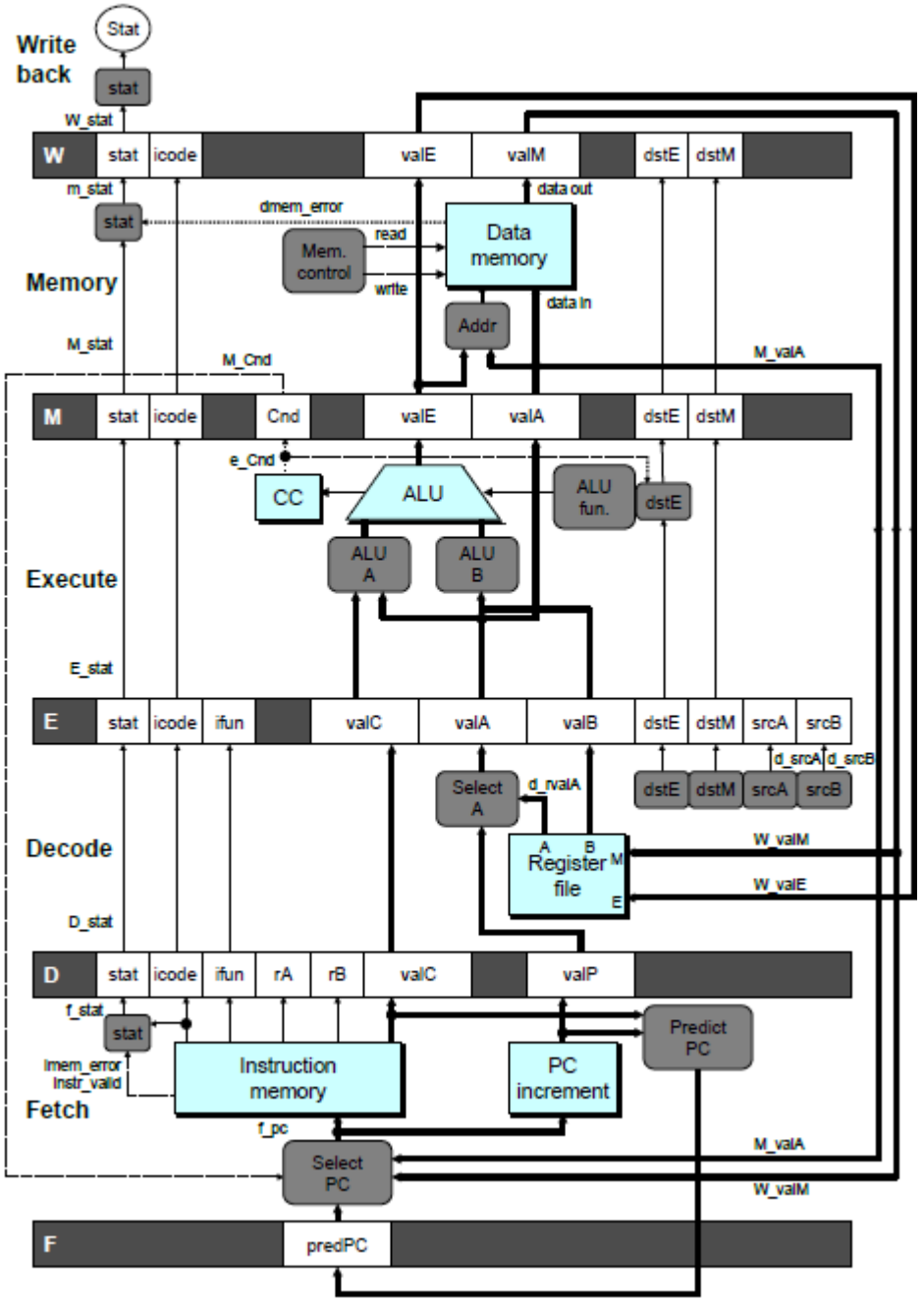
Pipeline Stages

- 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



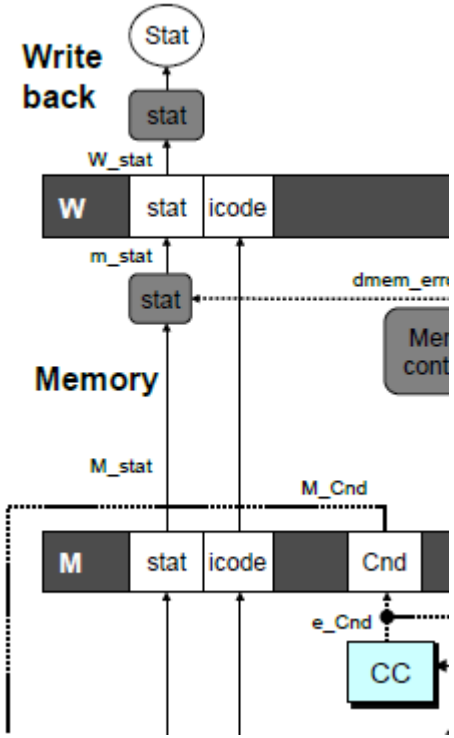
PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump over other stages
    - e.g., valC passes through decode



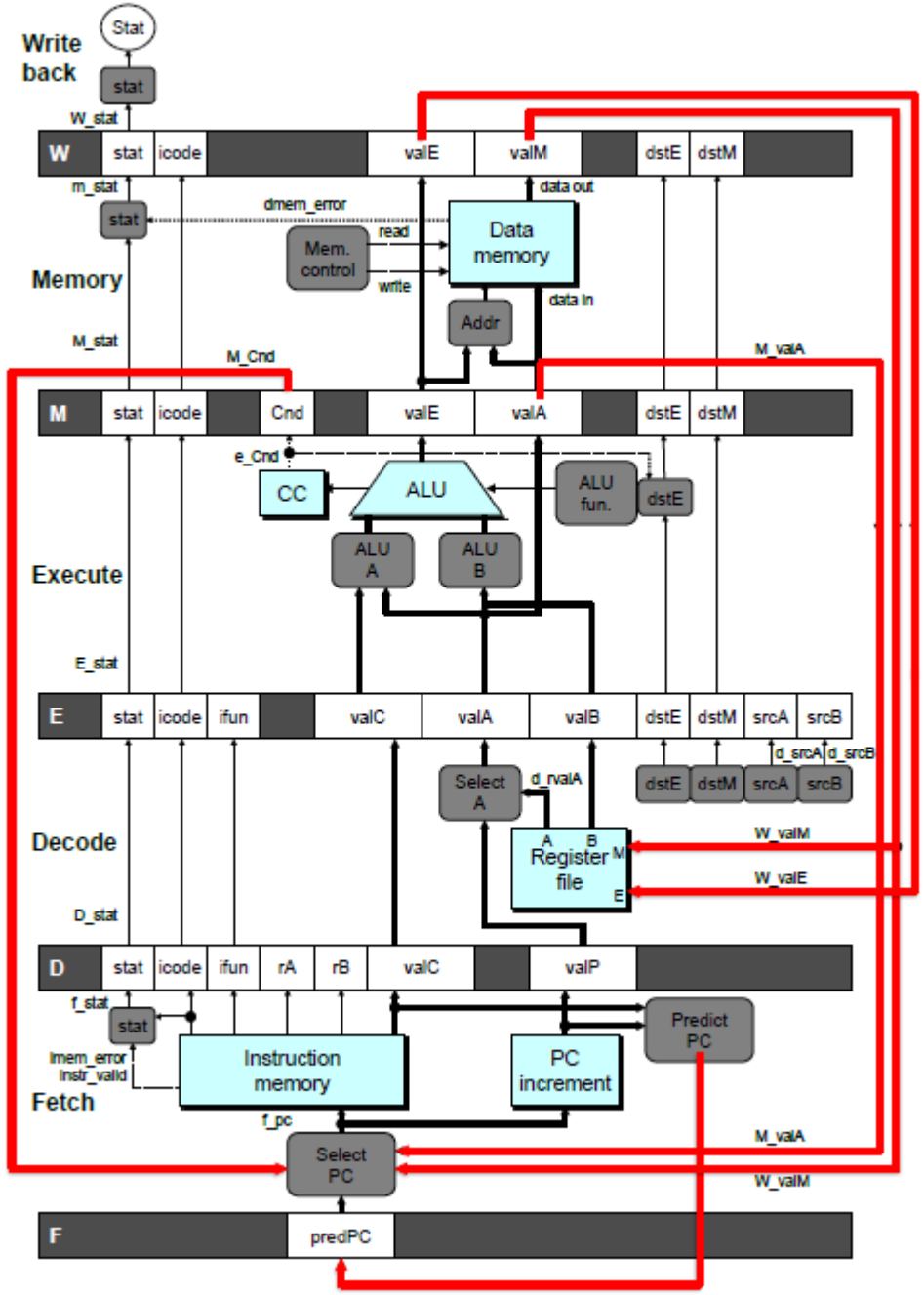
Signal Naming Conventions

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

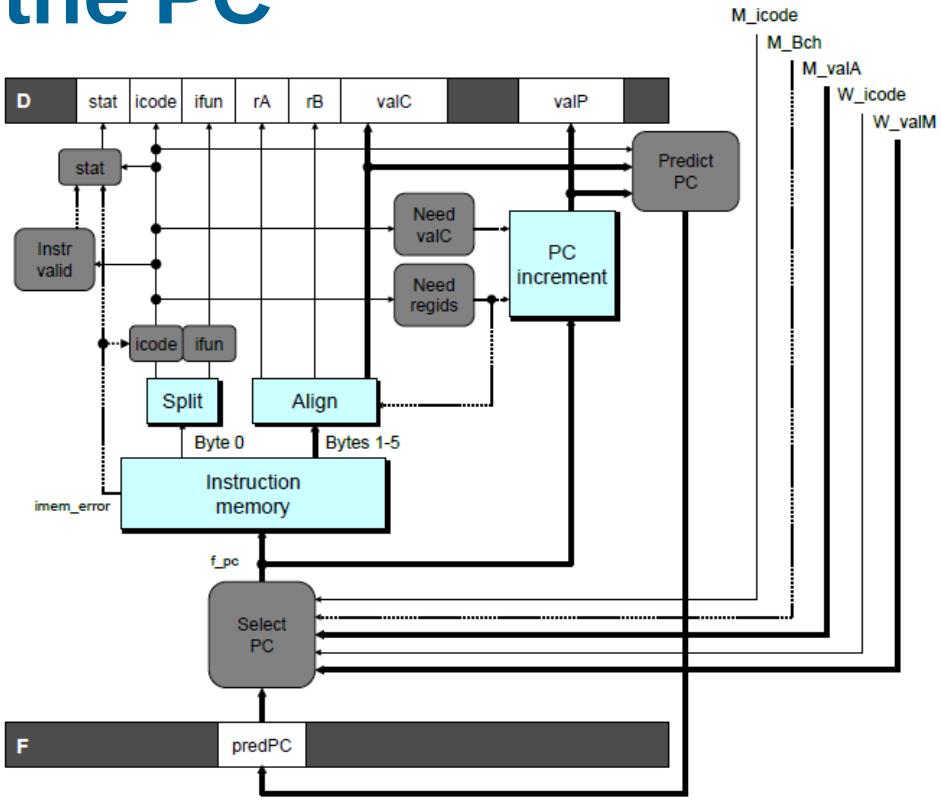


Feedback Paths

- 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

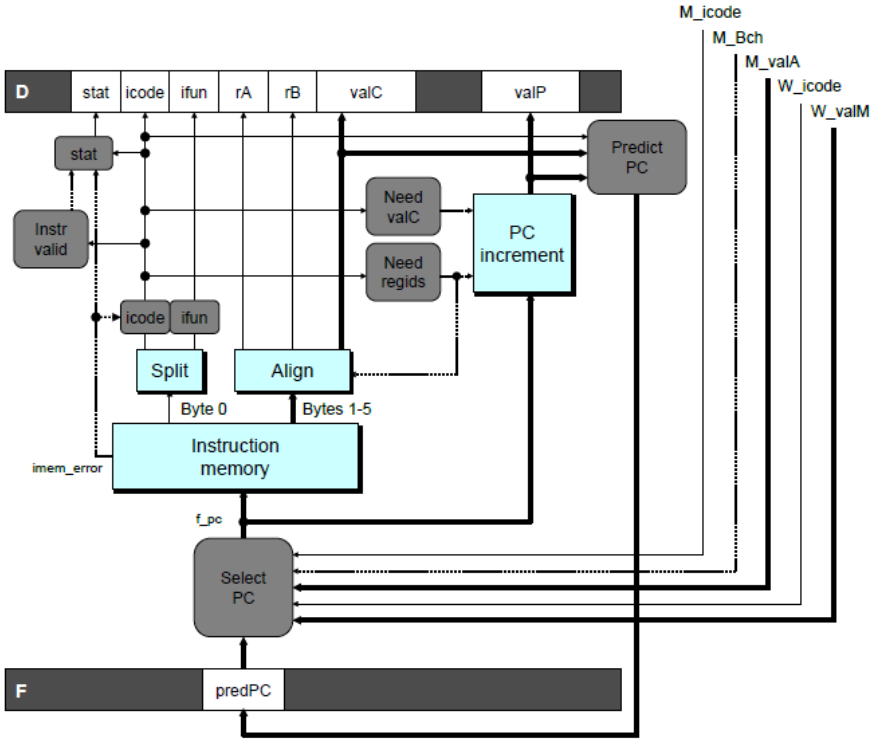


- Start fetch of new instruction after current one has completed fetch stage
  - Not possible to determine the next instruction 100% correctly
- Guess which instruction will follow
  - Recover if prediction was incorrect

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

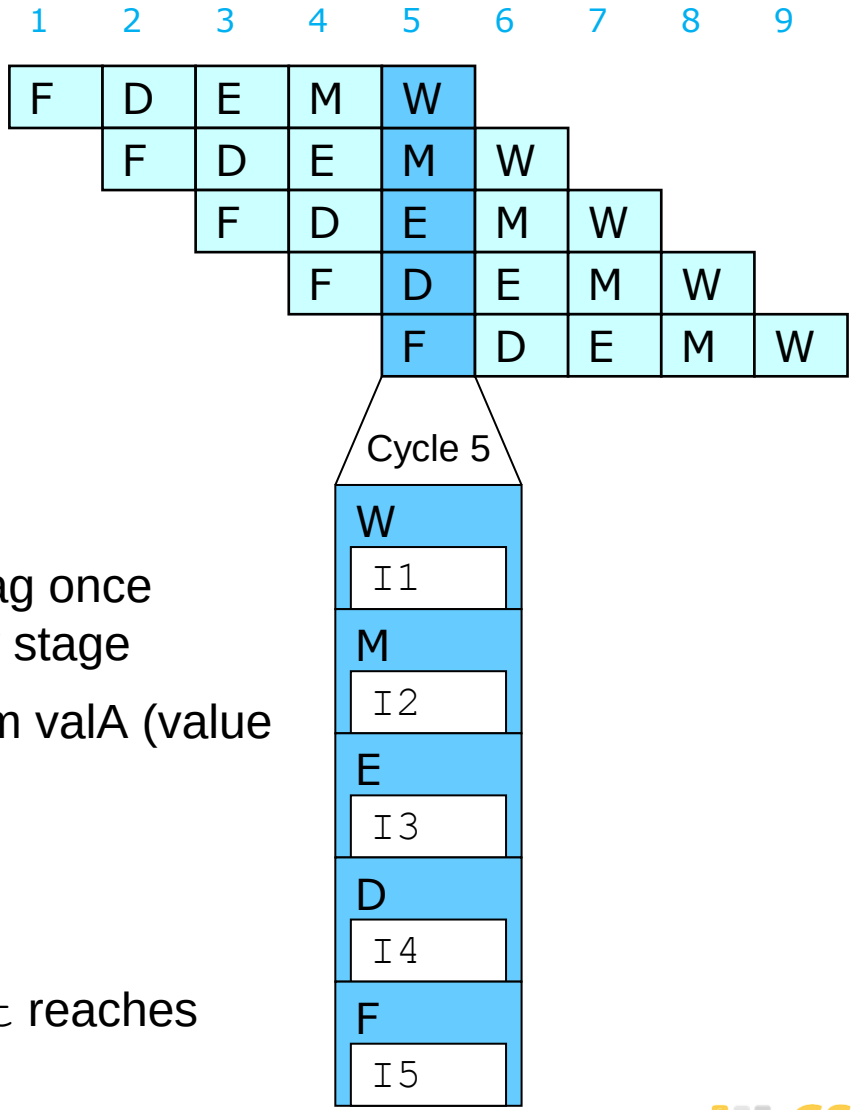
Recovering from PC Misprediction



- 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

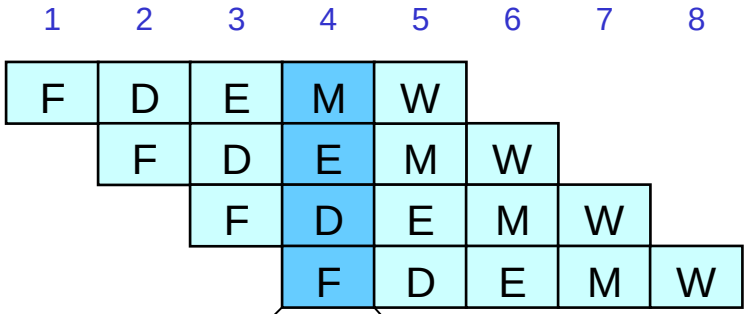
```
irmovl    $1,%eax    #I1
irmovl    $2,%ecx    #I2
irmovl    $3,%edx    #I3
irmovl    $4,%ebx    #I4
halt                 #I5
```

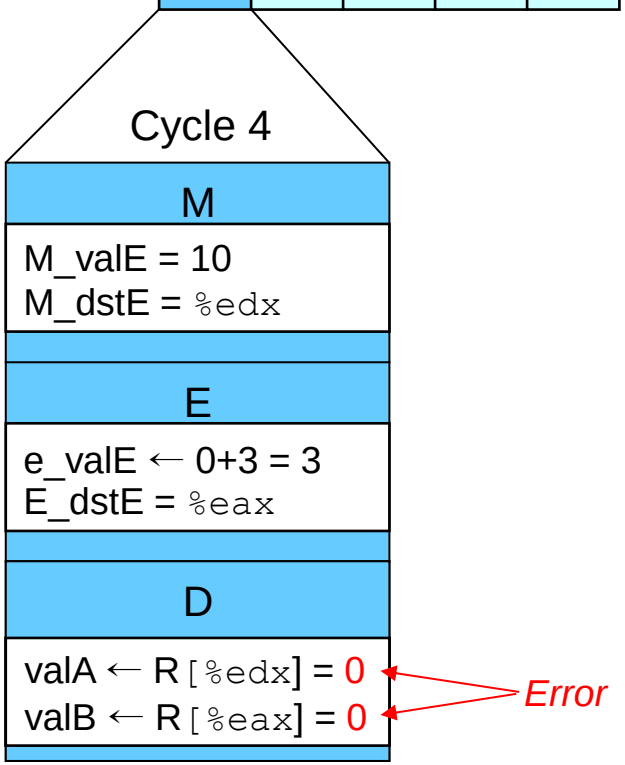


- Mispredicted Jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from valA (value M_valA)
- Return Instruction
  - Will get return PC when `ret` reaches write-back stage (W_valM)

Data Dependencies: No Nop

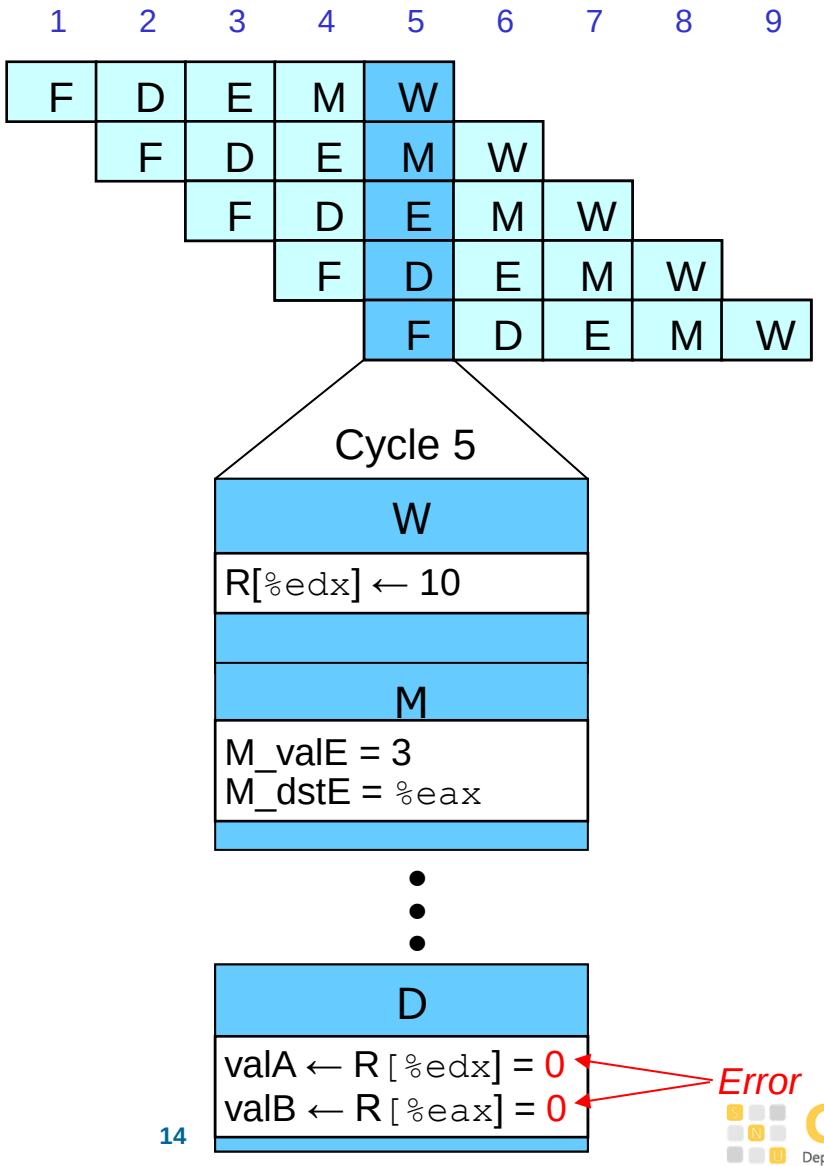
```
0x000: irmovl   $10, %edx
0x006: irmovl    $3, %eax
0x00c: addl    %edx, %eax
0x00e: halt
```





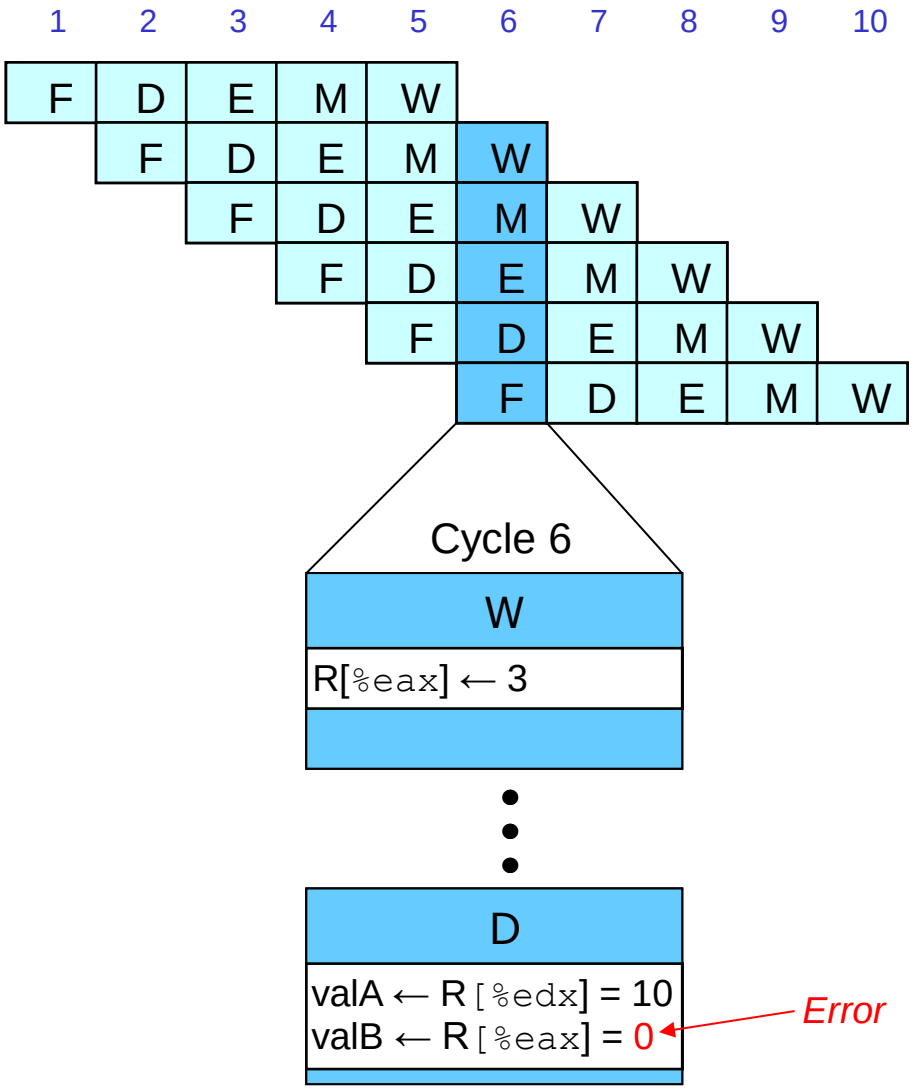
Data Dependencies: 1 Nop

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



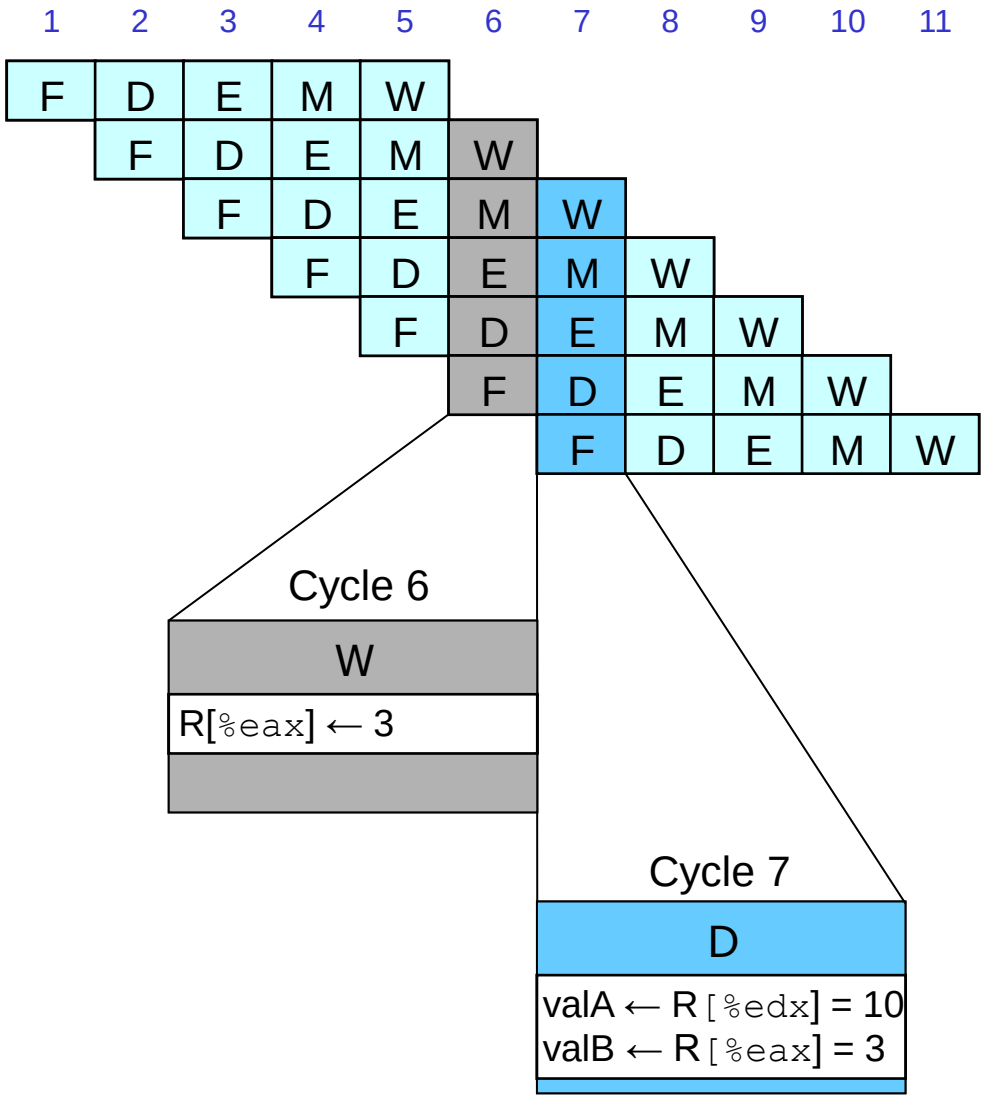
Data Dependencies: 2 Nop's

```
0x000: irmovl   $10, %edx
0x006: irmovl    $3, %eax
0x00c: nop
0x00d: nop
0x00e: addl    %edx, %eax
0x010: halt
```



Data Dependencies: 3 Nop's

```
0x000: irmovl   $10, %edx
0x006: irmovl    $3, %eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl    %edx, %eax
0x011: halt
```



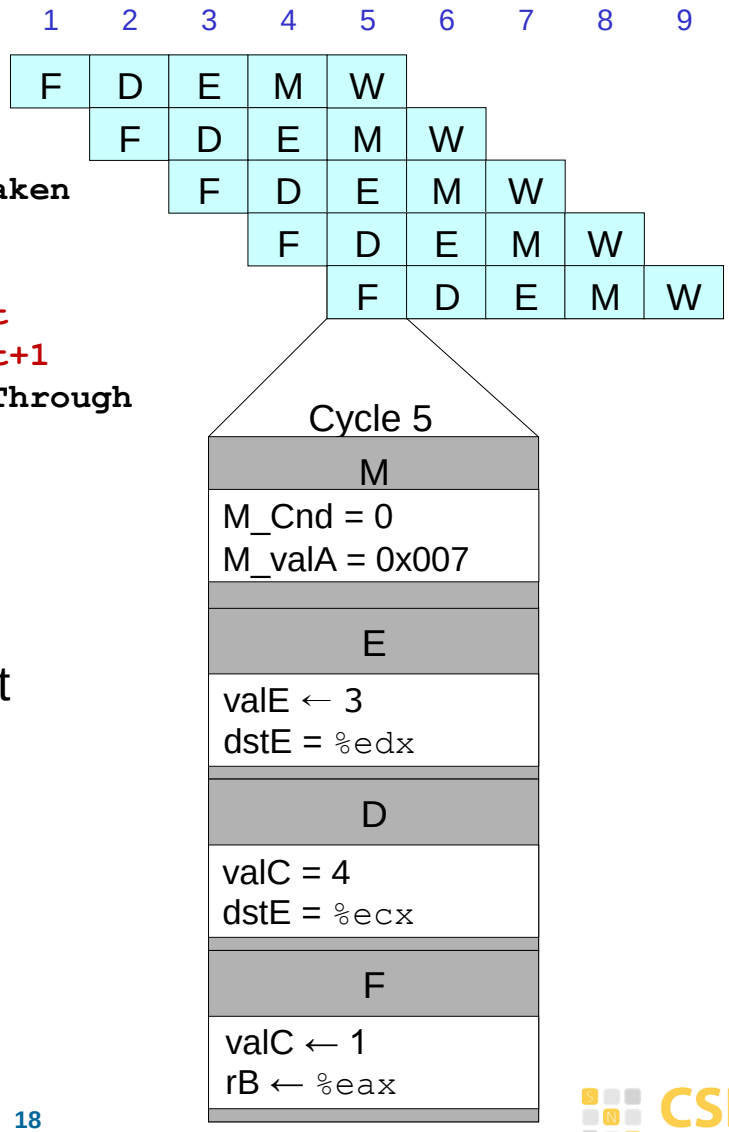
Branch Misprediction Example

- Predict ‘Taken’ at 0x002

```
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
```

Branch Misprediction Trace

```
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
```



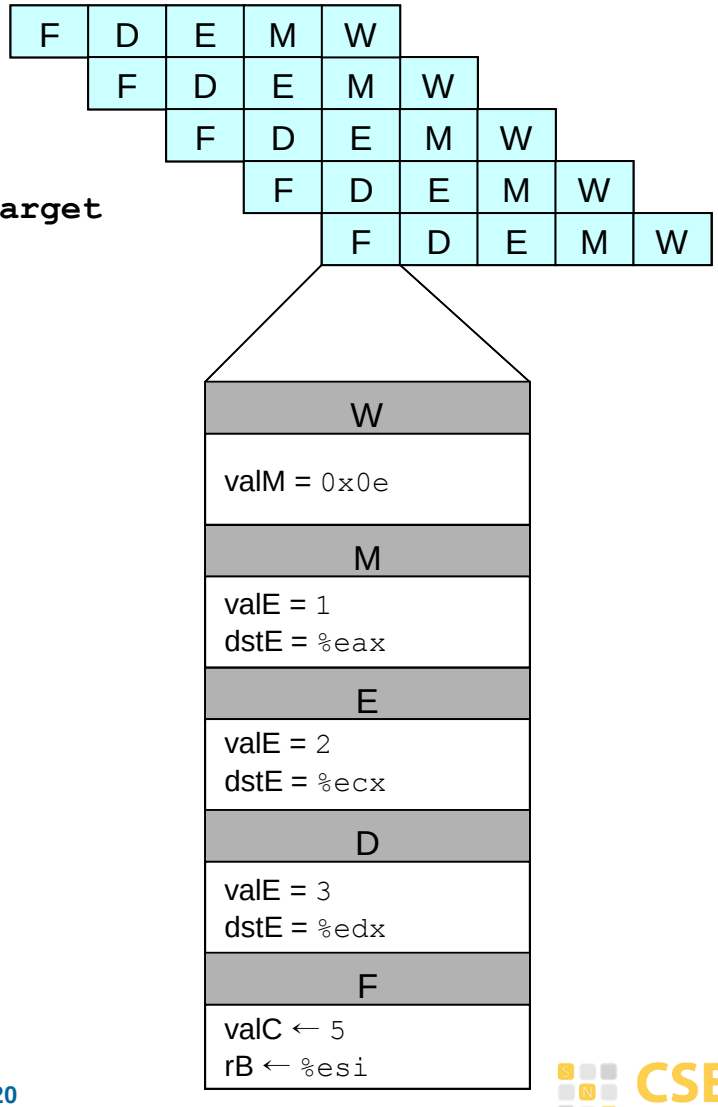
- Incorrectly execute two instructions at branch target

Return Example

```
0x000:     irmovl Stack,%esp  # Initialize 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:  nop                # procedure
0x021:     nop
0x022:     nop
0x023:     ret
0x024:     irmovl $1,%eax     # Should not execute
0x02a:     irmovl $2,%ecx     # Should not execute
0x030:     irmovl $3,%edx     # Should not execute
0x036:     irmovl $4,%ebx     # Should not execute
0x100: .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

Incorrect Return Example

```
0x023:    ret
0x024:    irmovl $1,%eax      # ret+1
0x02a:    irmovl $2,%ecx      # ret+2
0x030:    irmovl $3,%edx      # ret+3
0x00e:    irmovl $5,%esi      # return target
```



- Incorrectly execute three instructions following `ret`

SEQ+ 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 next time