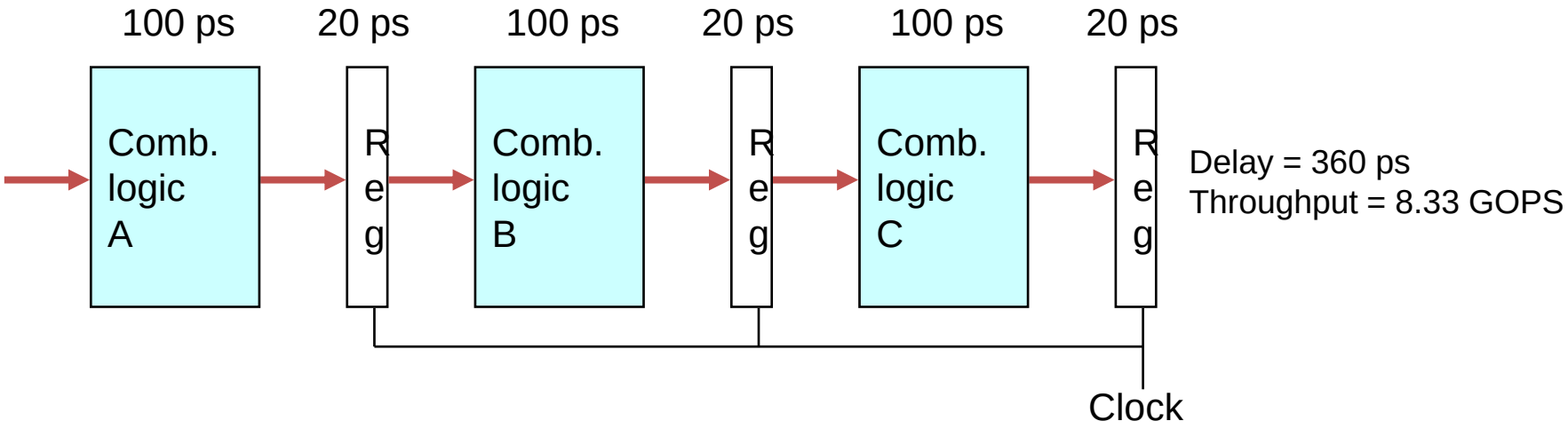
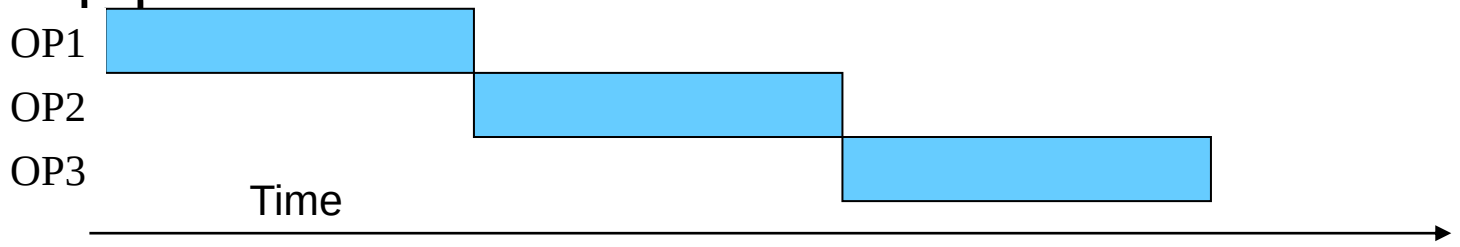
3-Way Pipelined Version



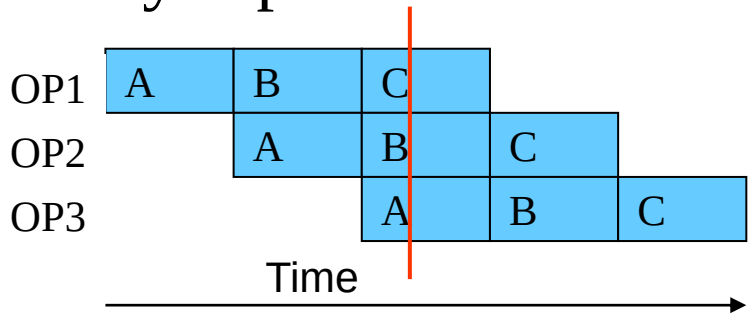
- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

Pipeline Diagrams

- Unpipelined

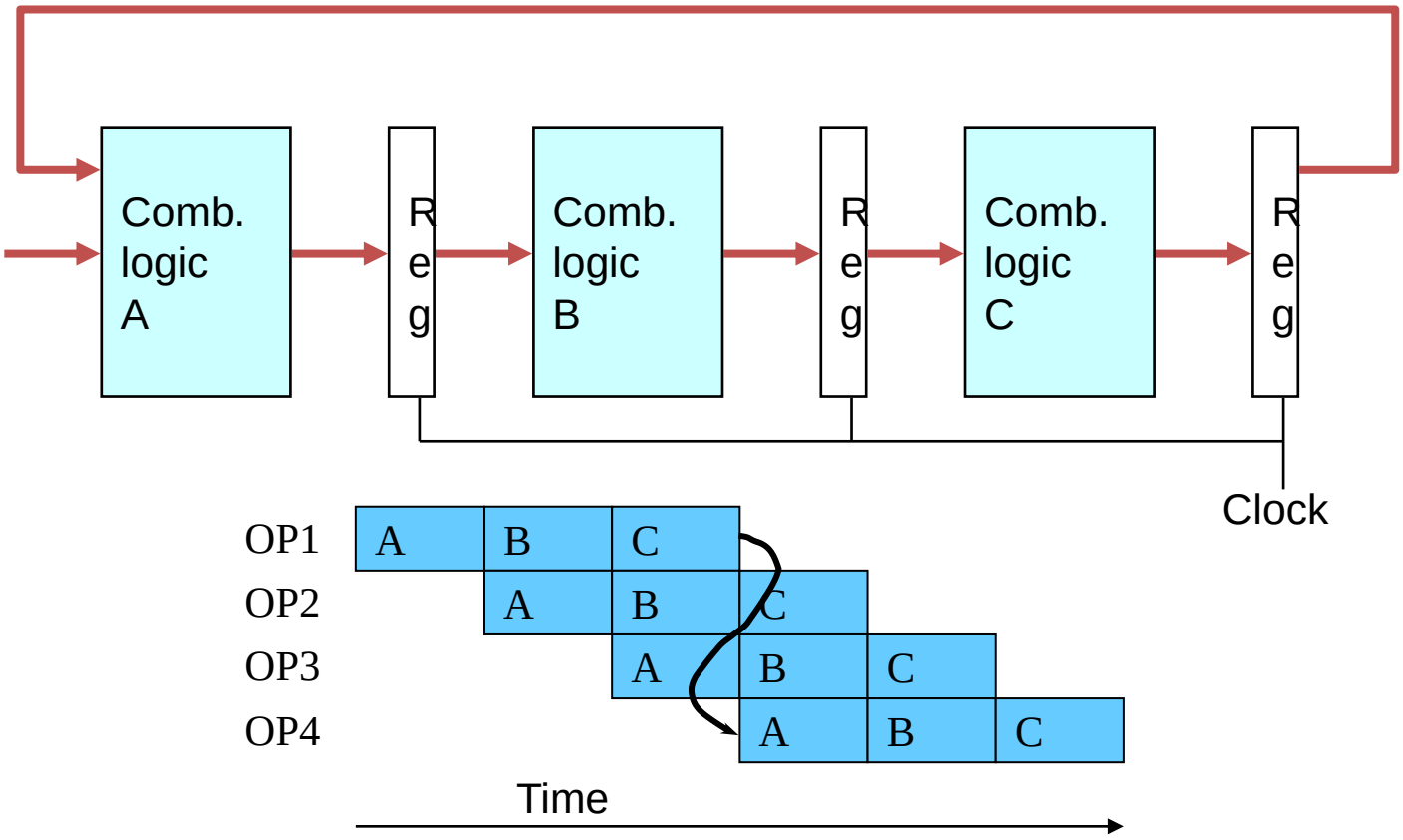


  - Cannot start new operation until previous one completes
- 3-Way Pipelined



  - Up to 3 operations in process simultaneously

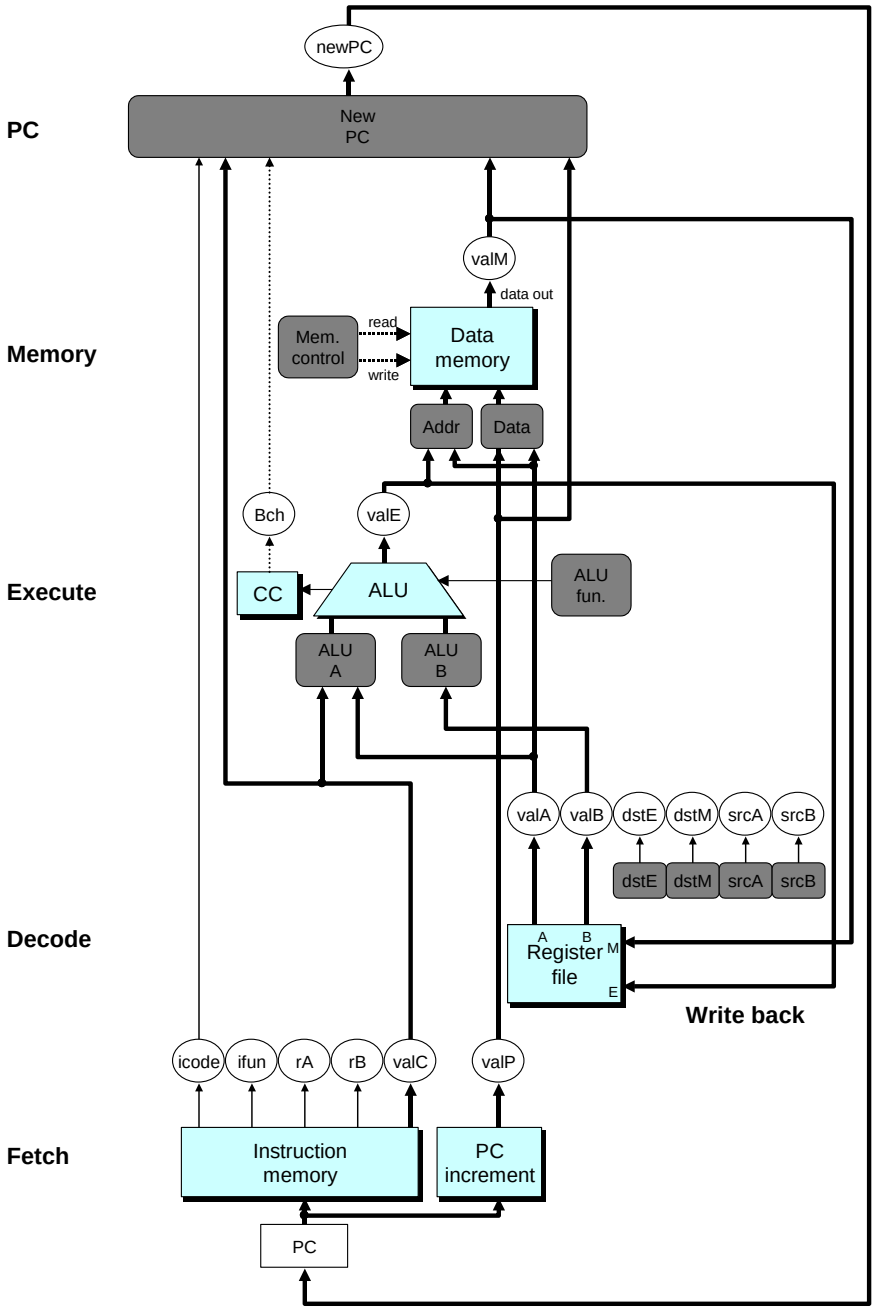
Data Hazards



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

SEQ Hardware

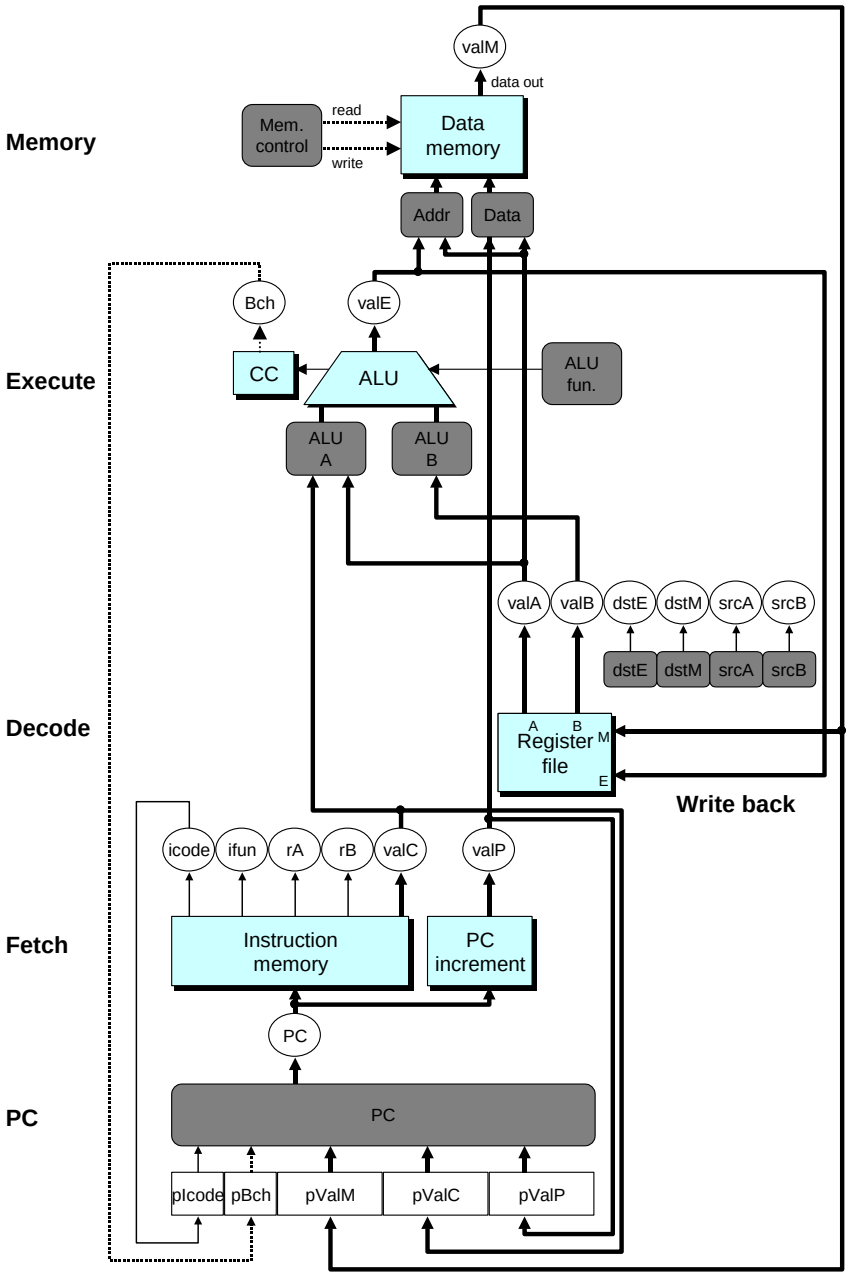
- Stages occur in sequence
- One operation in process at a time



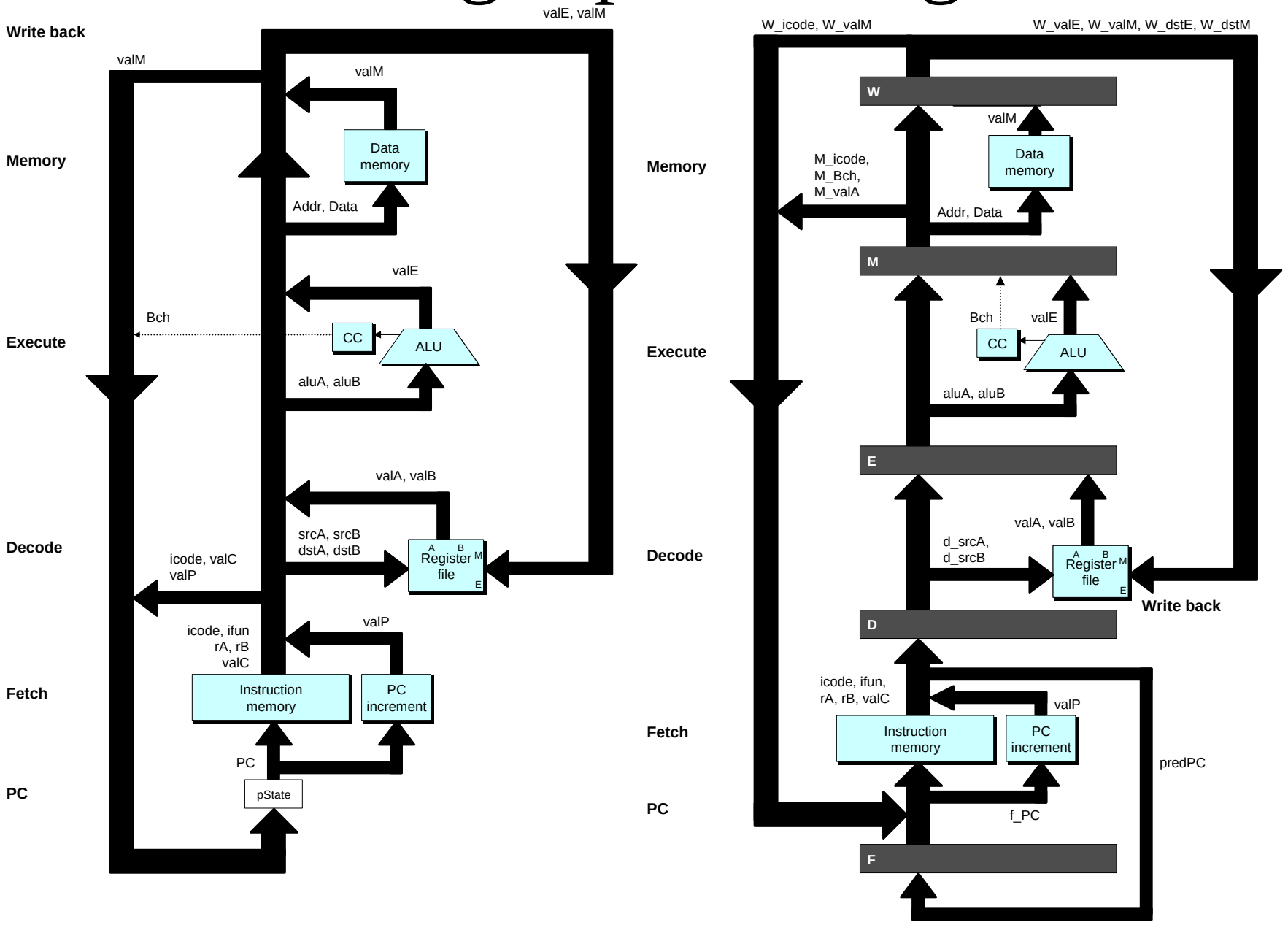
SEQ+ Hardware

- 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

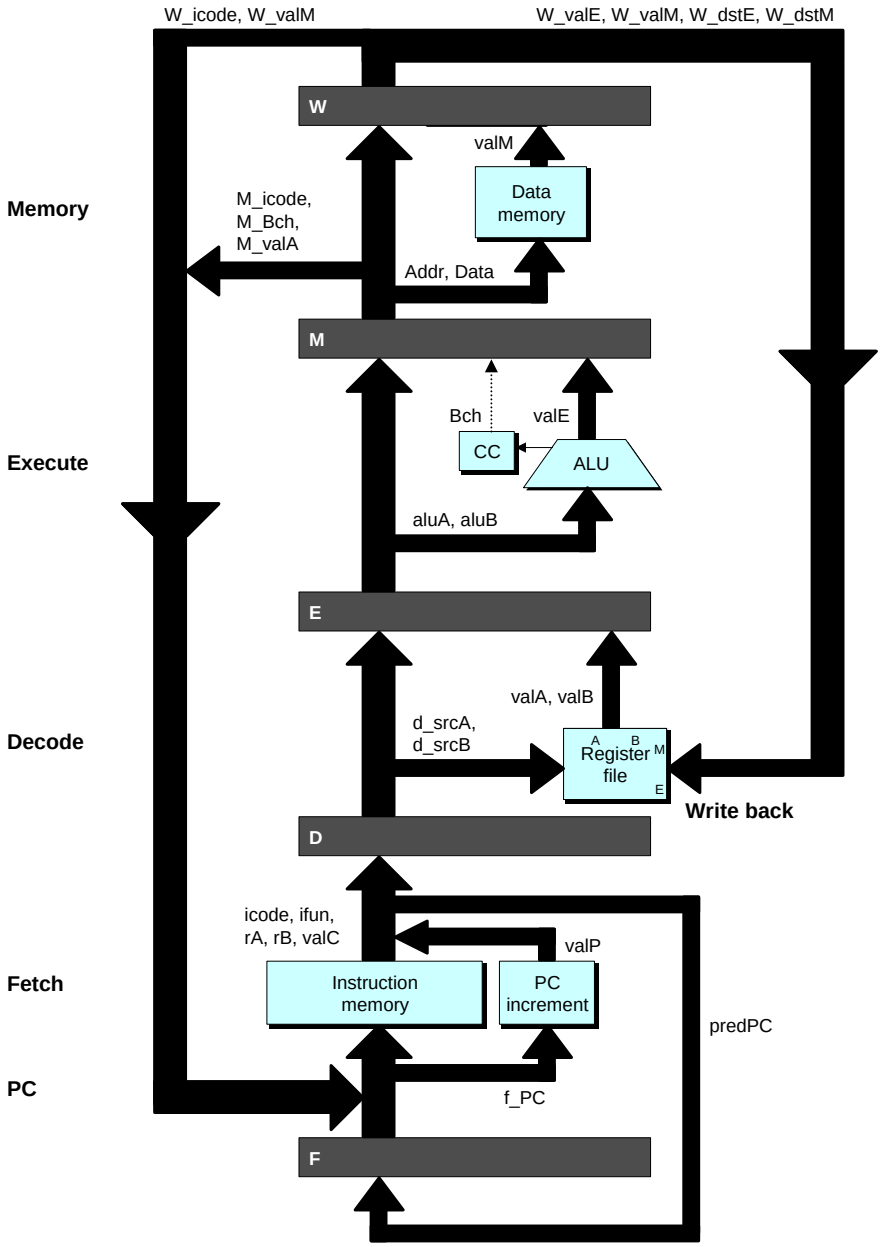


Adding Pipeline Registers

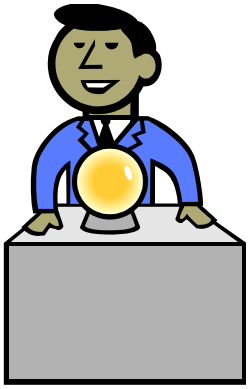


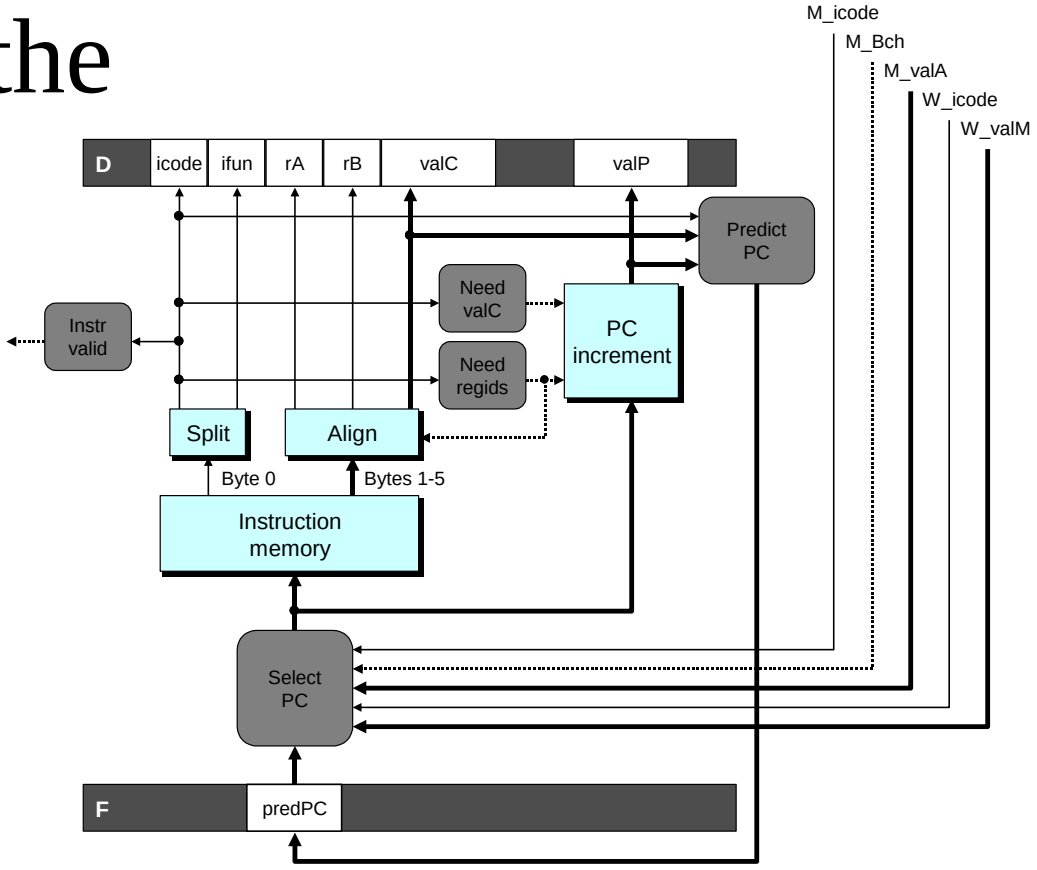
Pipeline Stages

- 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



Predicting the PC



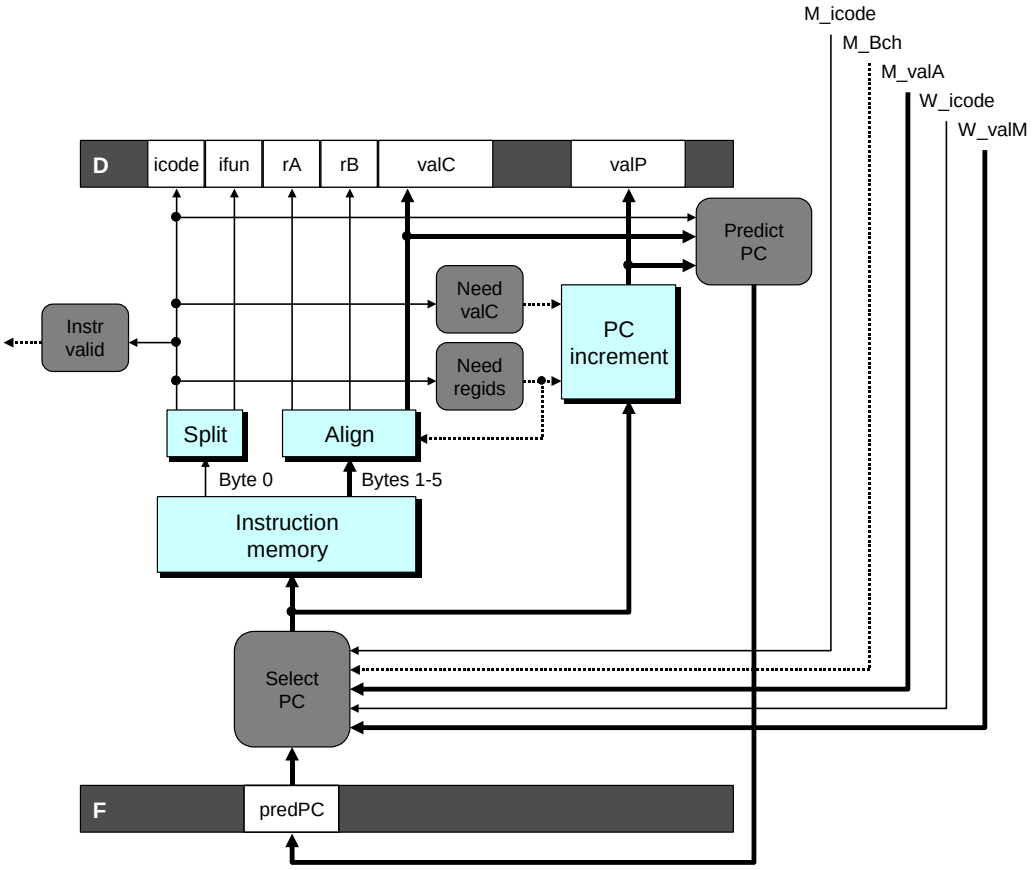


- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

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

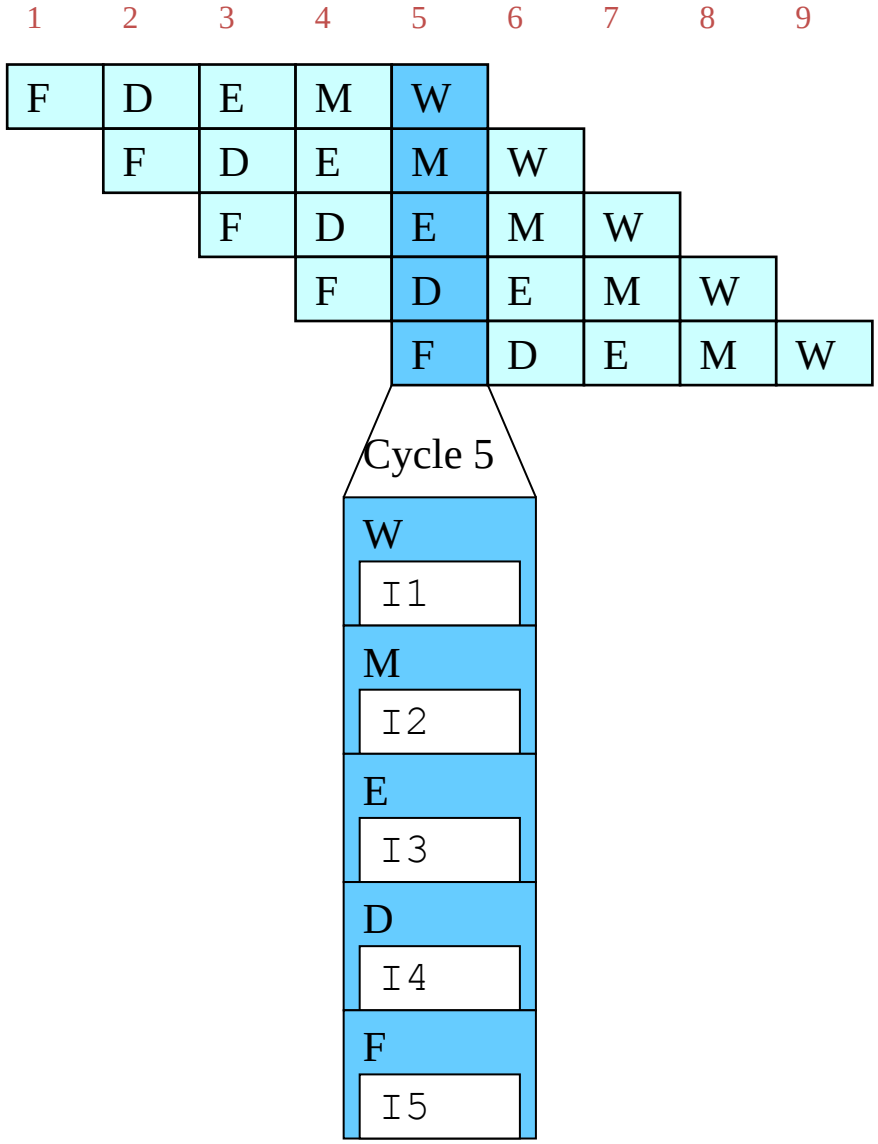
Recovering from PC Misprediction



- 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

Pipeline Demonstration

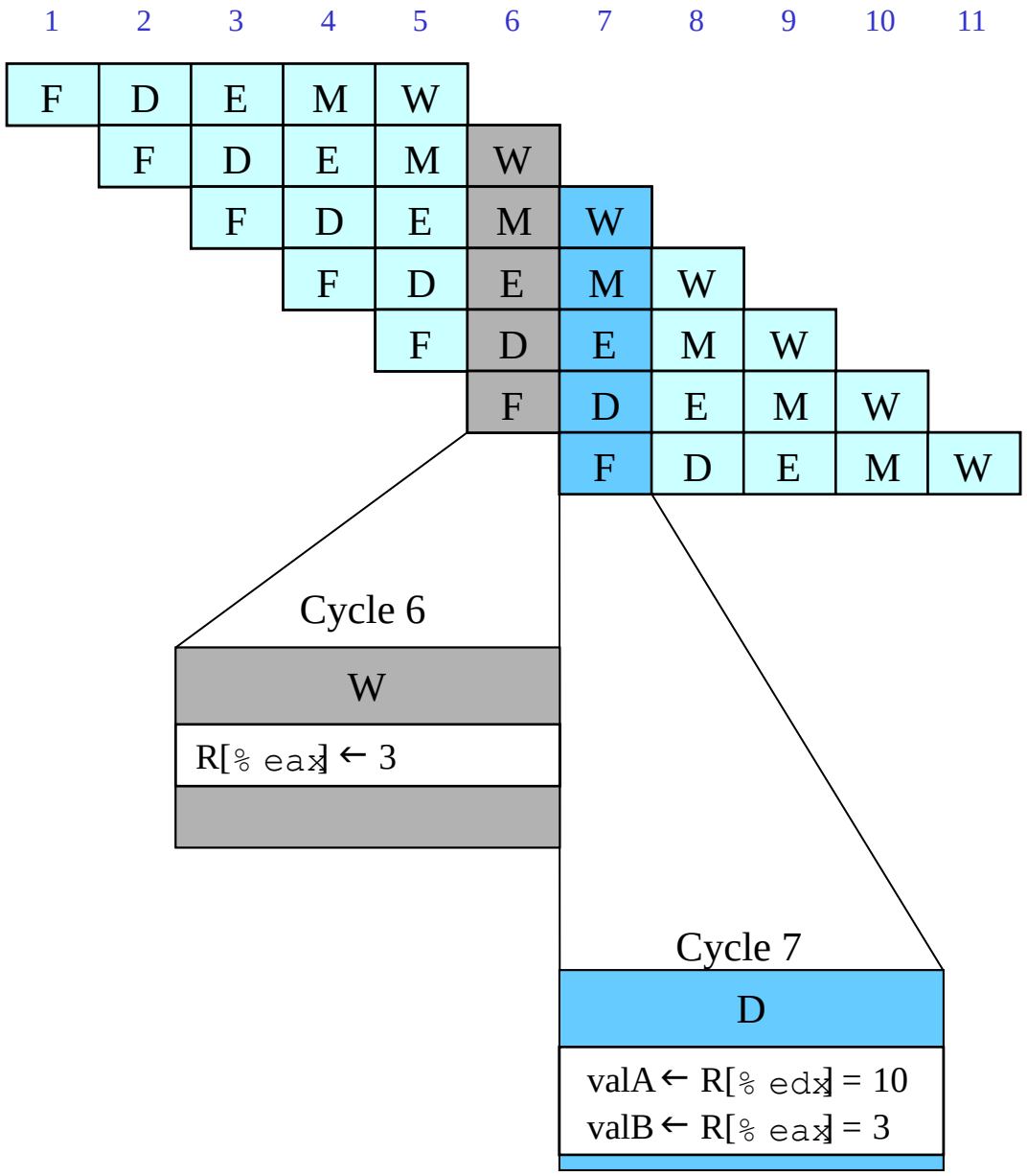
```
irmovl    $1,%eax   #I1
irmovl    $2,%ecx   #I2
irmovl    $3,%edx   #I3
irmovl    $4,%ebx   #I4
halt                #I5
```



- File: `demo-basic.ys`

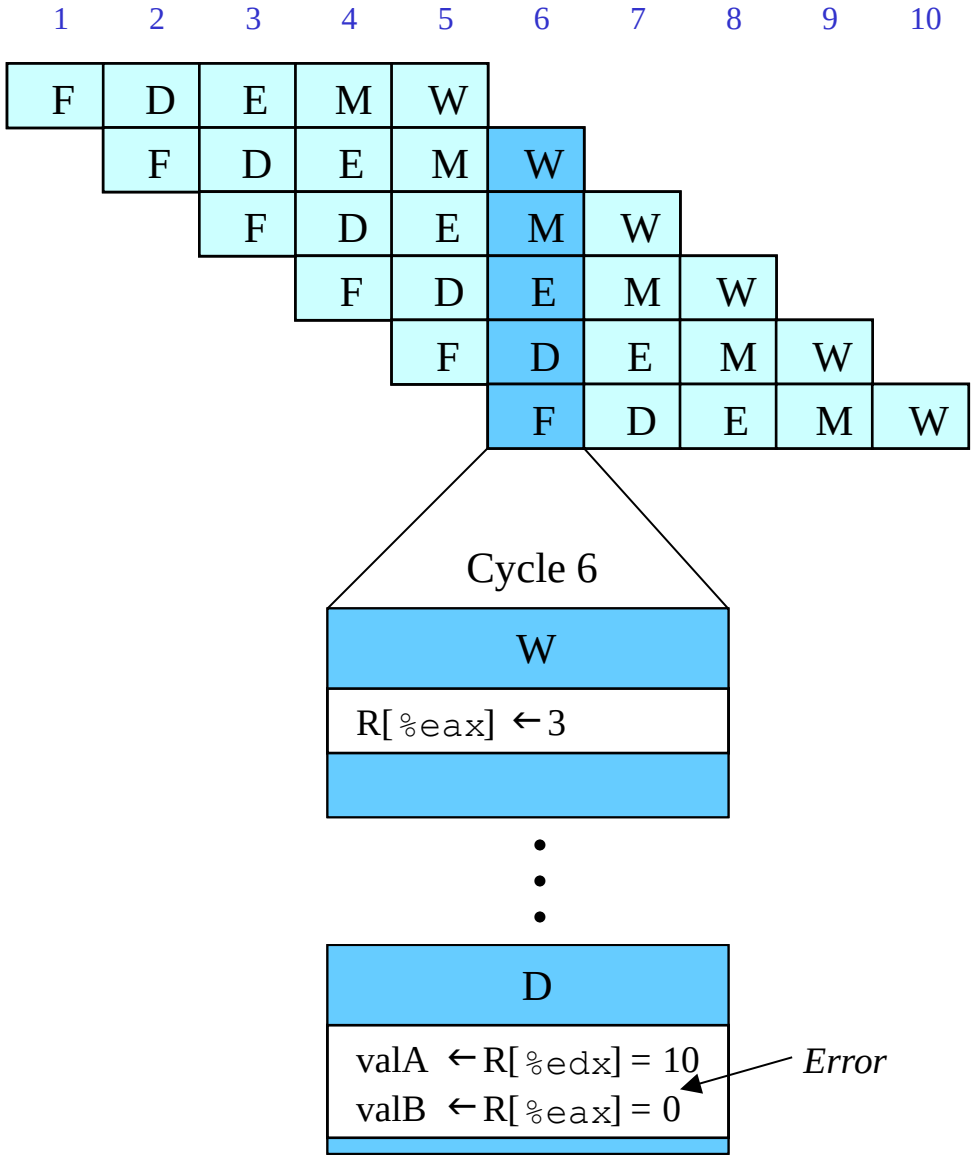
Data Dependencies: 3 Nop's

```
# 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

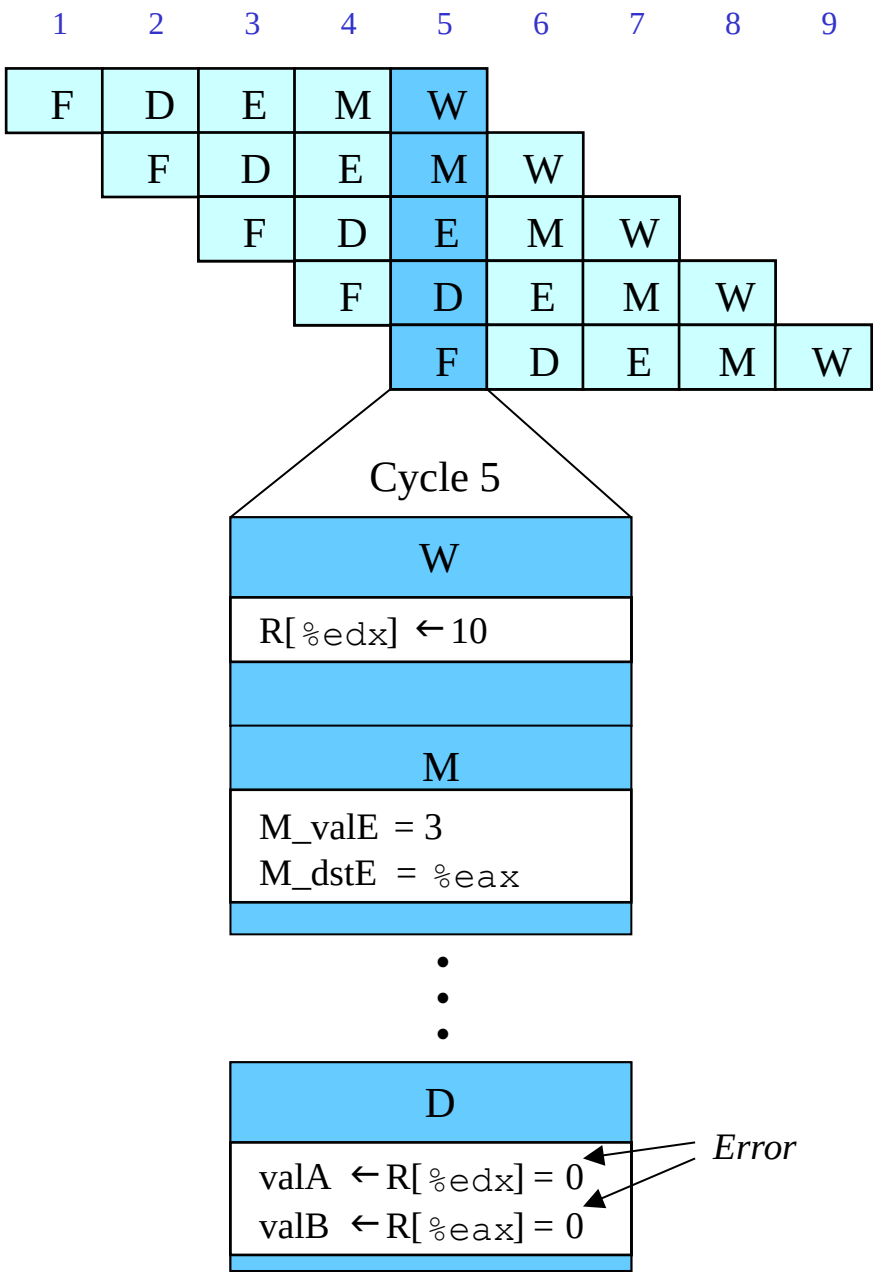
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



Data Dependencies: 1 Nop

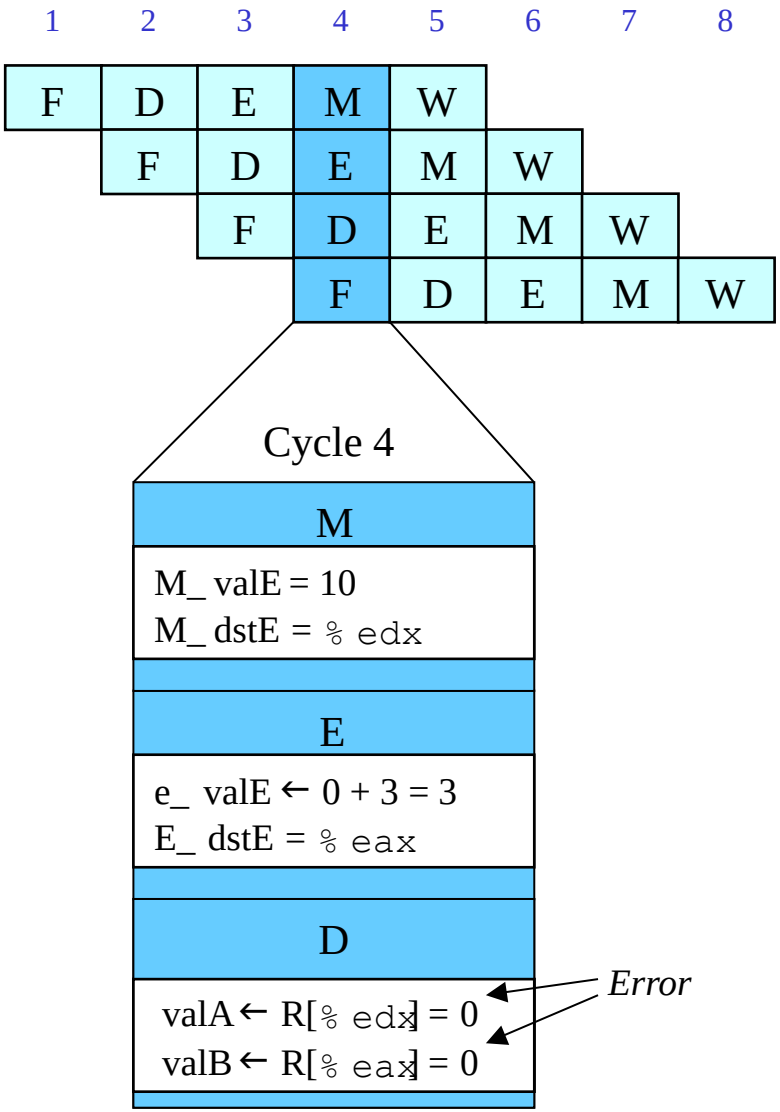
```
# demo-h1.ys

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



Data Dependencies: No Nop

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



Return Example

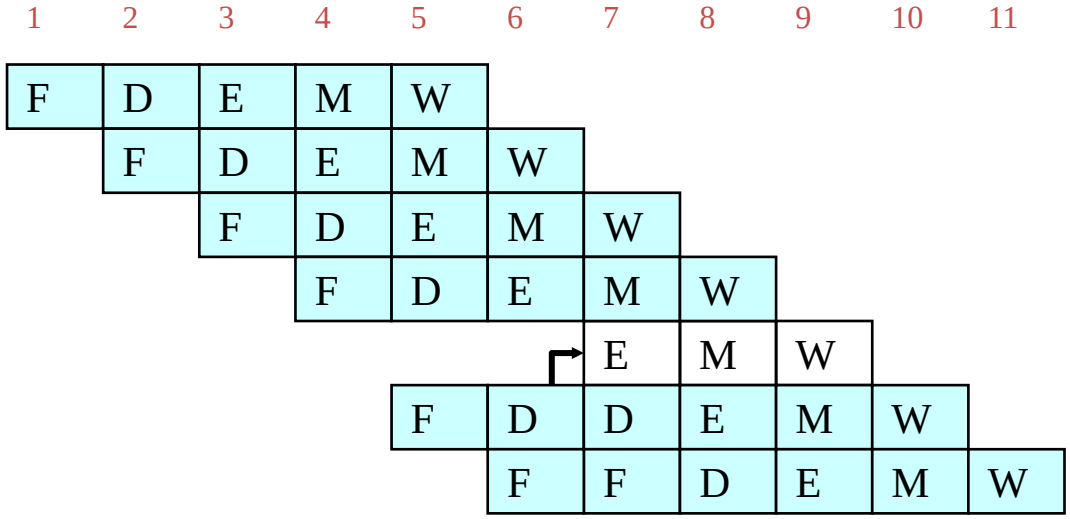
demo-ret.ys

```
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: nop                # procedure
0x021:    nop
0x022:    nop
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
```

- Require lots of nops to avoid data hazards

Stalling for Data Dependencies

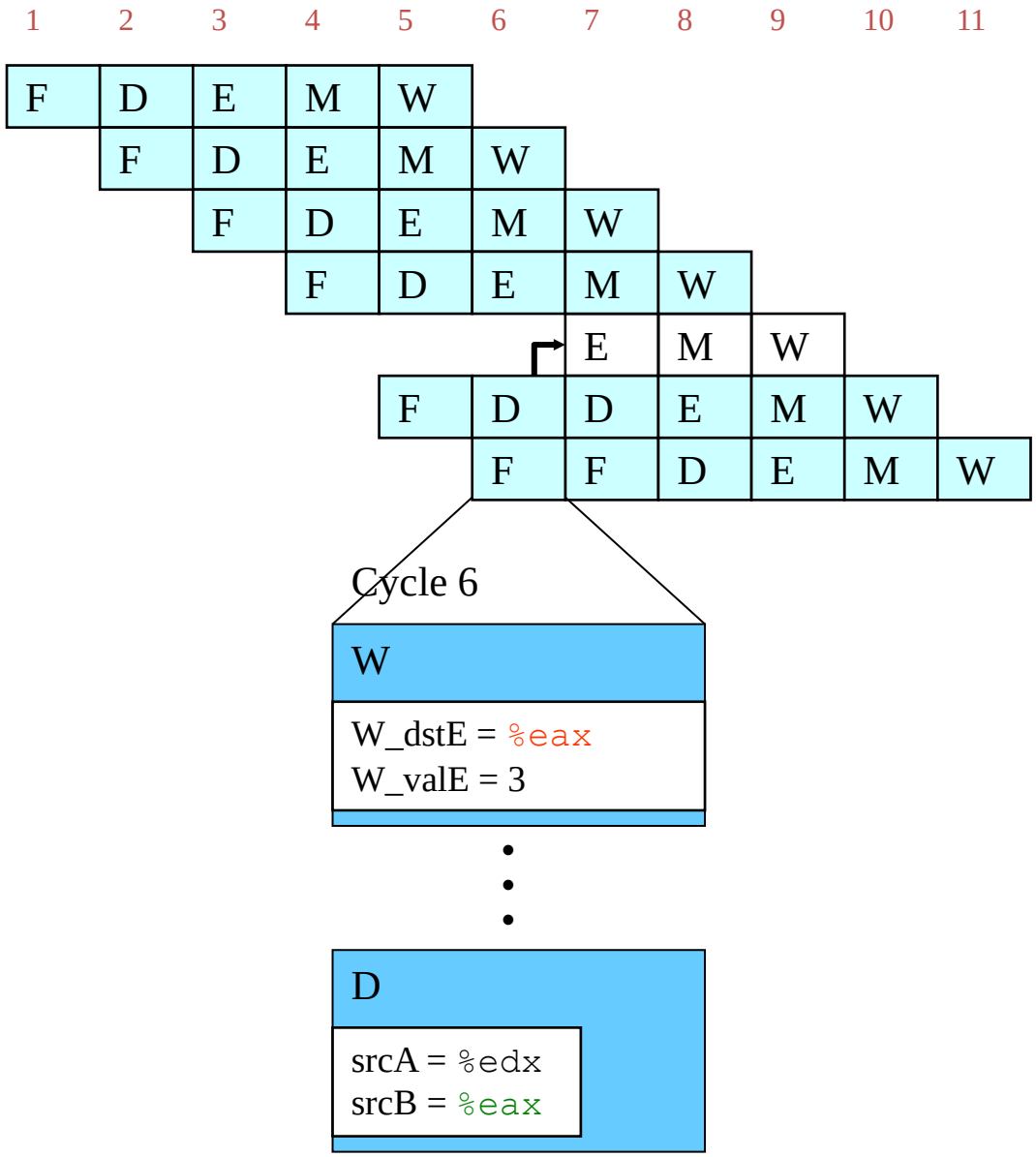
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```



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

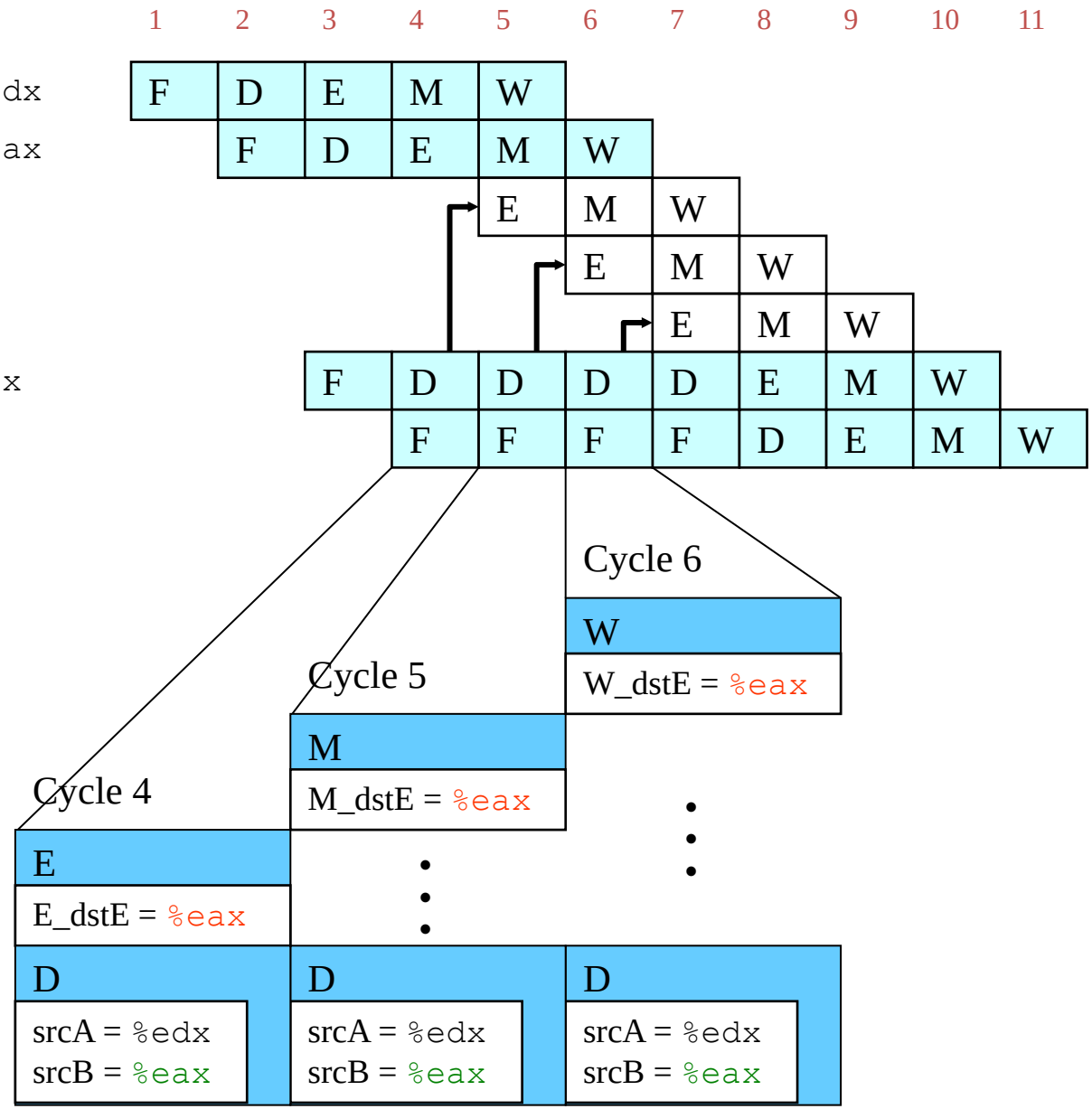
Detecting Stall Condition

```
# 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
```

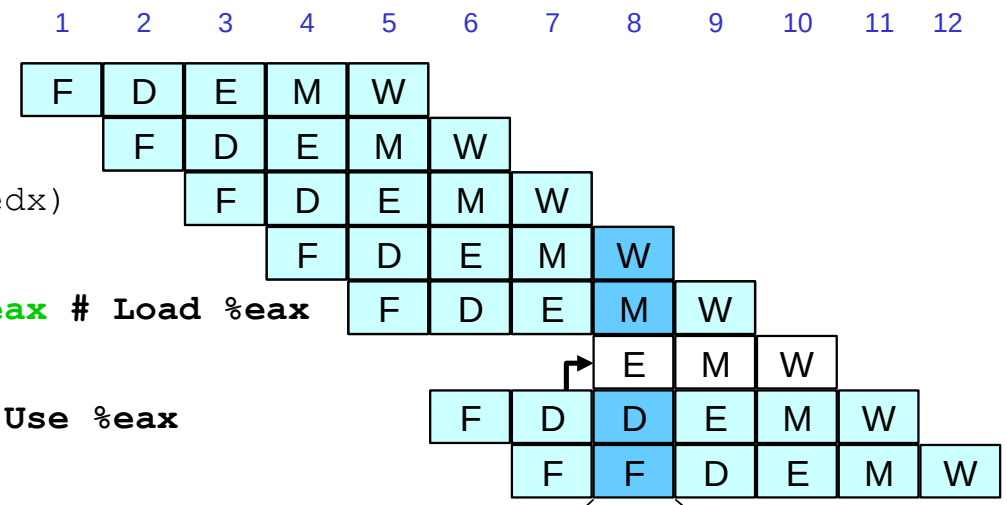
Cycle 8

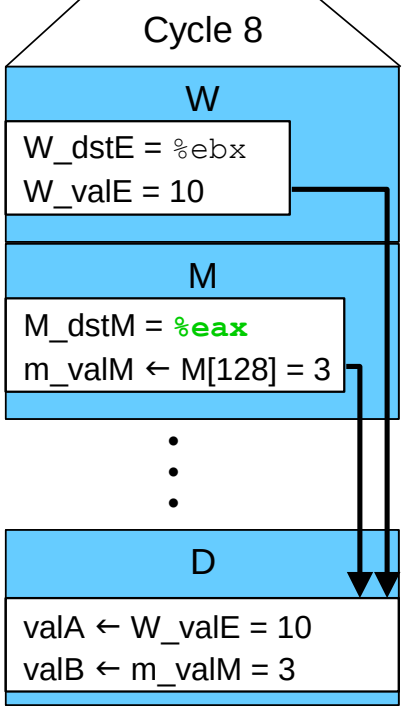
Write Back	*bubble*
Memory	*bubble*
Execute	`0x00c: addl %edx,%eax`
Decode	`0x00e: halt`
Fetch	

- 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

Avoiding Load/Use Hazard

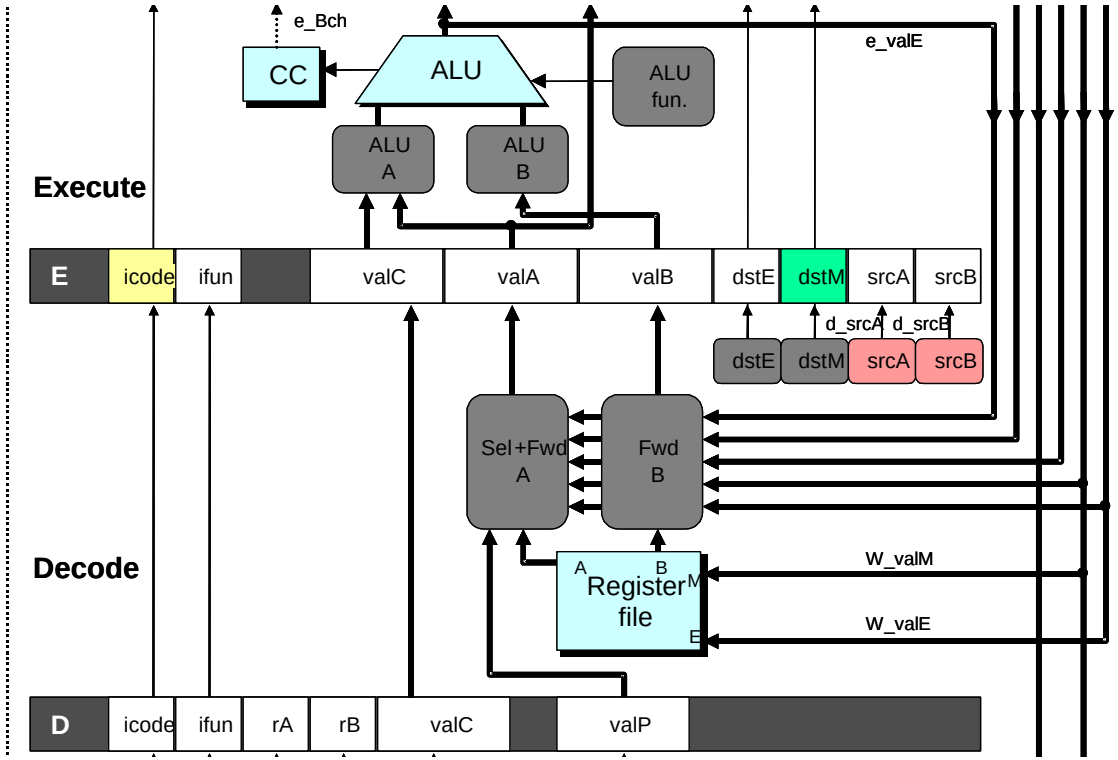
```
# demo-luh.ys
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
       bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```





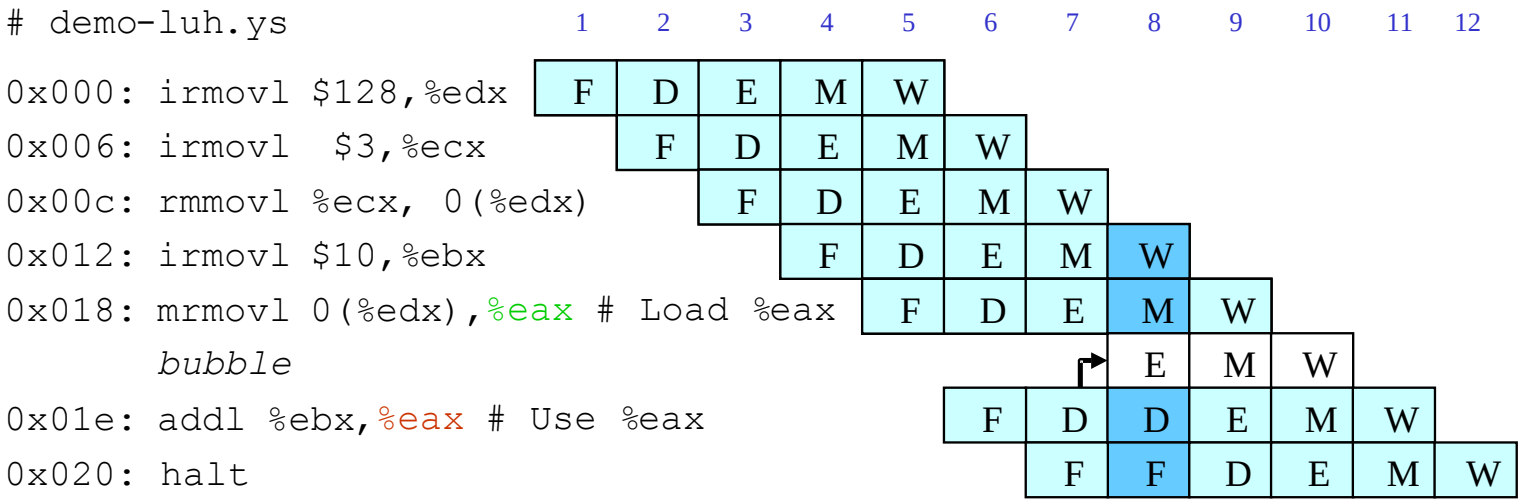
- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }

Control for Load/Use Hazard



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

Branch Misprediction Example

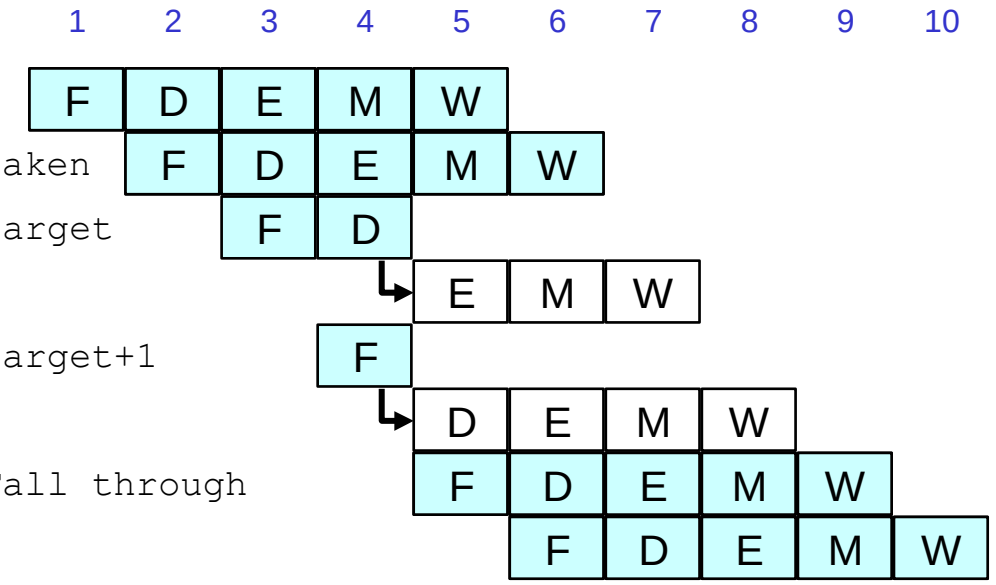
demo-j.ys

```
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 8 instructions

Handling Misprediction

```
# demo-j.ys
0x000:     xorl %eax,%eax
0x002:     jne target # Not taken
0x011: t:  irmovl $2,%edx # Target
           bubble
0x017:     irmovl $3,%ebx # Target+1
           bubble
0x007:     irmovl $1,%eax # Fall through
0x00d:     nop
```



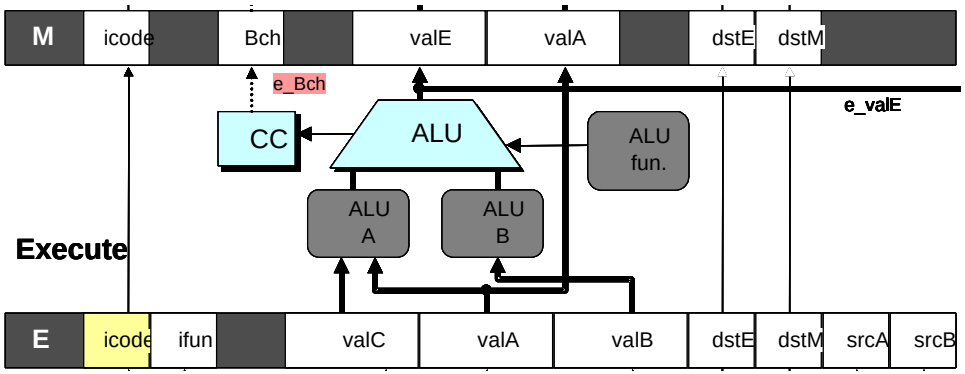
Predict branch as taken

- Fetch 2 instructions at target

Cancel 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

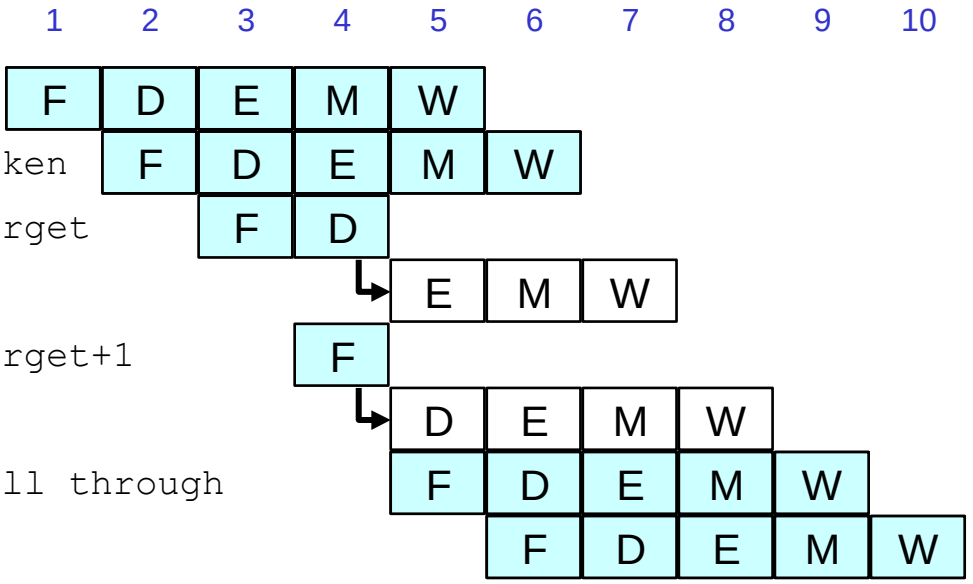
Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	E_icode = IJXX & !e_Bch

Control for Misprediction

```
# demo-j.ys
0x000:     xorl %eax,%eax
0x002:     jne target # Not taken
0x011: t:  irmovl $2,%edx # Target
           bubble
0x017:     irmovl $3,%ebx # Target+1
           bubble
0x007:     irmovl $1,%eax # Fall through
0x00d:     nop
```



Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

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

Pipeline Summary

- Data Hazards
  - Most handled by forwarding
    - No performance penalty
  - Load/use hazard requires one cycle stall
- Control Hazards
  - Cancel instructions when detect mispredicted branch
    - Two clock cycles wasted
  - Stall fetch stage while `ret` passes through pipeline
    - Three clock cycles wasted
- Control Combinations
  - Must analyze carefully
  - Multiple special cases simultaneously!