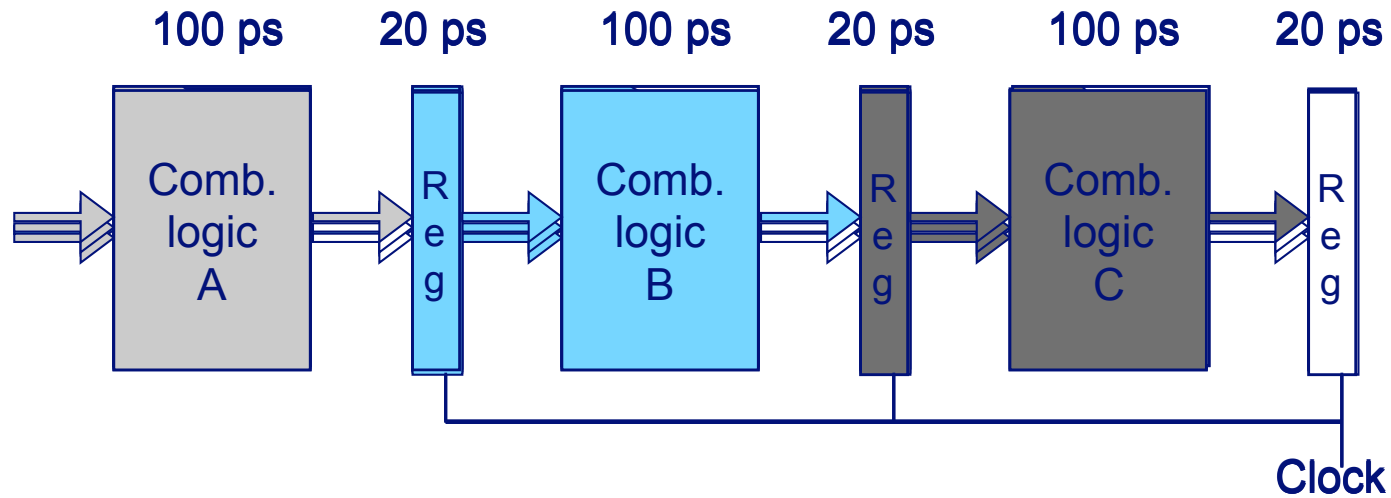
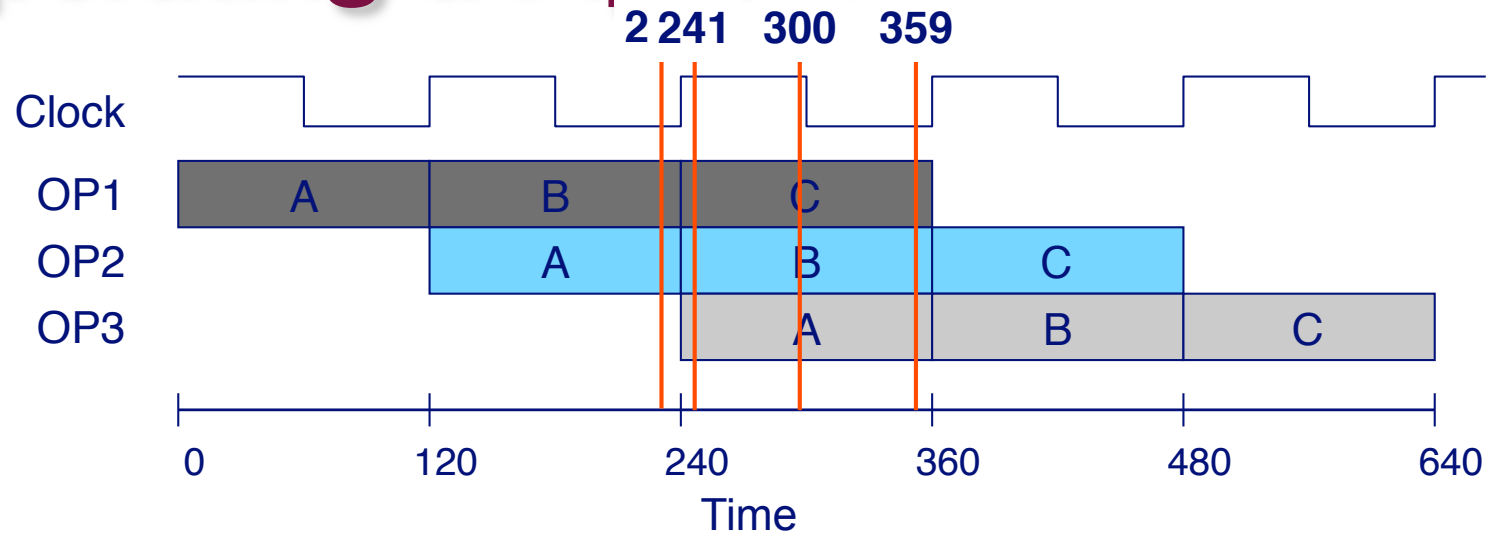
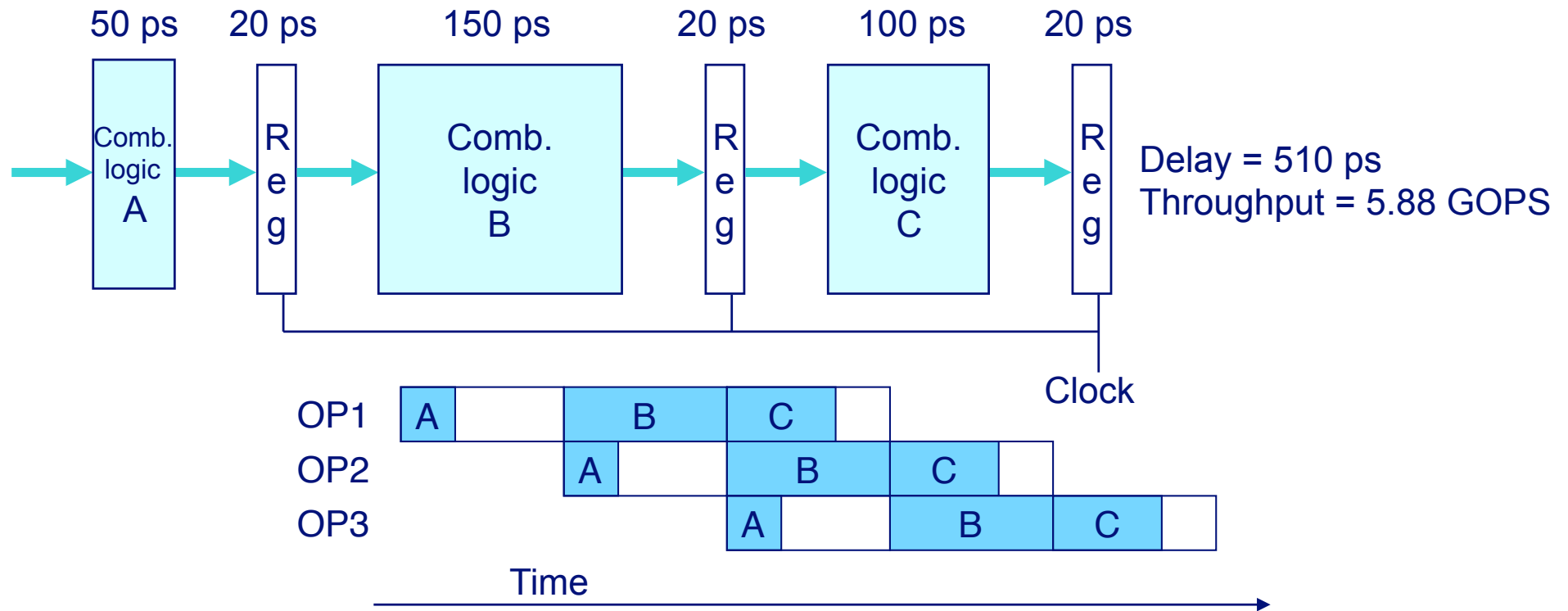


Operating a Pipeline

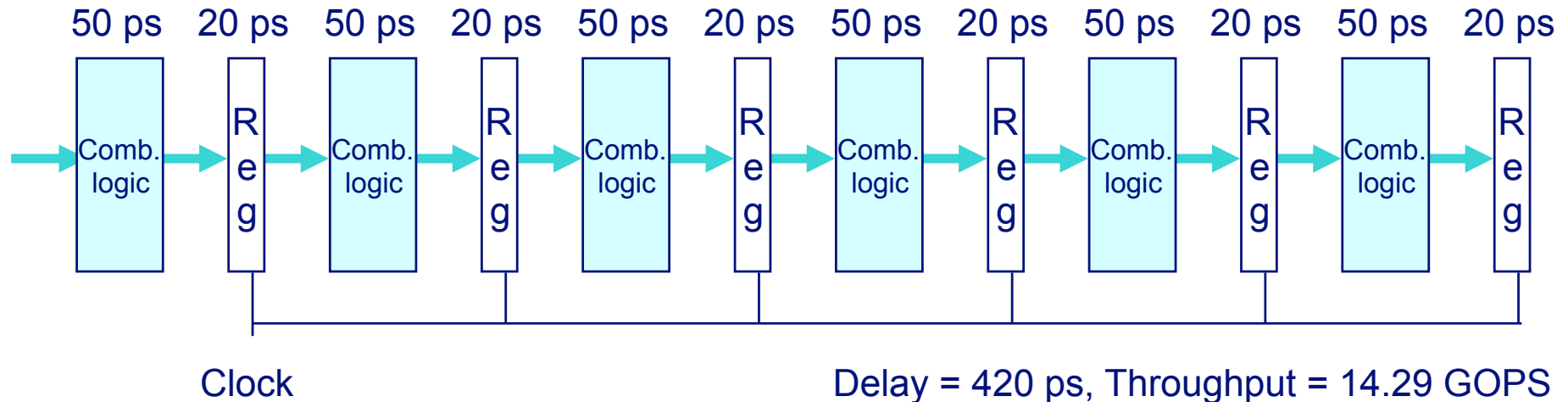


Limitations: Nonuniform Delays



- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
 - 1-stage pipeline: 6.25%
 - 3-stage pipeline: 16.67%
 - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

SEQ+ Hardware

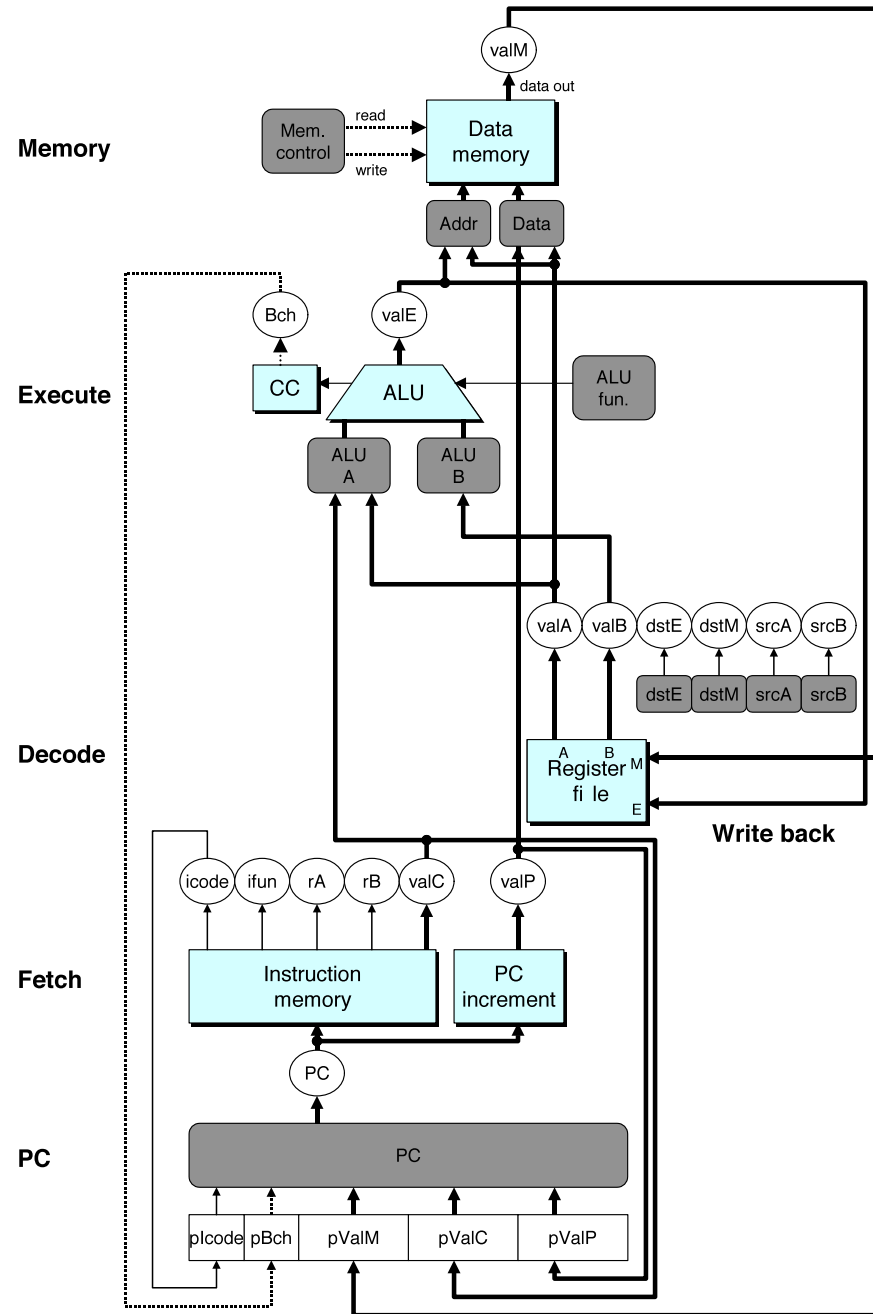
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

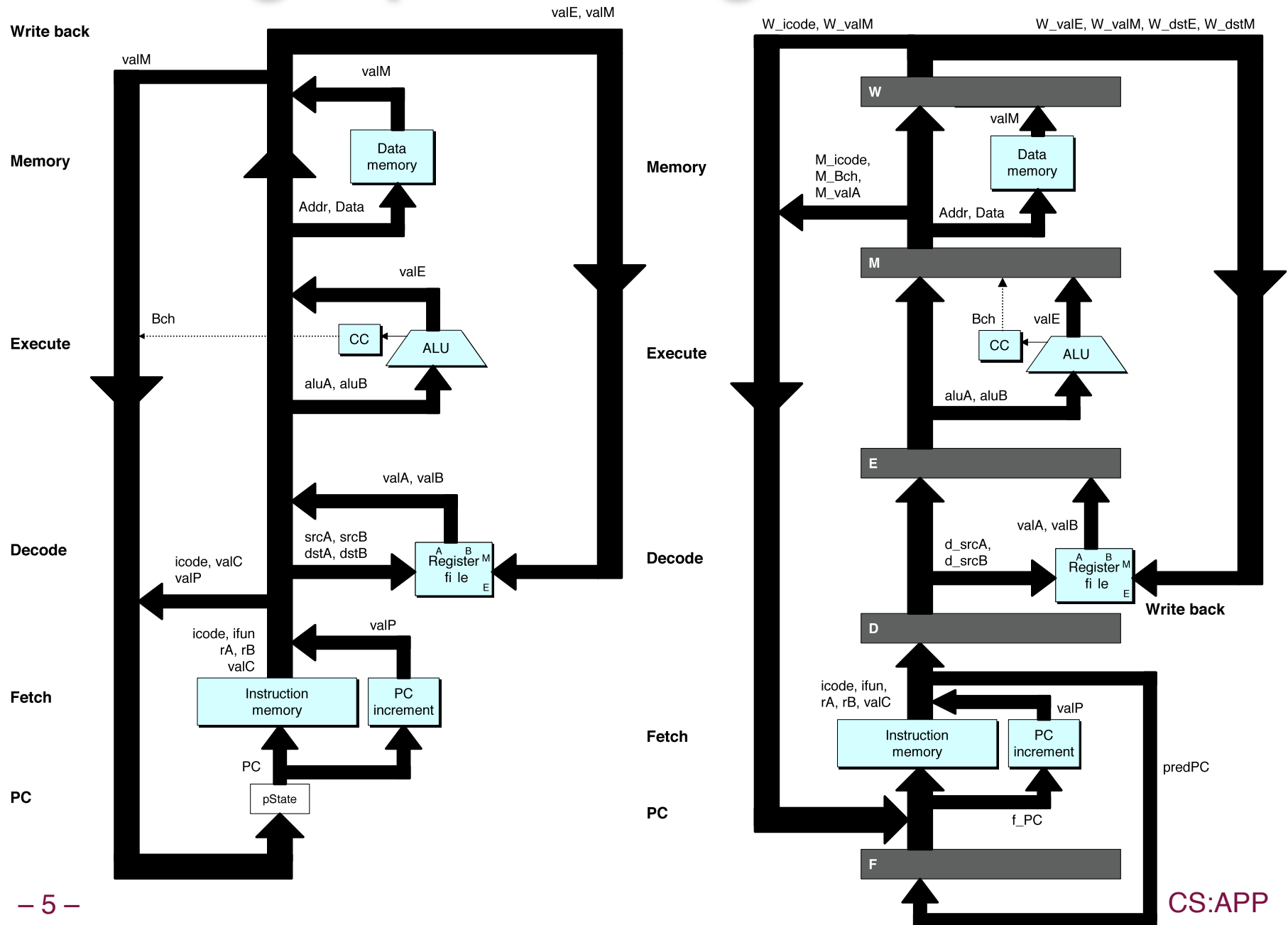
- Task is to select PC for current instruction
- Based on results computed by previous instruction

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

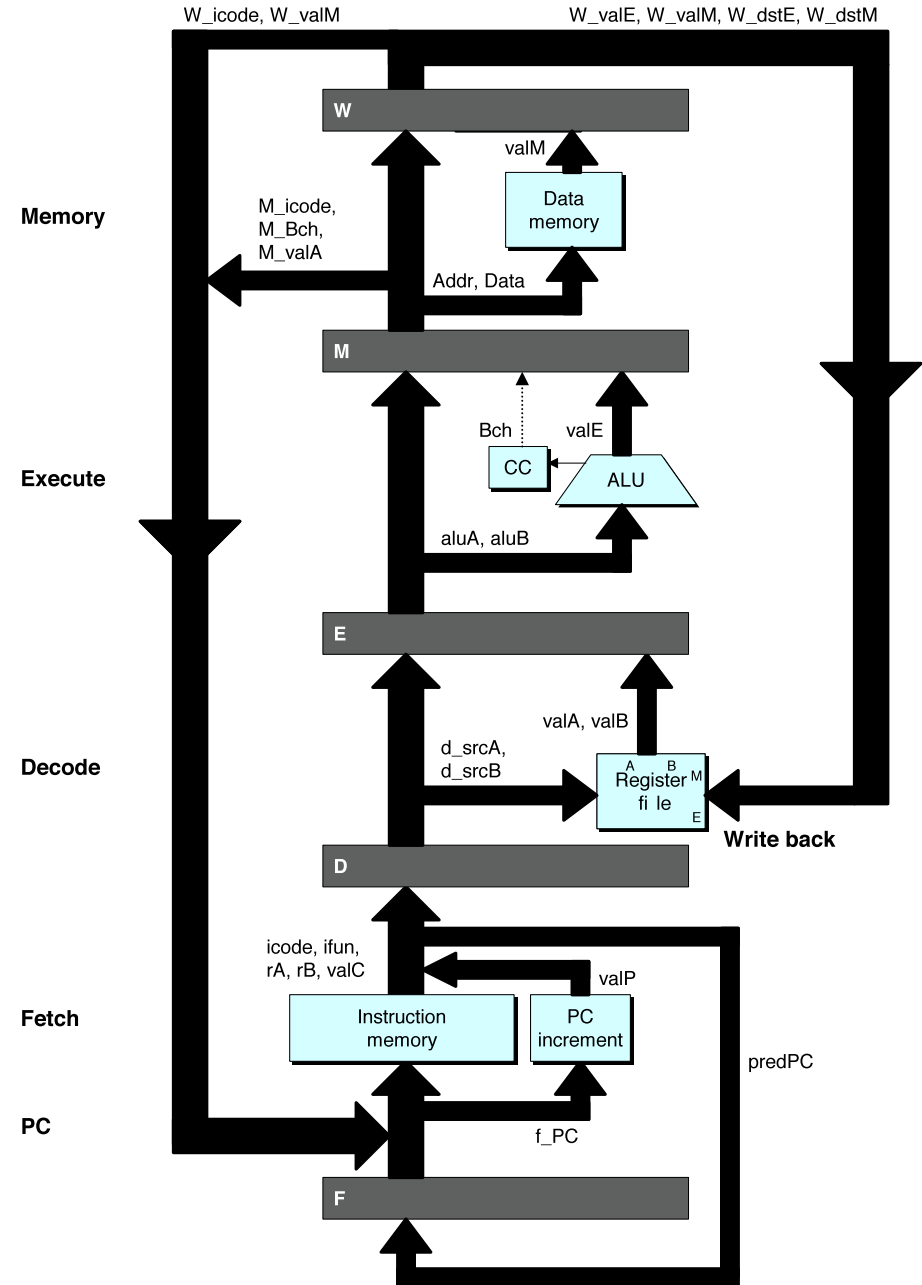
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file

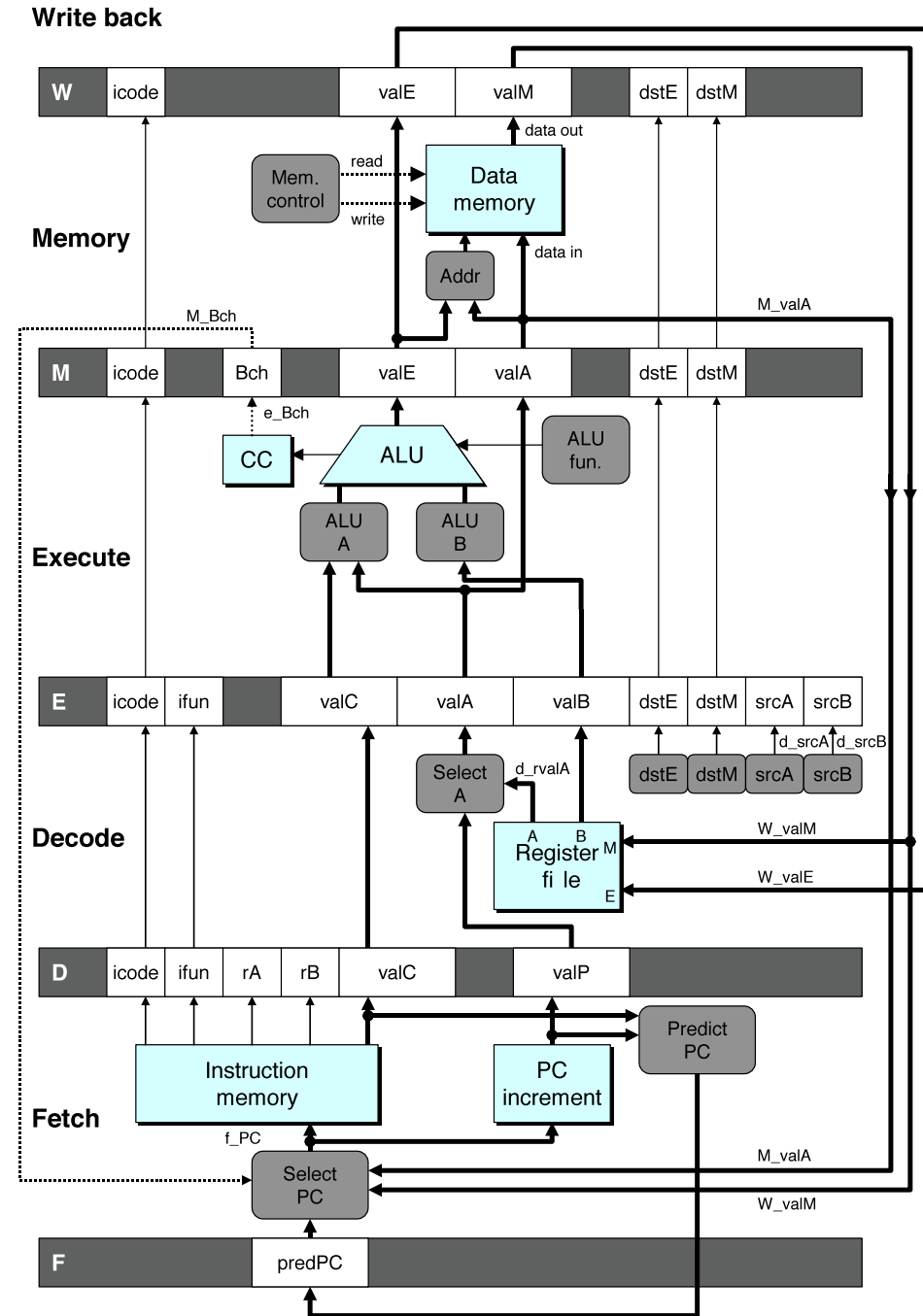


PIPE- Hardware

- 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



Feedback Paths

Predicted PC

- Guess value of next PC

Branch information

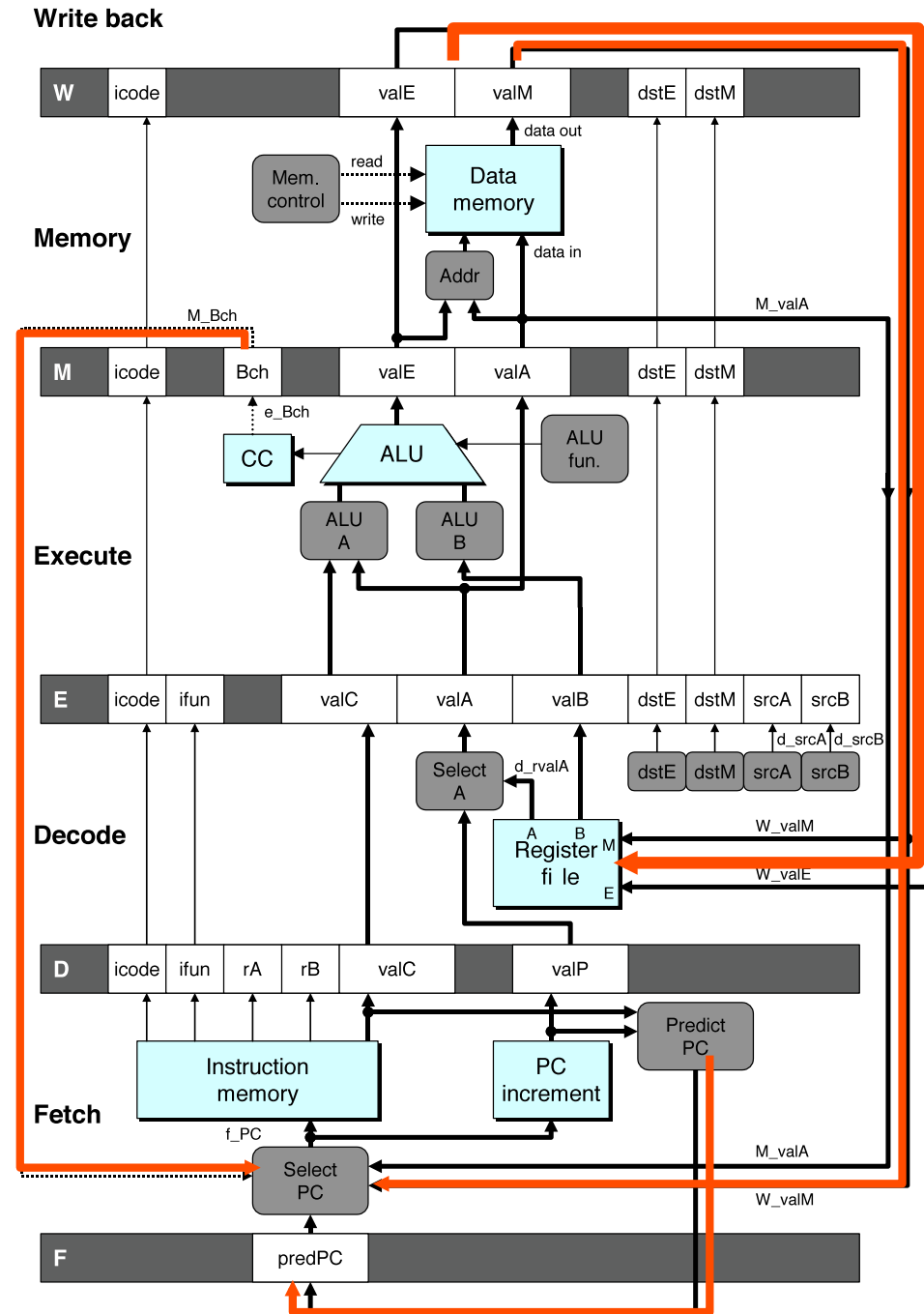
- Jump taken/not-taken
- Fall-through or target address

Return point

- Read from memory

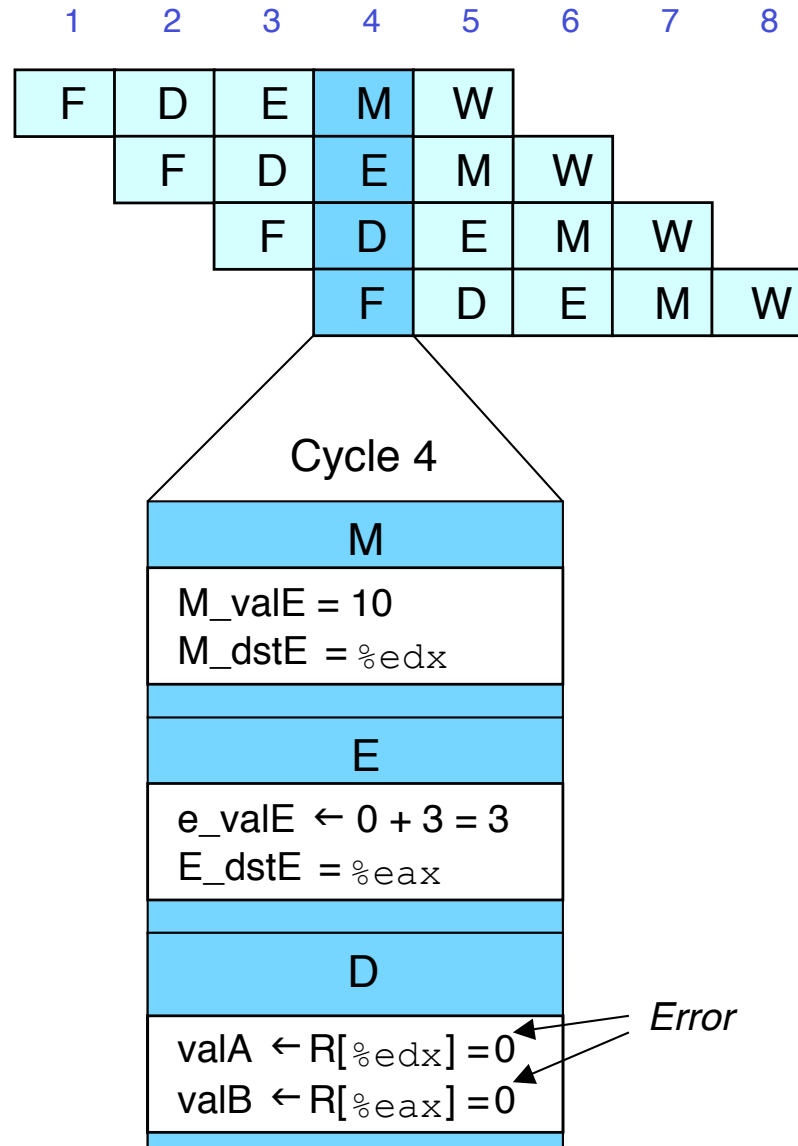
Register updates

- To register file write ports



Data Dependencies: No Nop

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



Data Dependencies: 2 Nop's

demo-h2.y

0x000: irmovl \$10,%edx

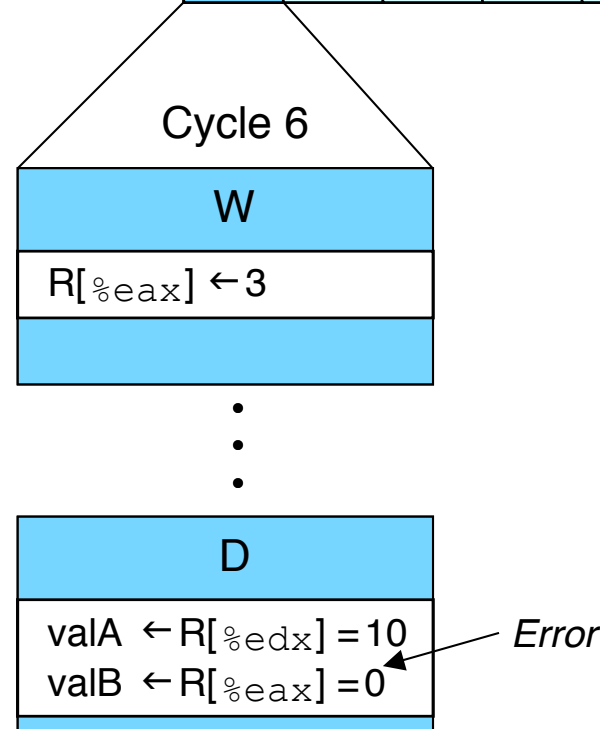
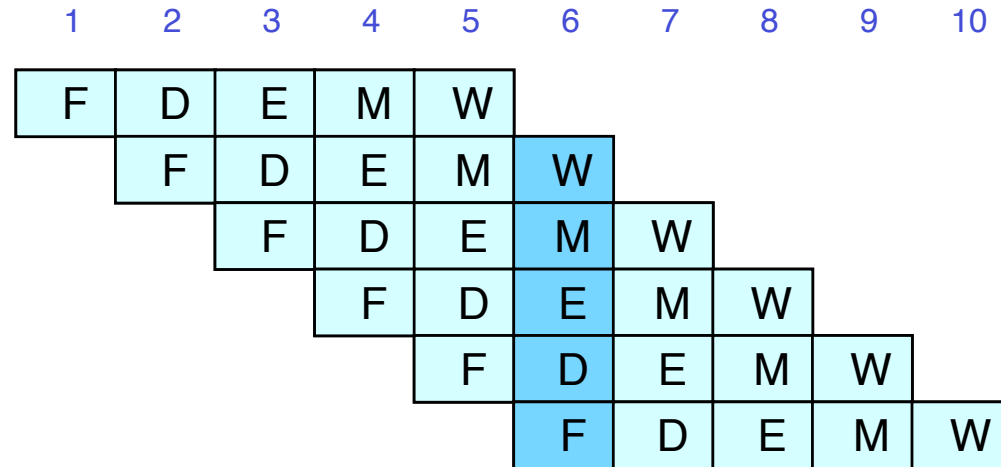
0x006: irmovl \$3,%eax

0x00c: nop

0x00d: nop

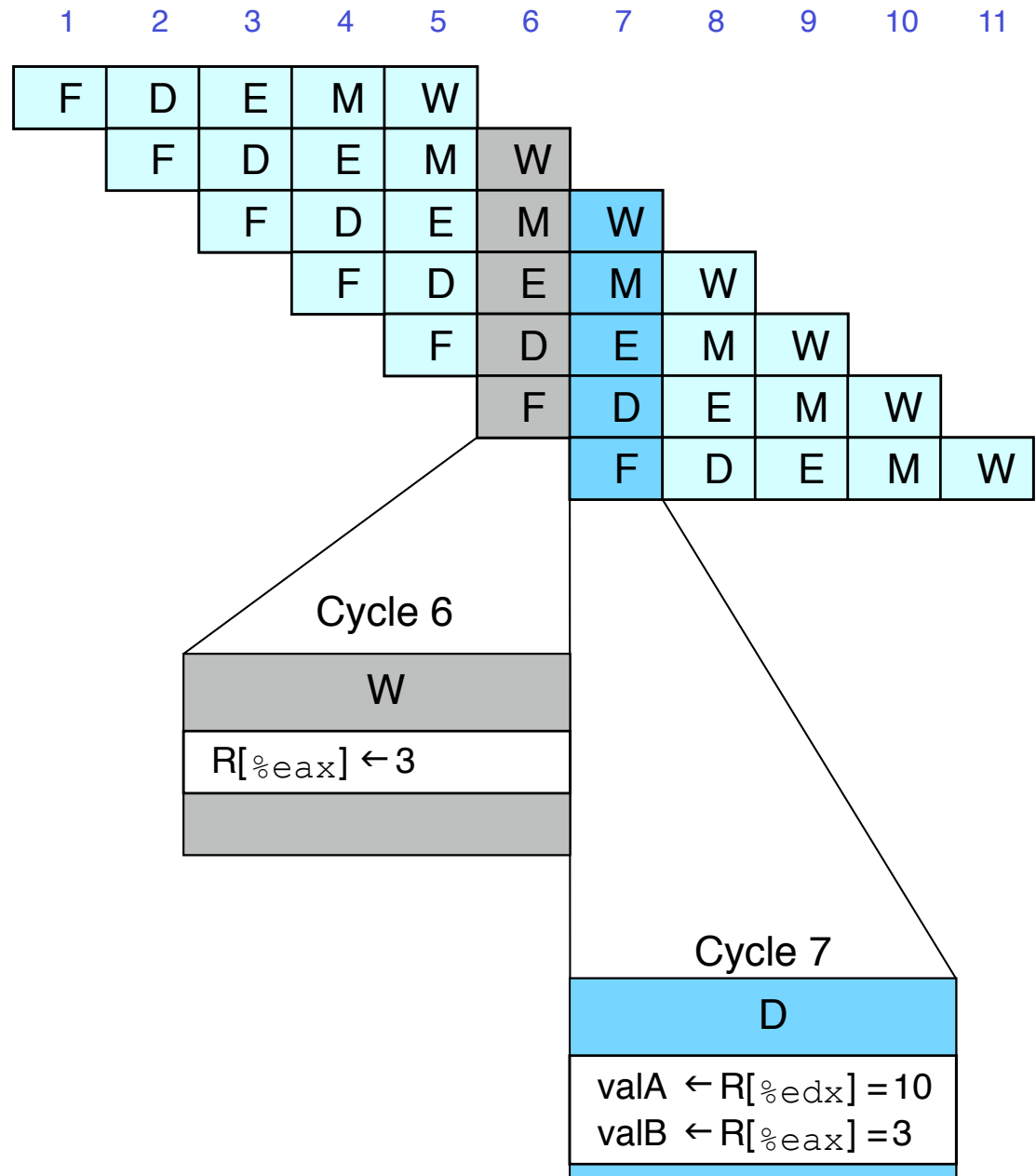
0x00e: addl %edx,%eax

0x010: halt



Data Dependencies: 3 Nop's

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



Stalling for Data Dependencies

```
# demo-h2.y
```

```
0x000: irmovl $10,%edx
```

```
0x006: irmovl $3,%eax
```

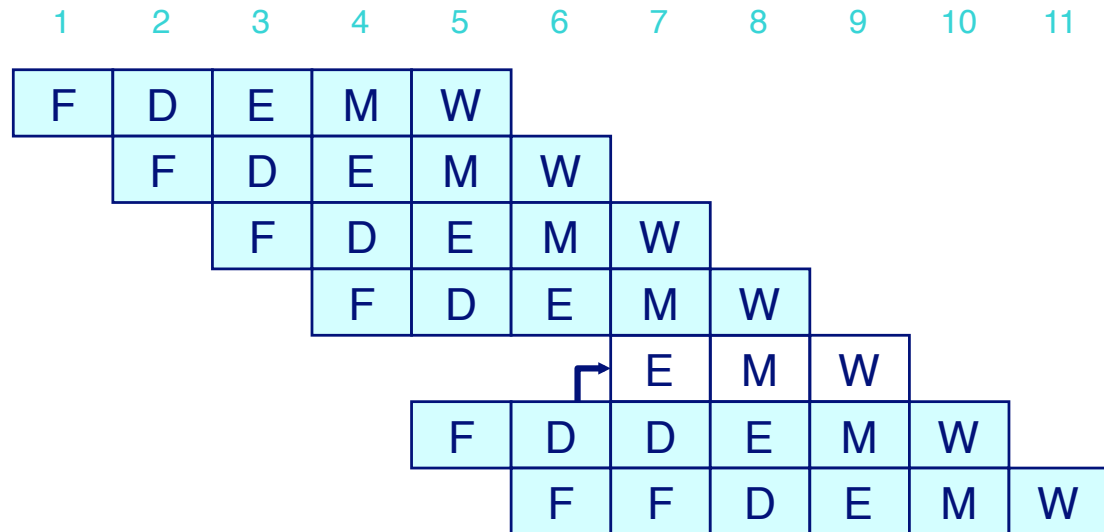
```
0x00c: nop
```

```
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

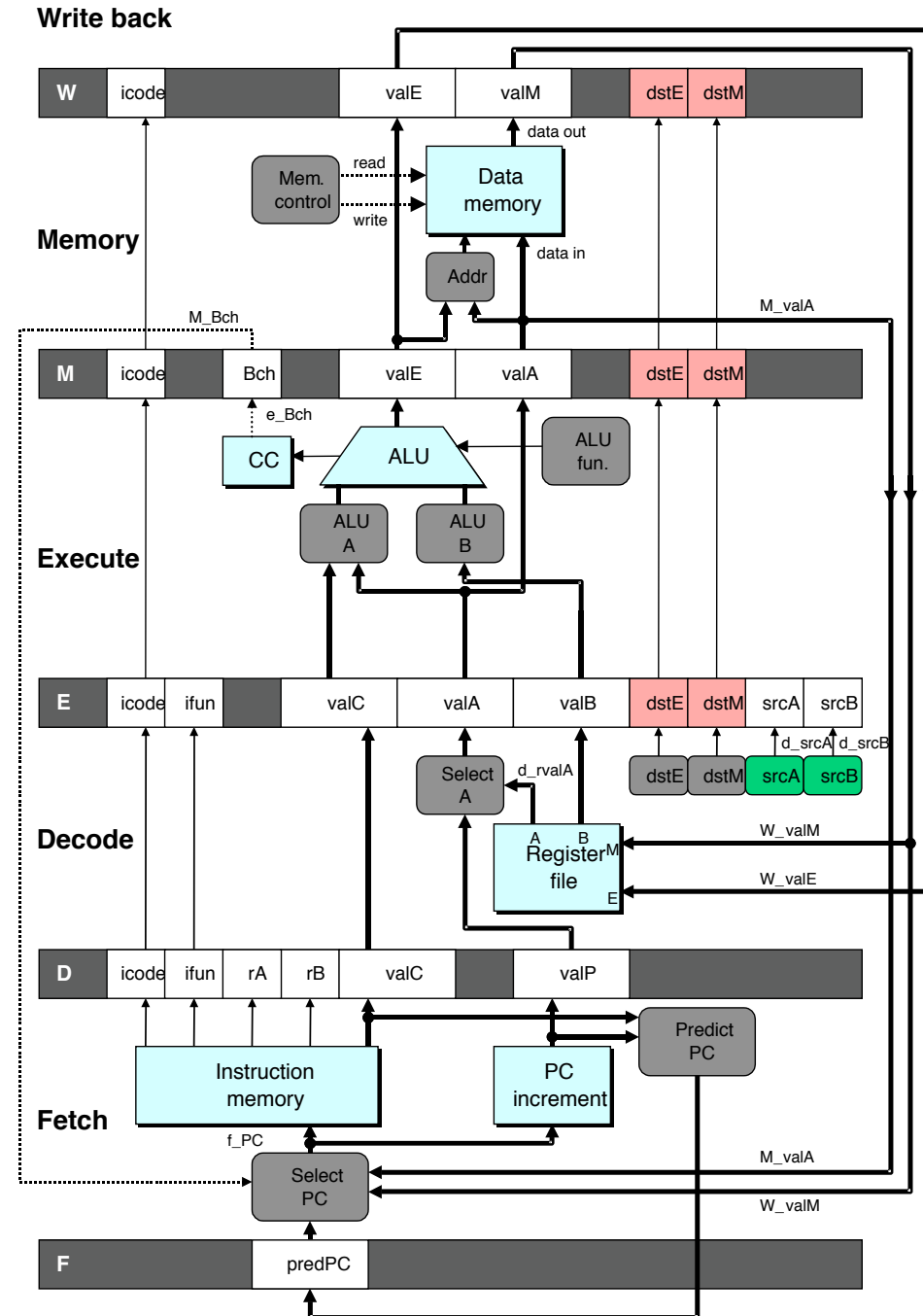
- srcA and srcB of current instruction in decode stage

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
```

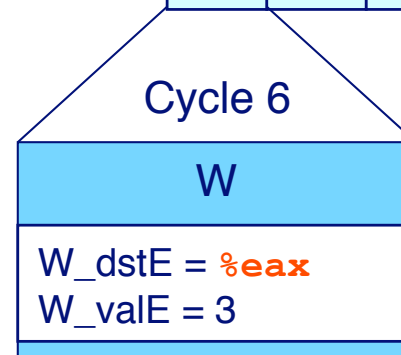
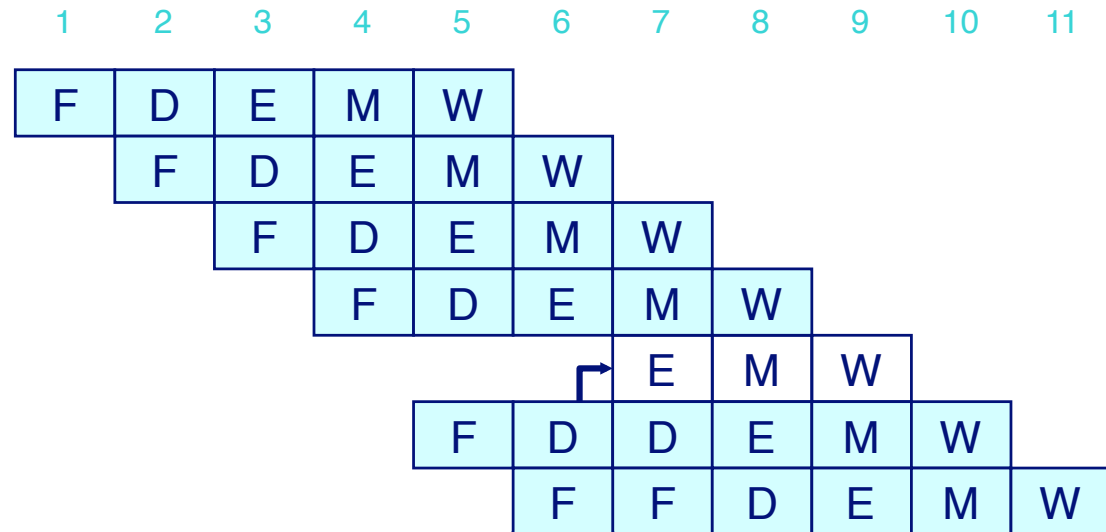
```
0x00c: nop
```

```
0x00d: nop
```

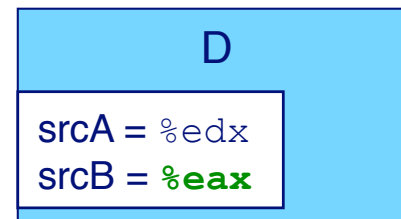
bubble

```
0x00e: addl %edx,%eax
```

```
0x010: halt
```

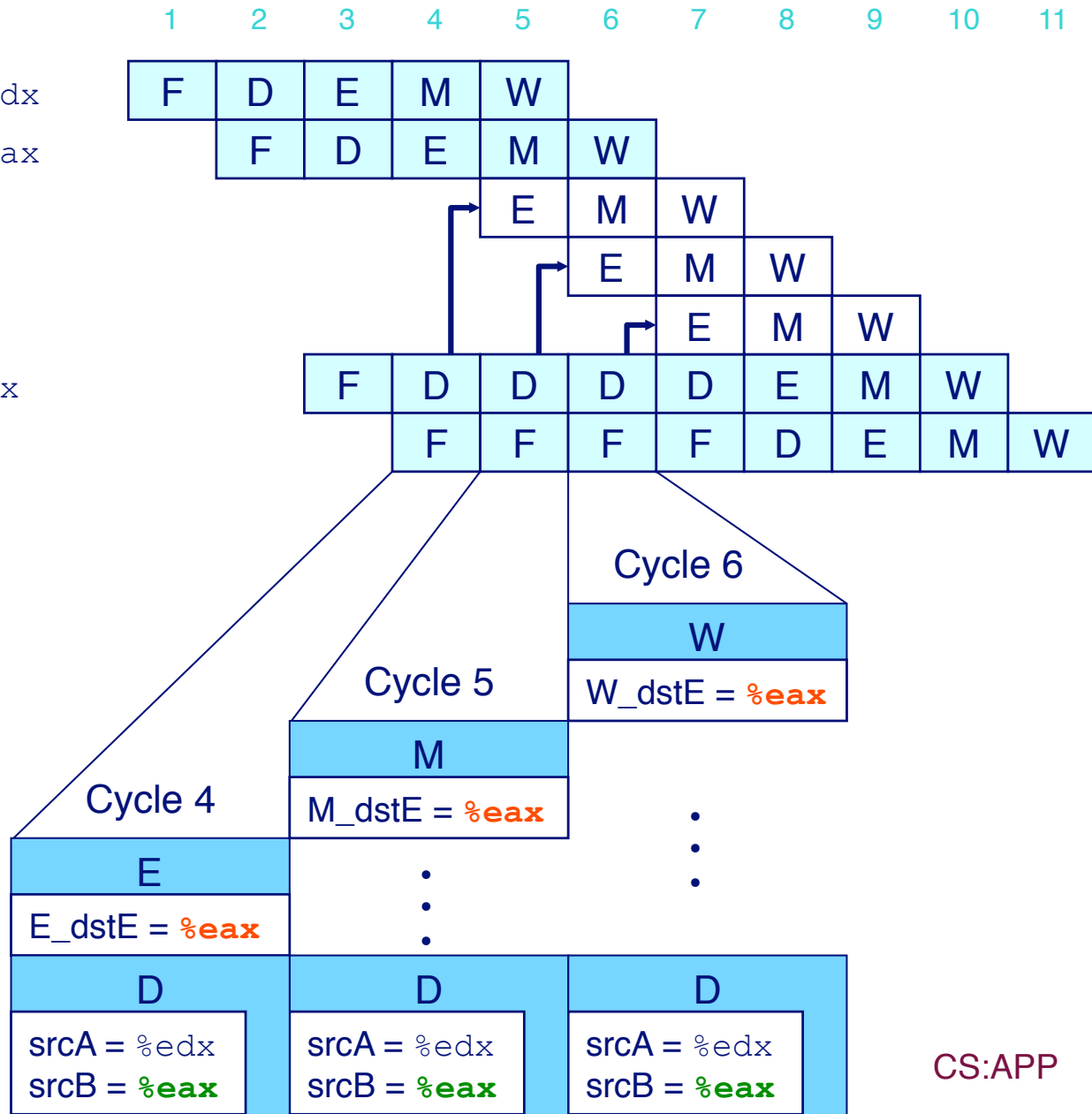


⋮



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
```

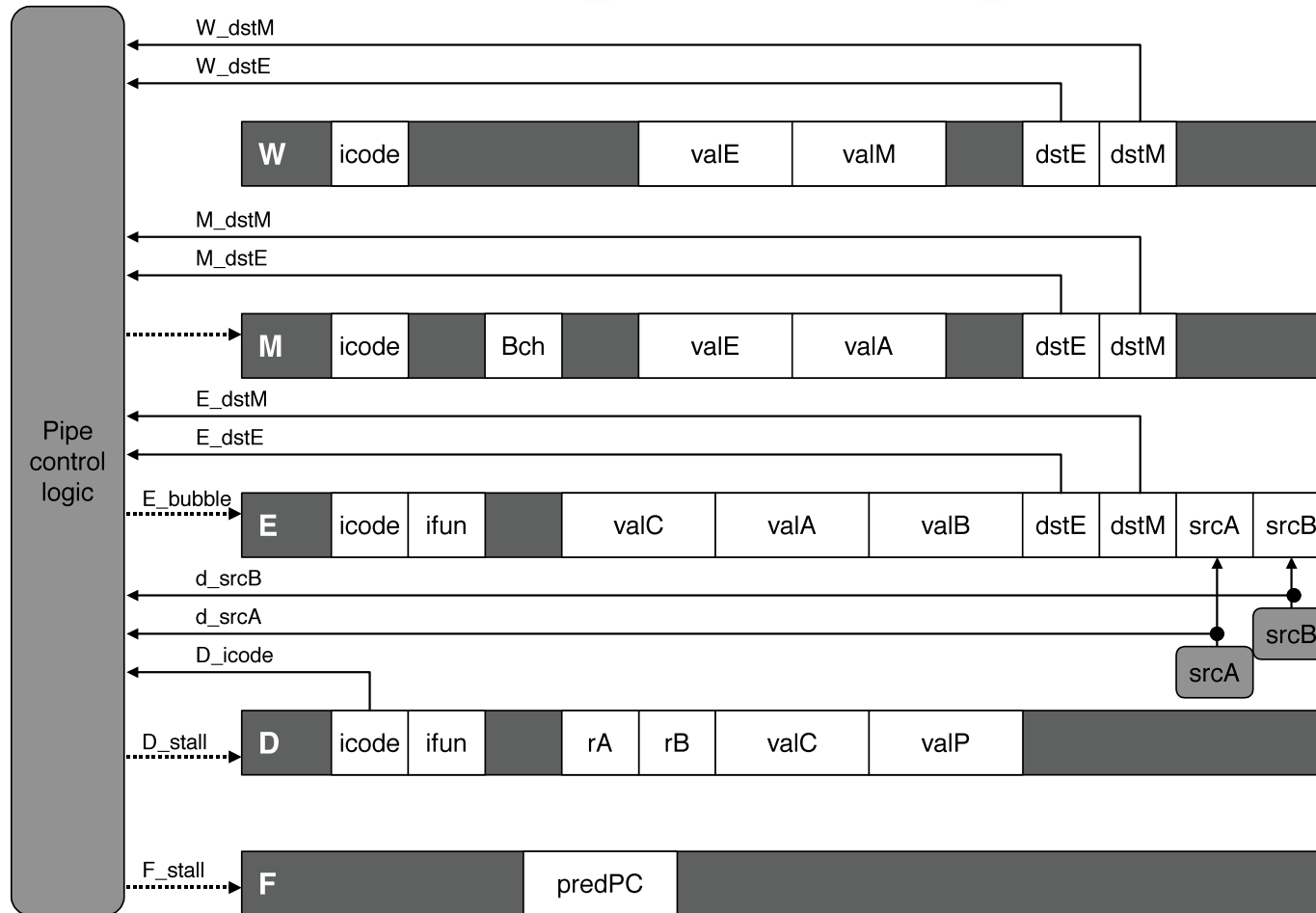
```
0x00e: halt
```

Cycle 8

| | |
|------------|-----------------------|
| Write Back | <i>bubble</i> |
| Memory | <i>bubble</i> |
| 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

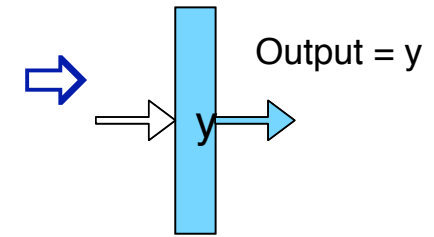
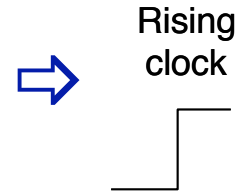
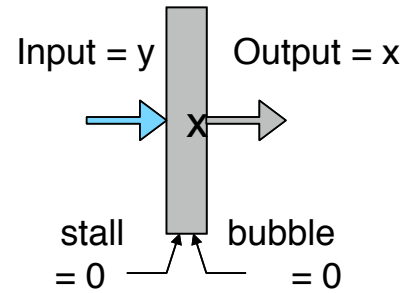


Pipeline Control

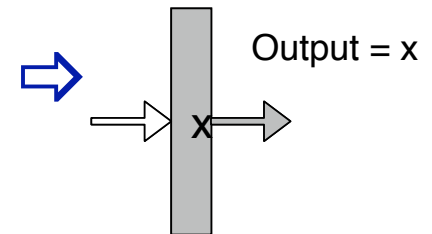
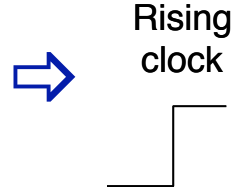
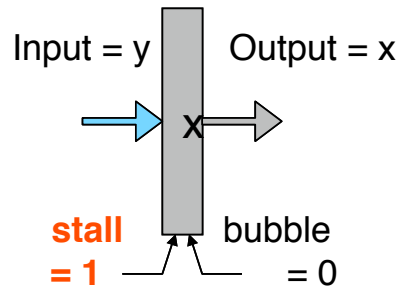
- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

Pipeline Register Modes

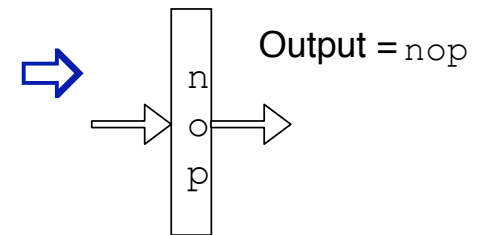
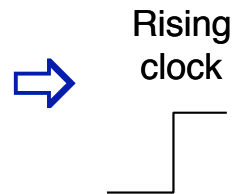
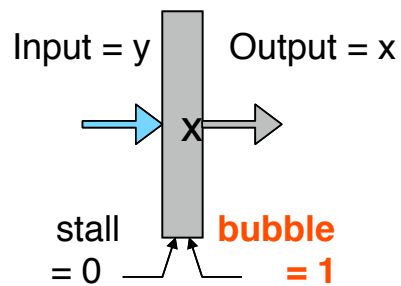
Normal



Stall



Bubble



Data Forwarding

Naïve Pipeline

- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
 - Needs to be in register file at start of stage

Observation

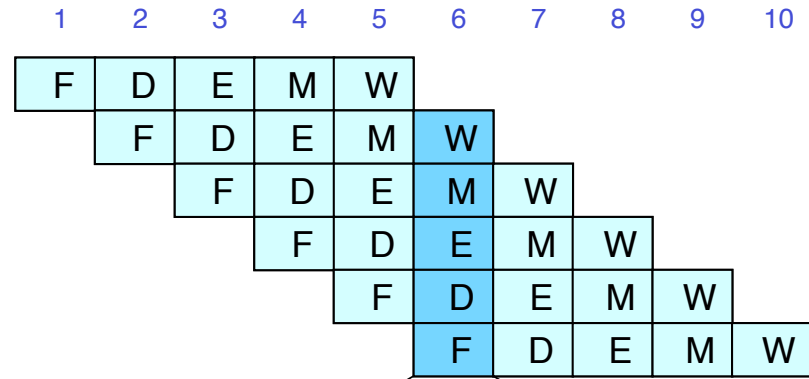
- Value generated in execute or memory stage

Trick

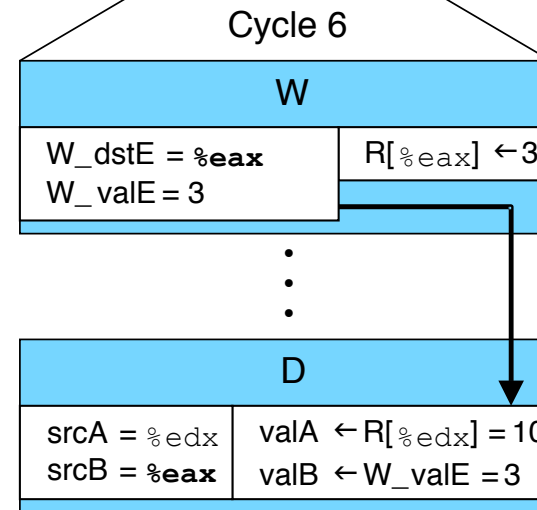
- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

Data Forwarding Example

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



- **irmovl** in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage



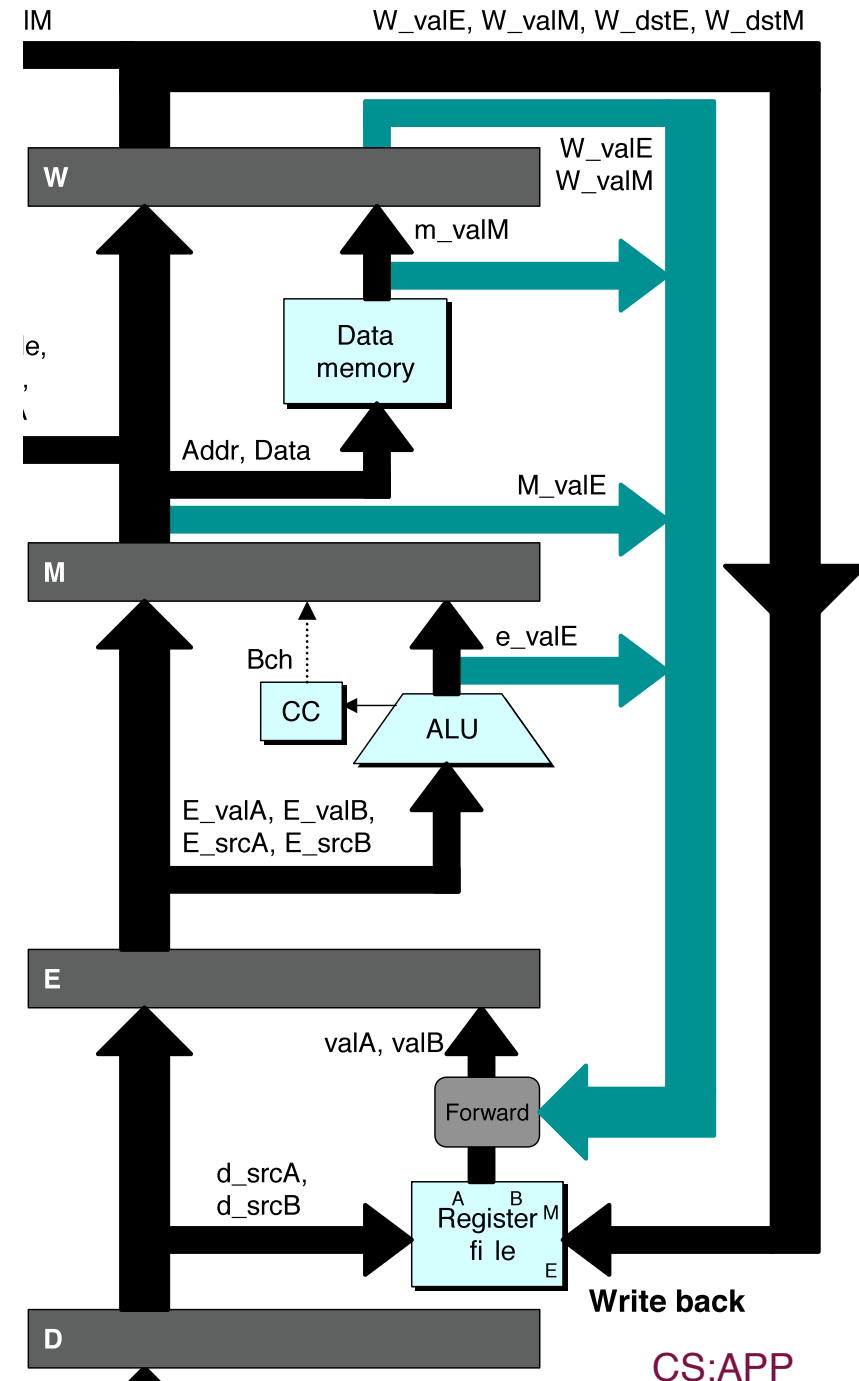
Bypass Paths

Decode Stage

- Forwarding logic selects valA and valB
- Normally from register file
- Forwarding: get valA or valB from later pipeline stage

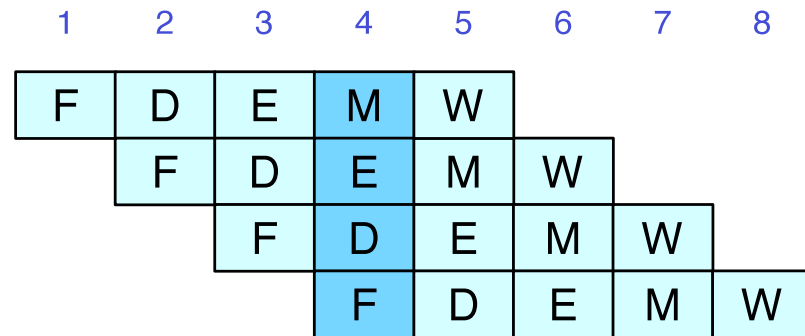
Forwarding Sources

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



Data Forwarding Example #2

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

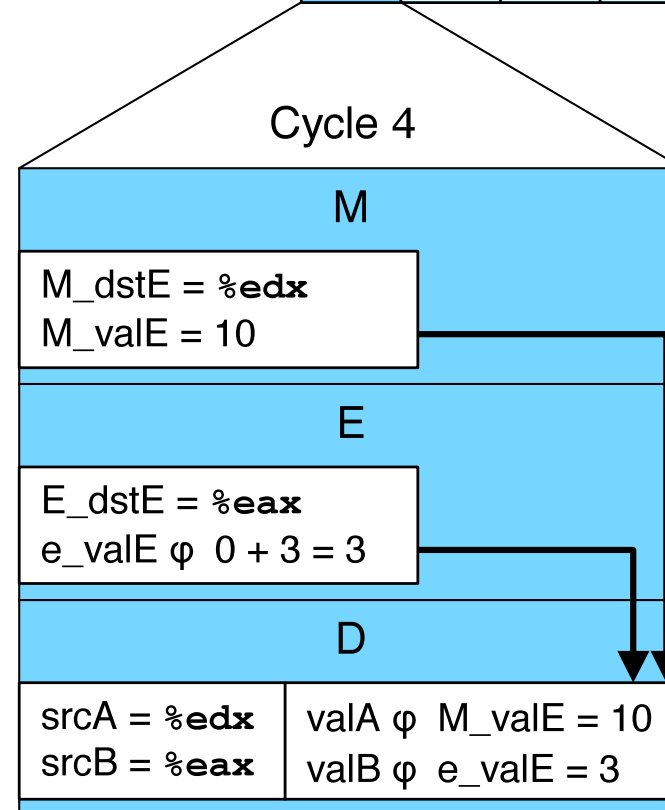


Register %edx

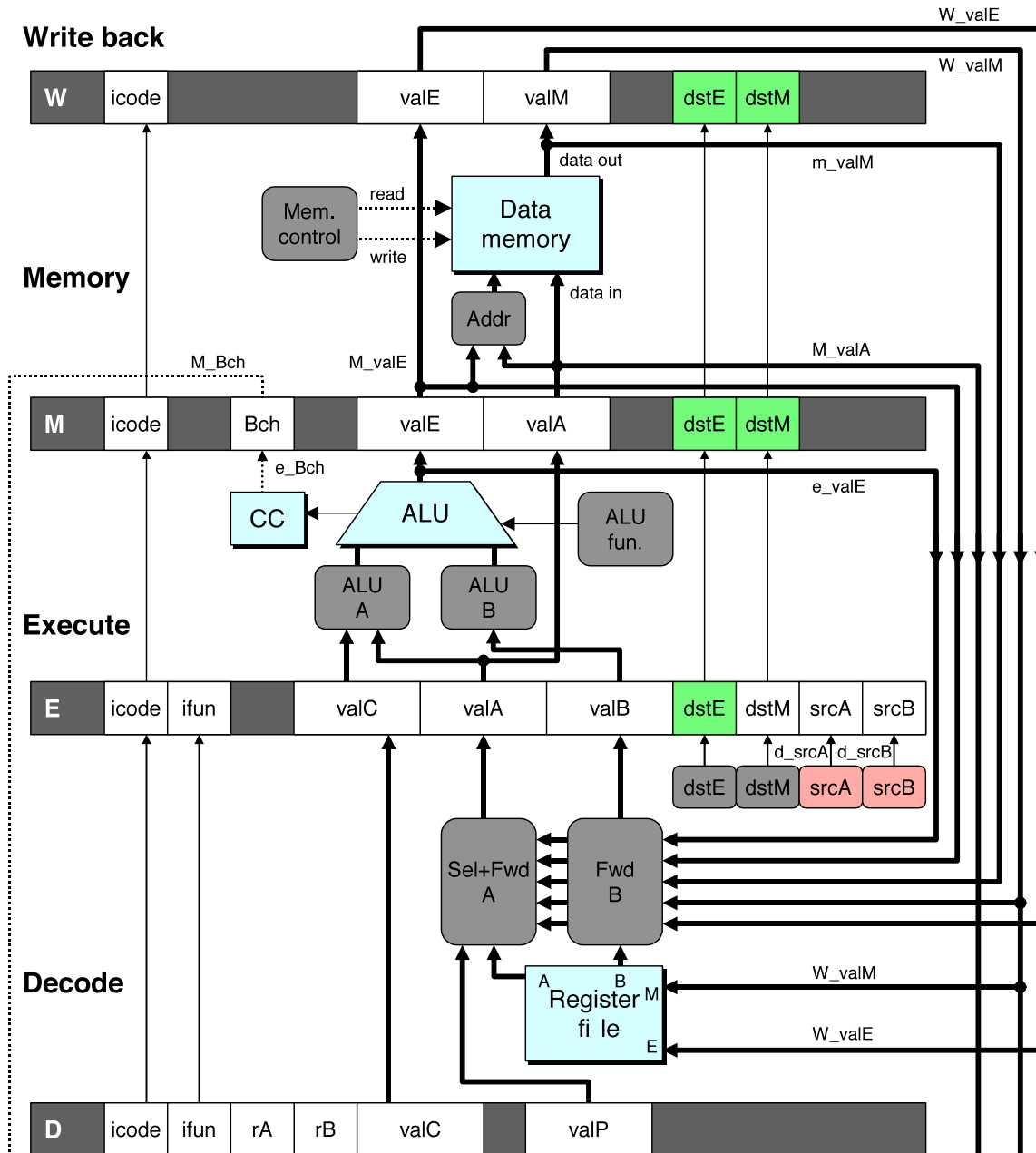
- Generated by ALU during previous cycle
- Forward from memory as valA

Register %eax

- Value just generated by ALU
- Forward from execute as valB

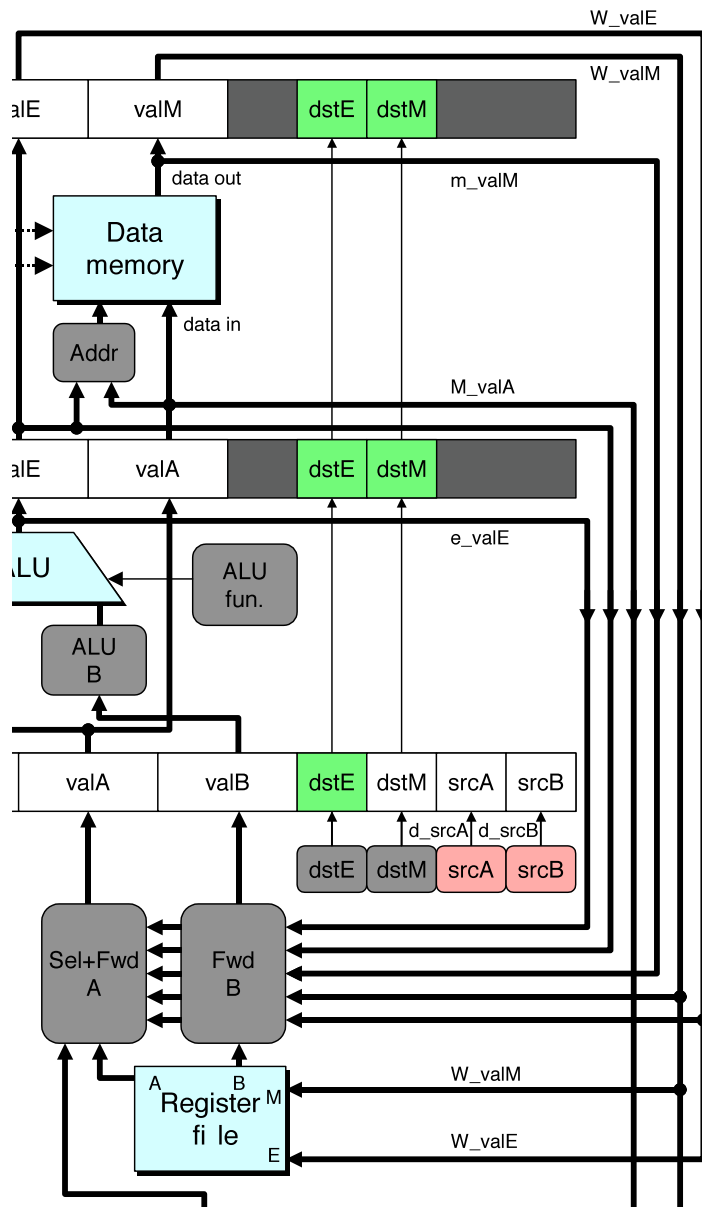


Implementing Forwarding



- Add additional feedback paths from E, M, and W pipeline registers into decode stage
- Create logic blocks to select from multiple sources for valA and valB in decode stage

Implementing Forwarding



```

## What should be the A value?
int new_E_valA = [
    # Use incremented PC
    D_icode in { ICALL, IJXX } : D_valP;
    # Forward valE from execute
    d_srcA == E_dstE : e_valE;
    # Forward valM from memory
    d_srcA == M_dstM : m_valM;
    # Forward valE from memory
    d_srcA == M_dstE : M_valE;
    # Forward valM from write back
    d_srcA == W_dstM : W_valM;
    # Forward valE from write back
    d_srcA == W_dstE : W_valE;
    # Use value read from register file
    1 : d_rvalA;
];

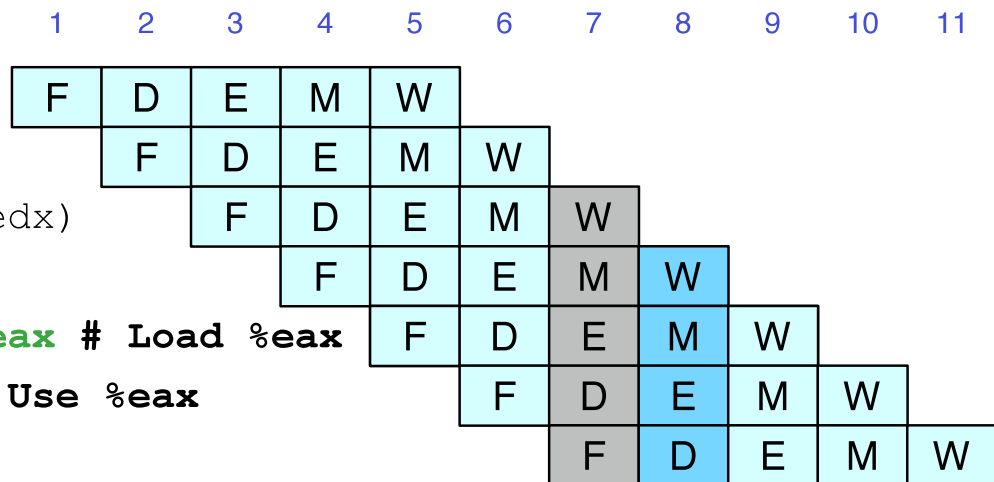
```


Limitation of Forwarding

demo-luh.js

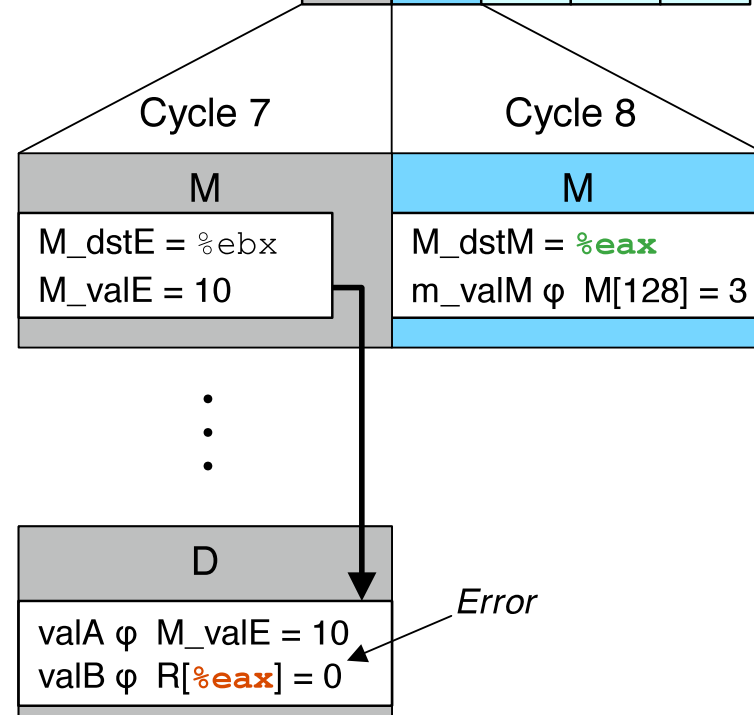
```

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

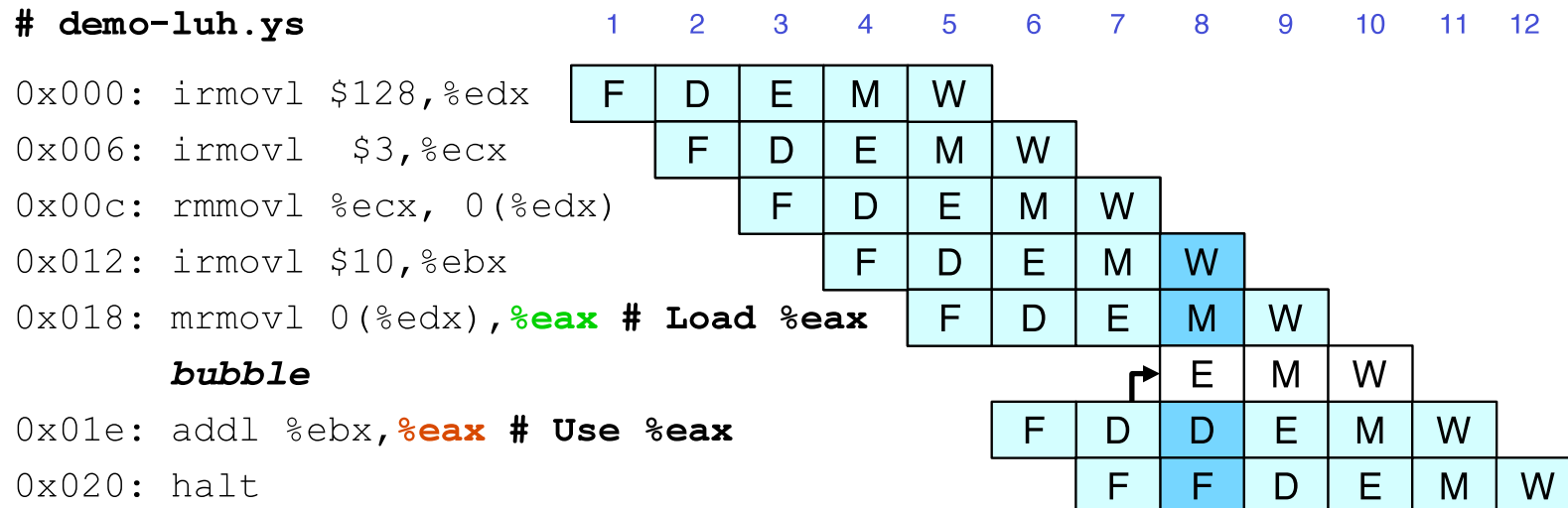


Load-use dependency

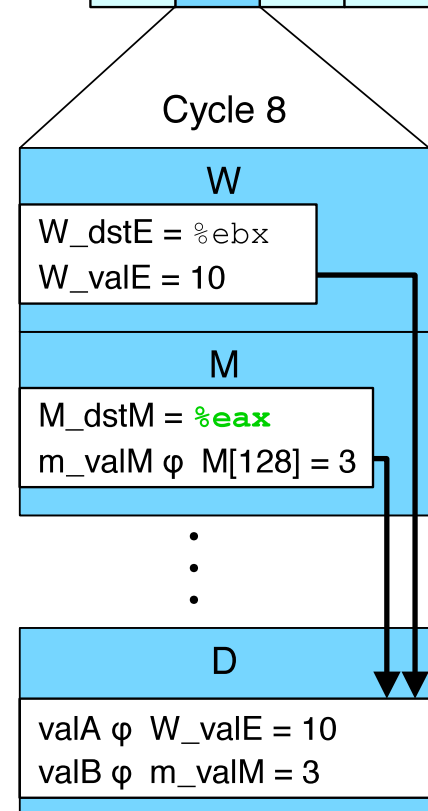
- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8



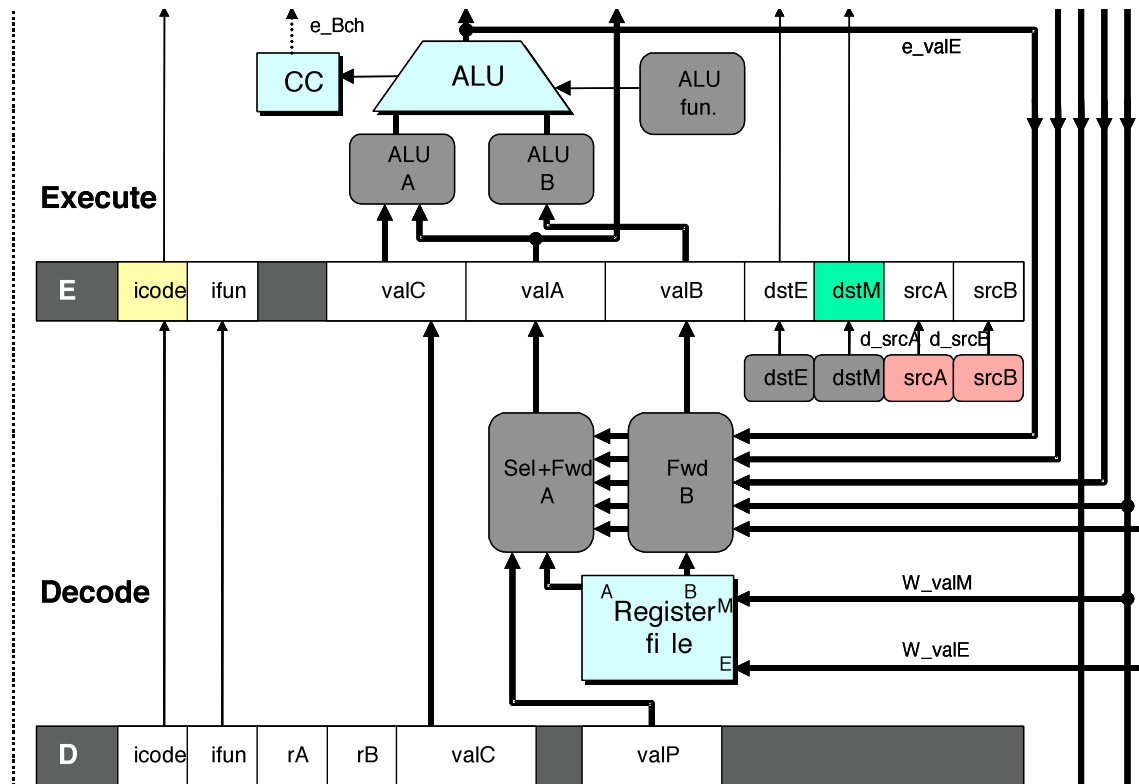
Avoiding Load/Use Hazard



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

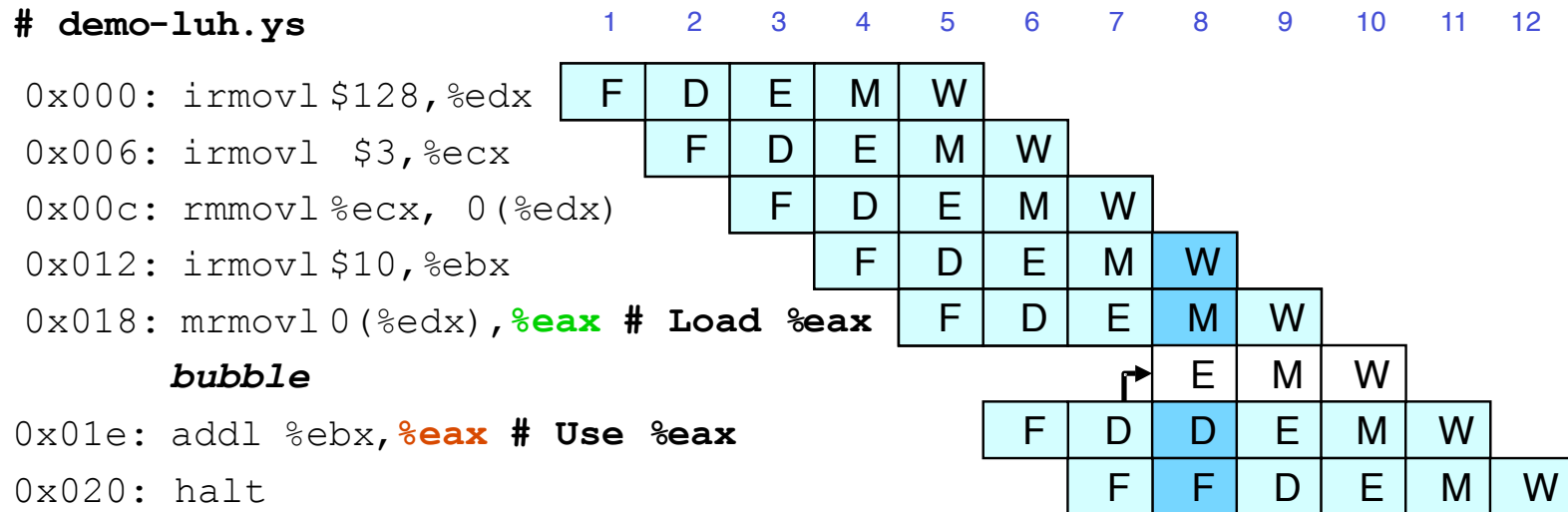


Detecting Load/Use Hazard



| 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 |

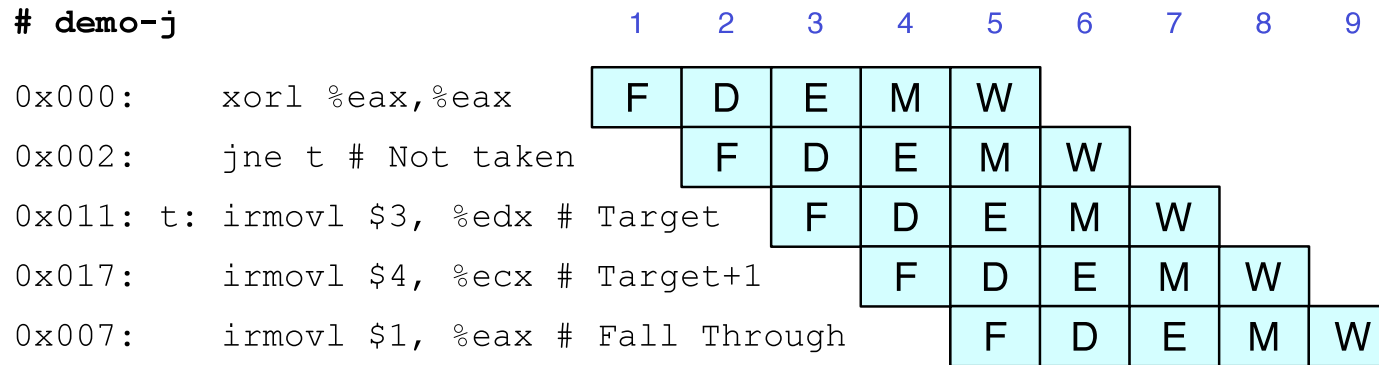
Branch Misprediction Example

demo-j.js

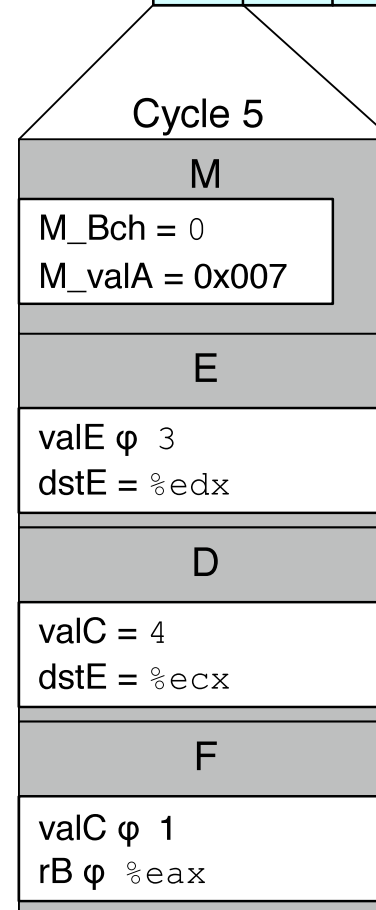
```
0x000:    xorl %eax,%eax
0x002:    jne  t                # Not taken
0x007:    irmovl $1, %eax     # Fall through
0x00d:    nop
0x00e:    nop
0x00f:    nop
0x010:    halt
0x011:  t:  irmovl $3, %edx   # Target (Should not execute)
0x017:    irmovl $4, %ecx     # Should not execute
0x01d:    irmovl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

- Should only execute first 8 instructions

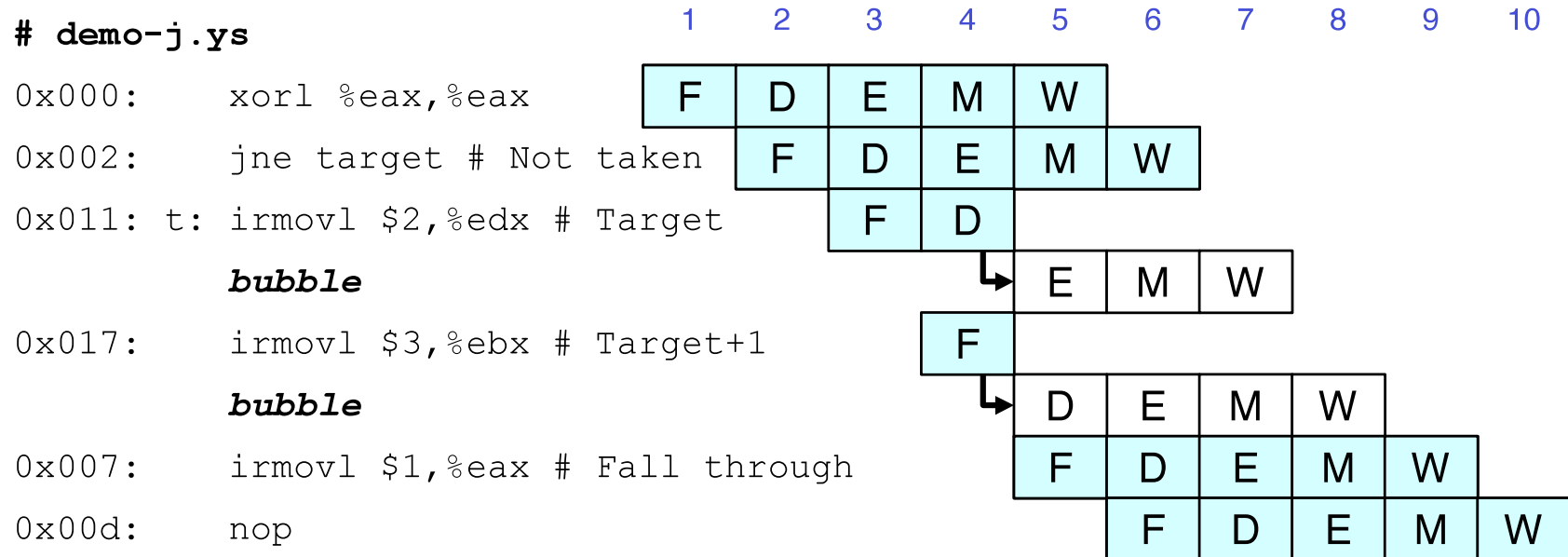
Branch Misprediction Trace



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



Handling Misprediction



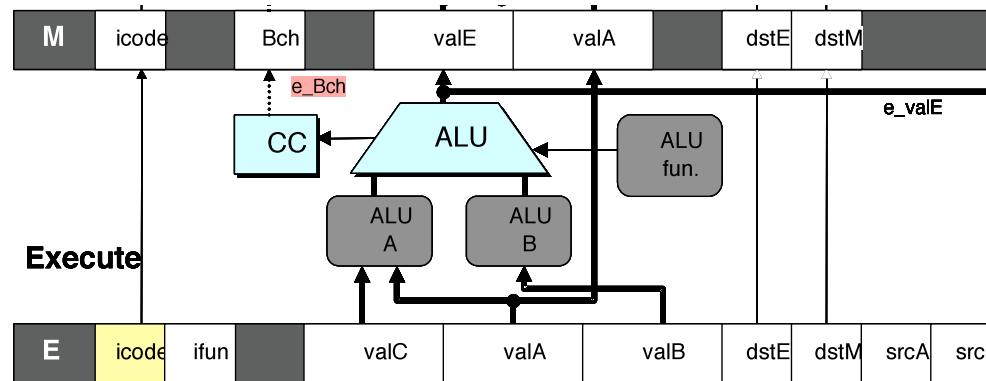
Predict branch as taken

- Fetch 2 instructions at target

Cancel when mispredicted

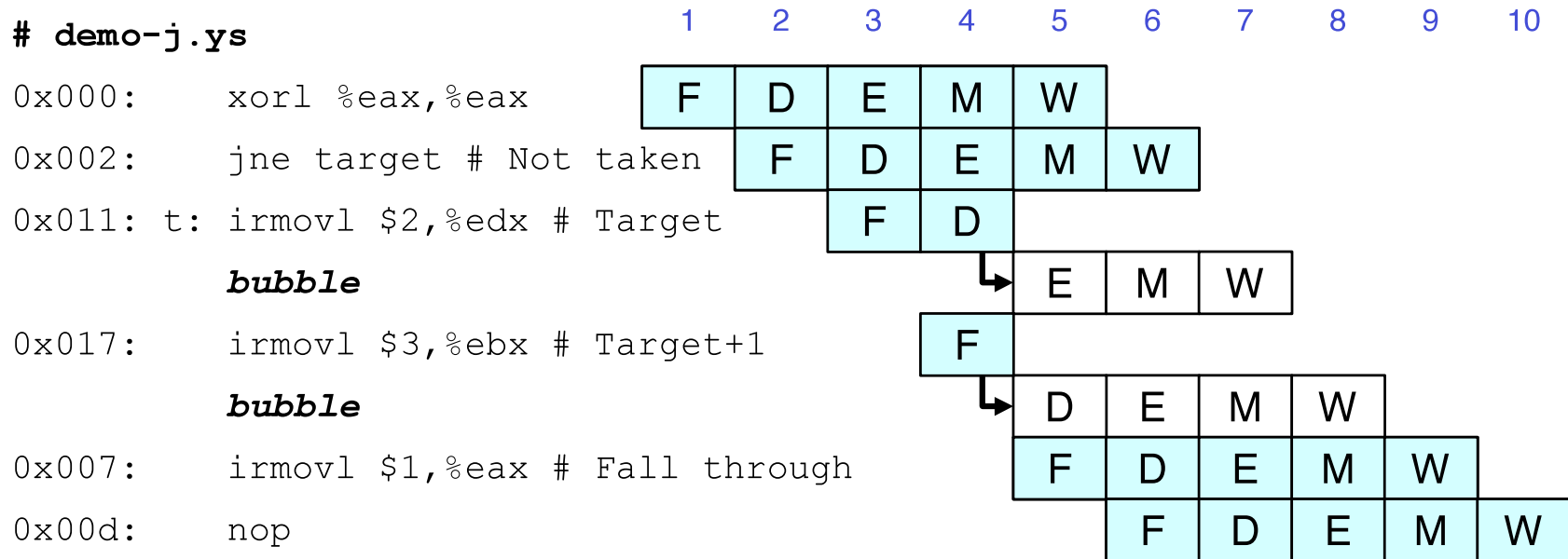
- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
- No side effects have occurred yet

Detecting Mispredicted Branch



| Condition | Trigger |
|---------------------|----------------------------------|
| Mispredicted Branch | $E_icode = IJXX \ \& \ !e_Bch$ |

Control for Misprediction



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

Return Example

demo-retb.ys

```
0x000:    irmovl Stack,%esp    # Initialize stack pointer
0x006:    call p                # Procedure call
0x00b:    irmovl $5,%esi       # Return point
0x011:    halt
0x020:    .pos 0x20
0x020:    p: irmovl $-1,%edi   # procedure
0x026:    ret
0x027:    irmovl $1,%eax       # Should not be executed
0x02d:    irmovl $2,%ecx       # Should not be executed
0x033:    irmovl $3,%edx       # Should not be executed
0x039:    irmovl $4,%ebx       # Should not be executed
0x100:    .pos 0x100
0x100:    Stack:                # Stack: Stack pointer
```

- Previously executed three additional instructions

Correct Return Example

```
# demo-retb
```

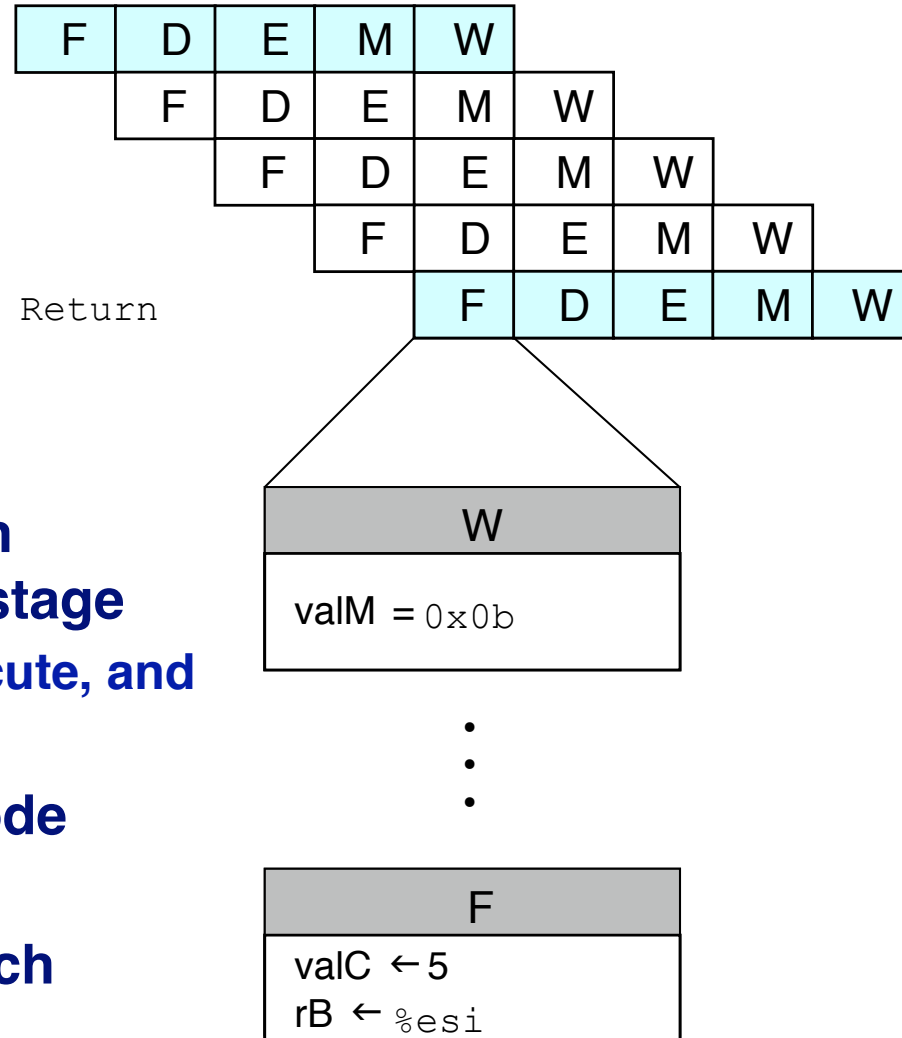
```
0x026:  ret
```

```
    bubble
```

```
    bubble
```

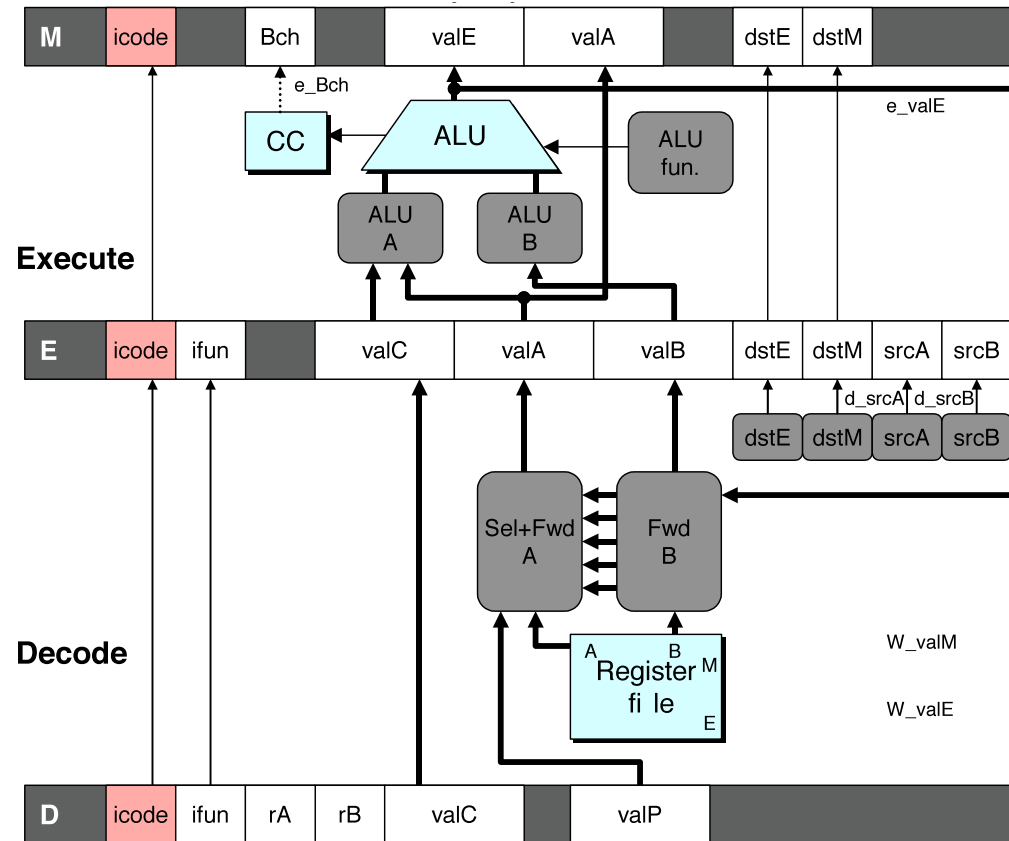
```
    bubble
```

```
0x00b:  irmovl $5,%esi # Return
```



- **As `ret` passes through pipeline, stall at fetch stage**
 - While in decode, execute, and memory stage
- **Inject bubble into decode stage**
- **Release stall when reach write-back stage**

Detecting Return



| Condition | Trigger |
|----------------|---------------------------------------|
| Processing ret | IRET in { D_icode, E_icode, M_icode } |

Control for Return

demo-retb

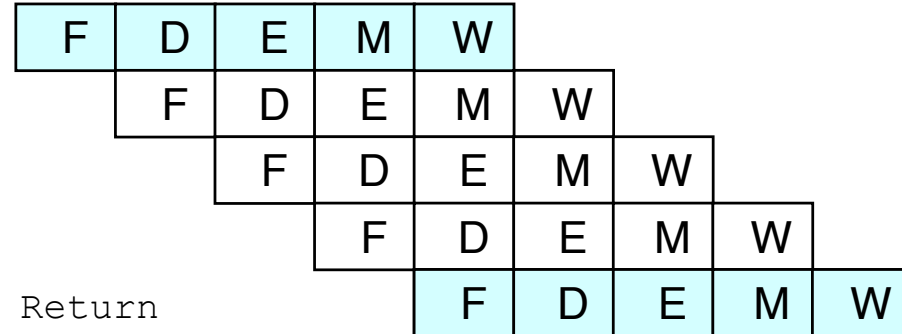
0x026: ret

bubble

bubble

bubble

0x00b: irmovl \$5,%esi # Return



| Condition | F | D | E | M | W |
|----------------|-------|--------|--------|--------|--------|
| Processing ret | stall | bubble | normal | normal | normal |

Special Control Cases

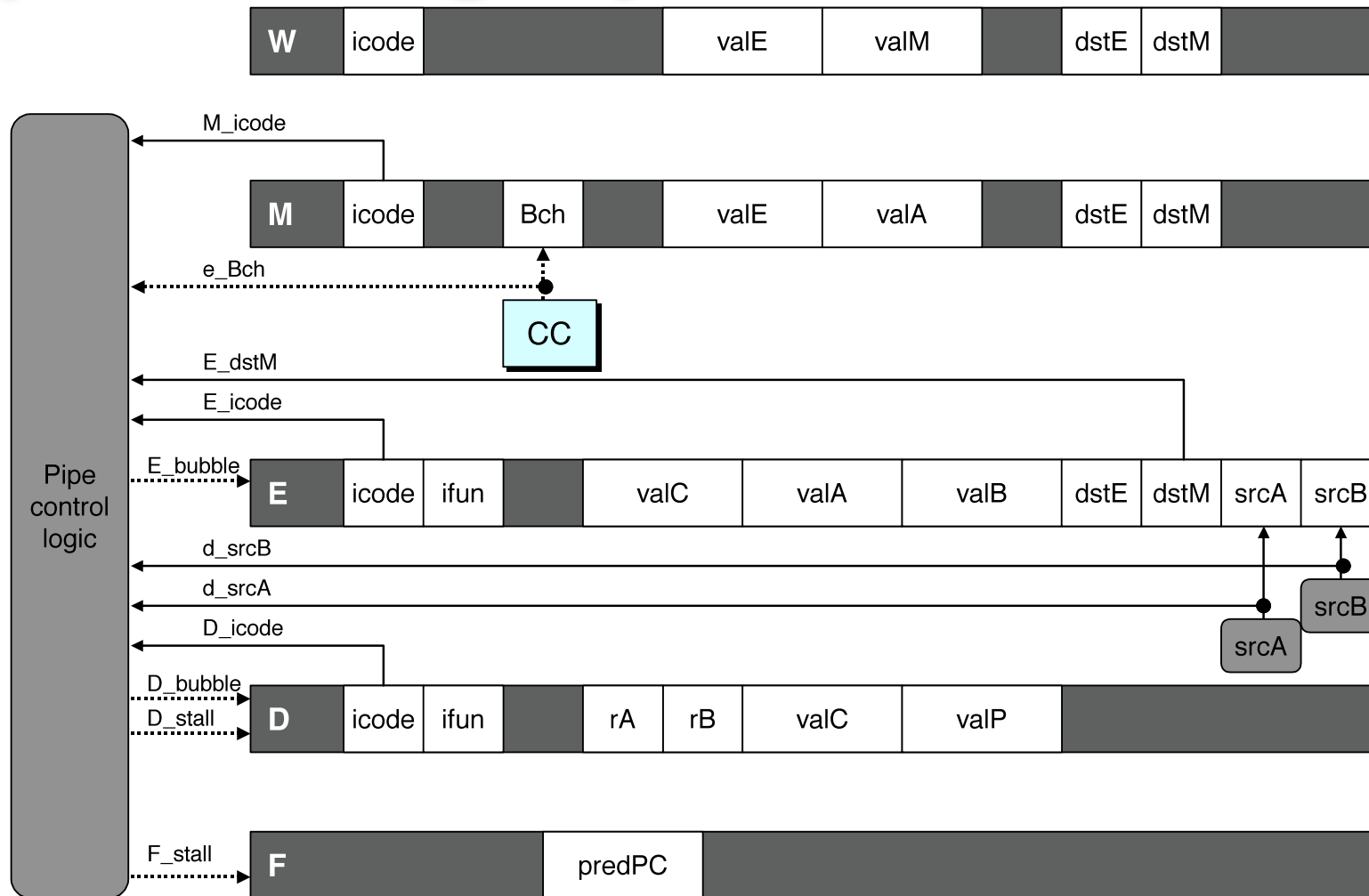
Detection

| Condition | Trigger |
|---------------------|--|
| Processing ret | IRET in { D_icode, E_icode, M_icode } |
| Load/Use Hazard | E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB } |
| Mispredicted Branch | E_icode = IJXX & !e_Bch |

Action (on next cycle)

| Condition | F | D | E | M | W |
|---------------------|--------|--------|--------|--------|--------|
| Processing ret | stall | bubble | normal | normal | normal |
| Load/Use Hazard | stall | stall | bubble | normal | normal |
| Mispredicted Branch | normal | bubble | bubble | normal | normal |

Implementing Pipeline Control



- **Combinational logic generates pipeline control signals**
- **Action occurs at start of following cycle**

Initial Version of Pipeline Control

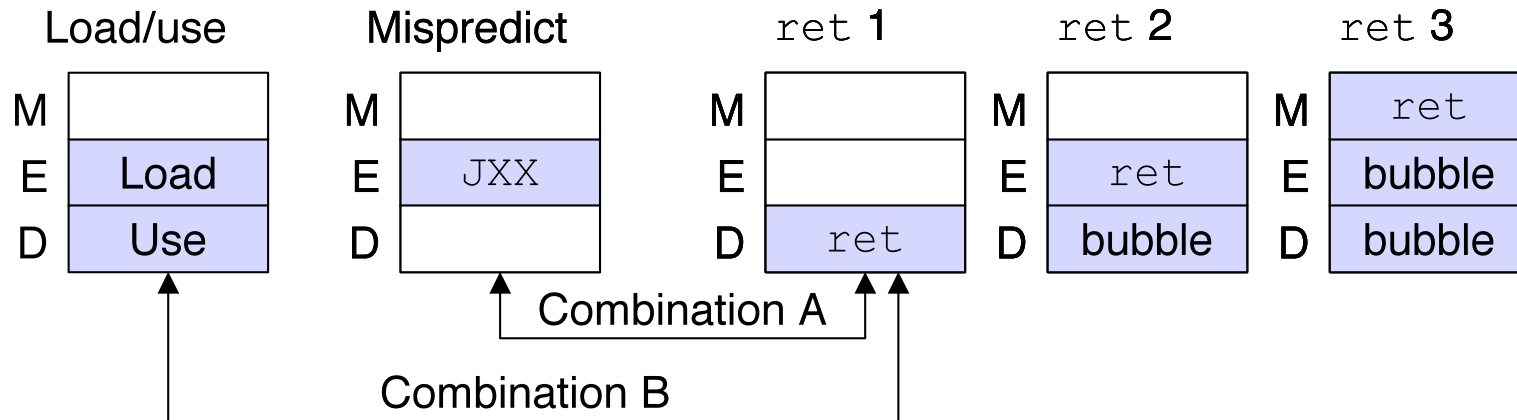
```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB};
```


Control Combinations



- Special cases that can arise on same clock cycle

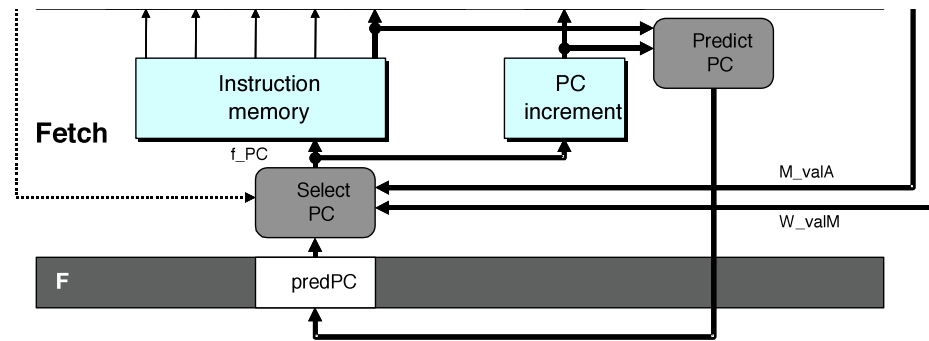
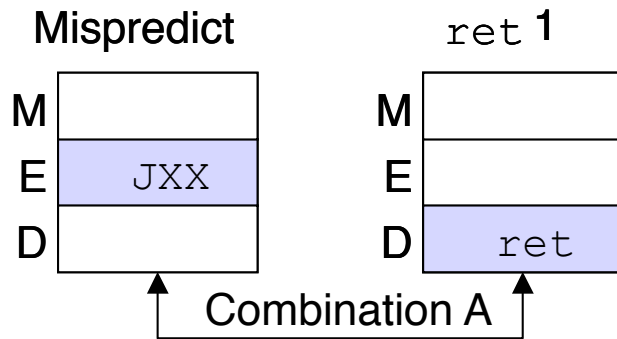
Combination A

- Not-taken branch
- `ret` instruction at branch target

Combination B

- Instruction that reads from memory to `%esp`
- Followed by `ret` instruction

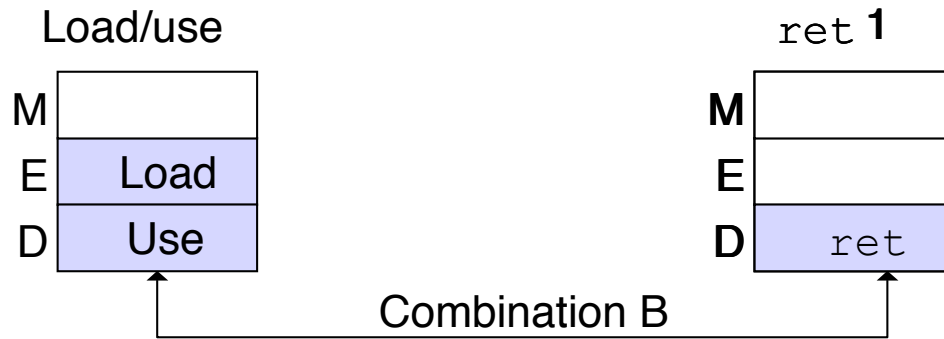
Control Combination A



| Condition | F | D | E | M | W |
|---------------------|--------------|---------------|---------------|---------------|---------------|
| Processing ret | stall | bubble | normal | normal | normal |
| Mispredicted Branch | normal | bubble | bubble | normal | normal |
| <i>Combination</i> | <i>stall</i> | <i>bubble</i> | <i>bubble</i> | <i>normal</i> | <i>normal</i> |

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M_valM anyhow

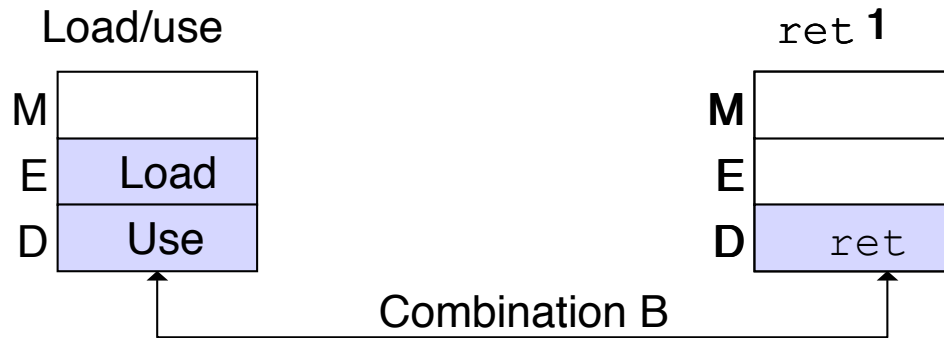
Control Combination B



| Condition | F | D | E | M | W |
|--------------------|--------------|-----------------------|---------------|---------------|---------------|
| Processing ret | stall | bubble | normal | normal | normal |
| Load/Use Hazard | stall | stall | bubble | normal | normal |
| <i>Combination</i> | <i>stall</i> | <i>bubble + stall</i> | <i>bubble</i> | <i>normal</i> | <i>normal</i> |

- Would attempt to bubble *and* stall pipeline register D
- Signaled by processor as pipeline error

Handling Control Combination B



| Condition | F | D | E | M | W |
|--------------------|--------------|--------------|---------------|---------------|---------------|
| Processing ret | stall | bubble | normal | normal | normal |
| Load/Use Hazard | stall | stall | bubble | normal | normal |
| <i>Combination</i> | <i>stall</i> | <i>stall</i> | <i>bubble</i> | <i>normal</i> | <i>normal</i> |

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

Corrected Pipeline Control Logic

```
bool D_bubble =  
    # Mispredicted branch  
    (E_icode == IJXX && !e_Bch) ||  
    # Stalling at fetch while ret passes through pipeline  
    IRET in { D_icode, E_icode, M_icode }  
    # but not condition for a load/use hazard  
    && !(E_icode in { IMRMOVL, IPOPL }  
        && E_dstM in { d_srcA, d_srcB });
```

| Condition | F | D | E | M | W |
|--------------------|--------------|--------------|---------------|---------------|---------------|
| Processing ret | stall | bubble | normal | normal | normal |
| Load/Use Hazard | stall | stall | bubble | normal | normal |
| <i>Combination</i> | <i>stall</i> | <i>stall</i> | <i>bubble</i> | <i>normal</i> | <i>normal</i> |

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

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
- First version had subtle bug
 - Only arises with unusual instruction combination

Return Example

demo-ret.y

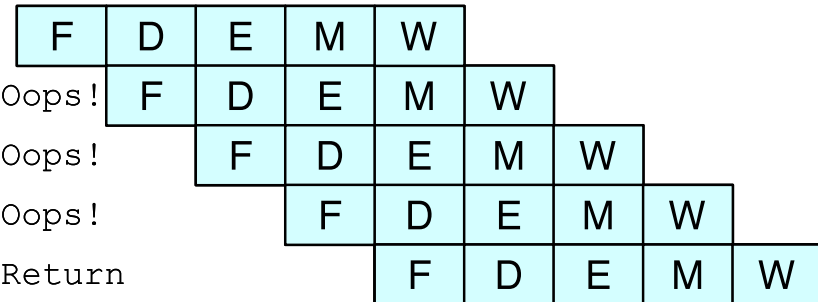
```
0x000:    irmovl Stack,%esp    # Intialize stack pointer
0x006:    nop                  # Avoid hazard on %esp
0x007:    nop
0x008:    nop
0x009:    call p                # Procedure call
0x00e:    irmovl $5,%esi       # Return point
0x014:    halt
0x020:    .pos 0x20
0x020:    p: nop                # 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

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**

