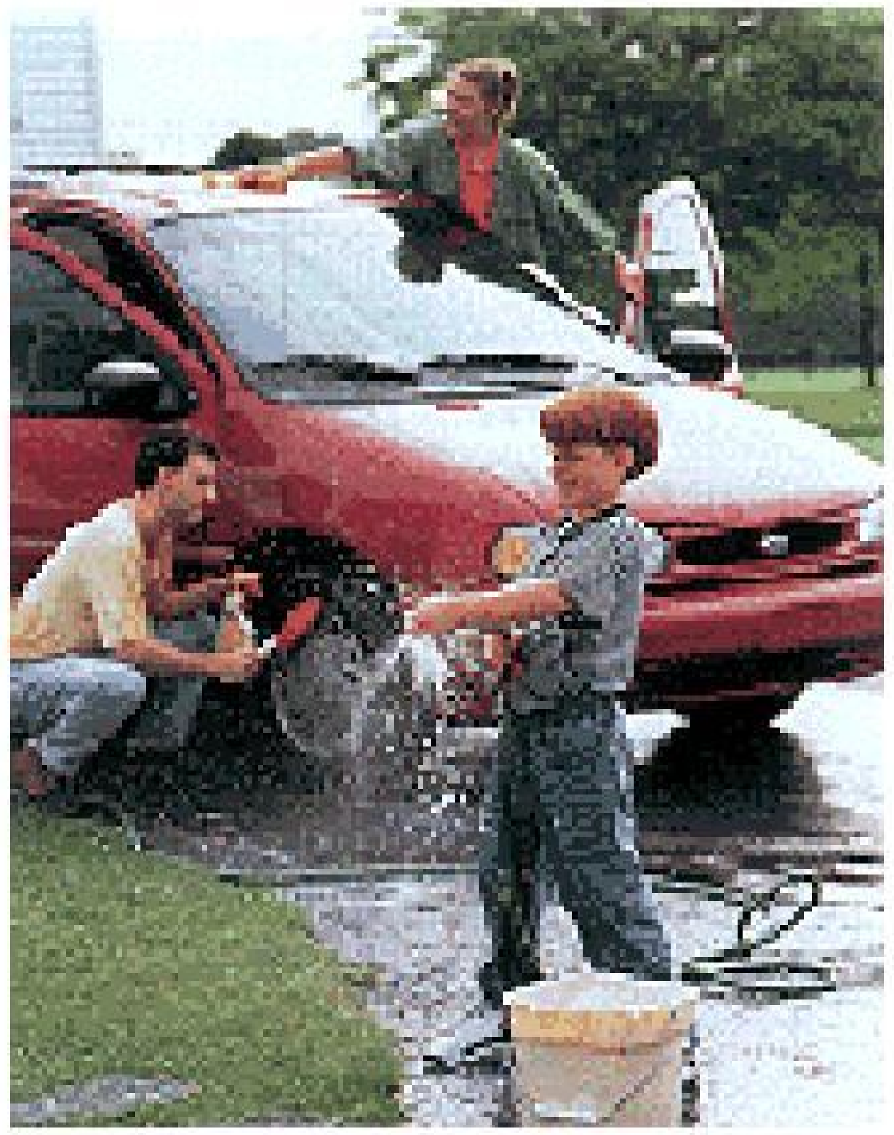
Real-World Pipelines: Car Washes

Sequential

Parallel

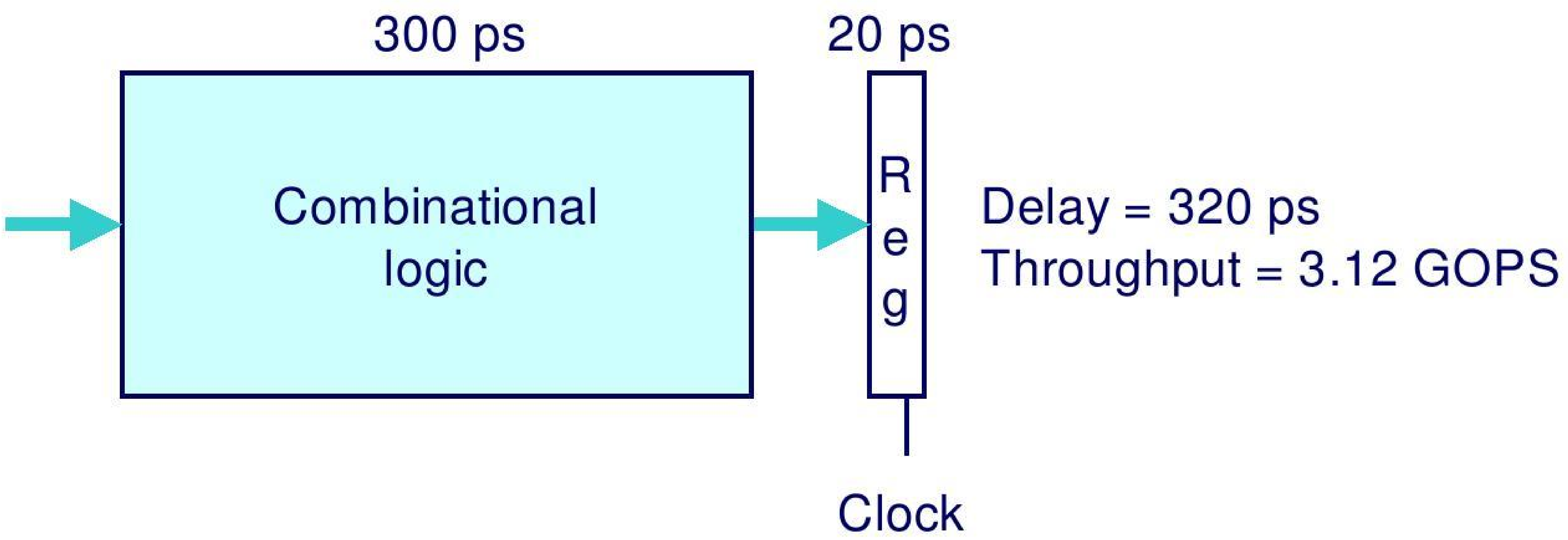


Pipelined

Idea

- **Divide process into independent stages**
- **Move objects through stages in sequence**
- **At any given times, multiple objects being processed**

Computational Example

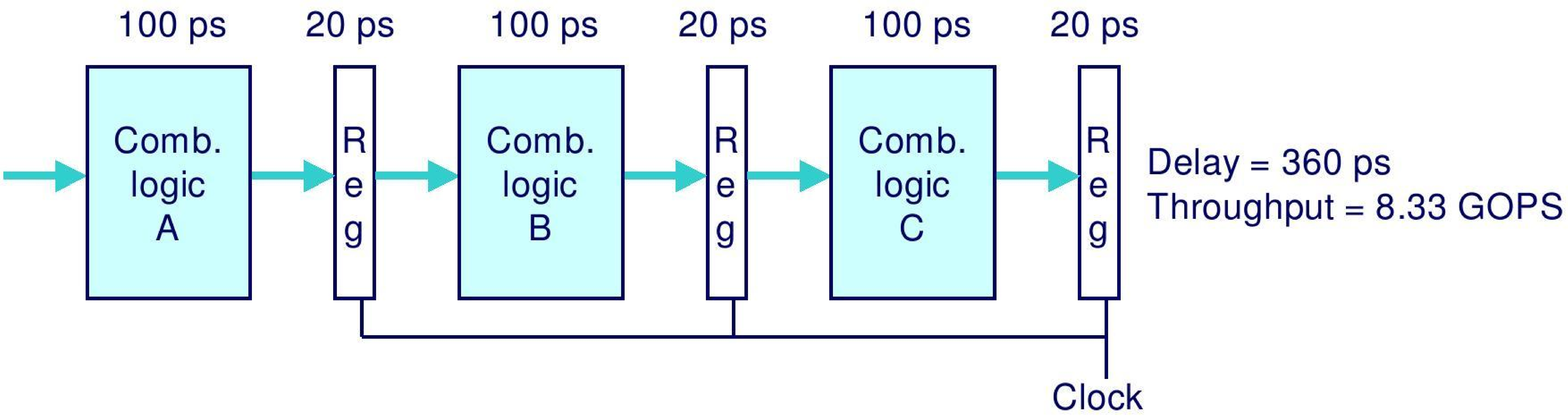


System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Clock cycle must be at least 320 ps

$$\text{Throughput} = \frac{\text{1 operation}}{\text{320 ps}} \times \frac{\text{1000 ps}}{\text{1 ns}} \approx \text{3.12 GOPS}$$

3-Way Pipelined Version

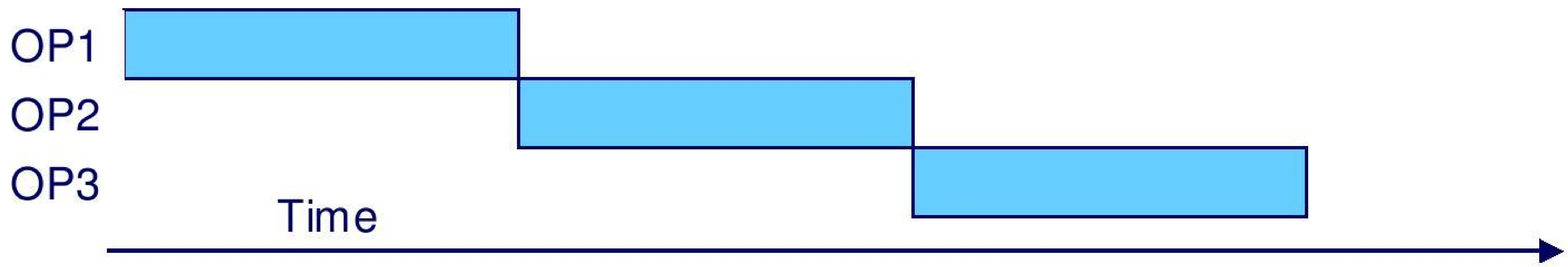


System

- 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

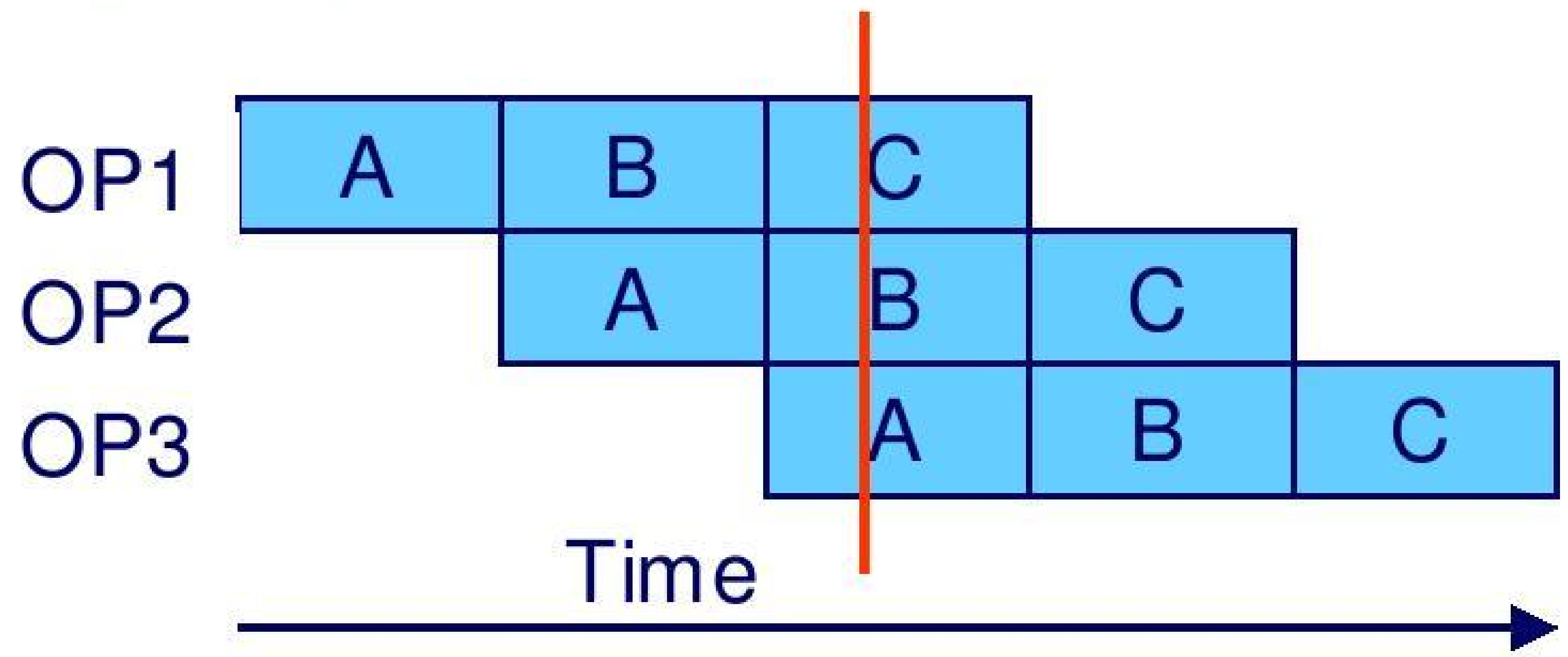
Pipeline Diagrams

Unpipelined



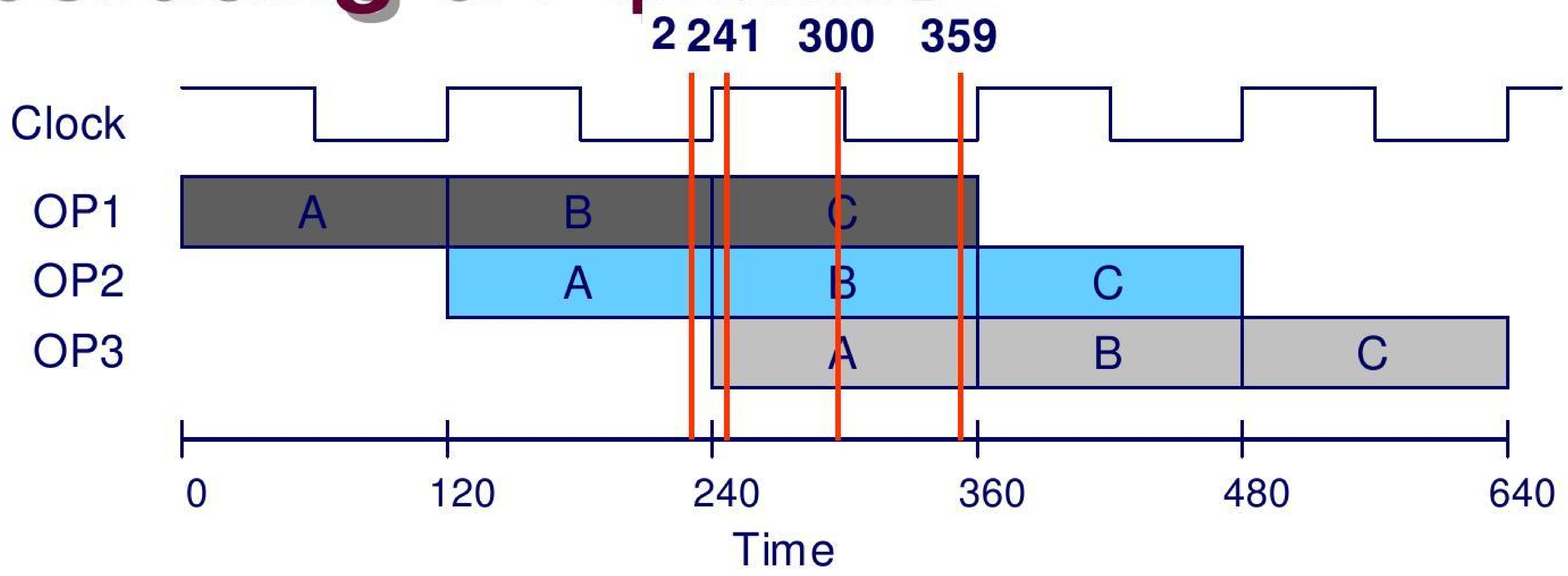
- Cannot start new operation until previous one completes

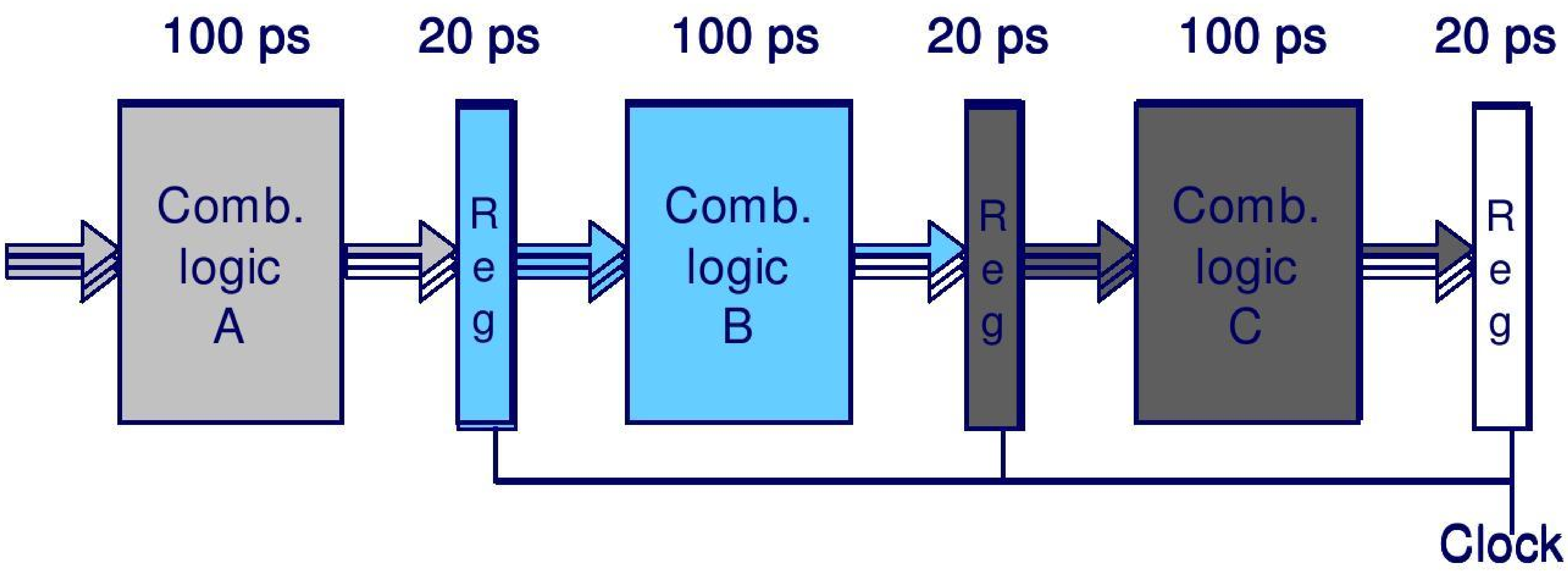
3-Way Pipelined



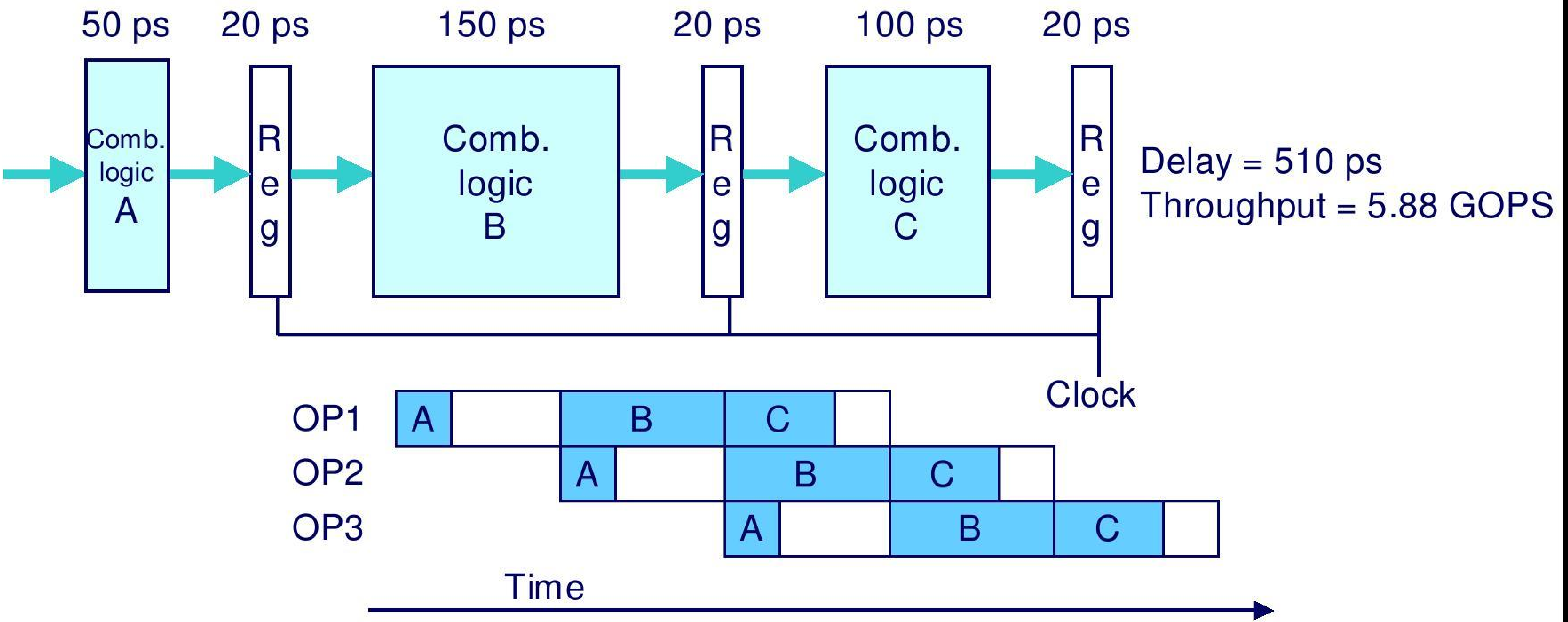
- Up to 3 operations in process simultaneously

Operating a Pipeline



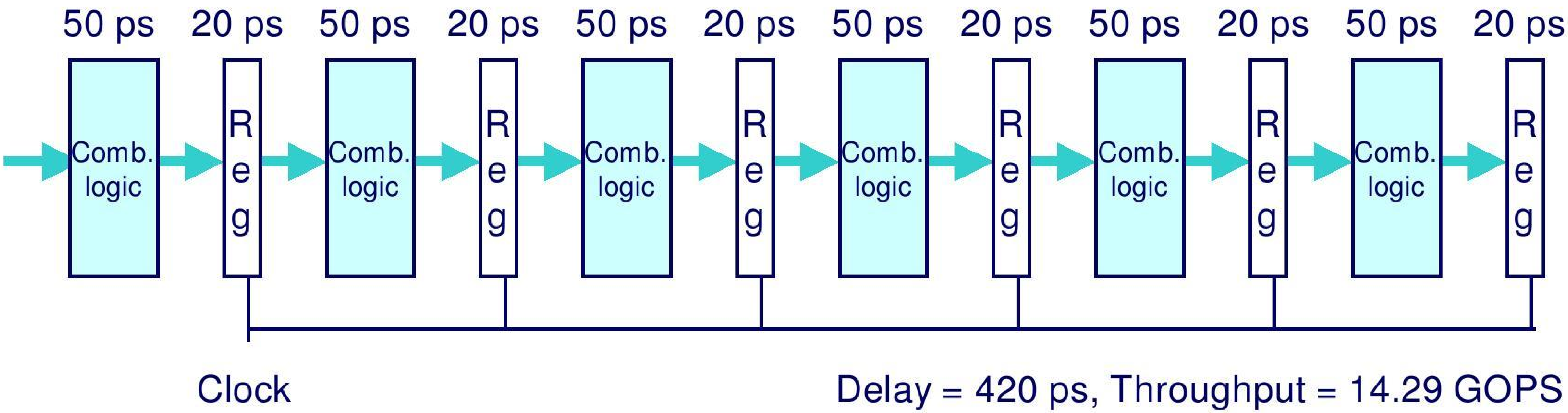


Limitations: Nonuniform Delays



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

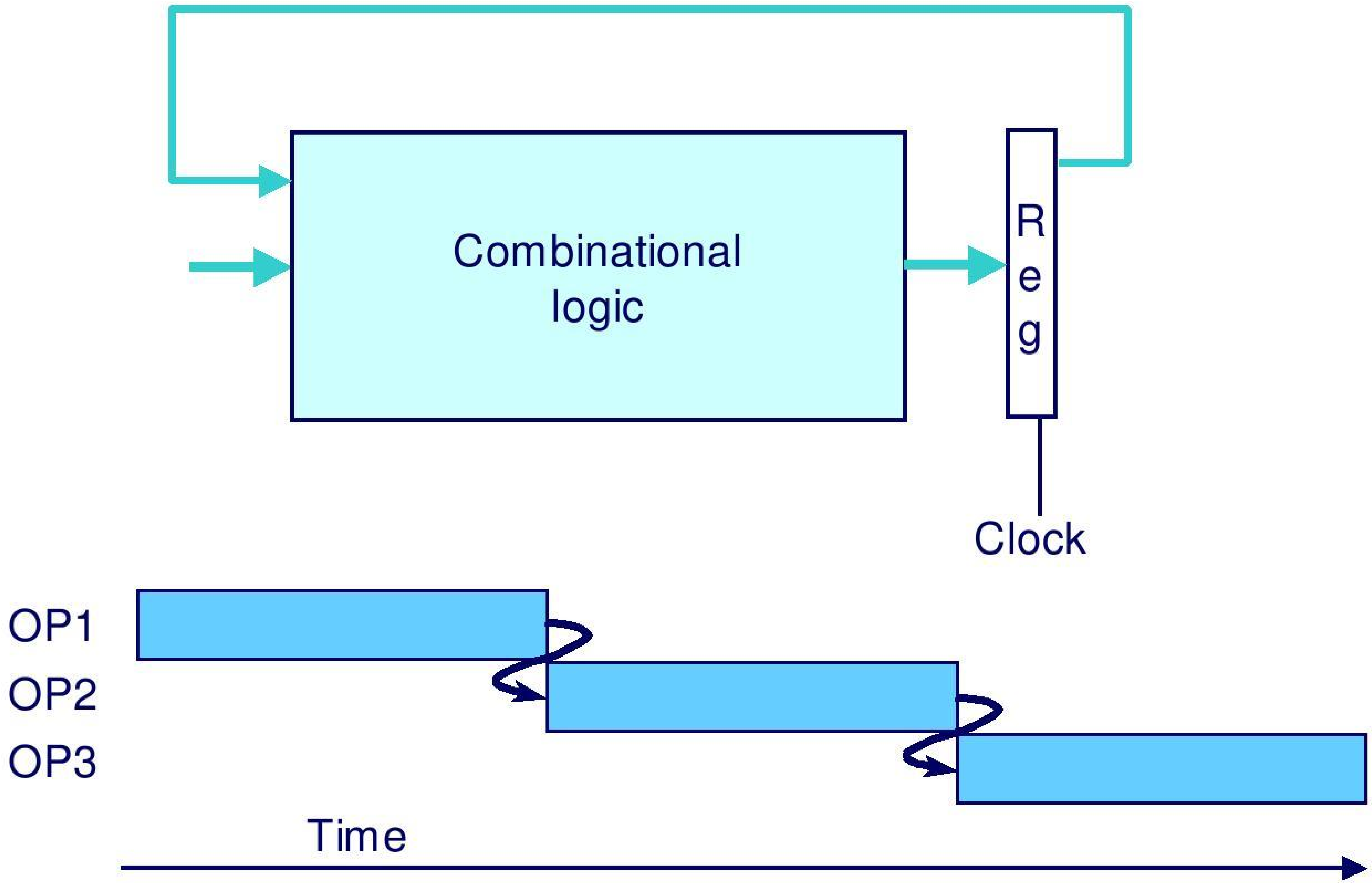
Limitations: Register Overhead



Delay = 420 ps, Throughput = 14.29 GOPS

- **As pipeline deepens, 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**

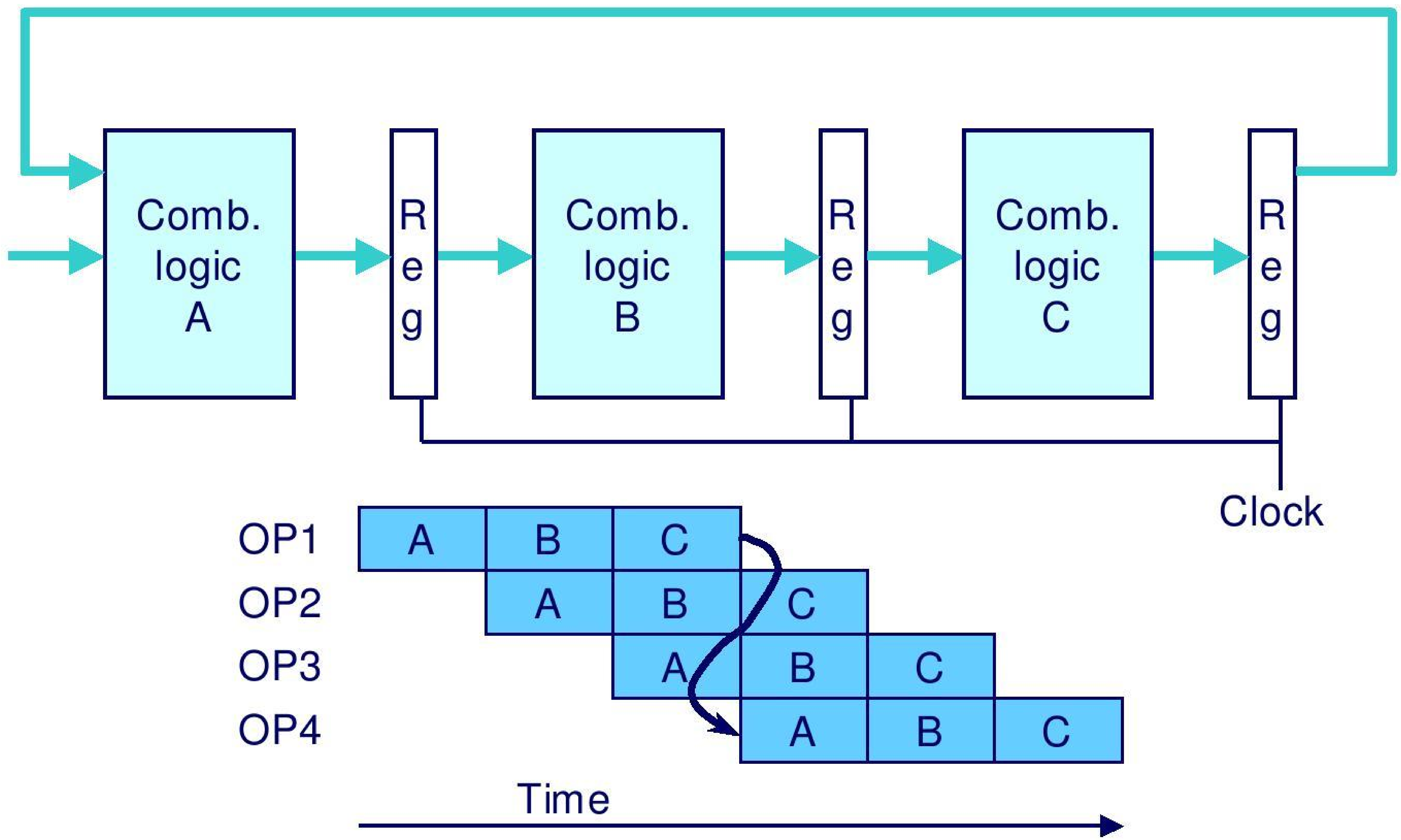
Data Dependencies



System

- **Each operation depends on result from preceding one**

Data Hazards



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

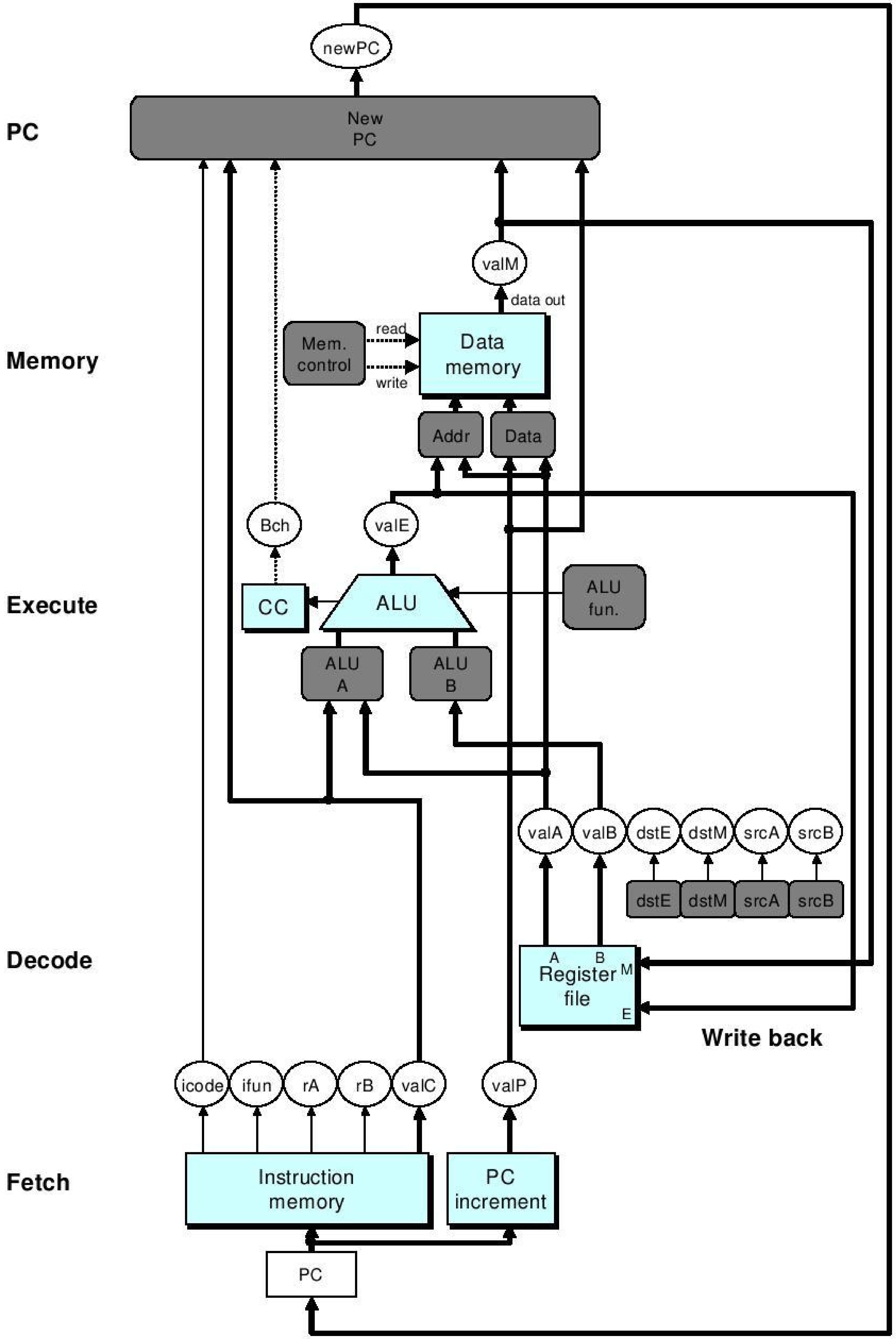
Data Dependencies in Processors

```
irmovl $50, %eax

addl %eax ,  %ebx

mrmovl 100( %ebx ),  %edx
```

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

SEQ Hardware

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



SEQ+ Hardware

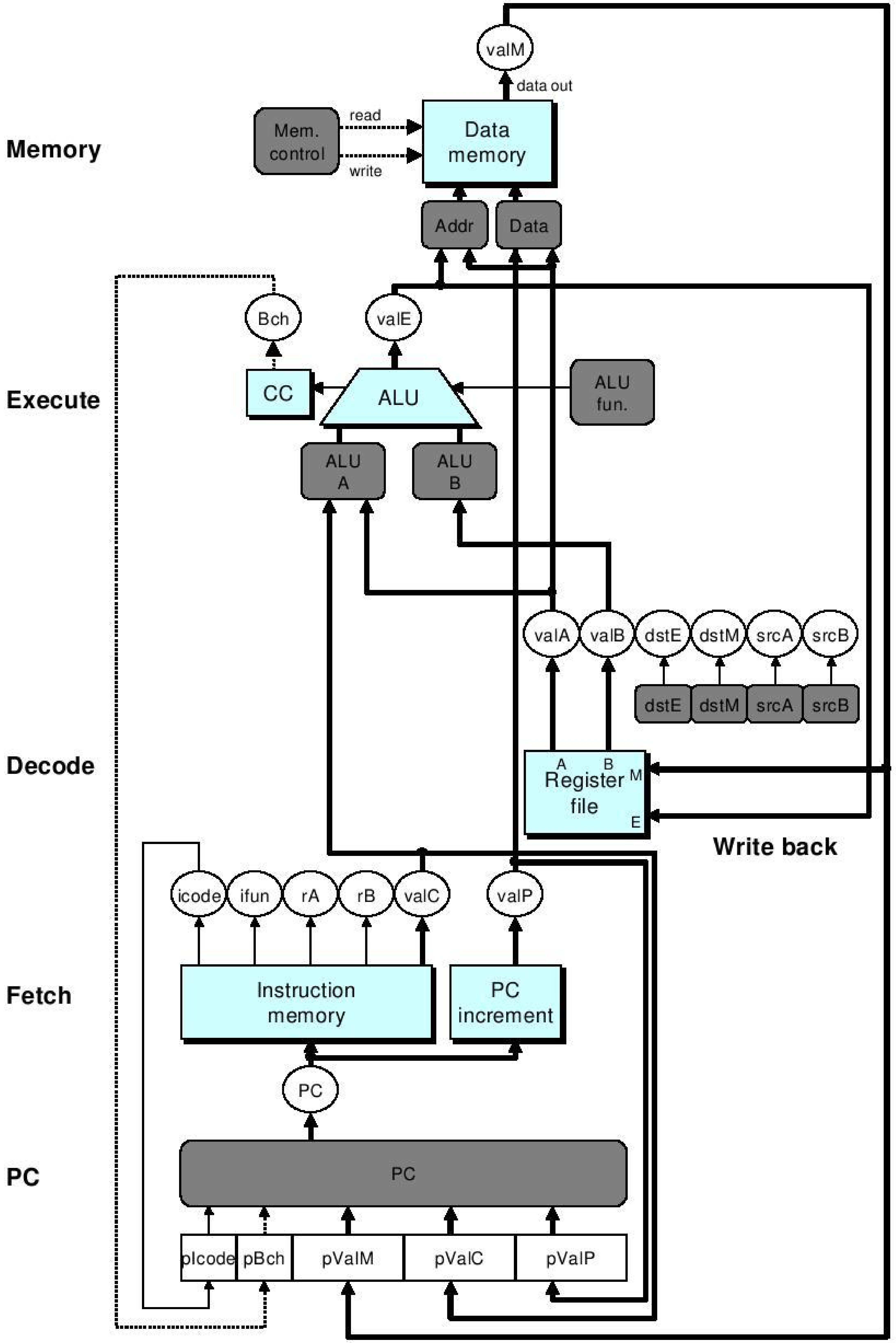
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

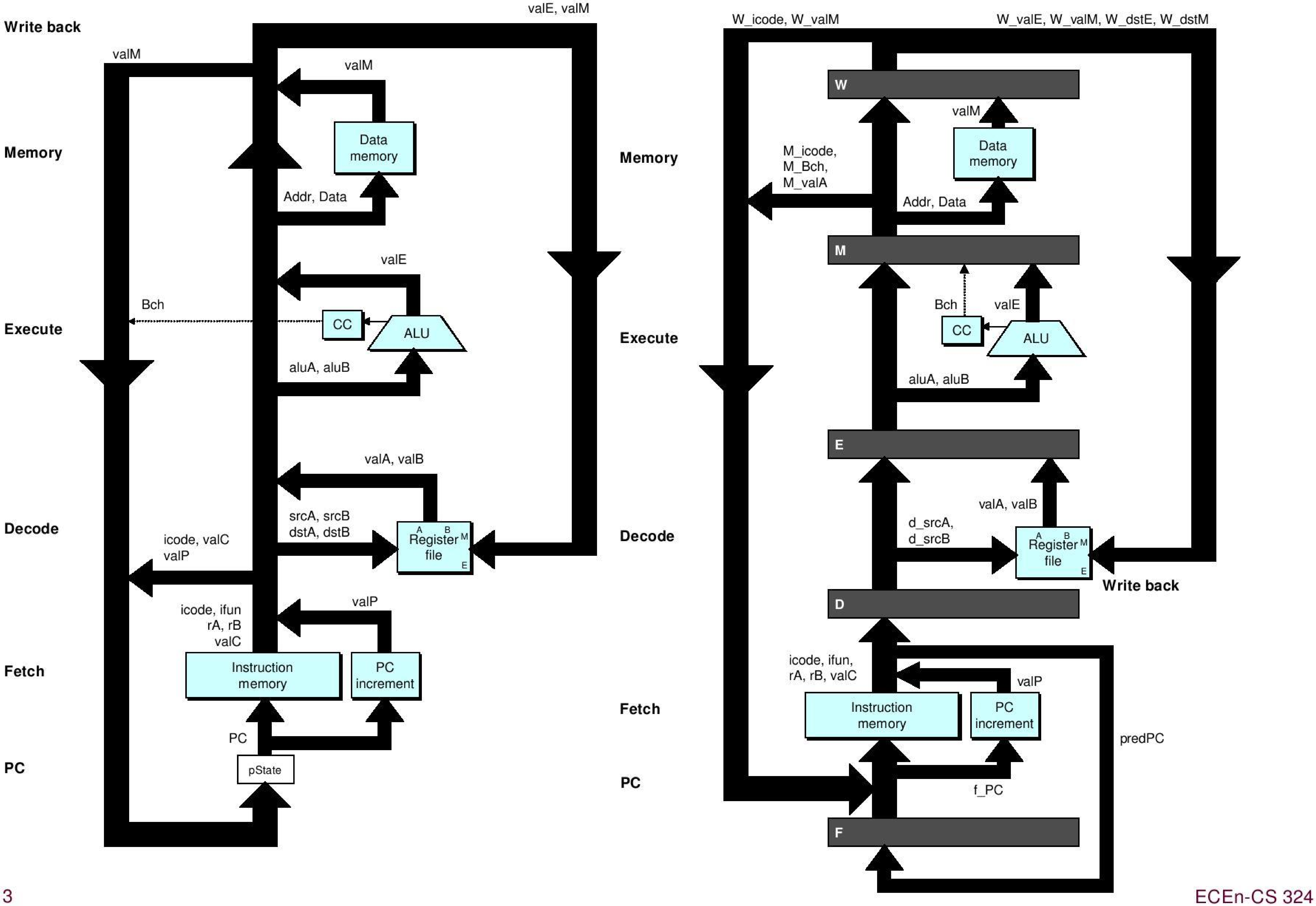
- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- PC is determined from other stored information



Adding Pipeline Registers



Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

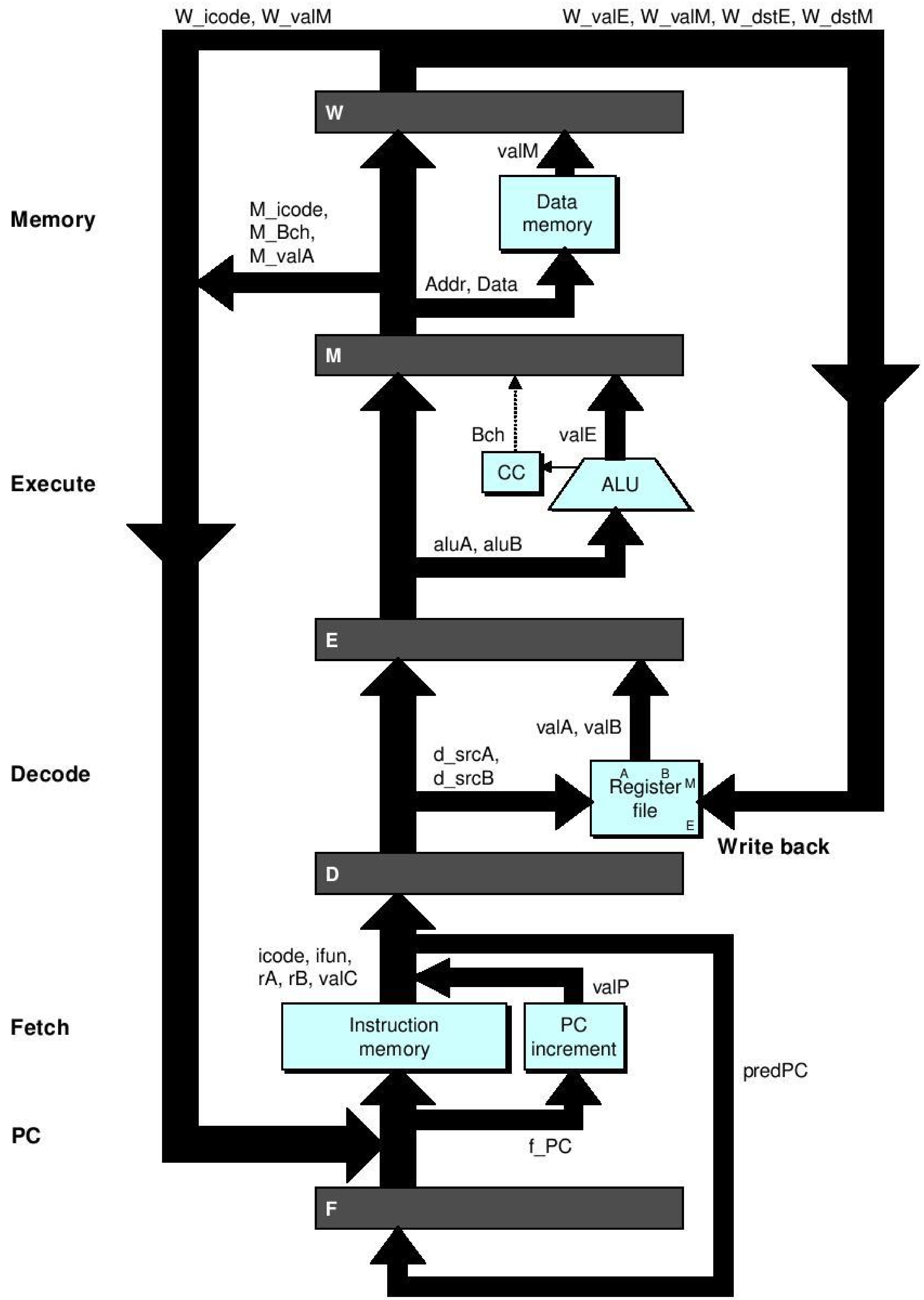
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file

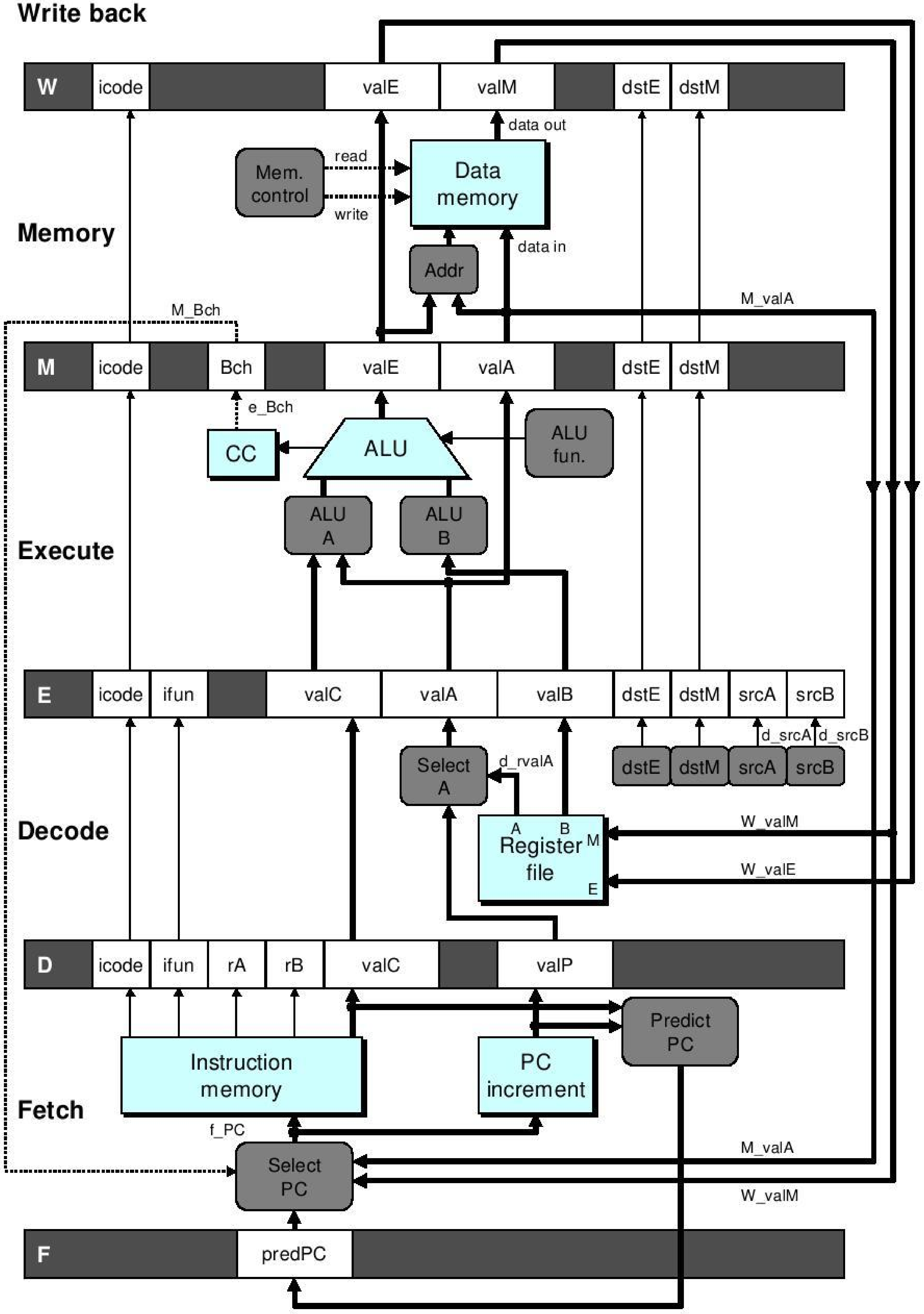


PIPE- Hardware

- 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

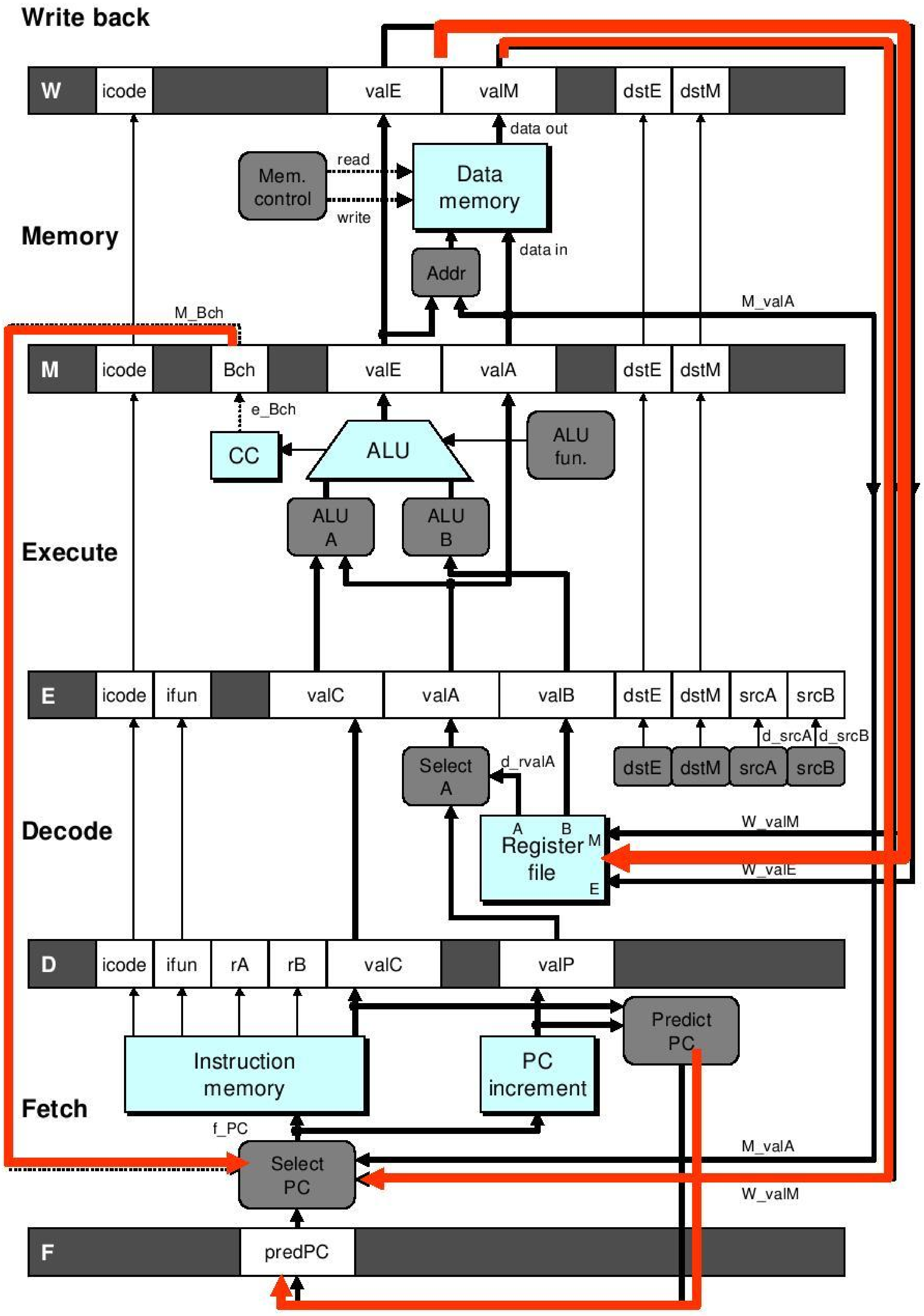
- Jump taken/not-taken
- Fall-through or target address

Return address

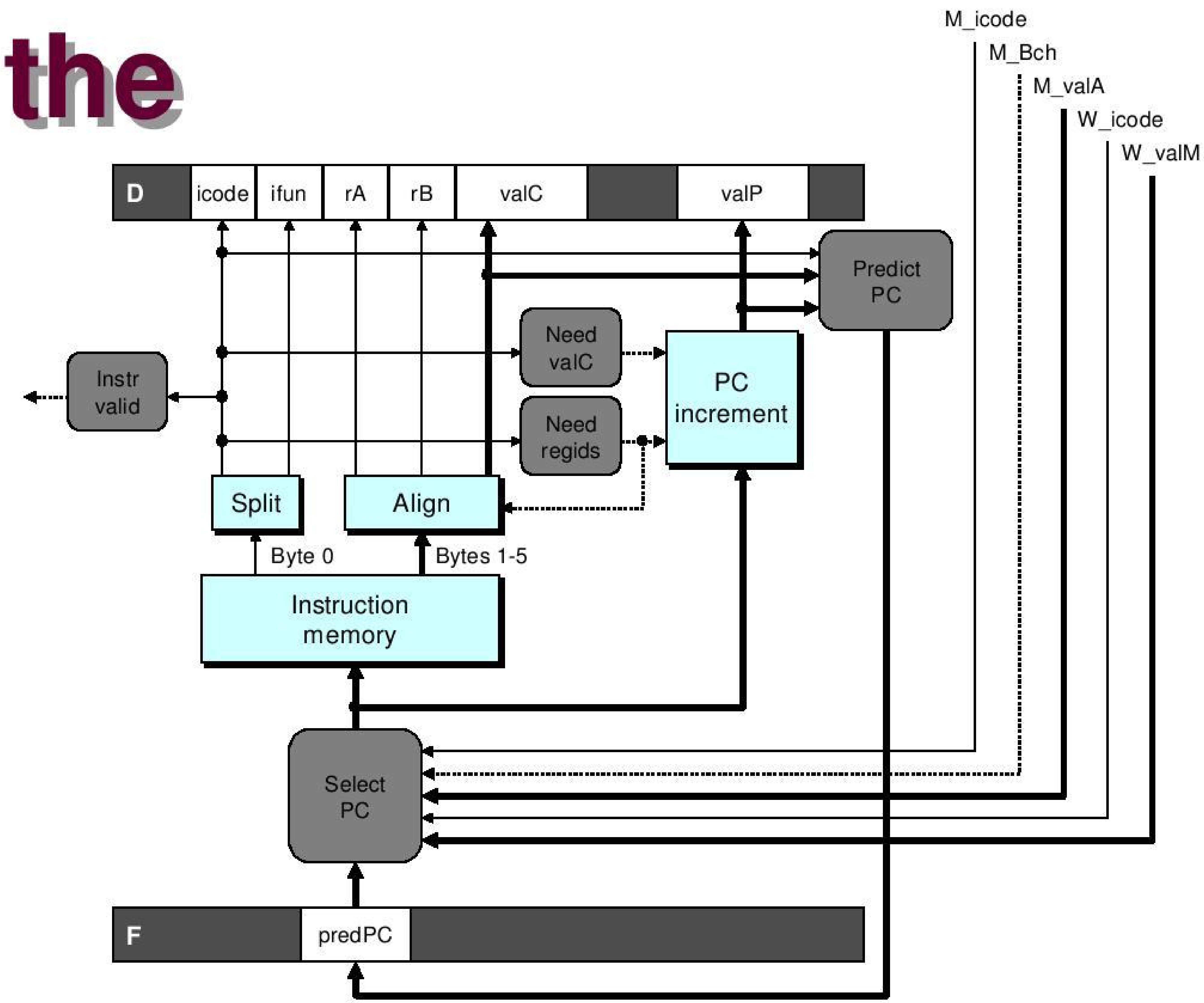
- Read from memory

Register updates

- To register file write ports



Predicting the PC



- Pipeline timing requires next instruction fetch to begin immediately after current instruction is fetched
  - Not enough time to reliably determine next instruction
  - Can be done for all but conditional jumps, return
- Solution for conditional jumps: predict taken (use valC)
  - Recover if prediction was incorrect

Simple 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

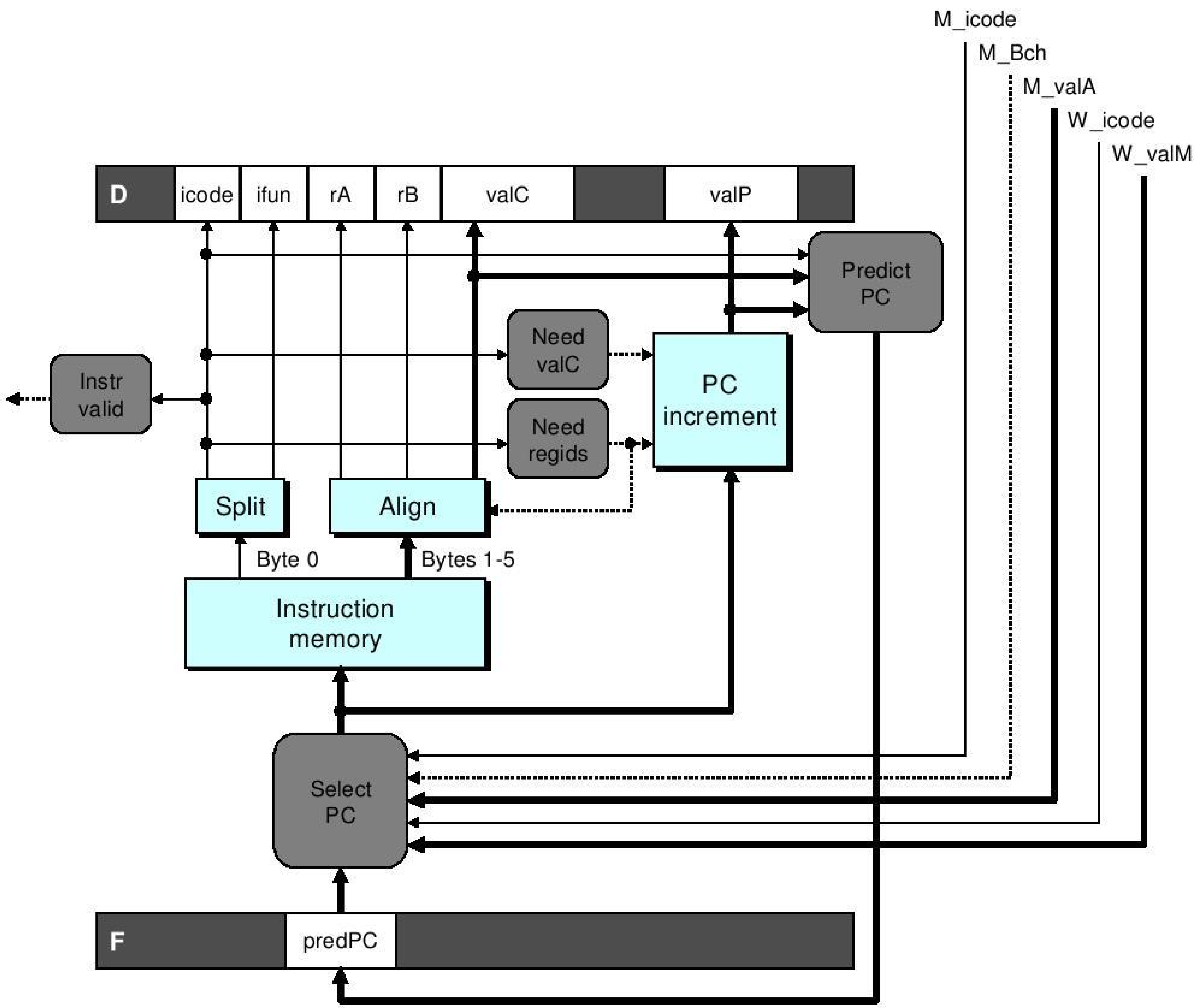
Conditional Jumps

- Predict next PC to be valC (destination)
- Correct only if branch is taken (~60% of time)
- Could predict all branches not taken (correct ~40% of time)

Return Instruction

- Don't try to predict

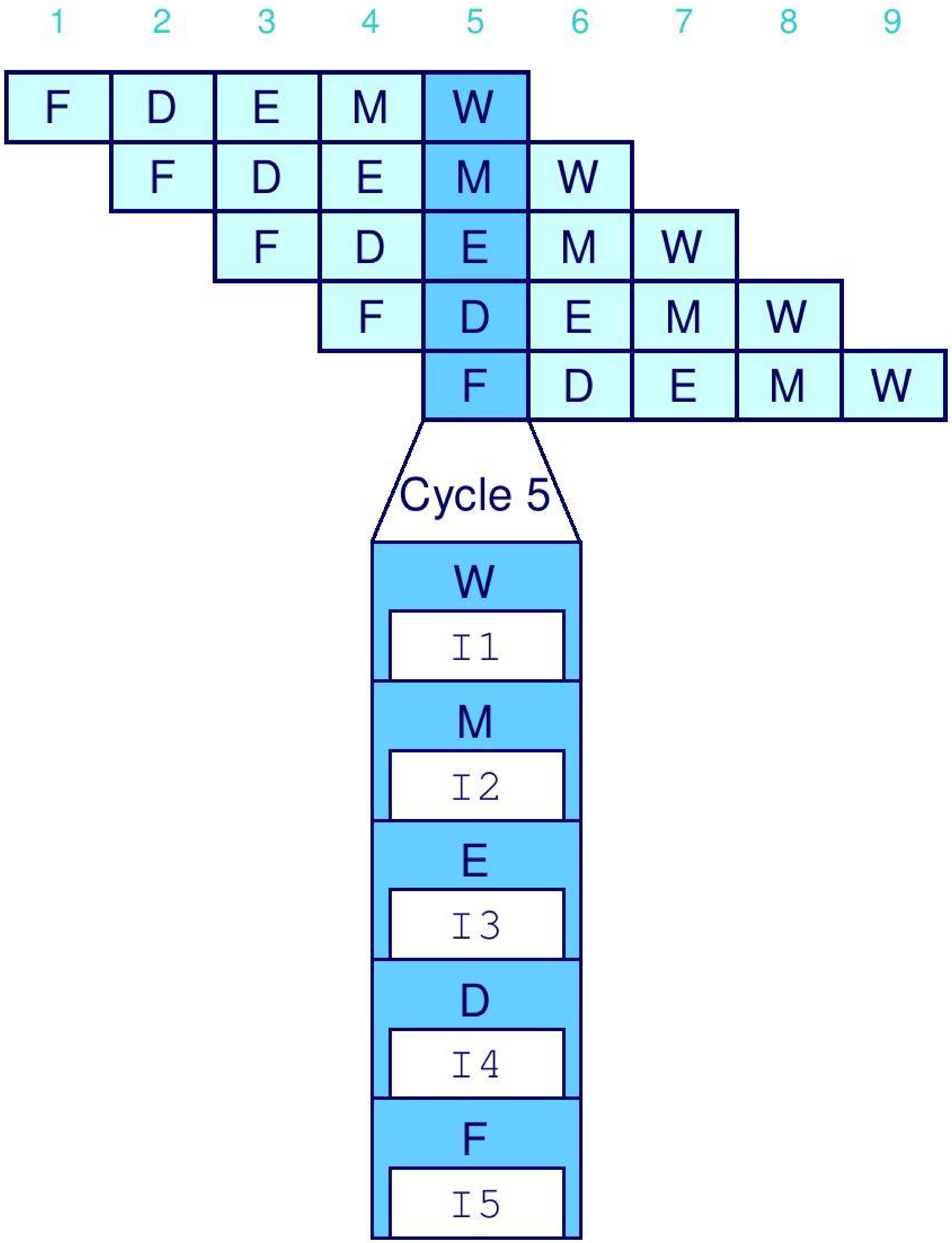
Recovering from PC Misprediction



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

Pipeline Demonstration

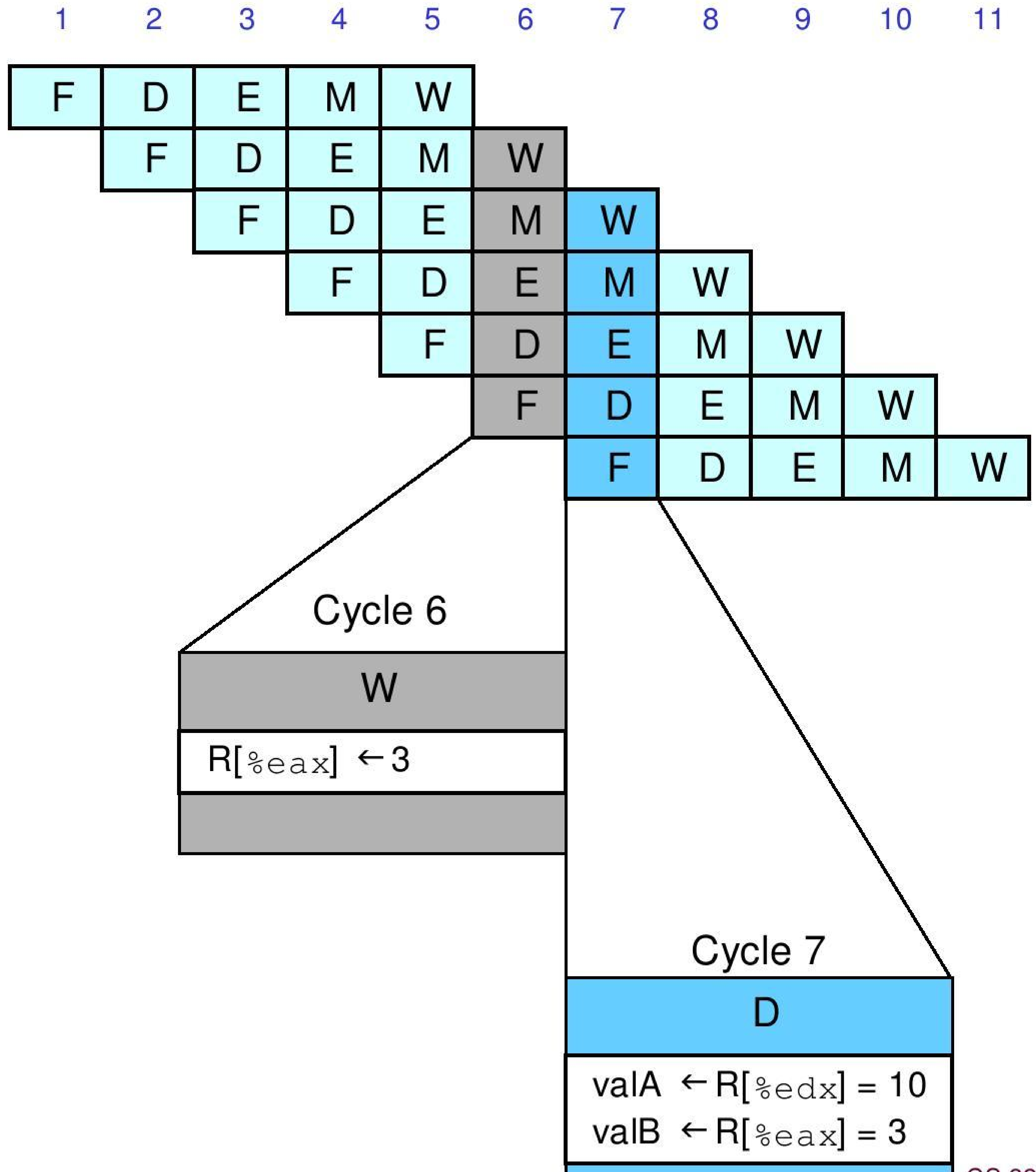
```
irmovl   $1,%eax   #I1
irmovl   $2,%ecx   #I2
irmovl   $3,%edx   #I3
irmovl   $4,%ebx   #I4
halt               #I5
```



File: `demo-basic.ys`

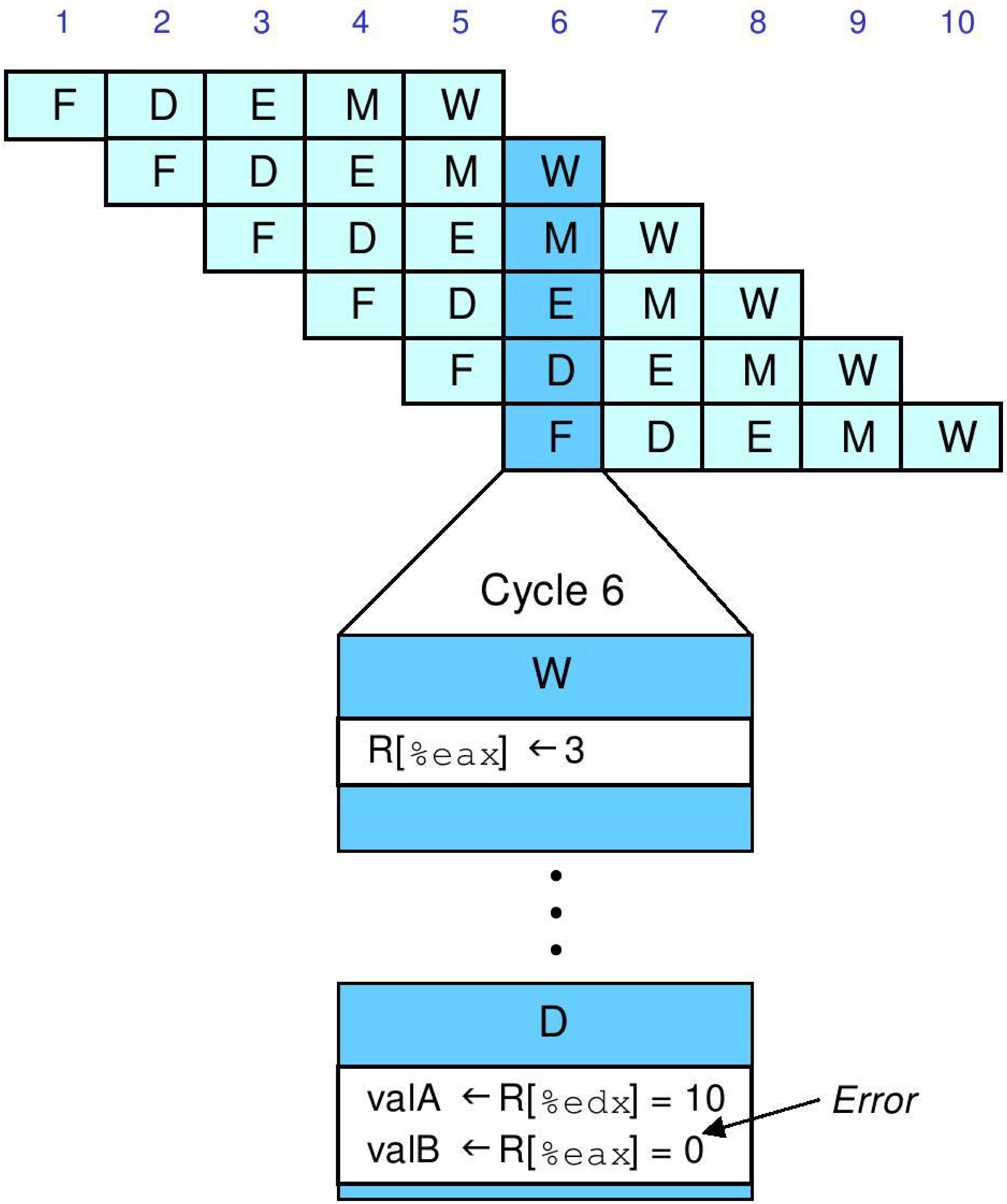
Data Dependencies: 3 Nop's

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



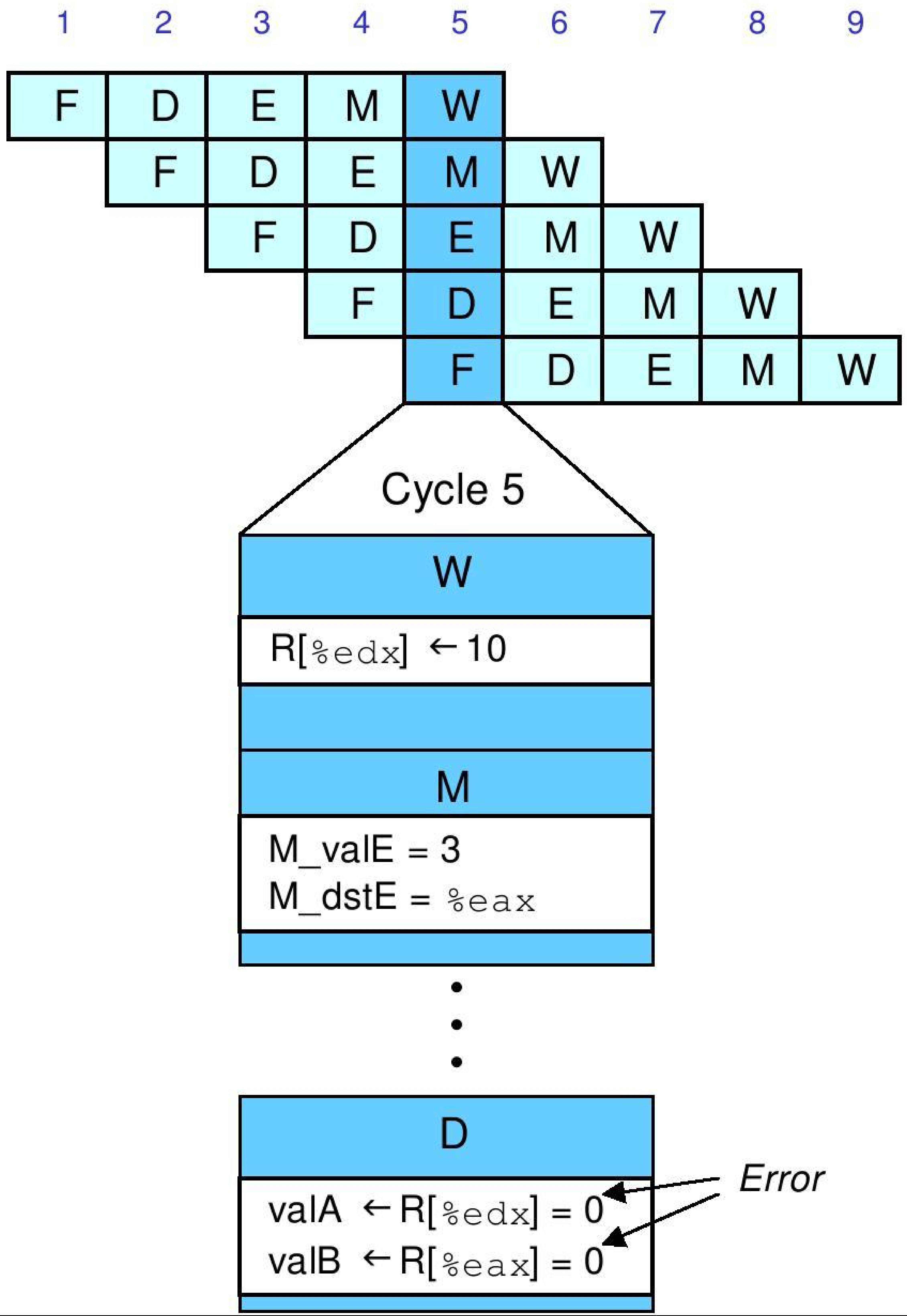
Data Dependencies: 2 Nop's

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



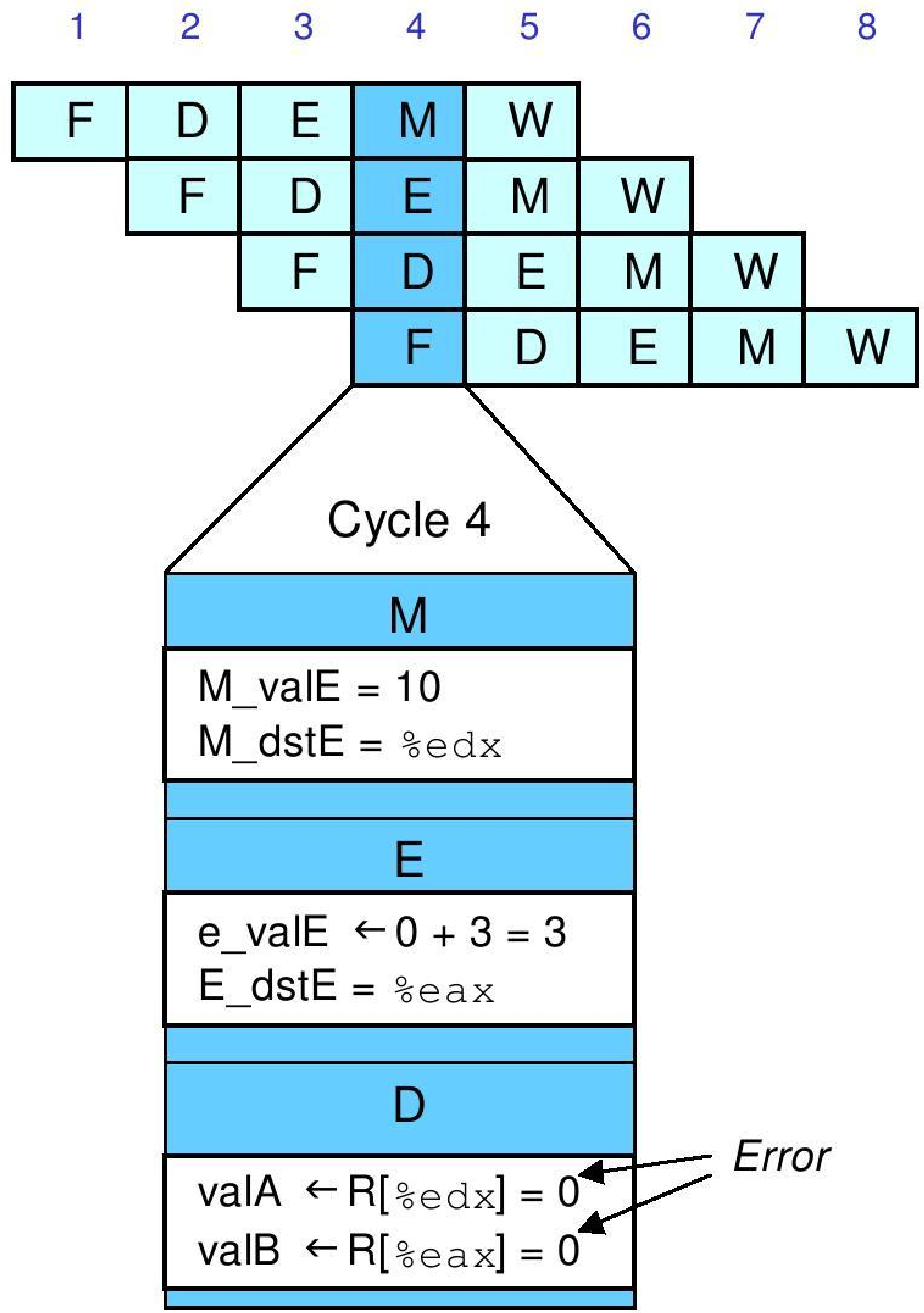
Data Dependencies: 1 Nop

```
# 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
```



Stalling for Data Dependencies

# demo-h2.ys	1	2	3	4	5	6	7	8	9	10	11
0x000: irmovl $10,%edx	F	D	E	M	W						
0x006: irmovl $3,%eax		F	D	E	M	W					
0x00c: nop			F	D	E	M	W				
0x00d: nop				F	D	E	M	W			
bubble							E	M	W		
0x00e: addl %edx,%eax					F	D	D	E	M	W	
0x010: halt						F	F	D	E	M	W

- **If instruction follows too closely after one that writes register, delay it**
- **Delay or stall always takes place in decode stage**
  - **Following instruction also delayed in fetch, but just one "stall cycle" is tallied**
- **Stall = dynamically injecting nop into execute stage**
- **One way of dealing with *hazards***
  - **Situation in which we get wrong answer without special action**

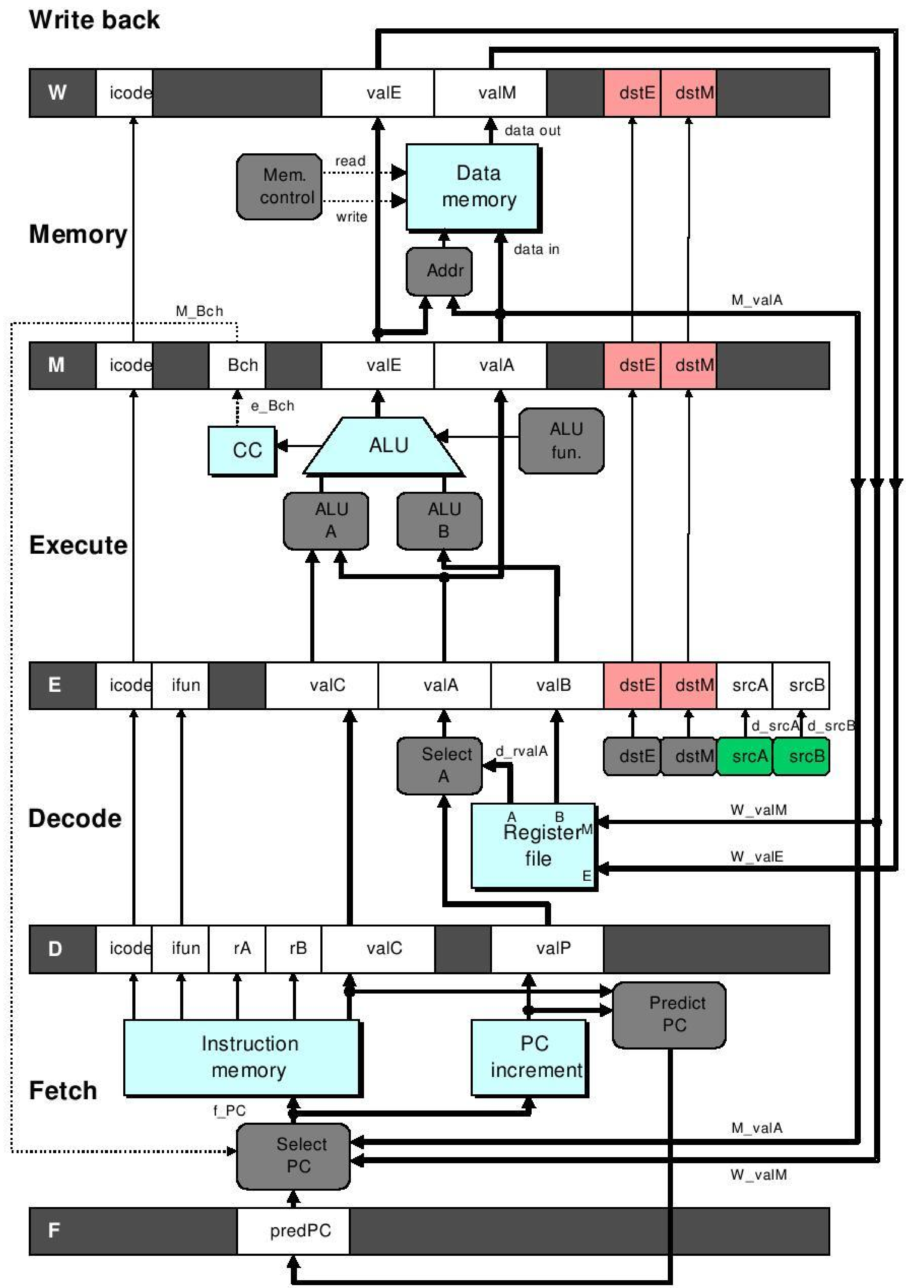
Stall Condition

Is There a Pending Write for Src Registers?

- Pending writes indicated by dstE and dstM fields
  - In execute, memory, and write-back stages
- Compare srcA and srcB in decode stage with all these fields
- If fields are equal, stall

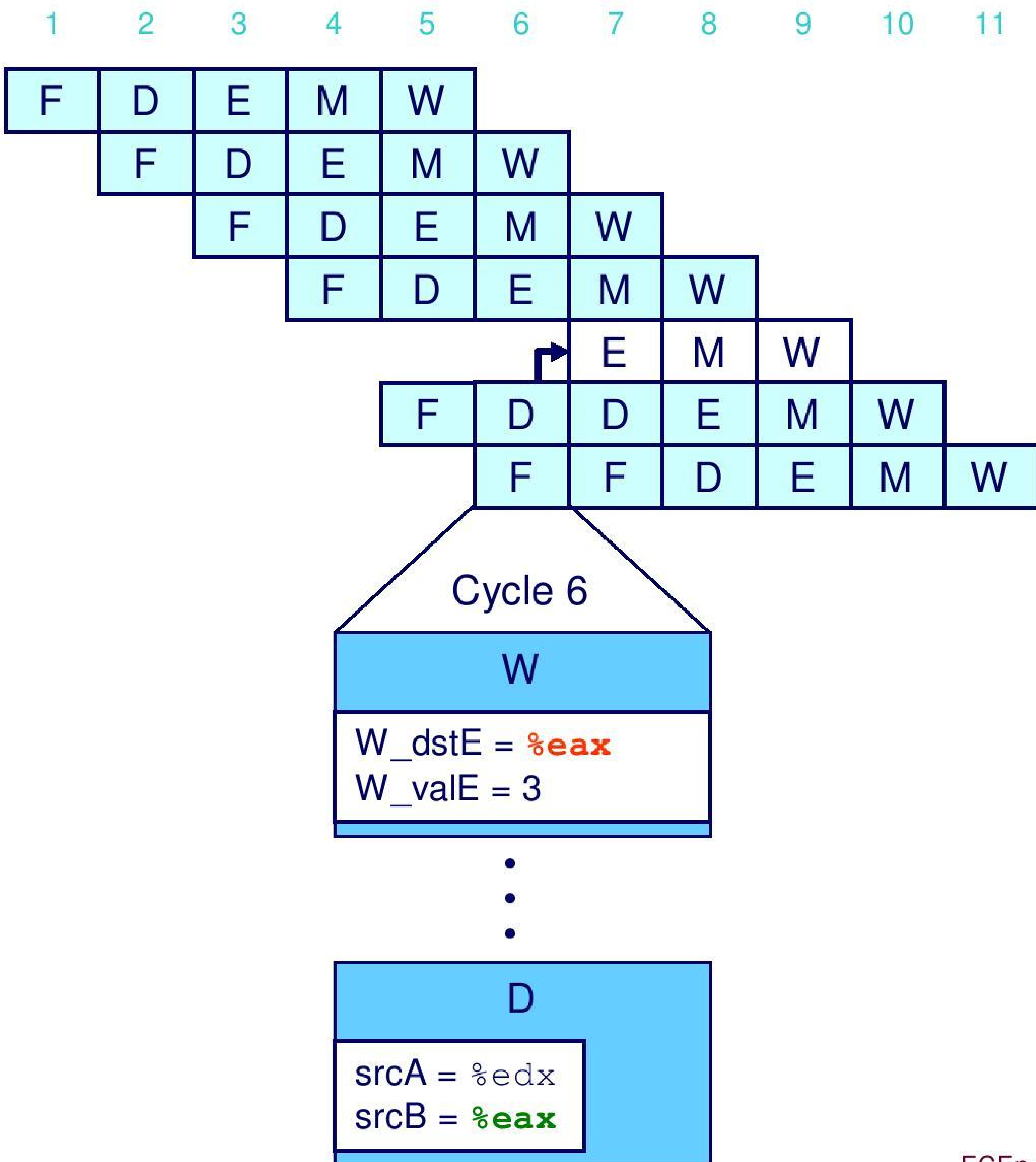
Special Case

- If srcX register ID is 8, don't stall even on match
  - Indicates absence of register operand



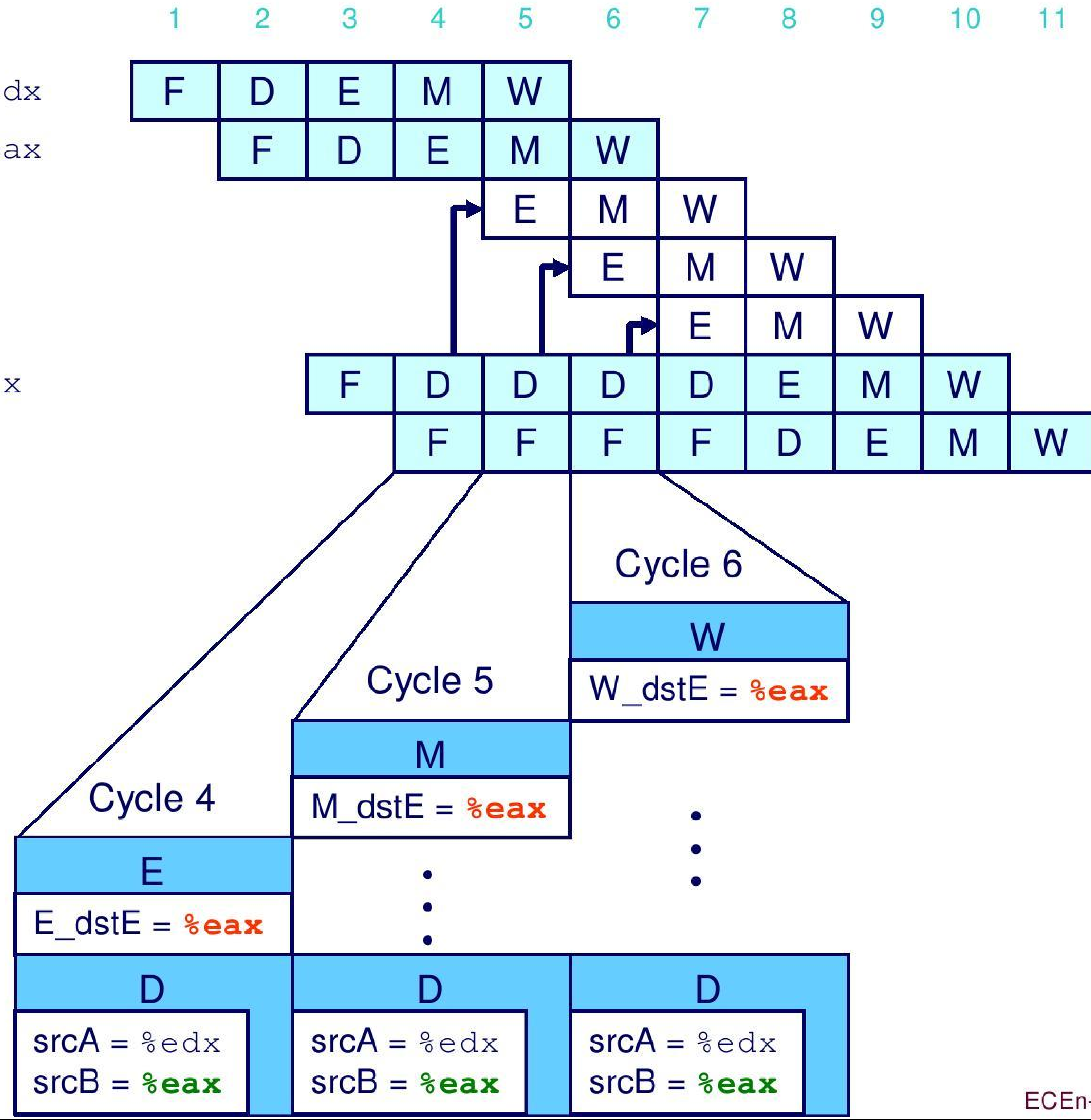
Detecting Stall Condition

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



Multi-cycle Stalls

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



Stalling: Another View

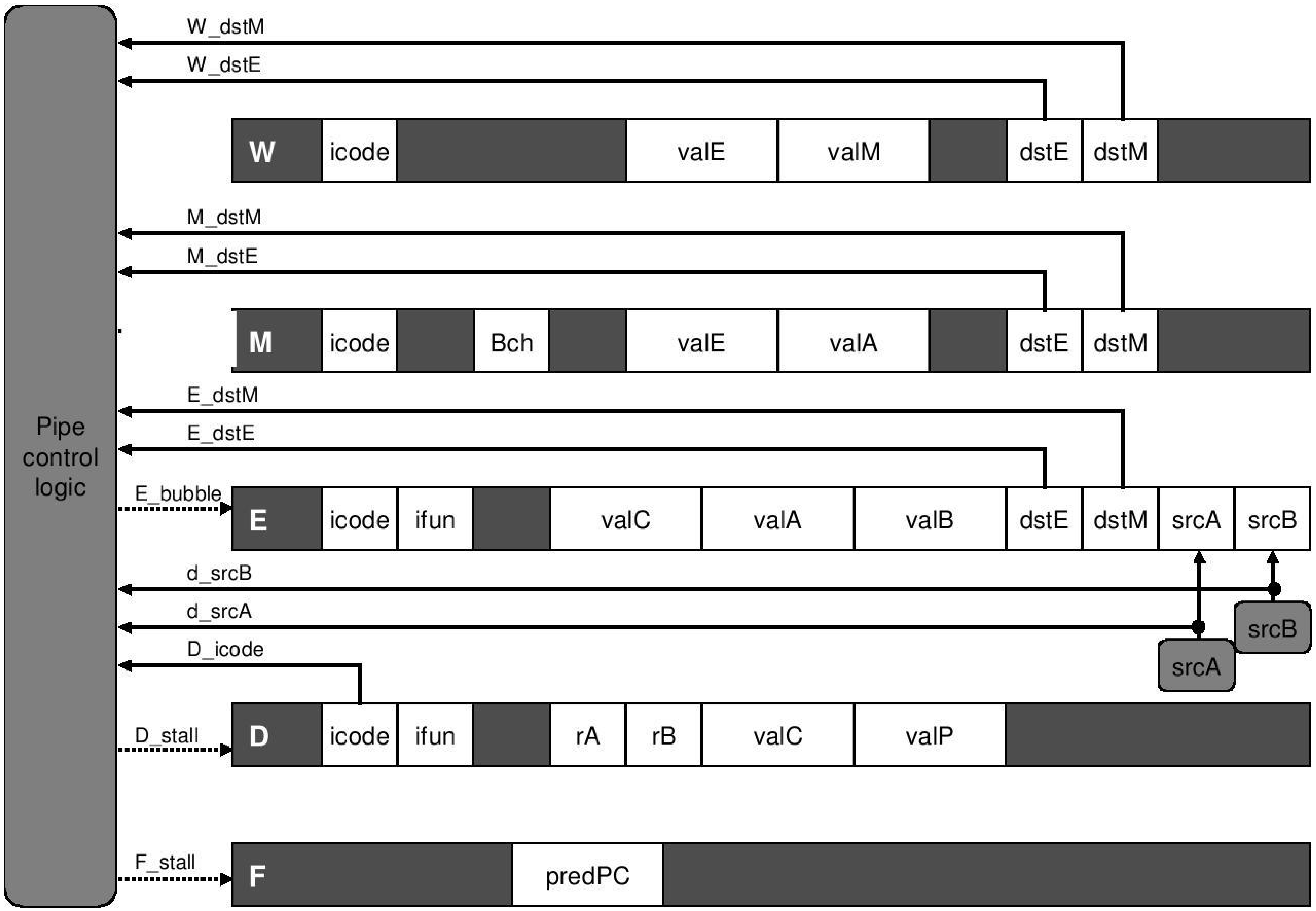
```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

	Cycle 8
Write Back	***bubble***
Memory	***bubble***
Execute	`0x00c: addl %edx,%eax`
Decode	`0x00e: halt`
Fetch	

Operational Rules:

- **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 stage-by-stage as regular instructions**

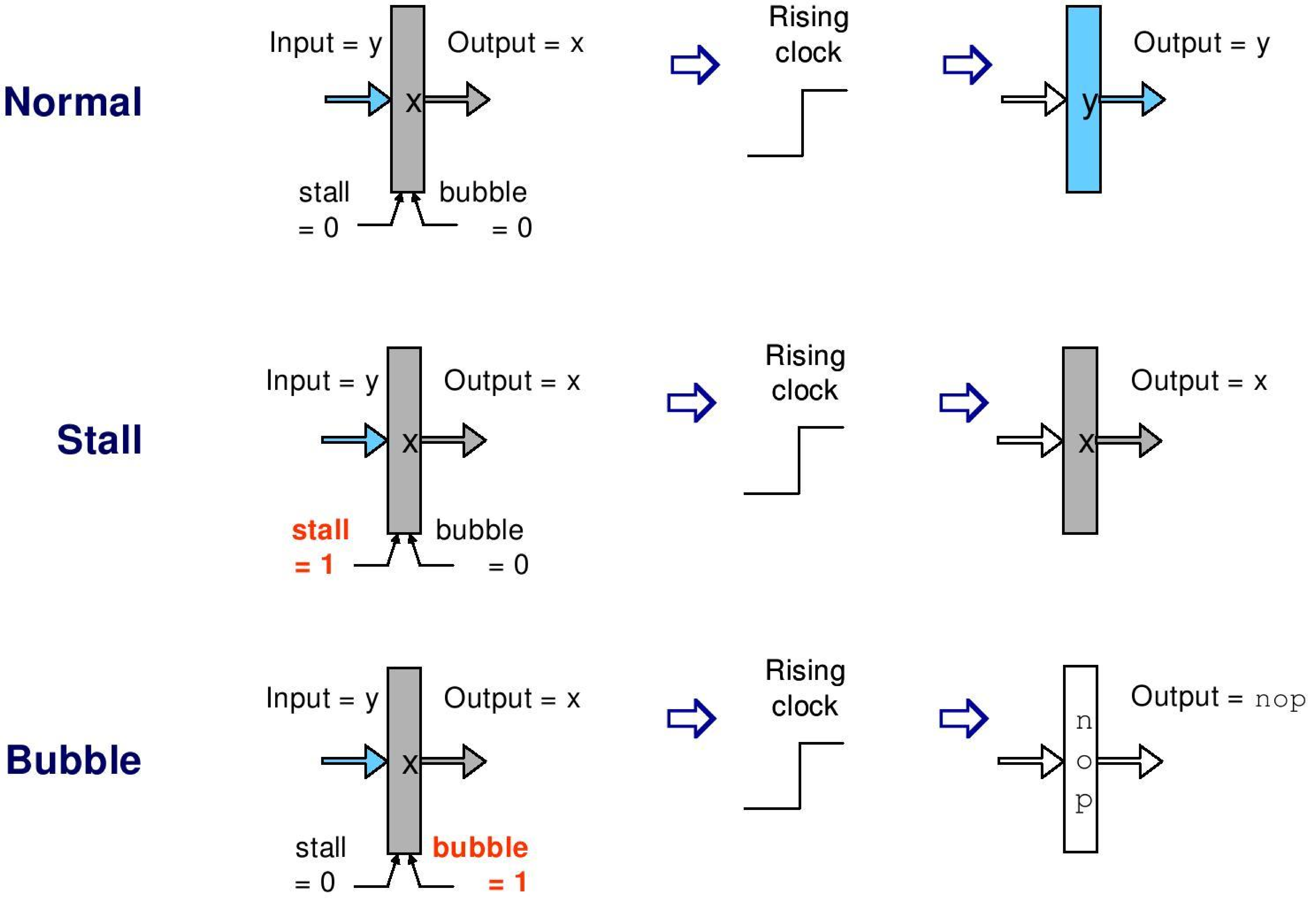
Implementing Stalling



Pipeline Control

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

Pipeline Register Modes



Data Forwarding

Our Naïve Pipeline Would Experience Many Data Stalls

- 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
  - Leads to many more stall cycles than necessary

Observation

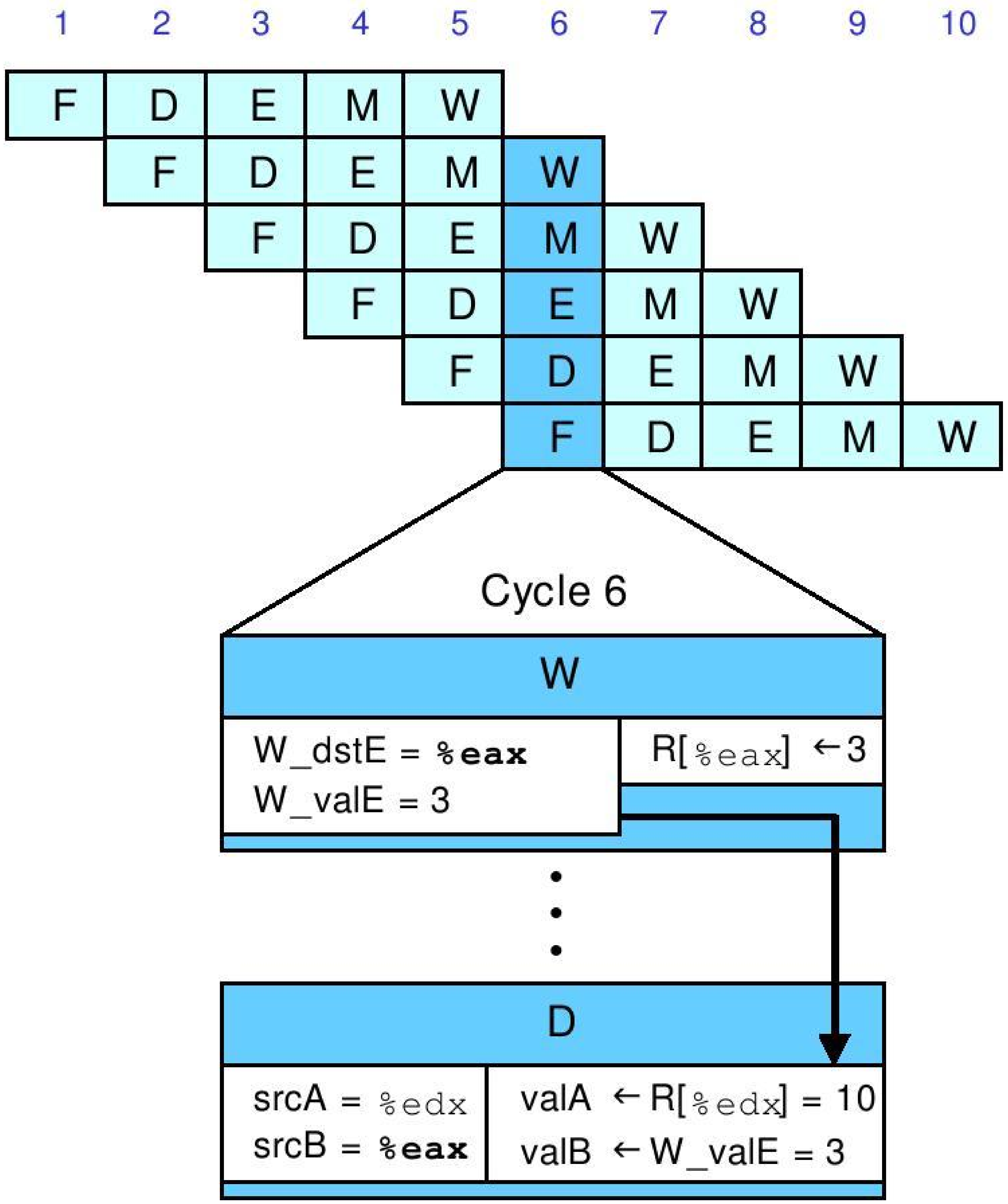
- The value we want is generated in execute or memory stage
- It is "available" 1-2 cycles before write-back

Trick: Go Get It!

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

Data Forwarding Example

```
# demo-h2.ys
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
- `addl` instruction can proceed without stalling

Bypass Paths

Decode Stage

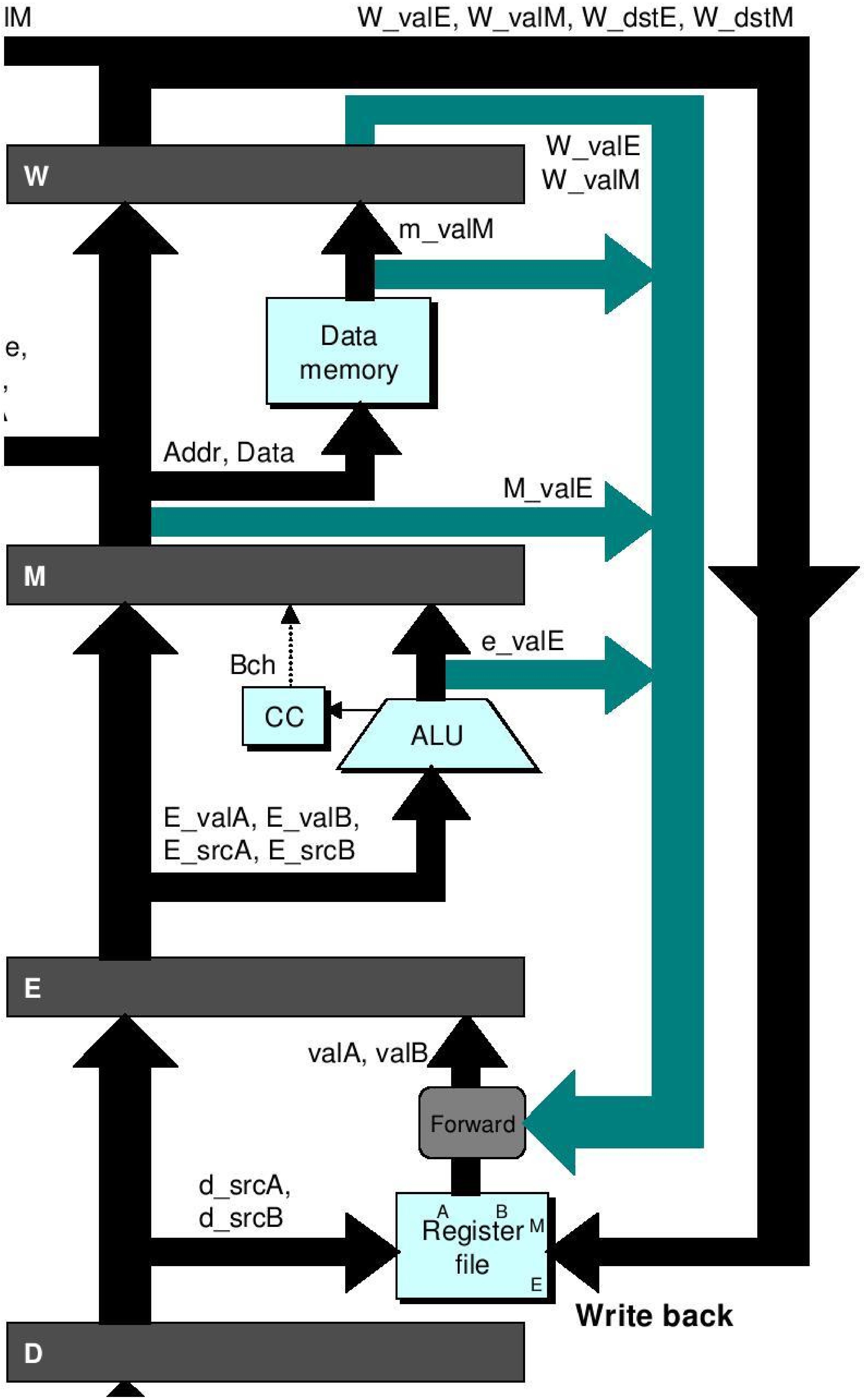
- Forwarding logic selects valA and valB
- Normally from register file
- When forwarding required, valA or valB come from later pipeline stage

Forwarding Sources

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

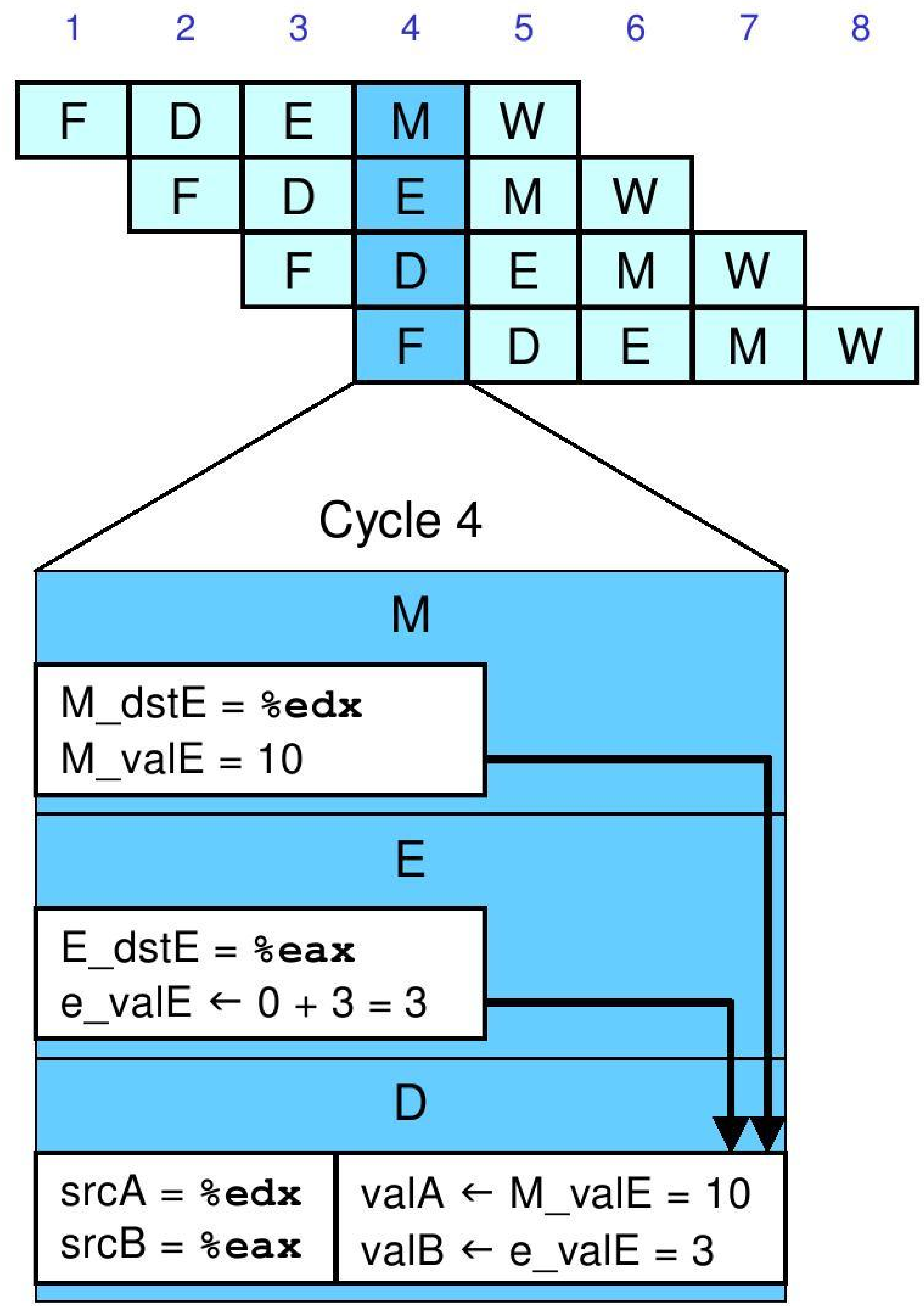
Detection

- Comparing regids, same as in previous stall examples



Data Forwarding Example #2

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



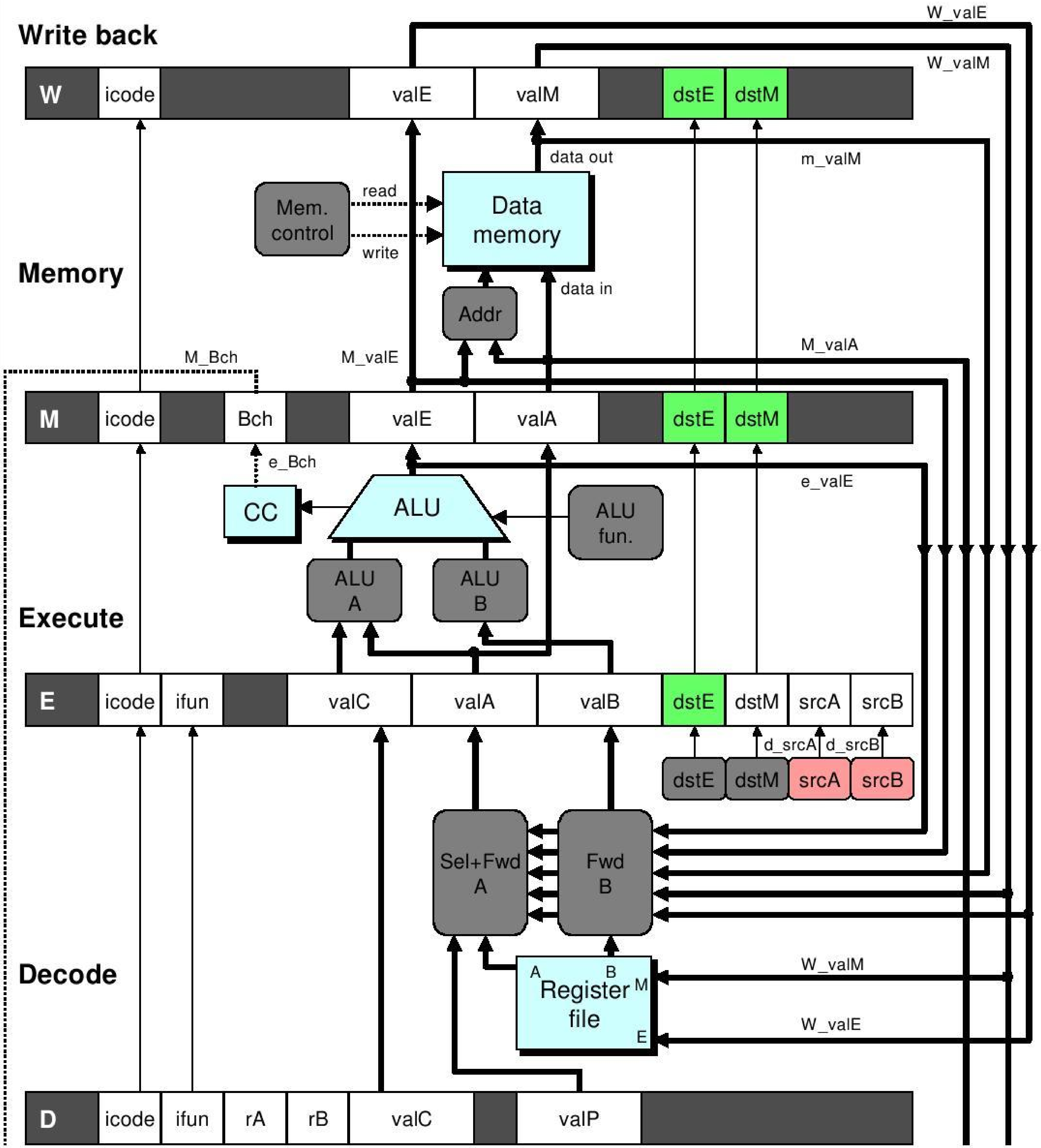
Register `%edx`

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

Register `%eax`

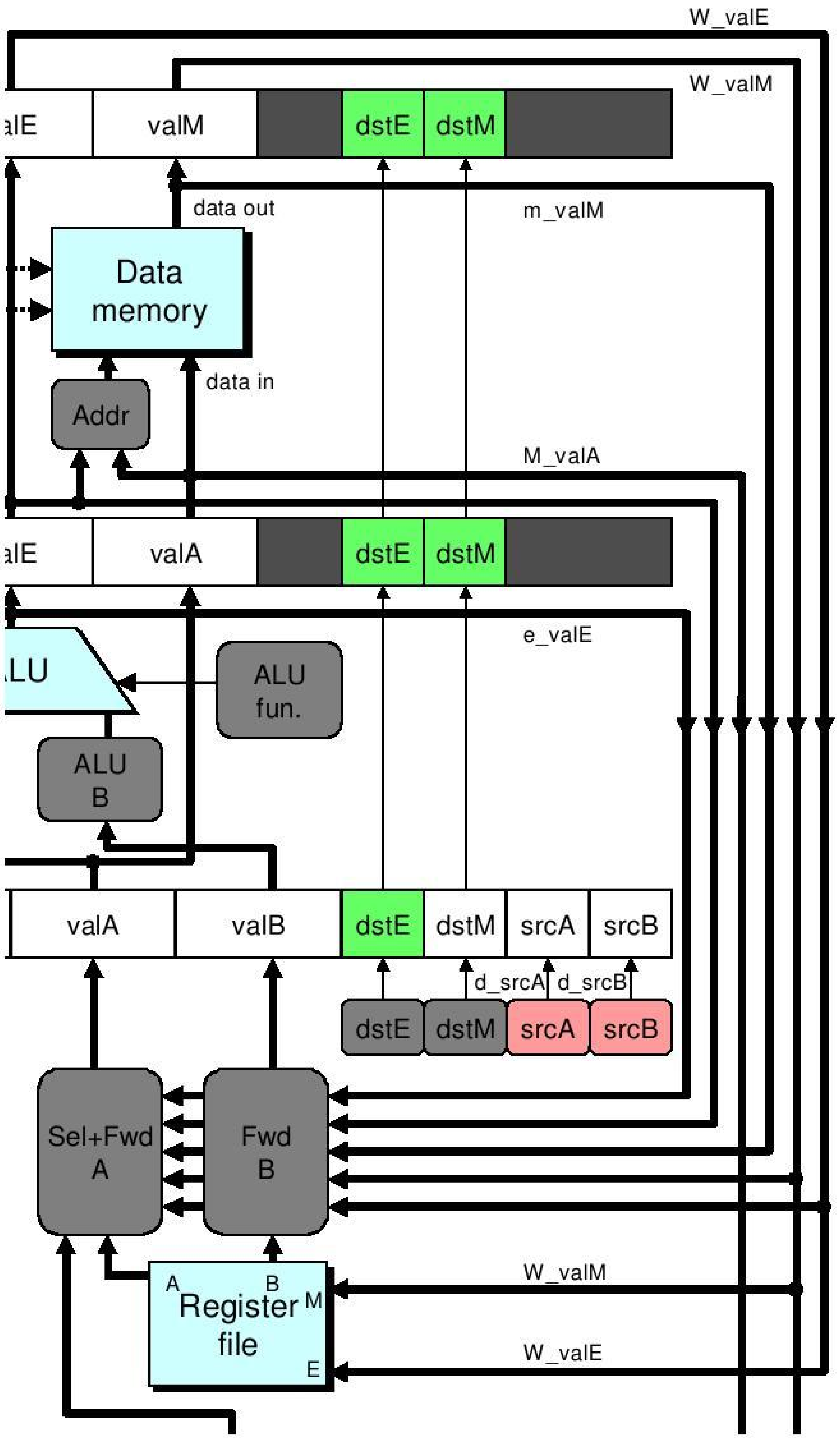
- Value just generated by ALU
- Forward from execute as valB

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
- Note: we either do this or use stall control logic
  - Forwarding results in much better performance

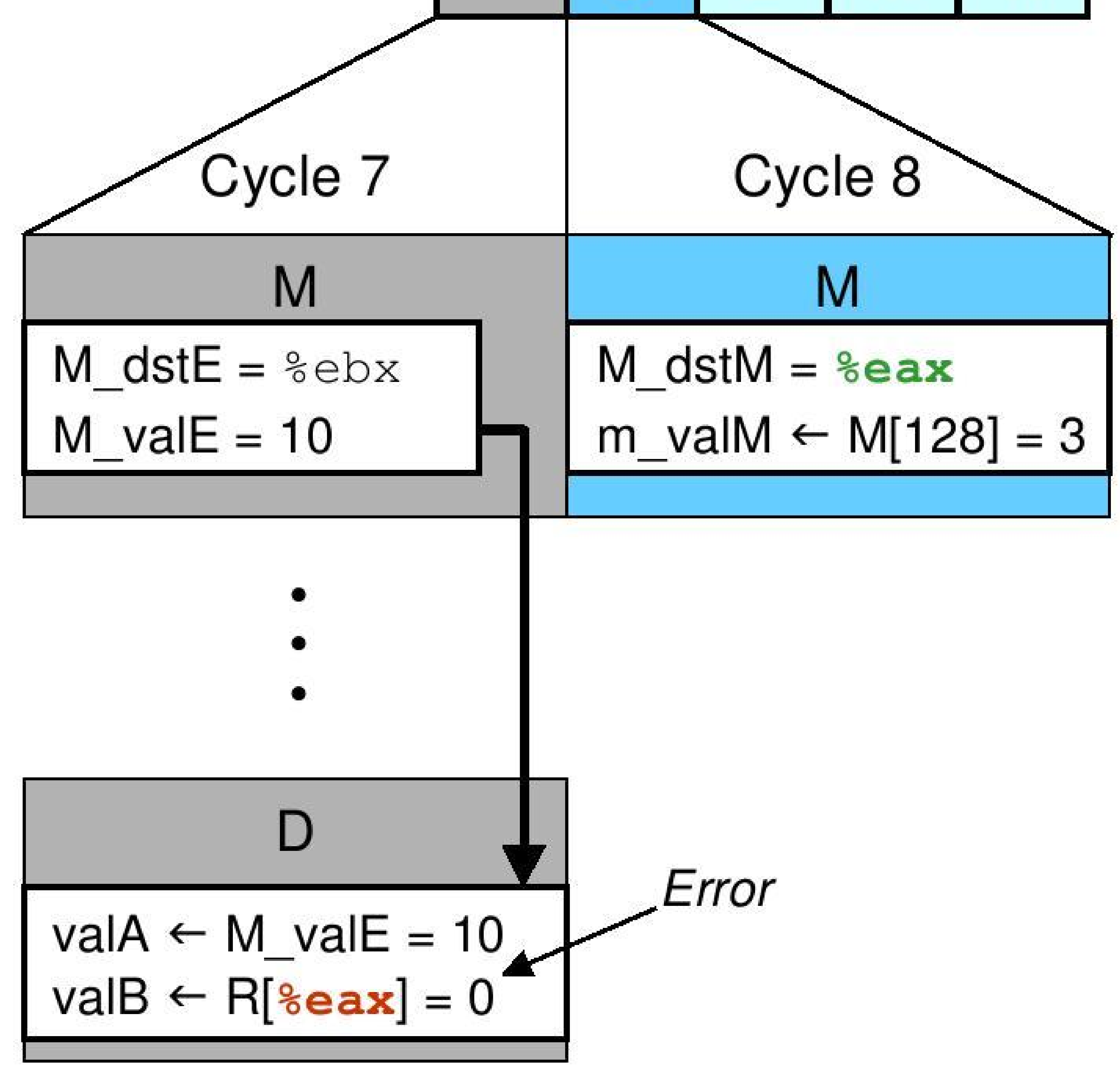
Implementing Forwarding




```
## 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.ys	1	2	3	4	5	6	7	8	9	10	11
0x000: irmovl $128,%edx	F	D	E	M	W						
0x006: irmovl $3,%ecx		F	D	E	M	W					
0x00c: rmmovl %ecx, 0(%edx)			F	D	E	M	W				
0x012: irmovl $10,%ebx				F	D	E	M	W			
0x018: mrmovl 0(%edx),**%eax** **# Load %eax**					F	D	E	M	W		
0x01e: addl %ebx,**%eax** **# Use %eax**						F	D	E	M	W	
0x020: halt							F	D	E	M	W



Load-use dependency

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

Terminology

- **This is a *load hazard***
- **One solution is a *load stall***