

Processor Architecture: Pipelined Implementation Part I

Lecture 6 - 2015
Mads Chr. Olesen

***Slides by Randal E. Bryant
Carnegie Mellon University***

Overview

General Principles of Pipelining

- Goal
- Difficulties

Creating a Pipelined Y86 Processor

- Rearranging SEQ
- Inserting pipeline registers
- Problems with data and control hazards

Real-World Pipelines: Car Washes

Sequential



Parallel



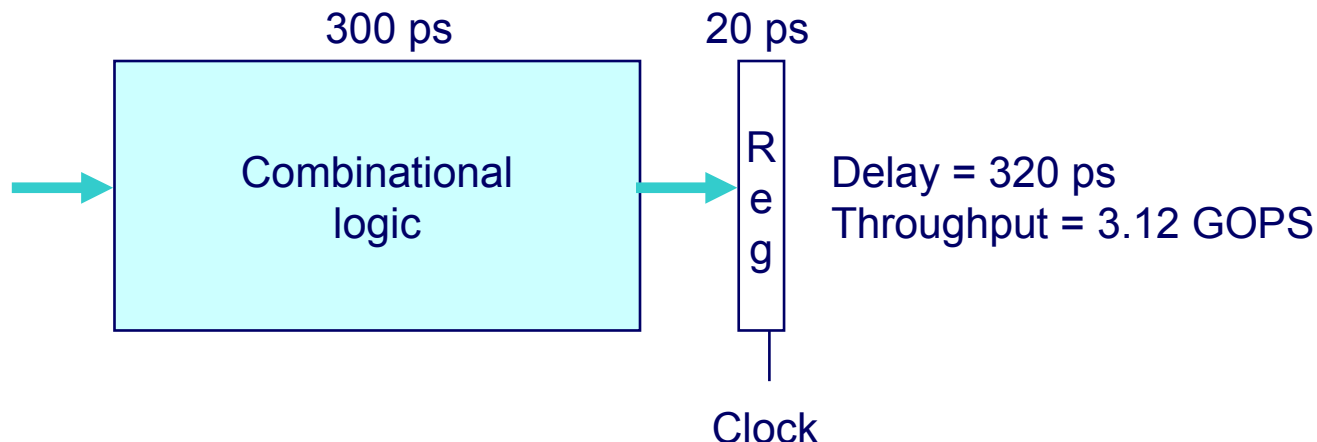
Pipelined



Idea

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

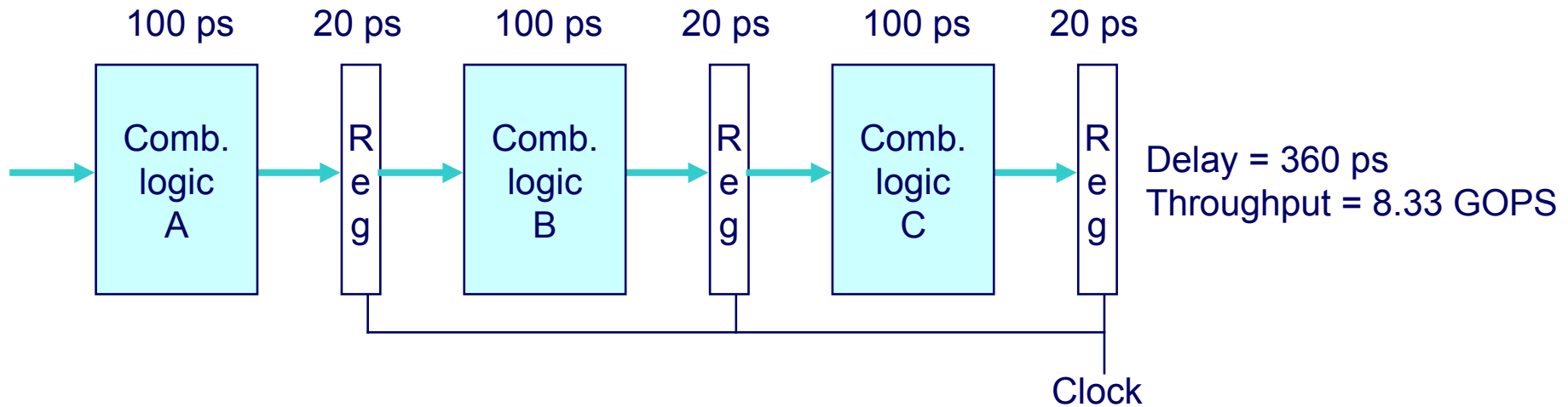
Computational Example



System

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

3-Way Pipelined Version

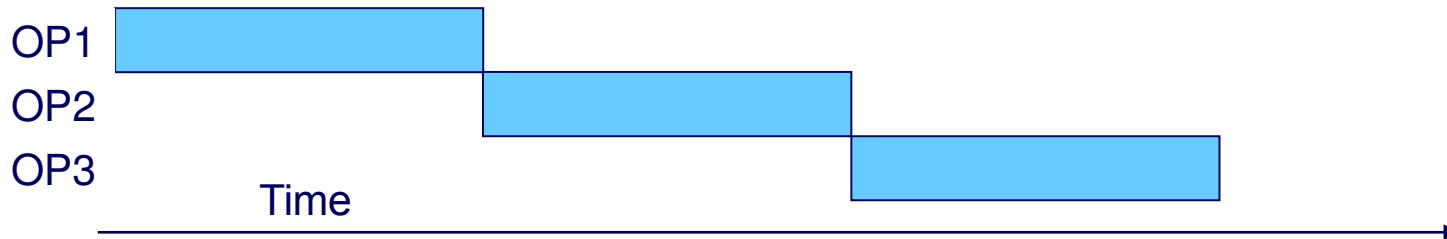


System

- 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

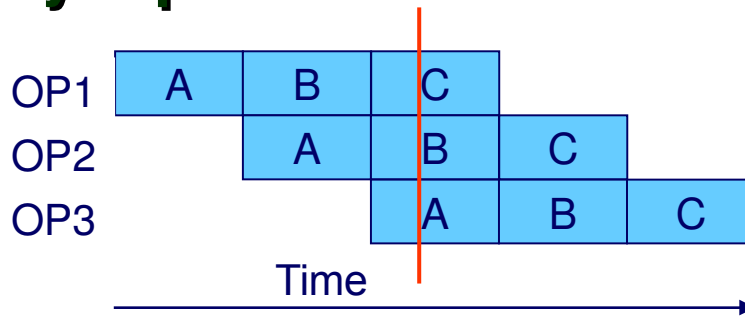
Pipeline Diagrams

Unpipelined



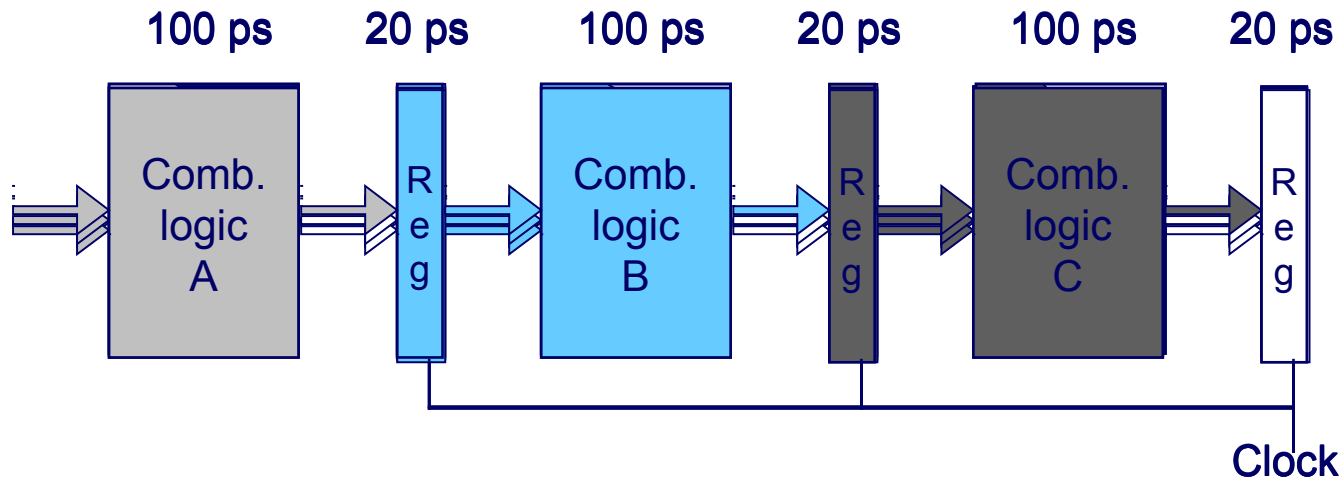
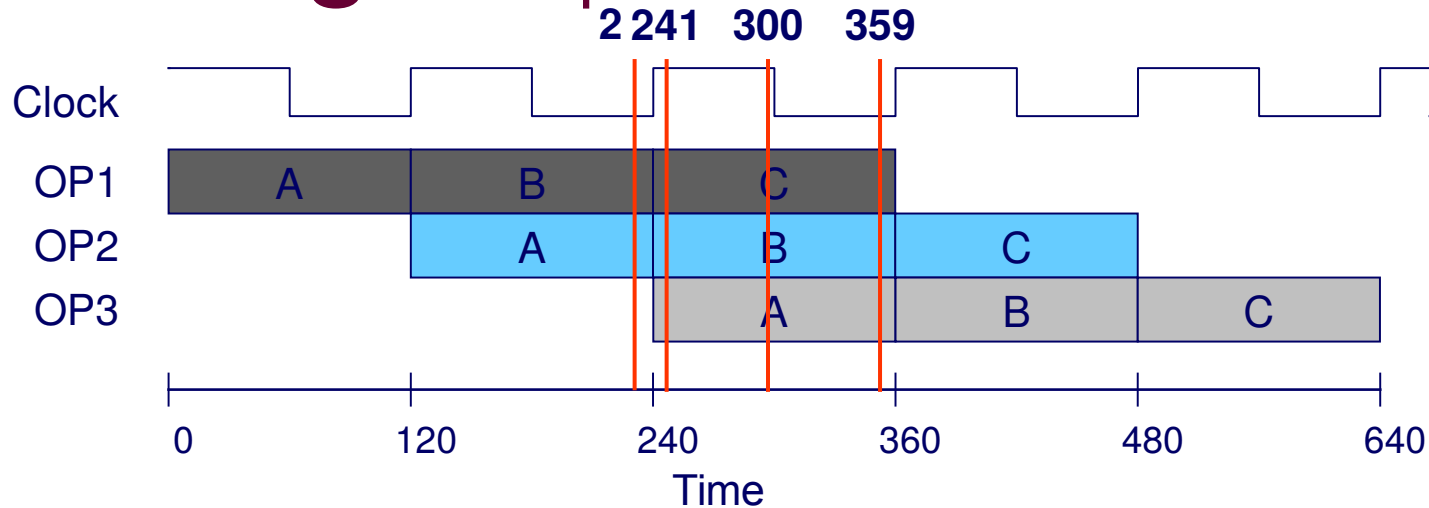
- Cannot start new operation until previous one completes

3-Way Pipelined

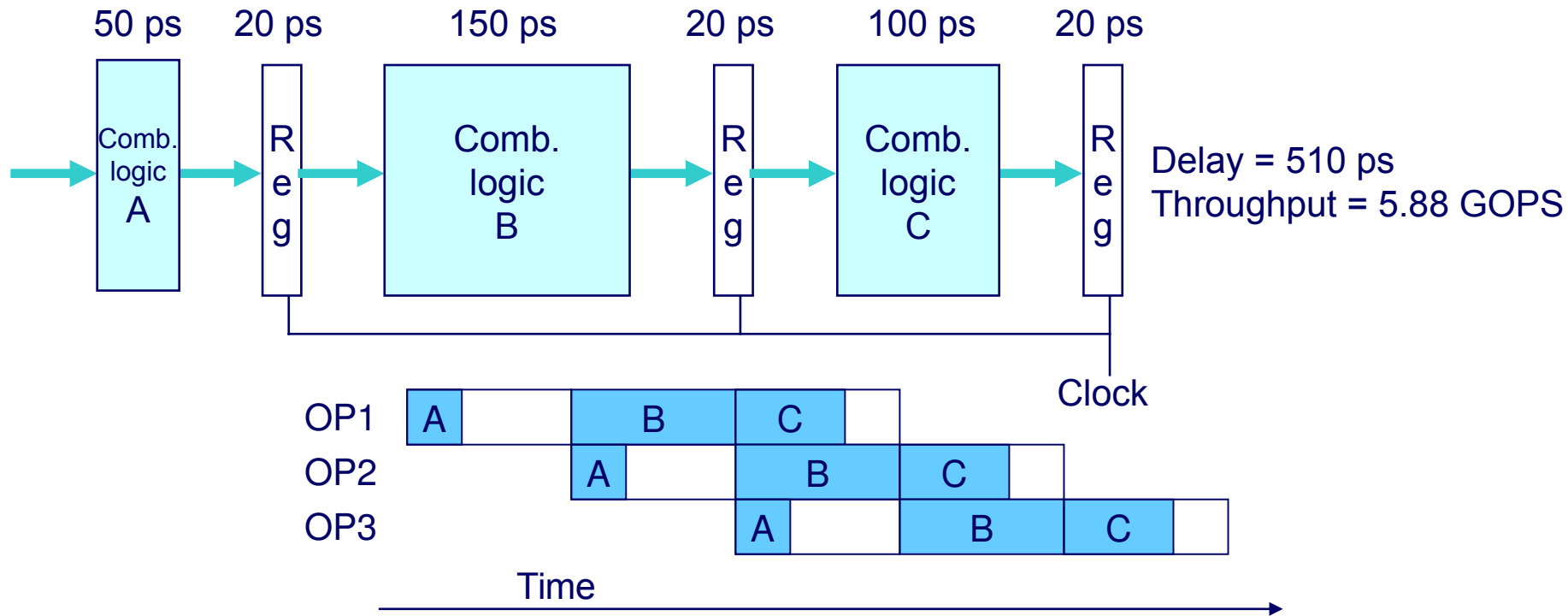


- Up to 3 operations in process simultaneously

Operating a Pipeline

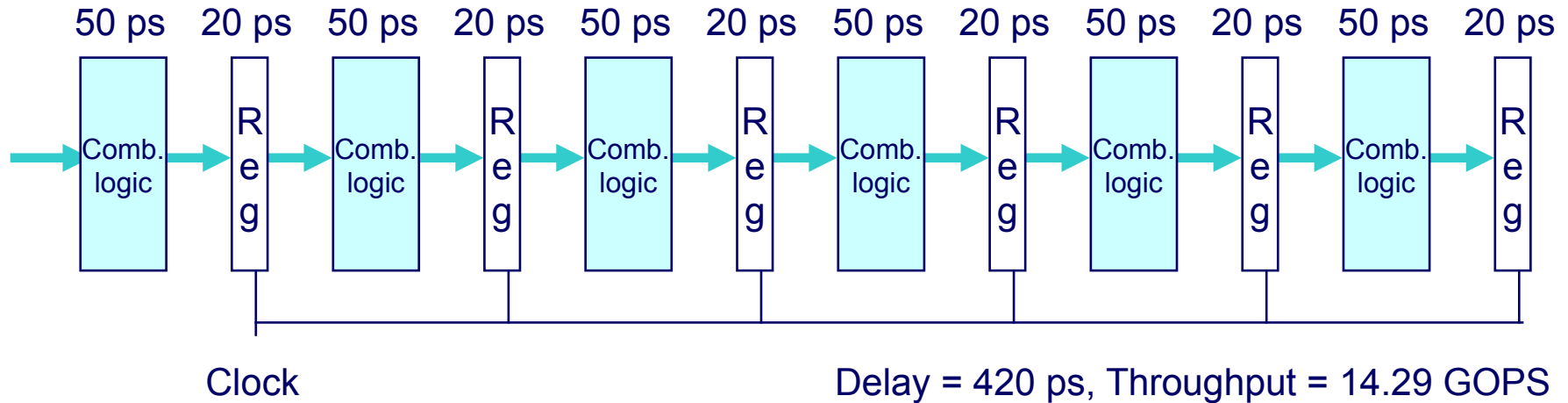


Limitations: Nonuniform Delays



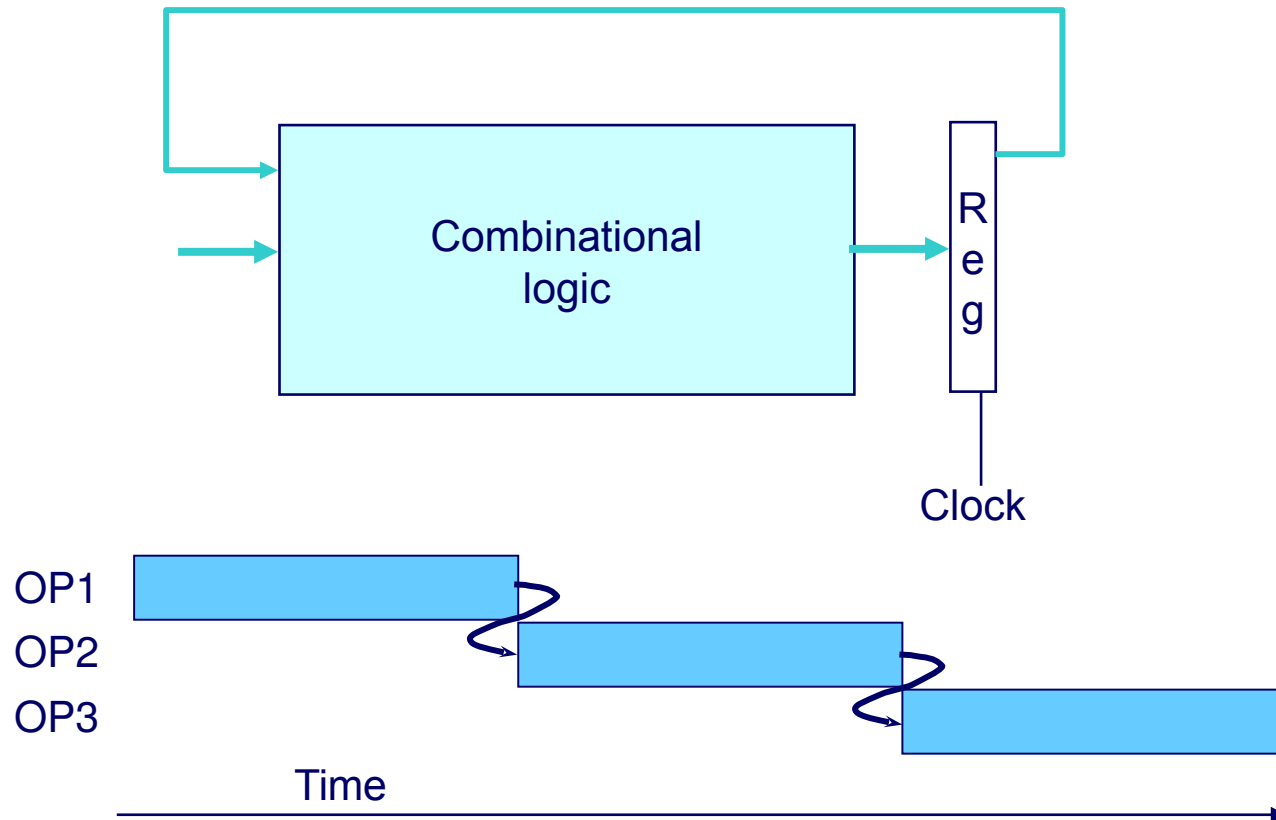
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



- 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

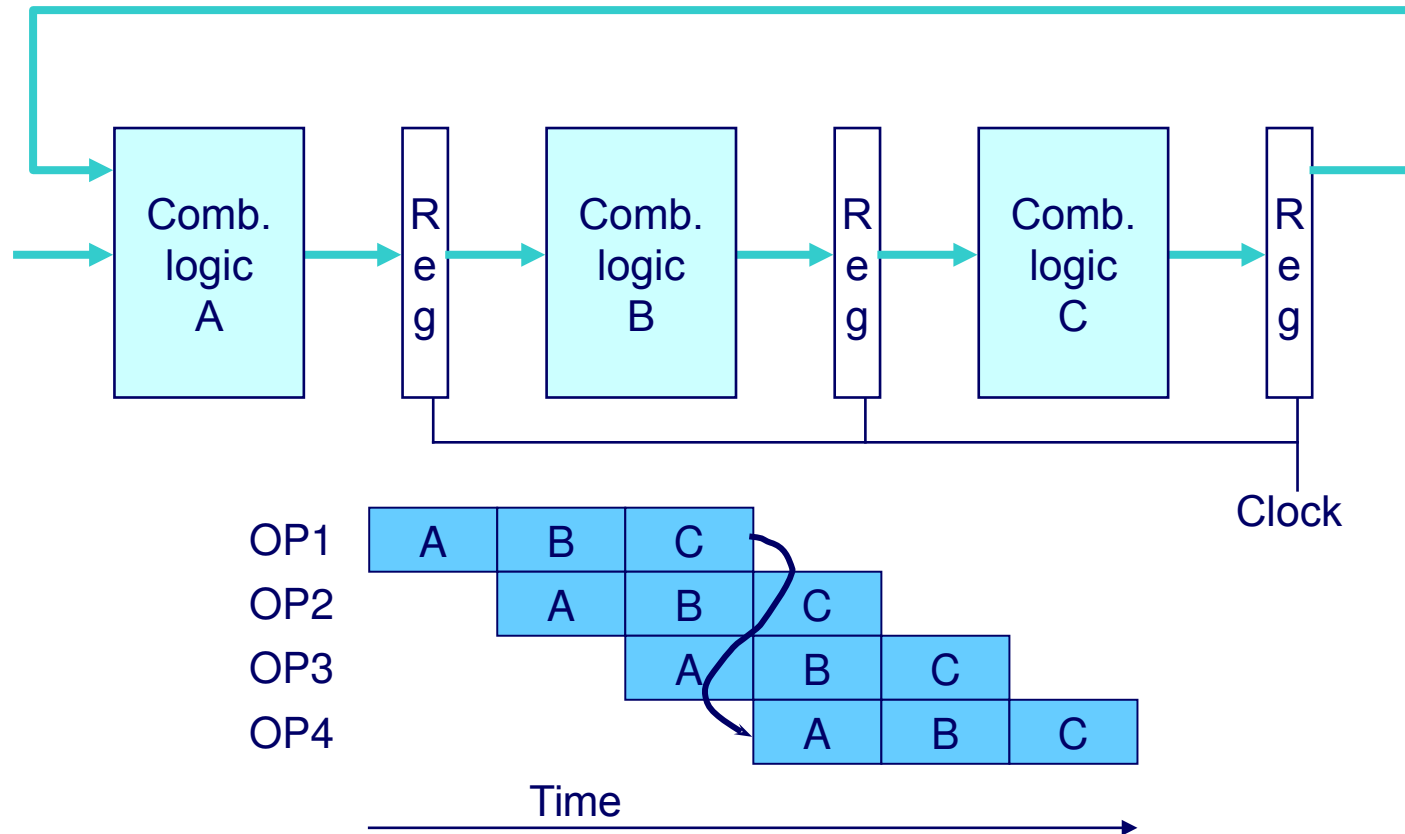
Data Dependencies



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

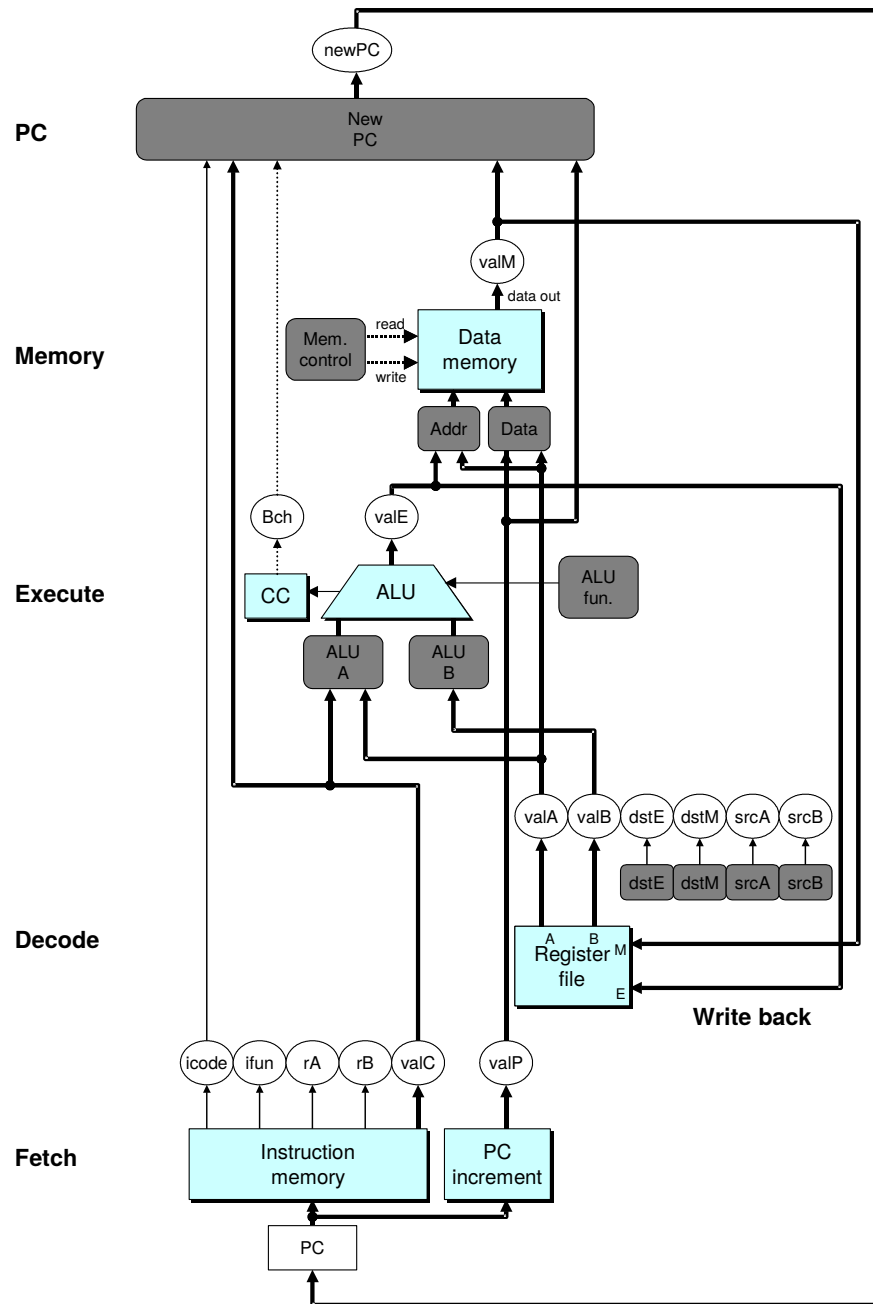
```
1    irmovl $50, %eax
2    addl %eax, %ebx
3    mrmovl 100( %ebx ), %edx
```

The diagram illustrates data dependencies between three instructions. Instruction 1 (irmovl \$50, %eax) writes to the %eax register. Instruction 2 (addl %eax, %ebx) reads the value from %eax and writes to %ebx. Instruction 3 (mrmovl 100(%ebx), %edx) reads the value from %ebx. Arrows show the data flow: from %eax in instruction 1 to %eax in instruction 2, and from %ebx in instruction 2 to %ebx in instruction 3.

- 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

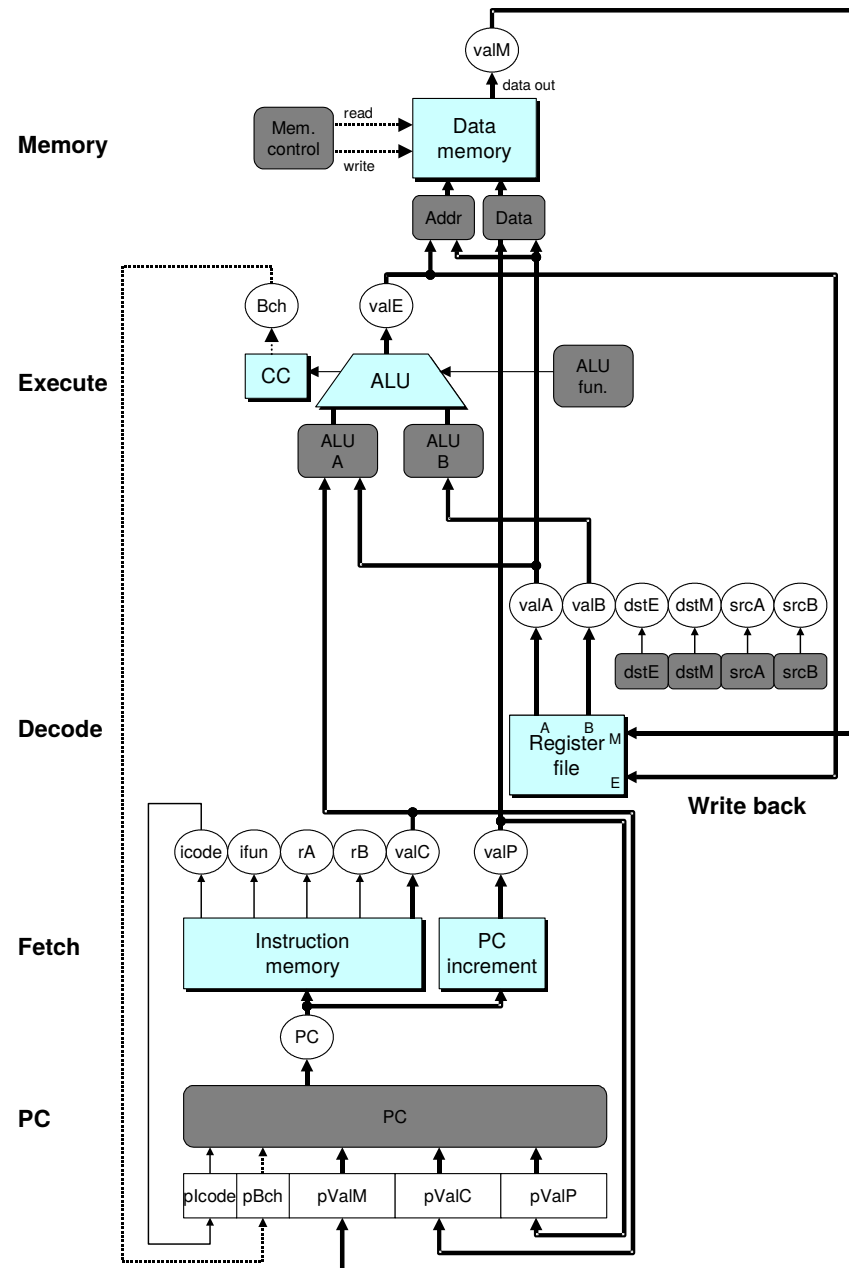
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

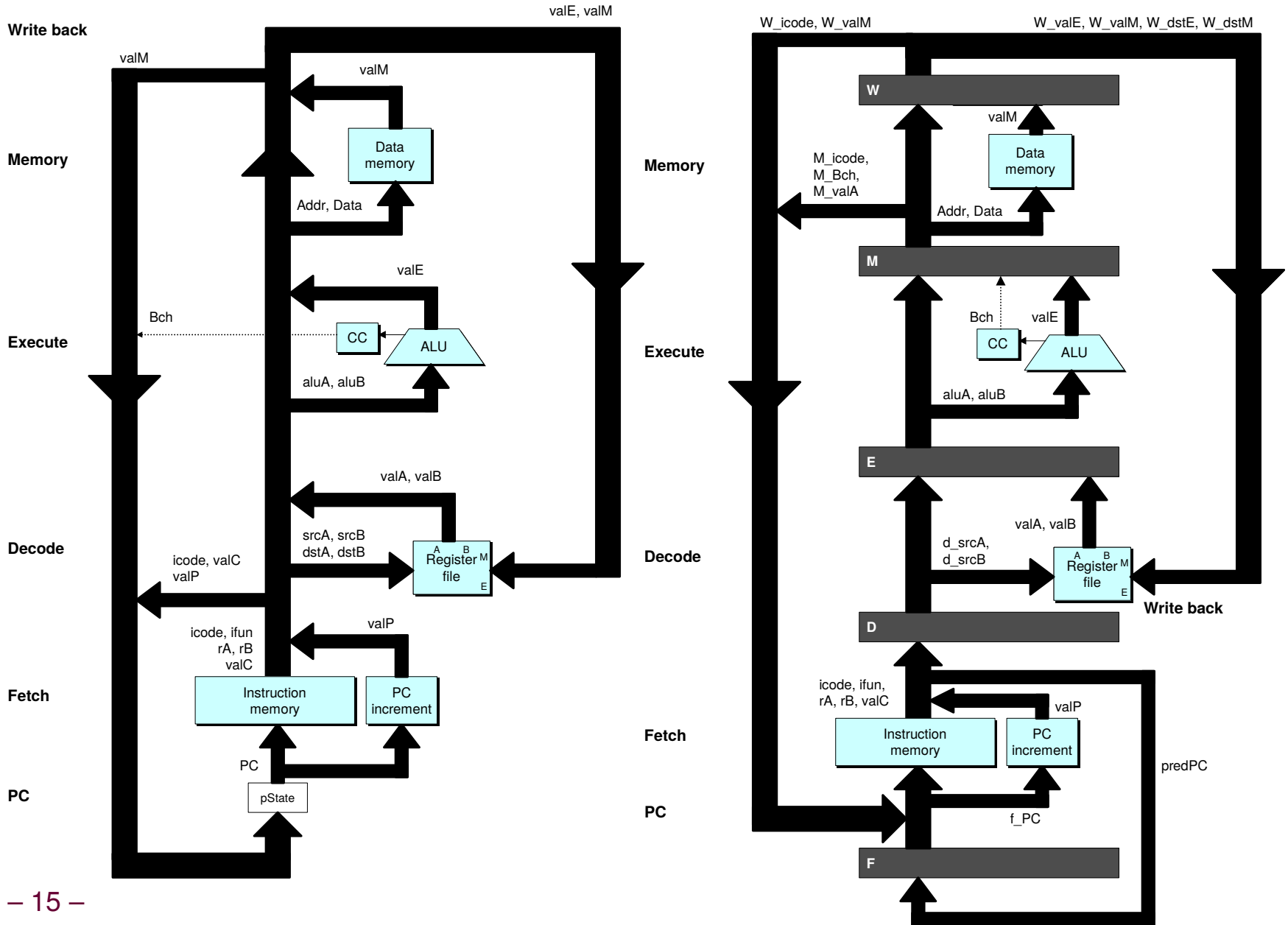
- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- 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

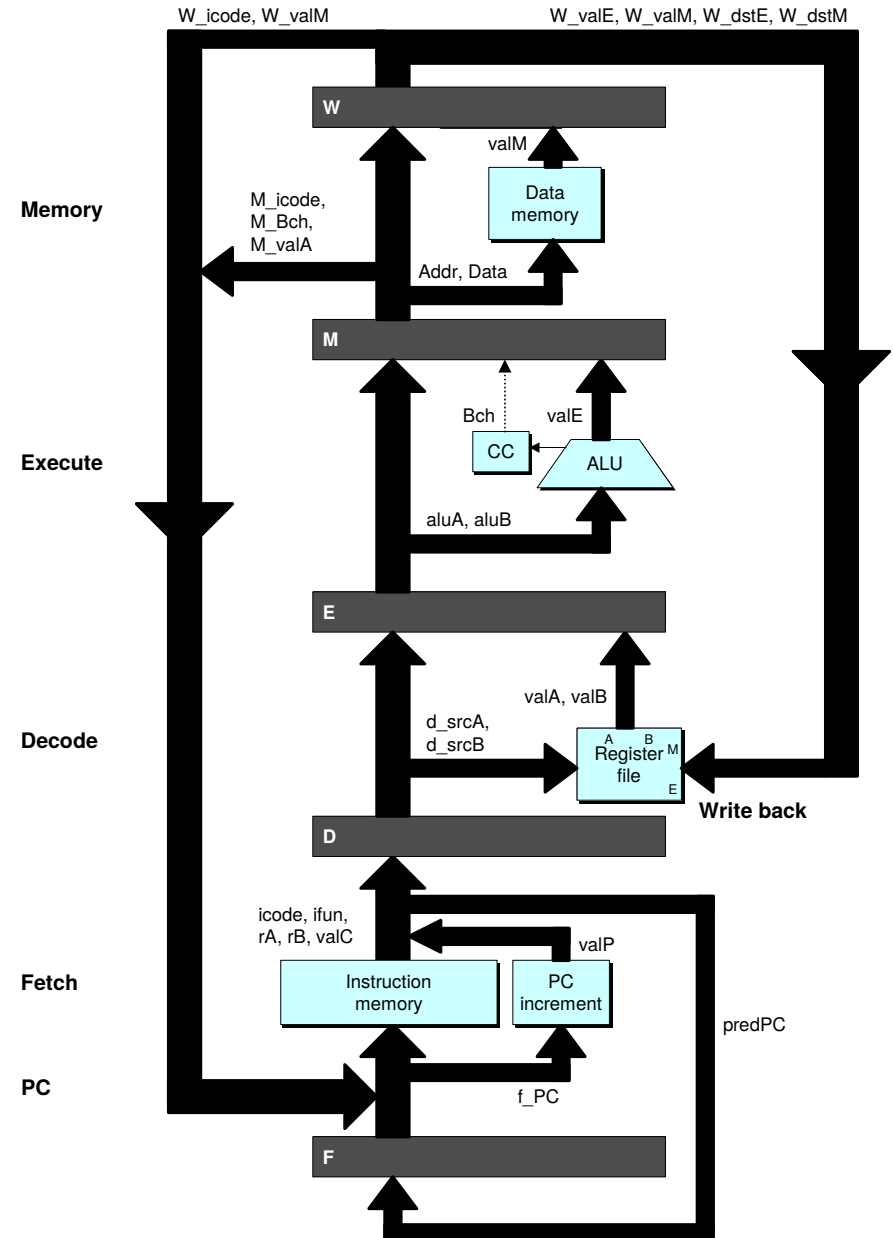
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file



HW Registers

F

- Holds a *predicted* value of PC.

D

- Holds the most recently fetched instruction.

E

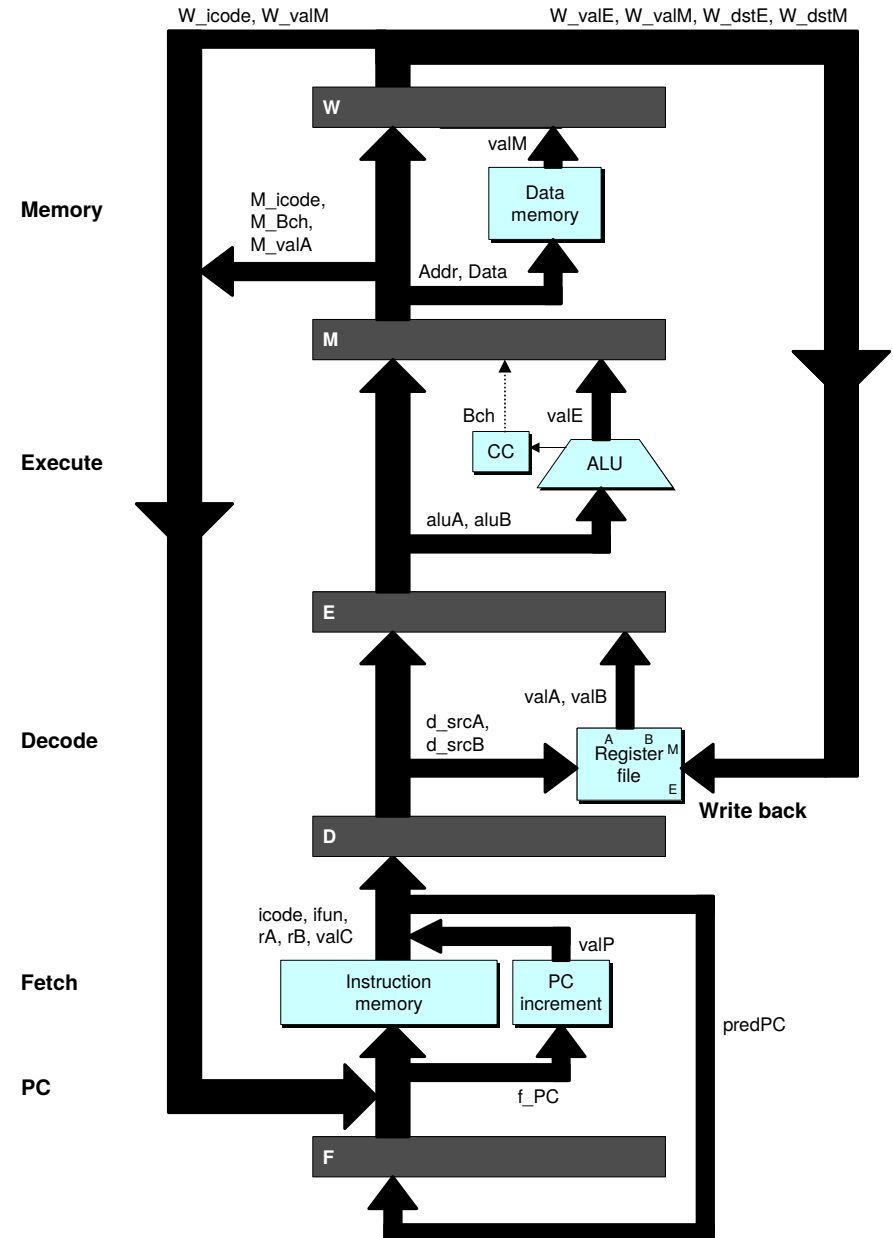
- Holds the most recently decoded instruction & register values.

M

- Holds results of the most recently executed instructions.
- Holds info on branch prediction and branch targets.

W

- Supplies results to the register file + return @ for PC for `ret` instructions.

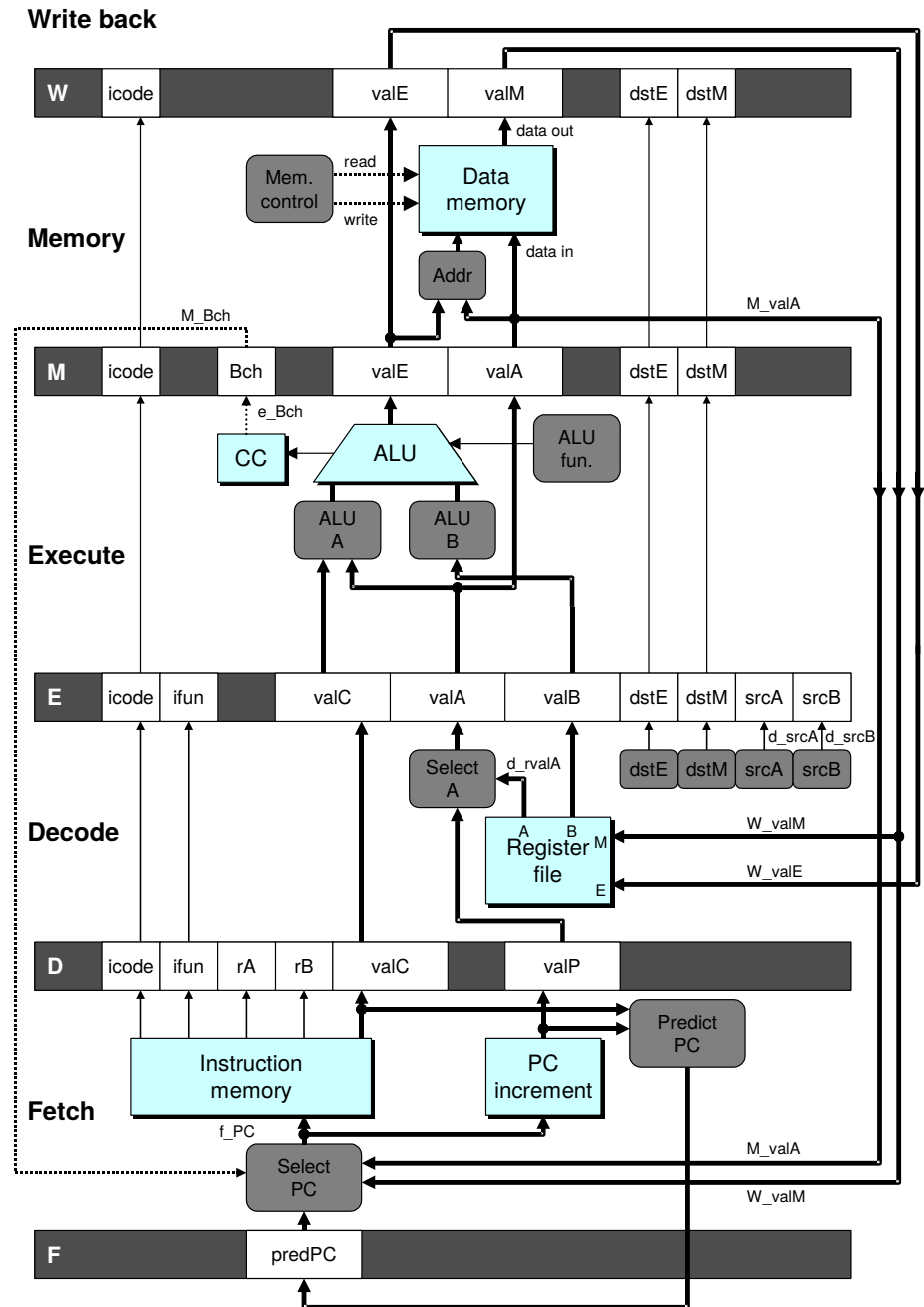


PIPE- Hardware

- 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

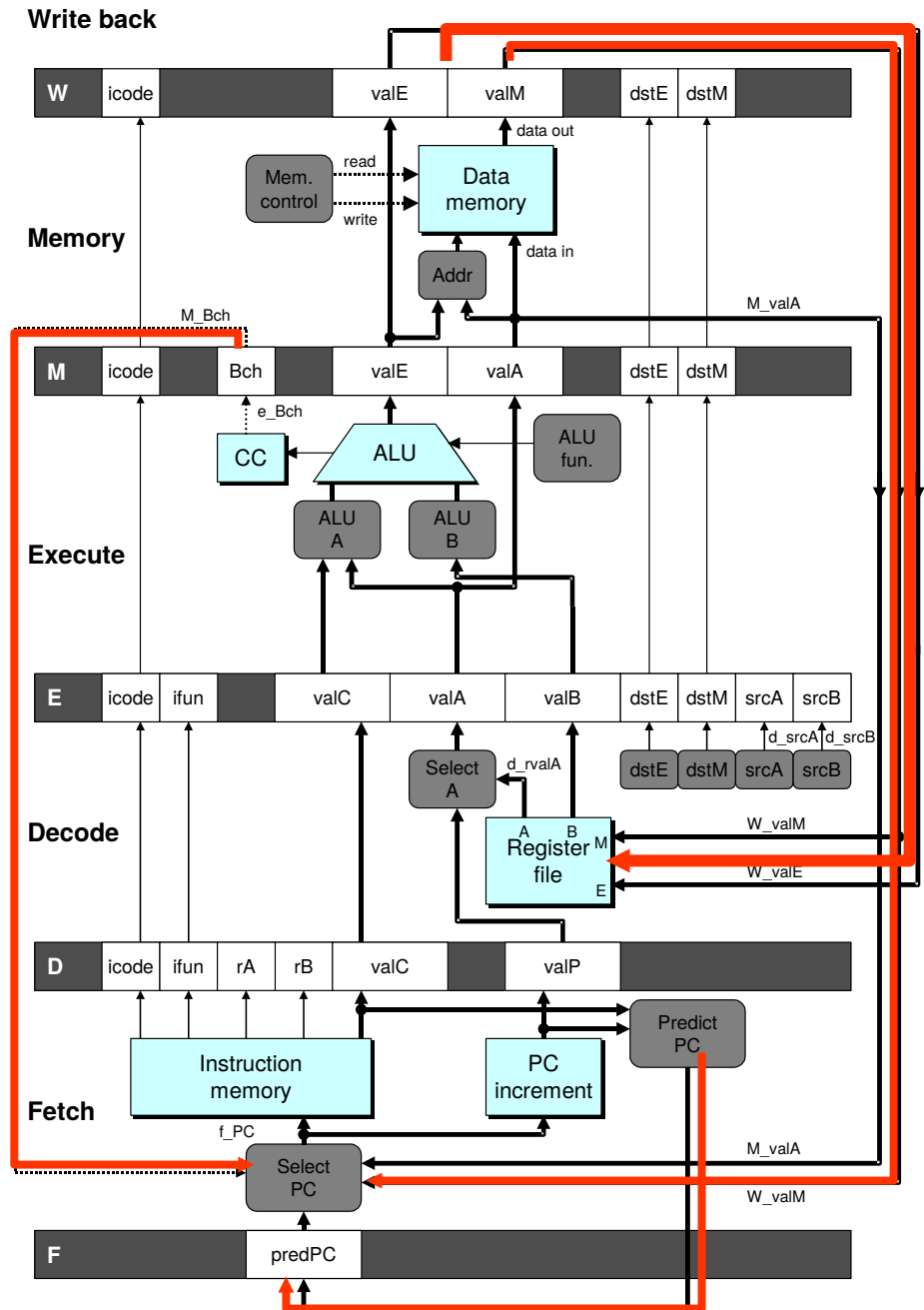
- Jump taken/not-taken
- Fall-through or target address

Return point

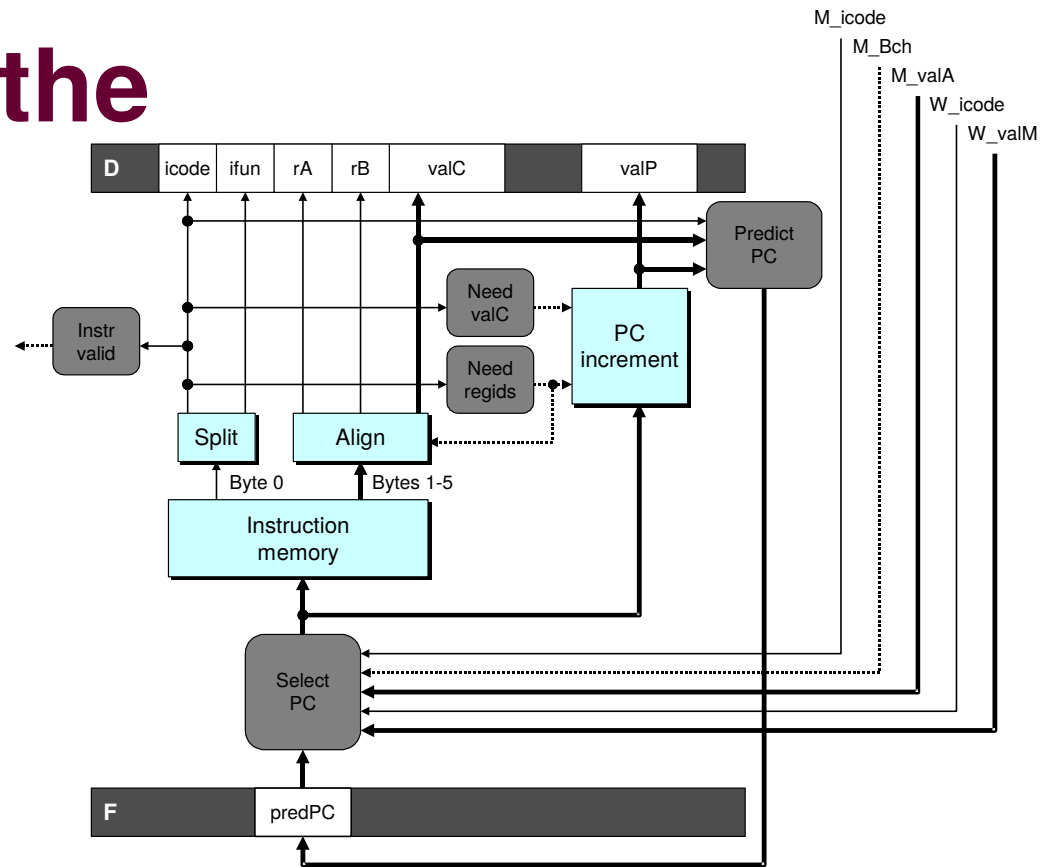
- Read from memory

Register updates

- To register file write ports



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

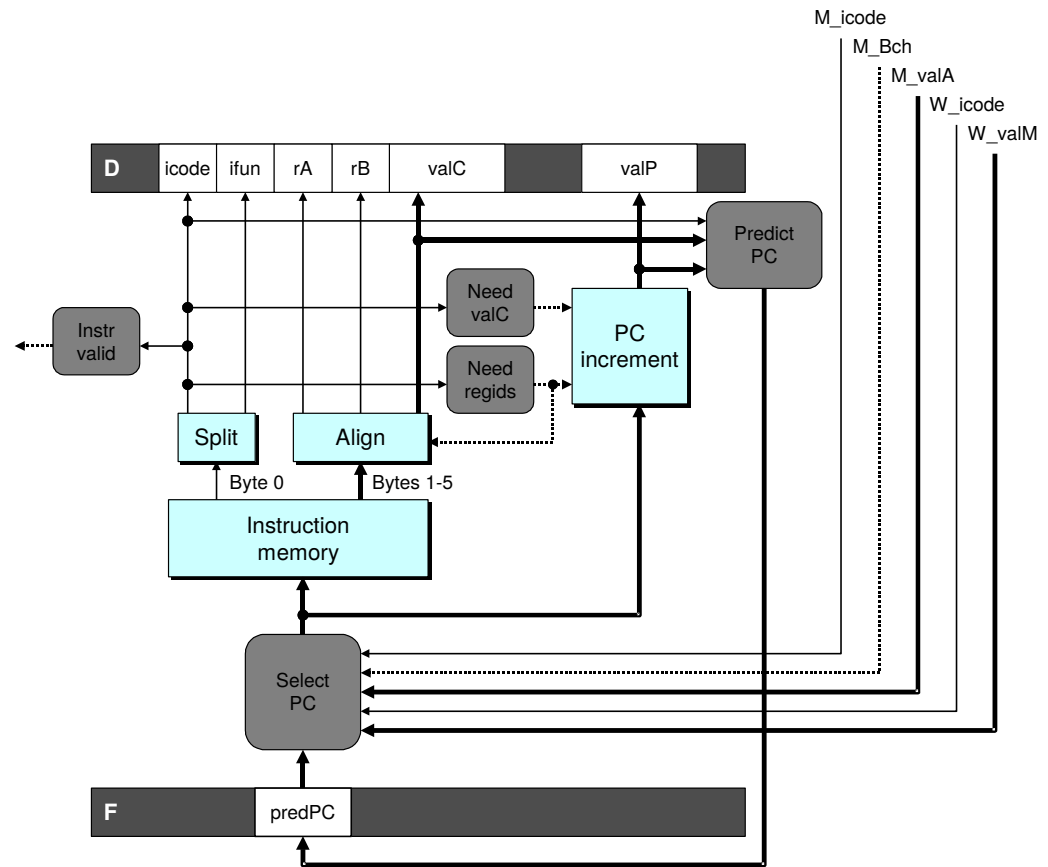
Conditional Jumps

- 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 – but in practice there's a way.

Recovering from PC Misprediction



■ Mispredicted Jump

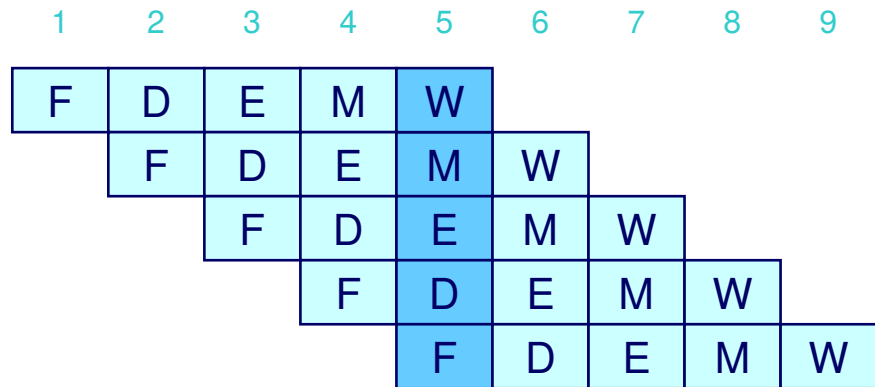
- Will see branch flag once instruction reaches memory stage
- Can get fall-through PC from valA

■ Return Instruction

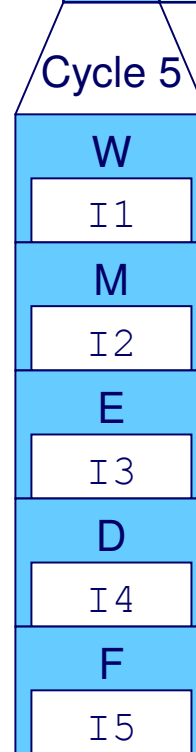
- Will get return PC when `ret` reaches write-back stage

Pipeline Demonstration

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



File: demo-basic.js



Branch Misprediction Example

demo-j.ys

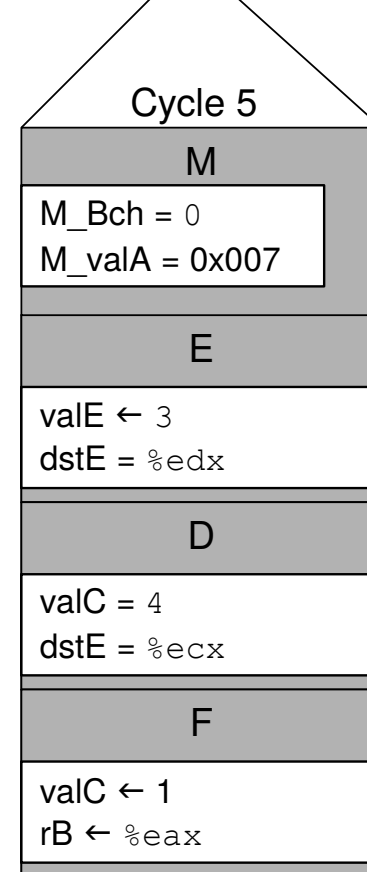
```
0x000:    xorl %eax,%eax
0x002:    jne  t                # Not taken
0x007:    irmovl $1, %eax        # Fall through
0x00d:    nop
0x00e:    nop
0x00f:    nop
0x010:    halt
0x011:  t:  irmovl $3, %edx    # Target (Should not execute)
0x017:    irmovl $4, %ecx    # Should not execute
0x01d:    irmovl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

- Should only execute first 8 instructions

Branch Misprediction Trace

# demo-j	1	2	3	4	5	6	7	8	9
0x000: xorl %eax,%eax	F	D	E	M	W				
0x002: jne t # Not taken		F	D	E	M	W			
0x011: t: irmovl \$3, %edx # Target			F	D	E	M	W		
0x017: irmovl \$4, %ecx # Target+1				F	D	E	M	W	
0x007: irmovl \$1, %eax # Fall Through					F	D	E	M	W

- Incorrectly execute two instructions at branch target



Data Dependencies: 3 Nop's

demo-h3.js

0x000: irmovl \$10,%edx

0x006: irmovl \$3,%eax

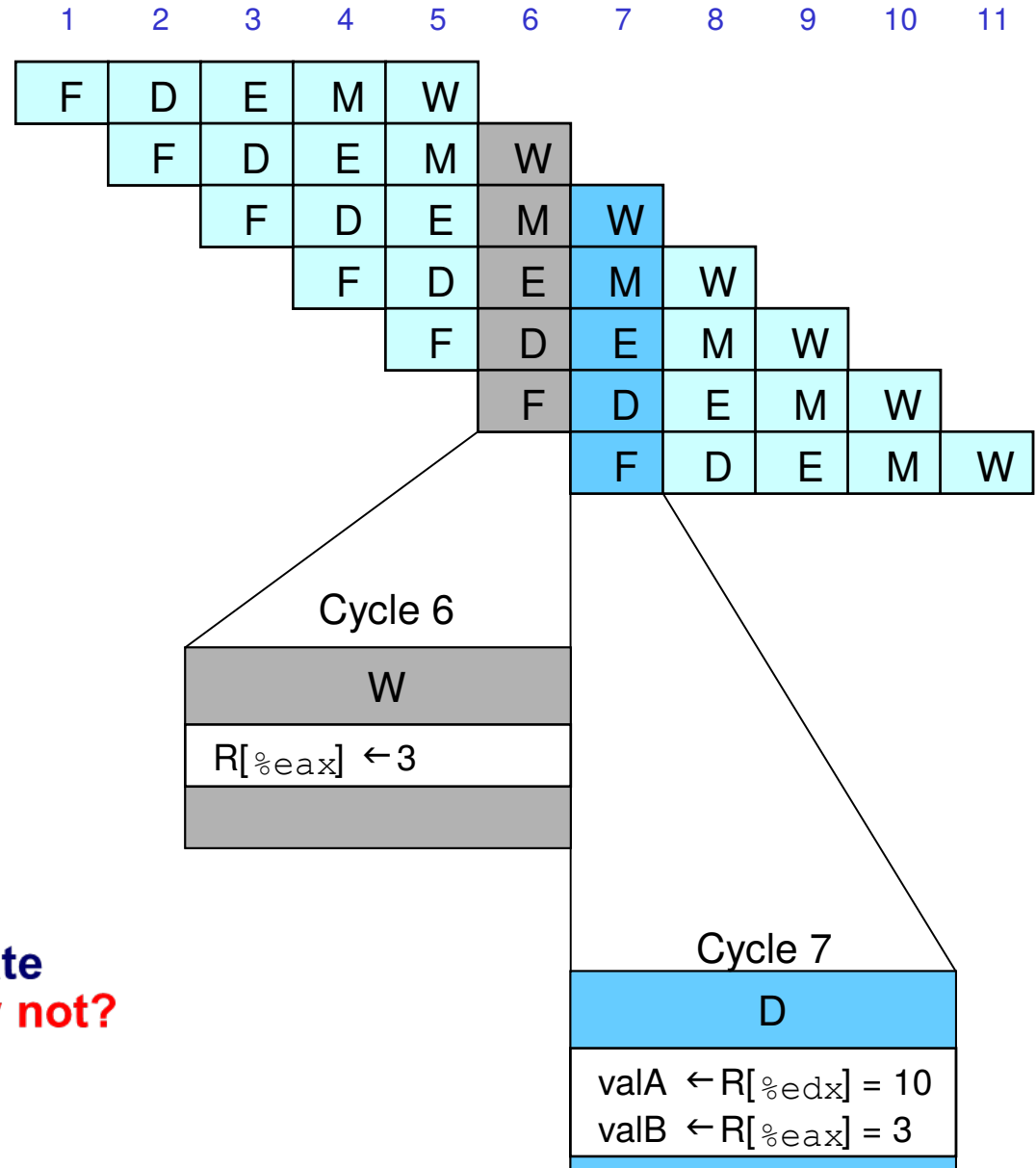
0x00c: nop

0x00d: nop

0x00e: nop

0x00f: addl %edx,%eax

0x011: halt



**Compilers do NOT generate
nop for that purpose. Why not?**

Data Dependencies: 2 Nop's

demo-h2.y_s

0x000: irmovl \$10,%edx

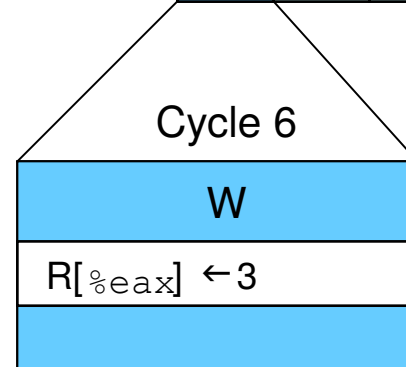
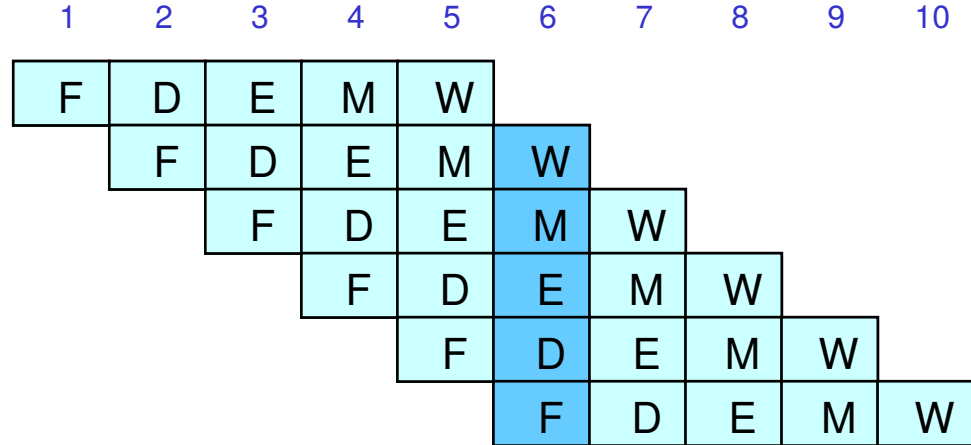
0x006: irmovl \$3,%eax

0x00c: nop

0x00d: nop

0x00e: addl %edx,%eax

0x010: halt



Data Dependencies: 1 Nop

demo-h1.y_s

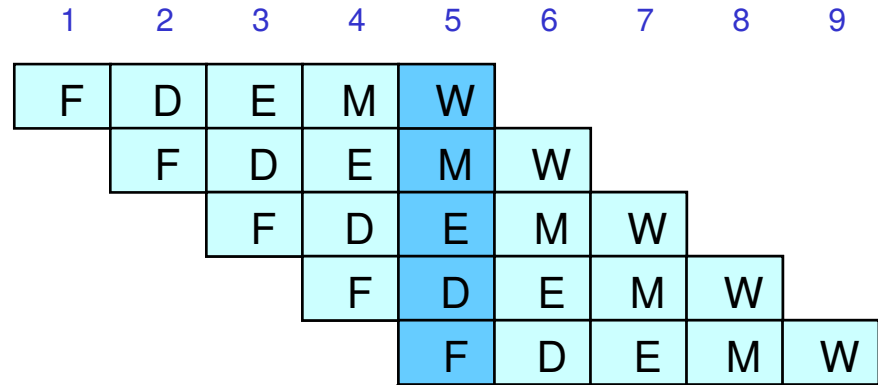
0x000: irmovl \$10,%edx

0x006: irmovl \$3,%eax

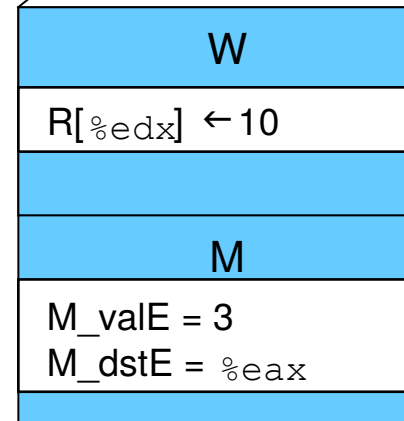
0x00c: nop

0x00d: addl %edx,%eax

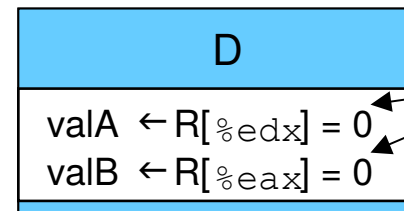
0x00f: halt



Cycle 5



⋮



Error

Data Dependencies: No Nop

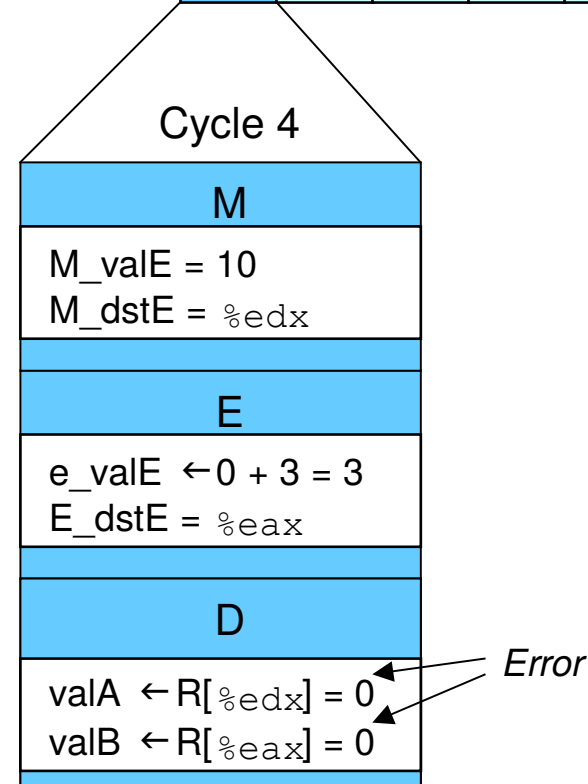
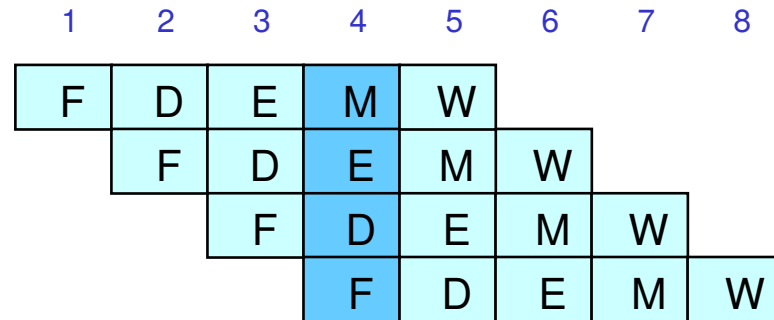
demo-h0.ys

0x000: irmovl \$10,%edx

0x006: irmovl \$3,%eax

0x00c: addl %edx,%eax

0x00e: halt



Stalling for Data Dependencies

```
# demo-h2.ys
```

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

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

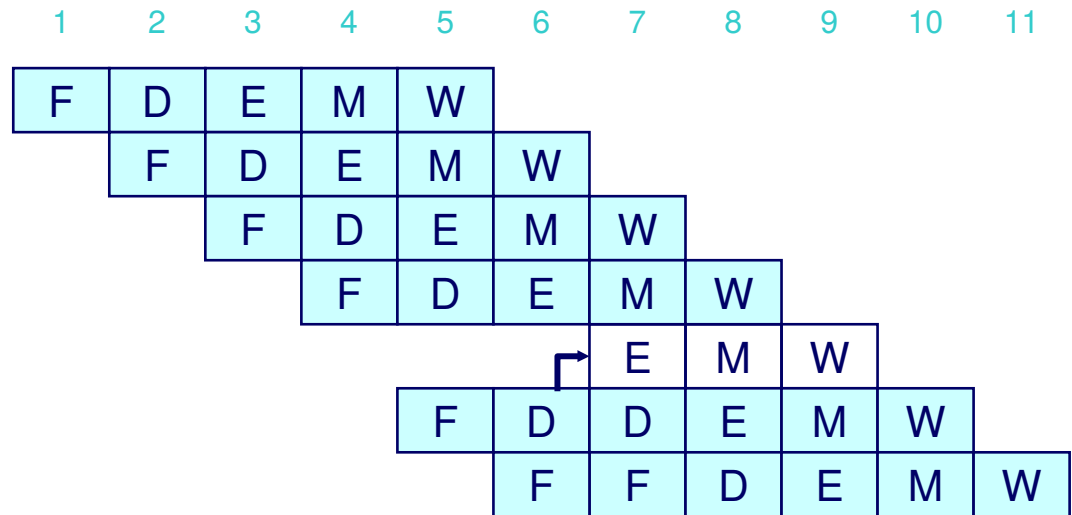
```
0x00c: nop
```

```
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

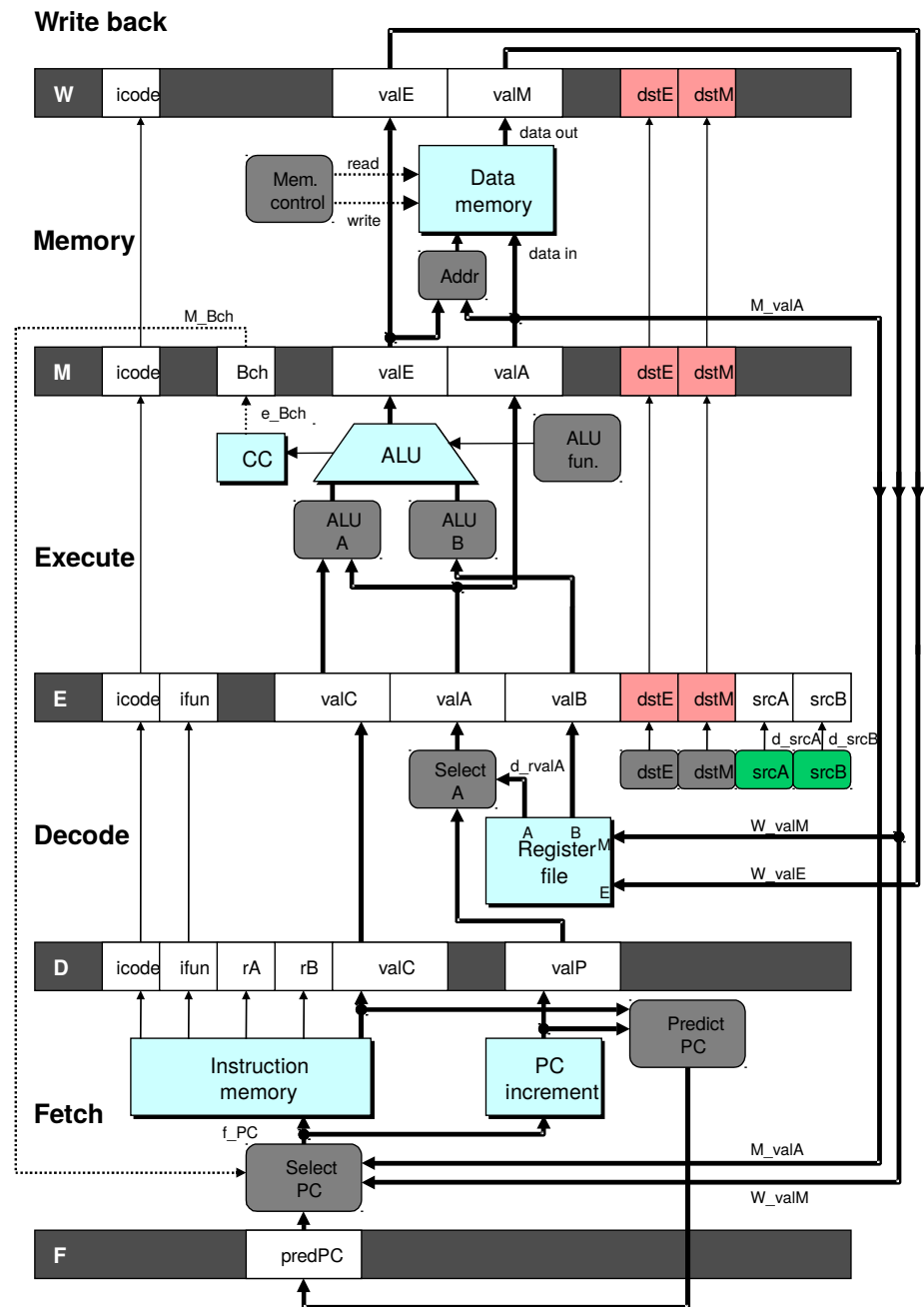
- srcA and srcB of current instruction in decode stage

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.y
```

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

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

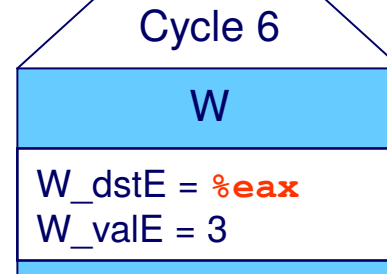
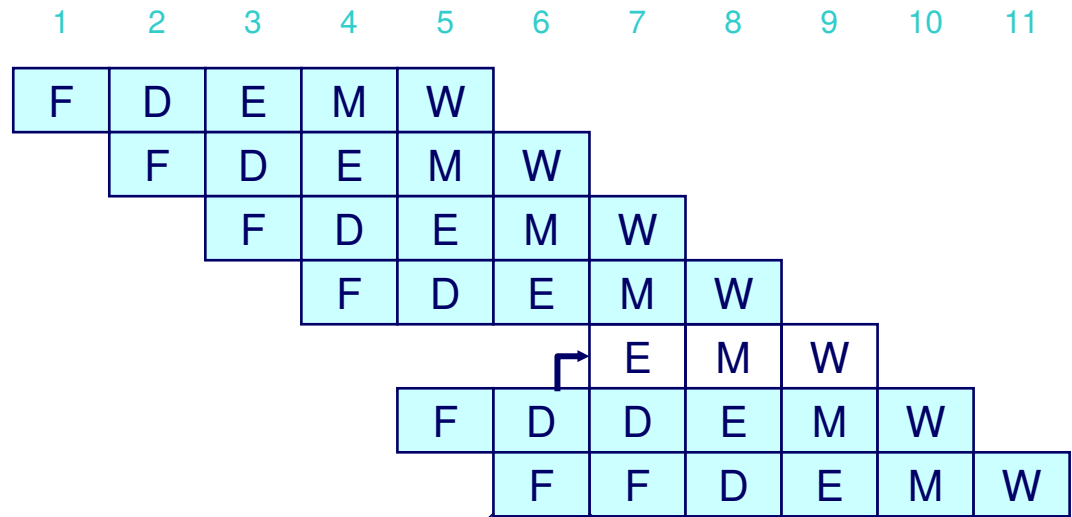
```
0x00c: nop
```

```
0x00d: nop
```

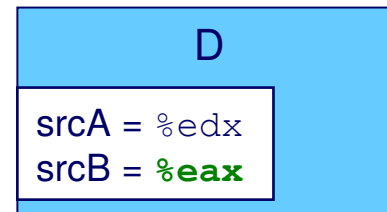
bubble

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

```
0x010: halt
```



⋮



Stalling X3

```
# demo-h0.ys
```

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

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

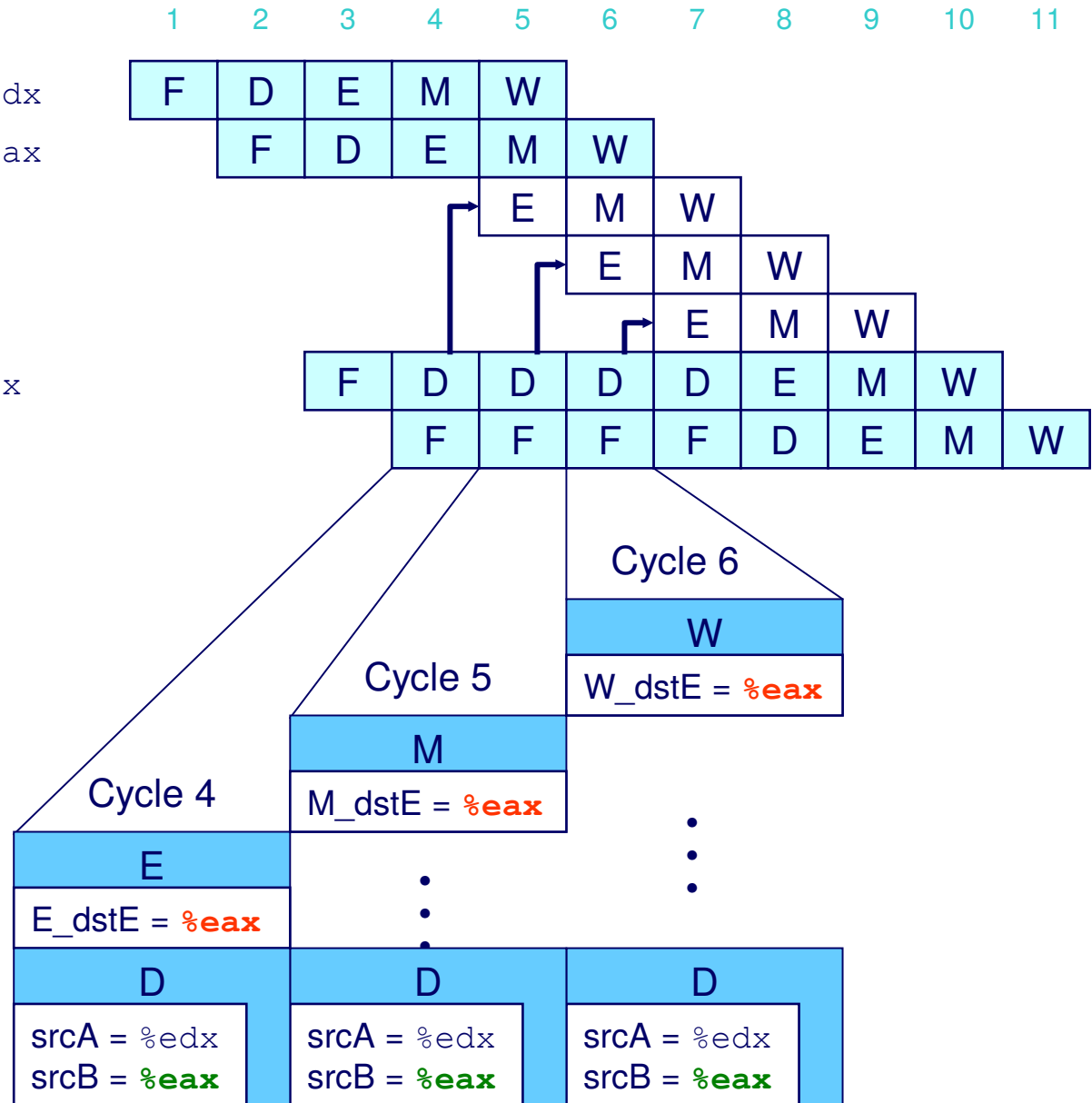
```
bubble
```

```
bubble
```

```
bubble
```

```
0x00c: addl %edx,%eax
```

```
0x00e: halt
```



What Happens When Stalling?

```
# demo-h0.ys
```

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

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

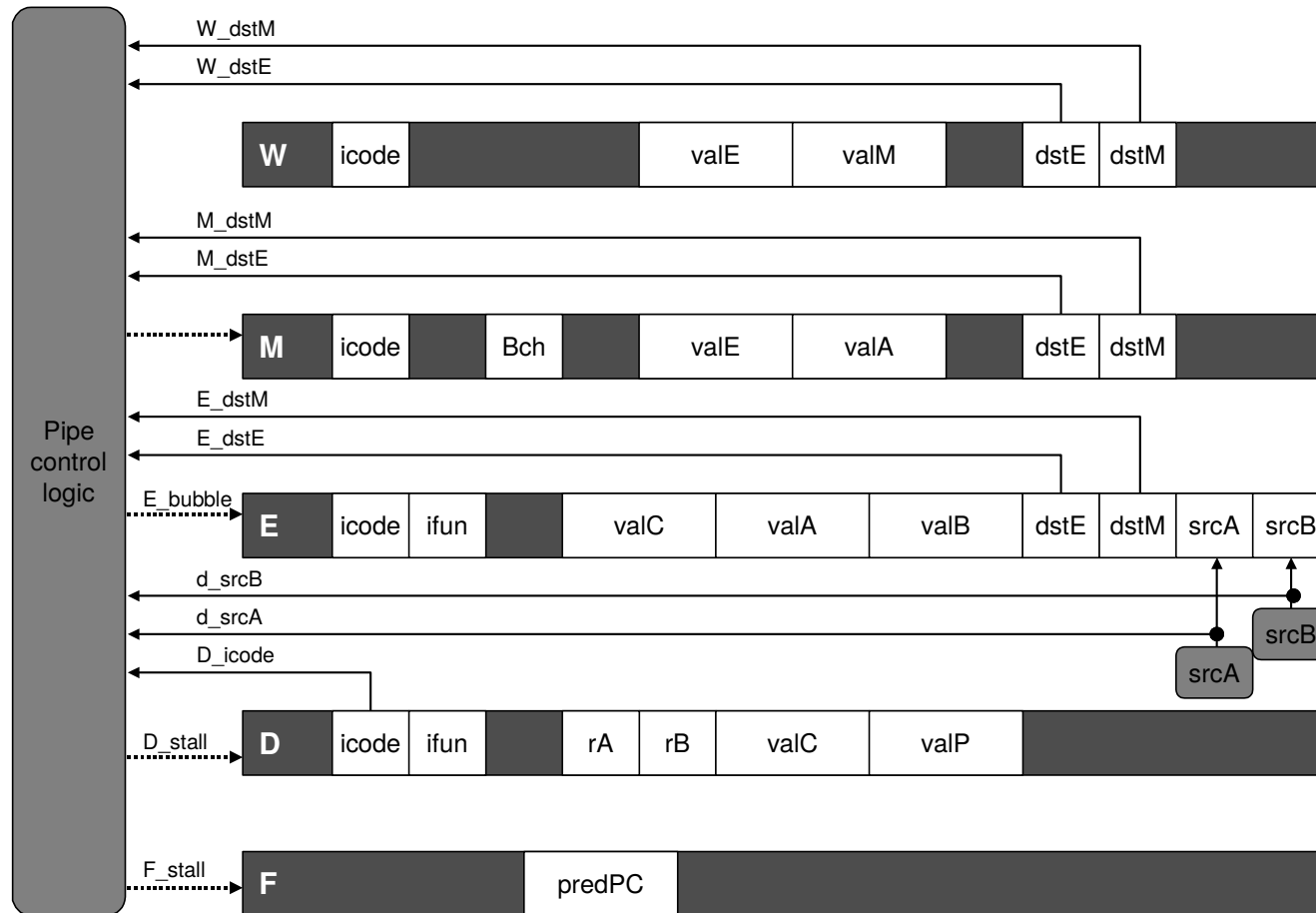
```
0x00c: addl %edx,%eax
```

```
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

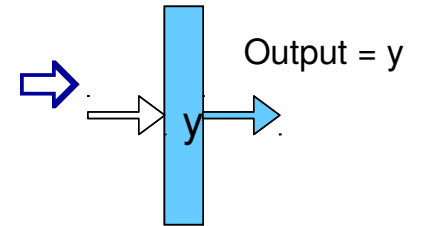
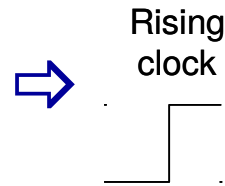
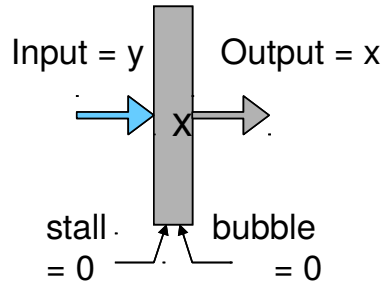


Pipeline Control

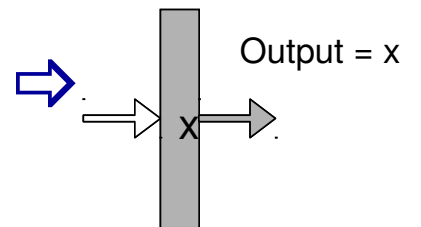
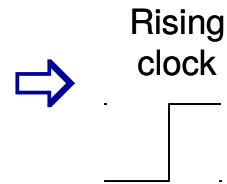
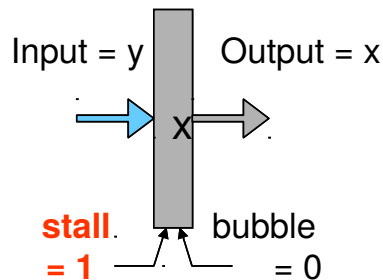
- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

Pipeline Register Modes

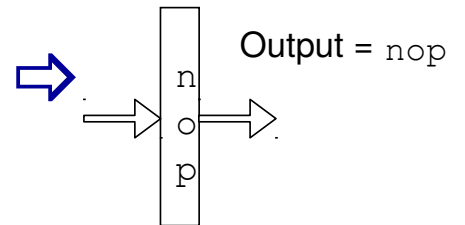
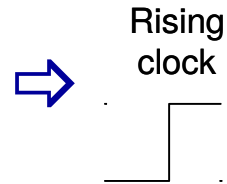
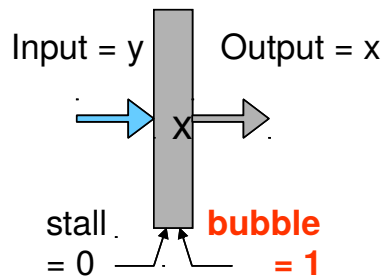
Normal



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

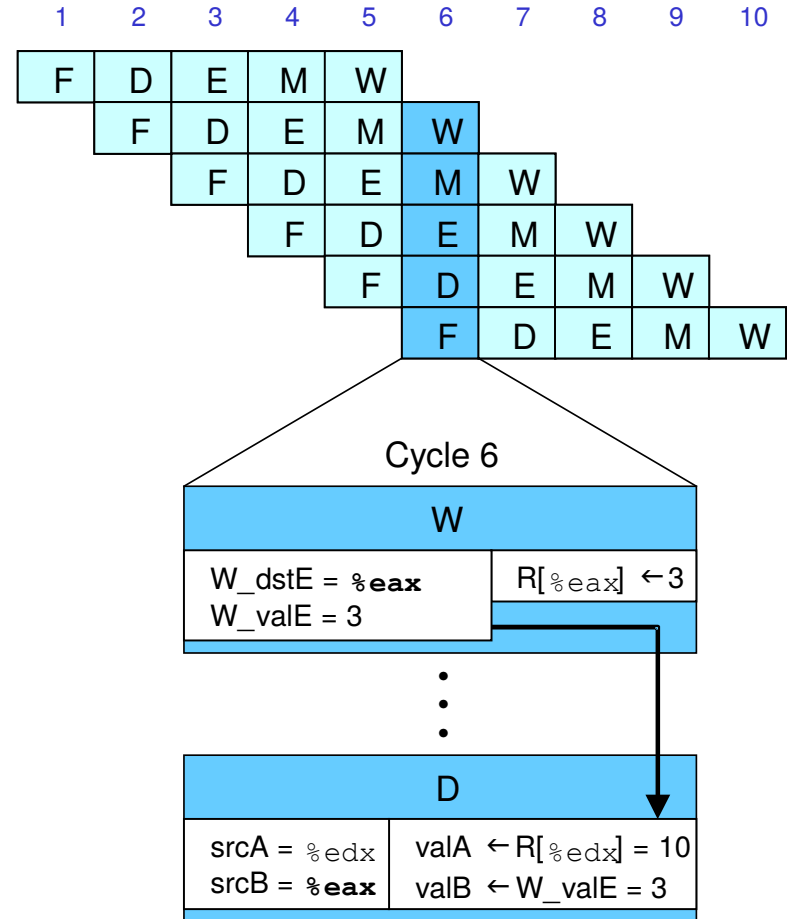
Trick

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

Data Forwarding Example

```
# demo-h2.y
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



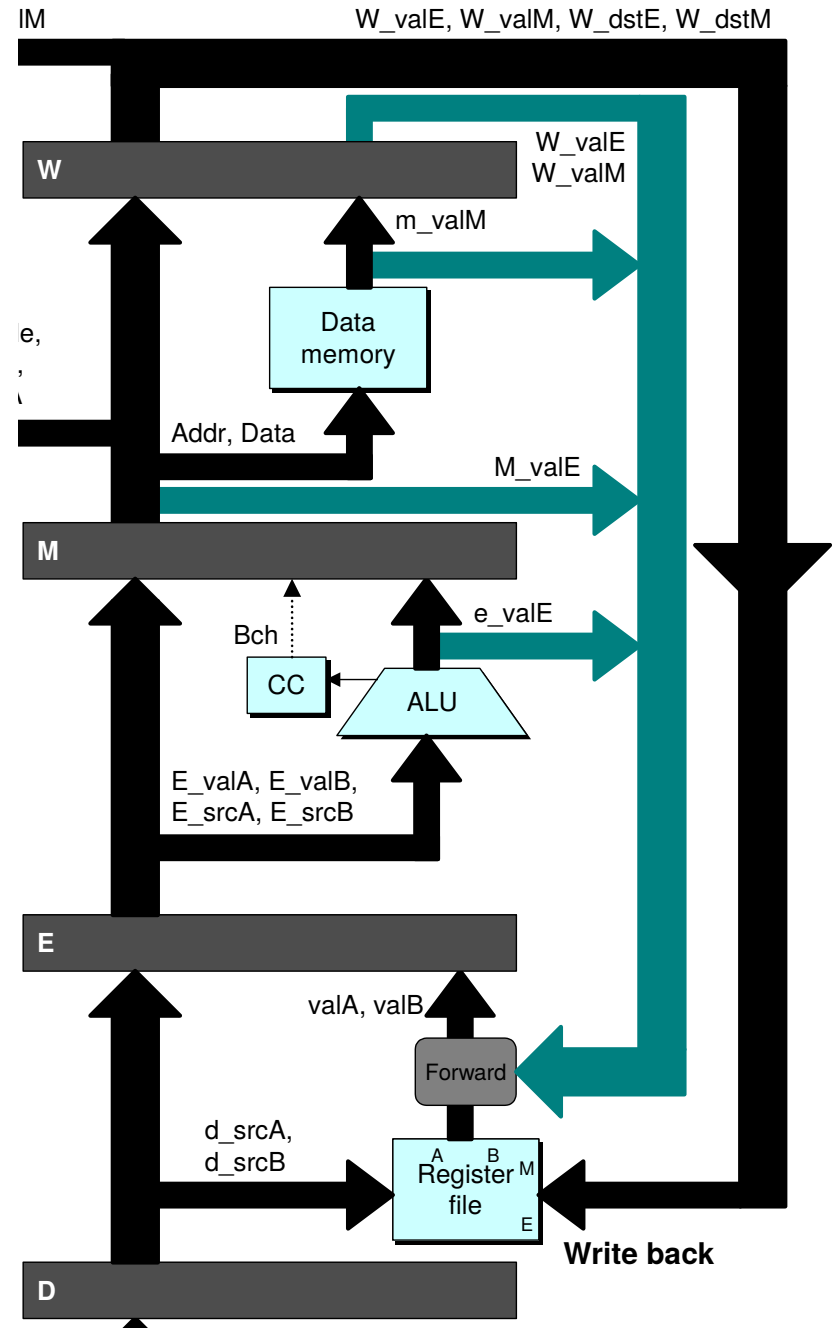
Bypass Paths

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

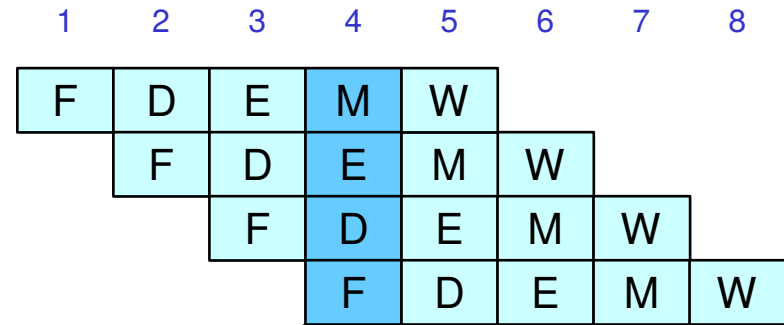
```
# demo-h0.ys
```

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

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

```
0x00c: addl %edx,%eax
```

```
0x00e: halt
```

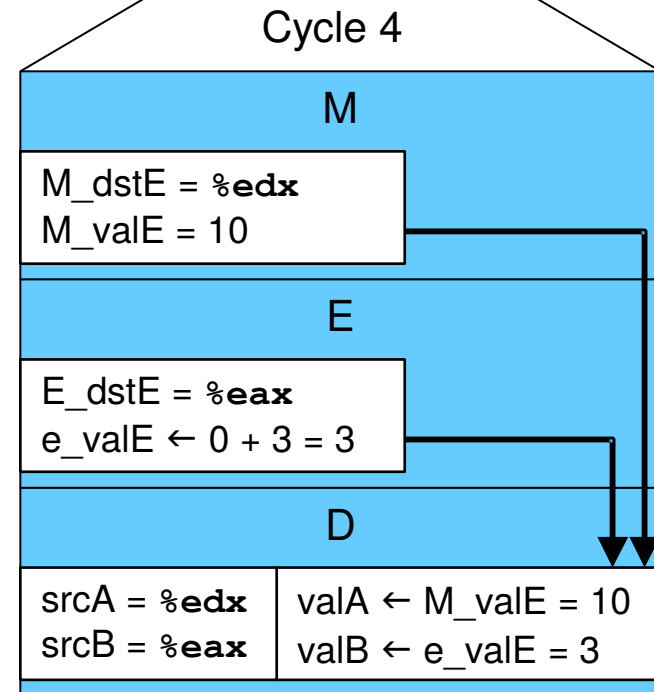


Register %edx

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

Register %eax

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



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

Data Hazards

- Most handled by forwarding
- No performance penalty