

Y86 Pipelined Implementation

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



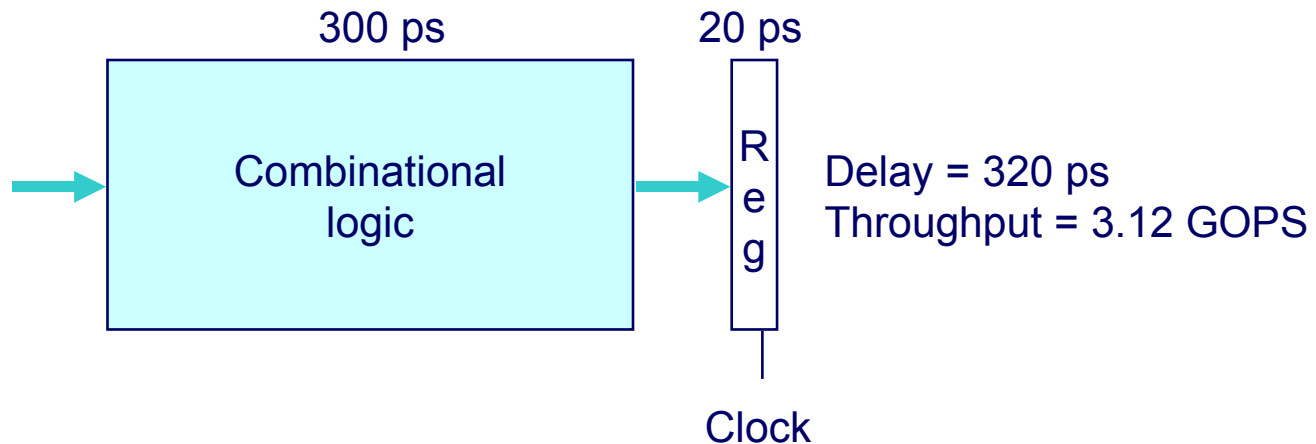
Pipelined



Idea

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

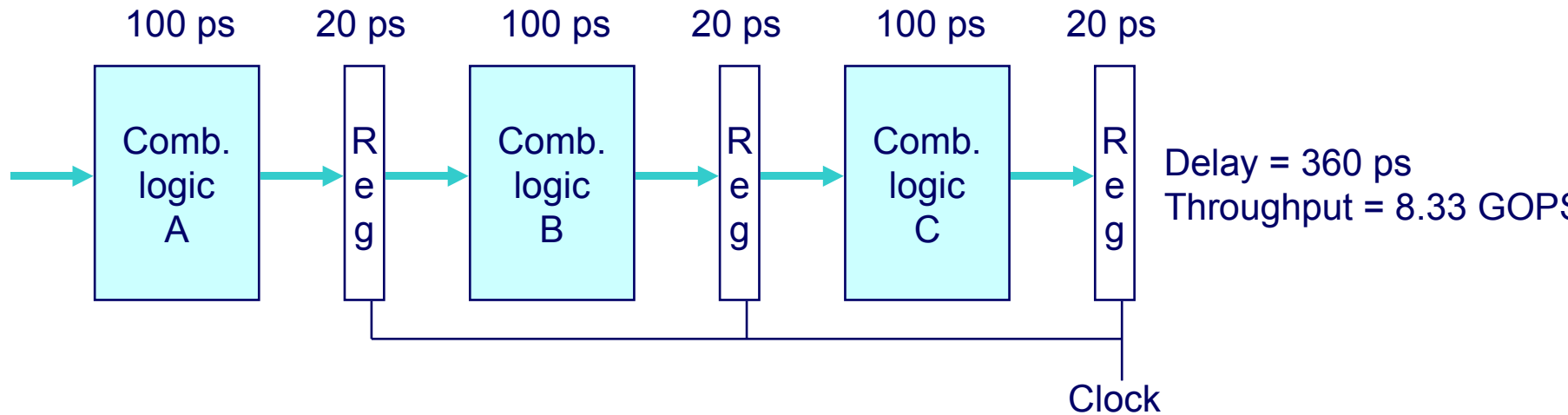
Computational Example



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

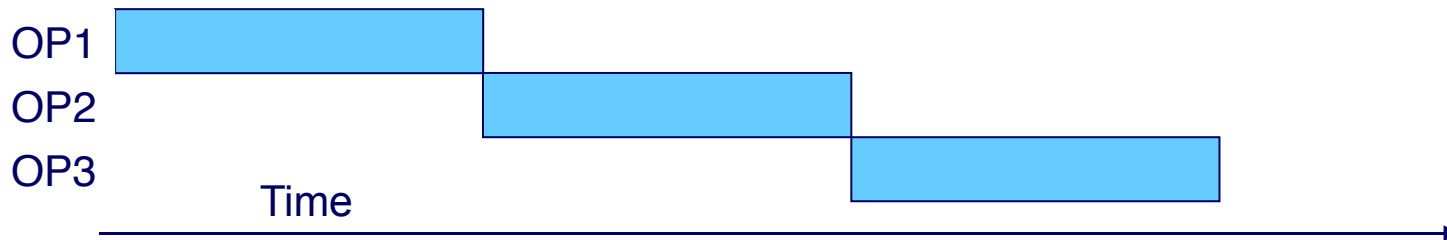


System

- **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**

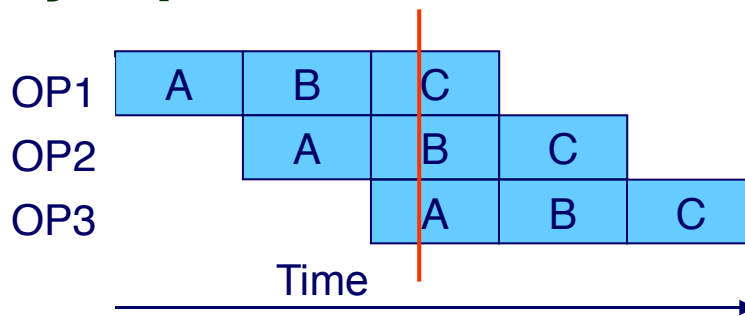
Pipeline Diagrams

Unpipelined



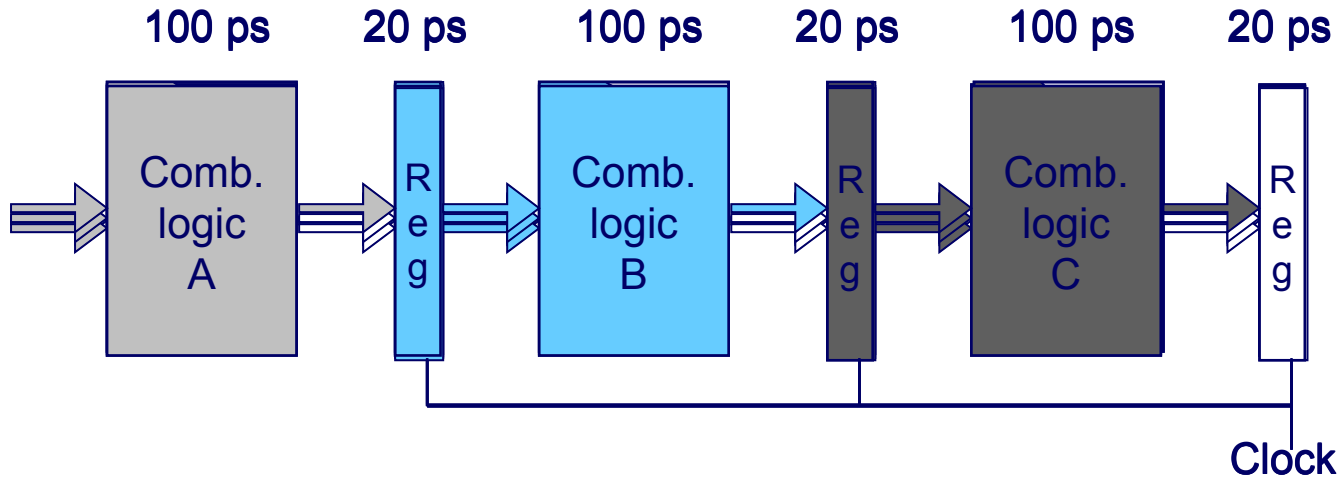
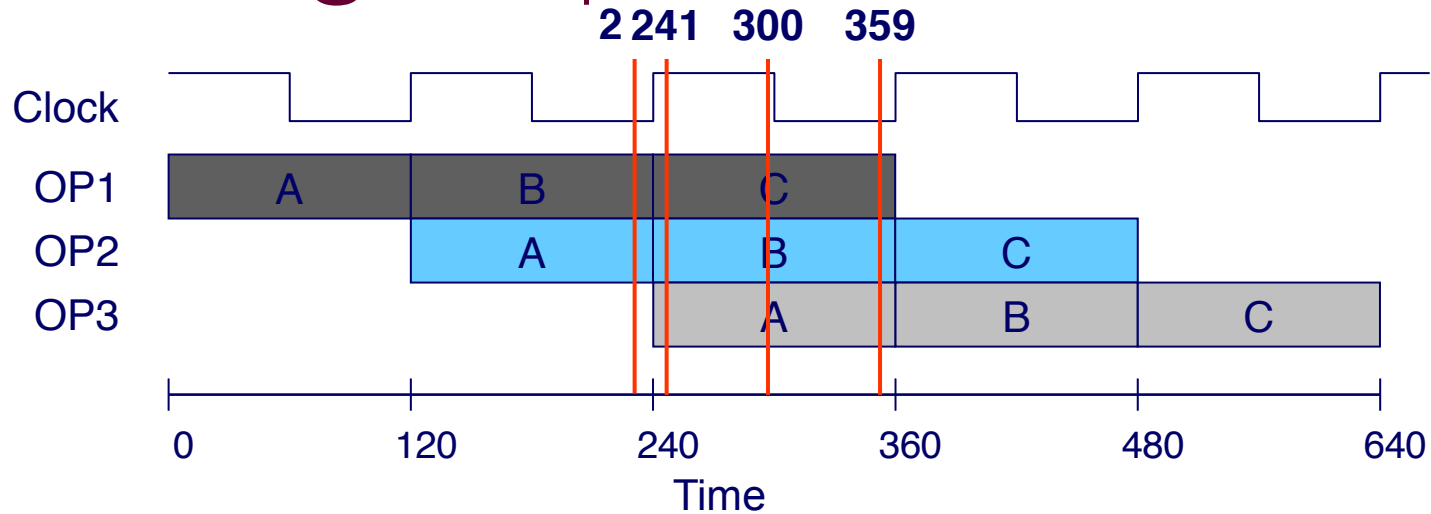
- Cannot start new operation until previous one completes

3-Way Pipelined

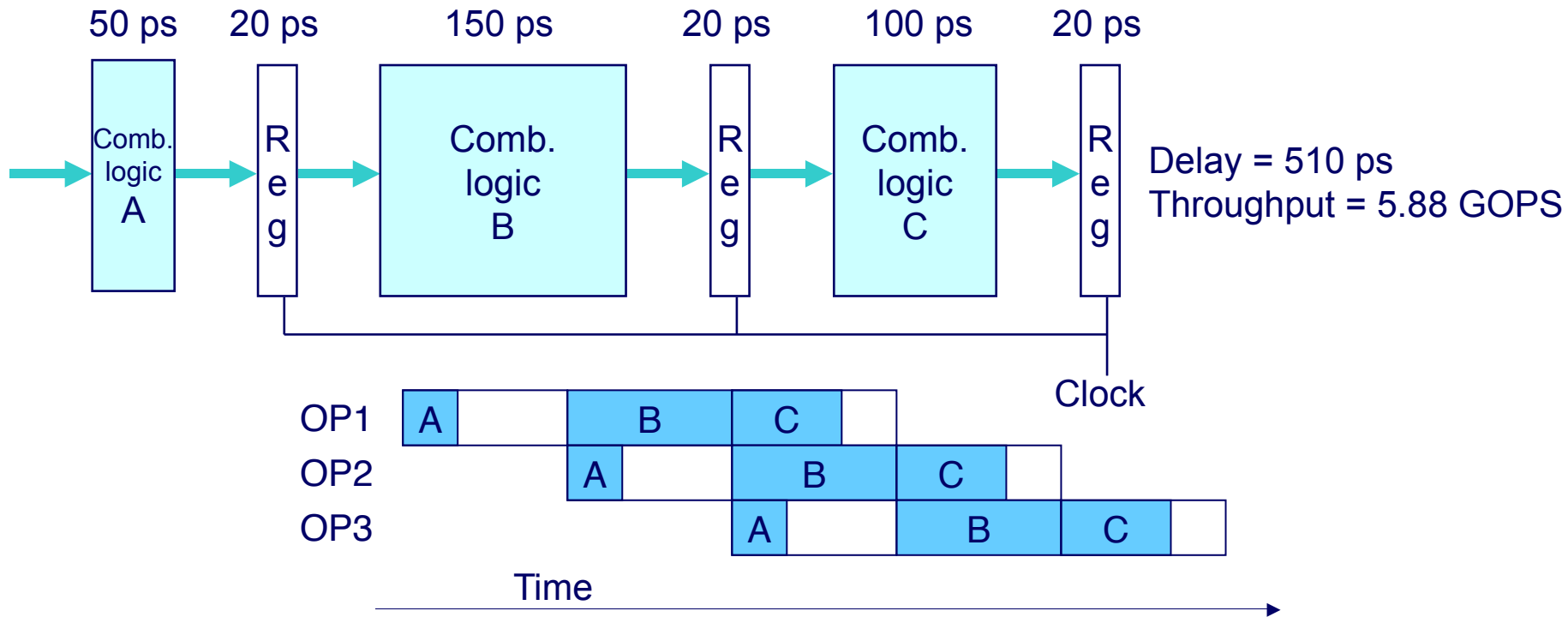


- Up to 3 operations in process simultaneously

Operating a Pipeline

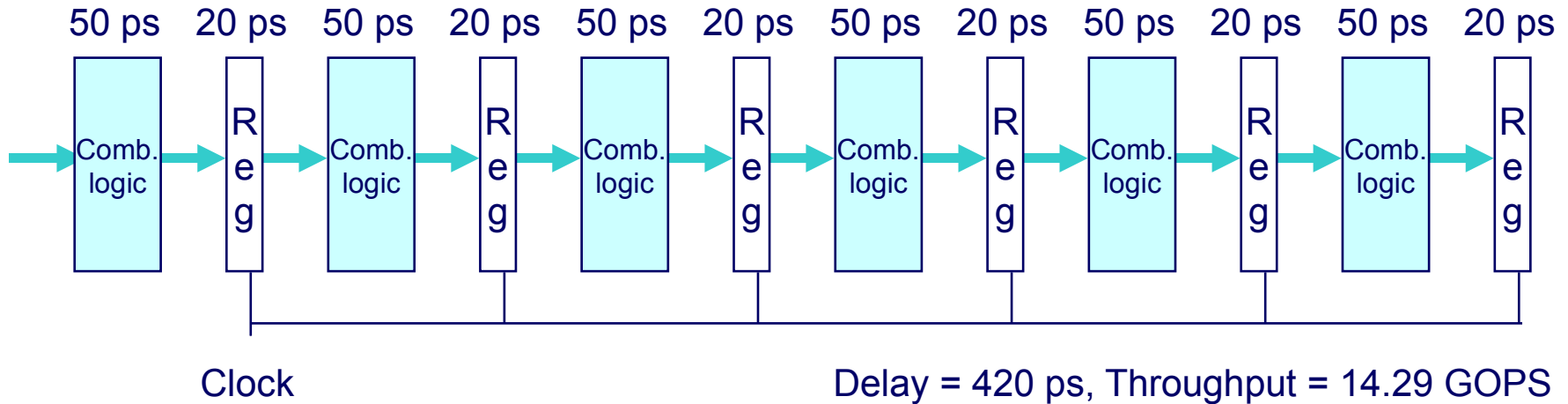


Limitations: Nonuniform Delays



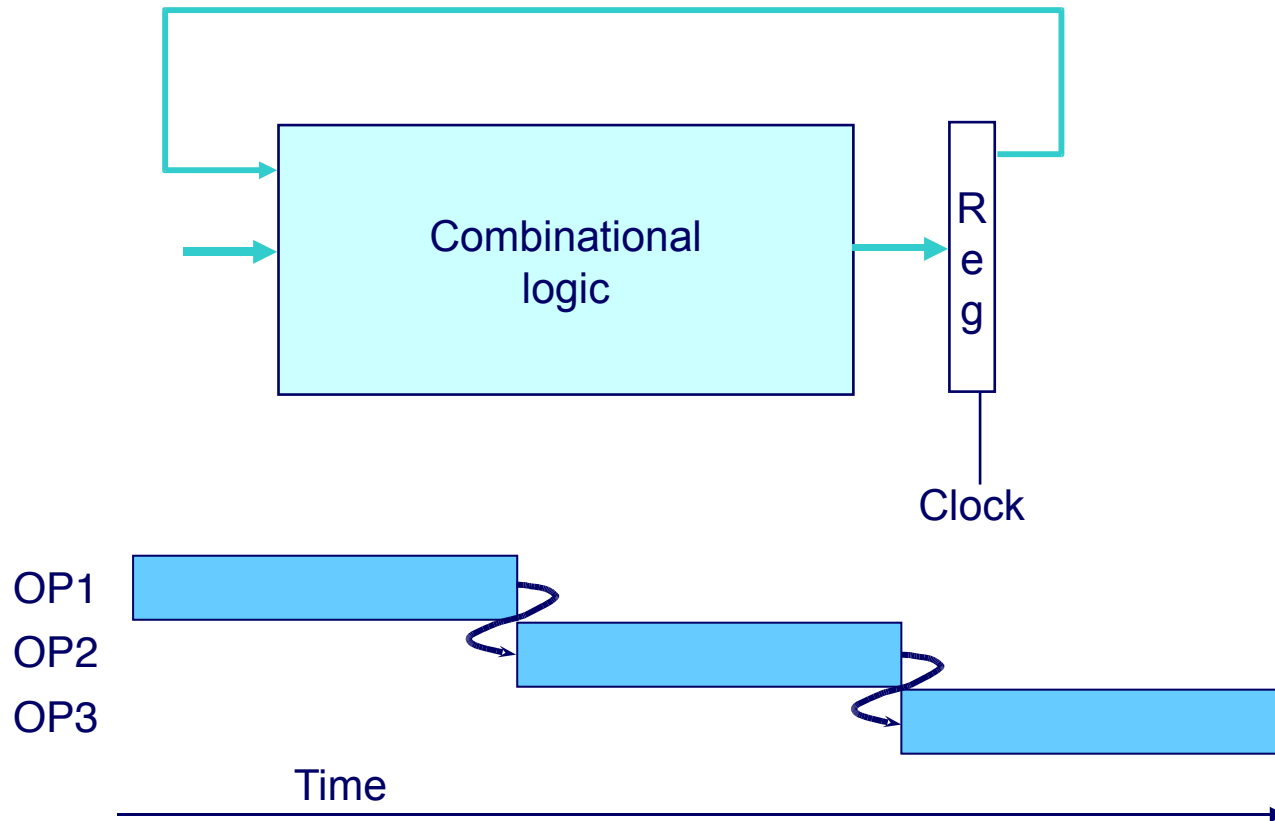
- **Throughput limited by slowest stage**
- **Other stages sit idle for much of the time**
- **Challenging to partition system into balanced stages**

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**

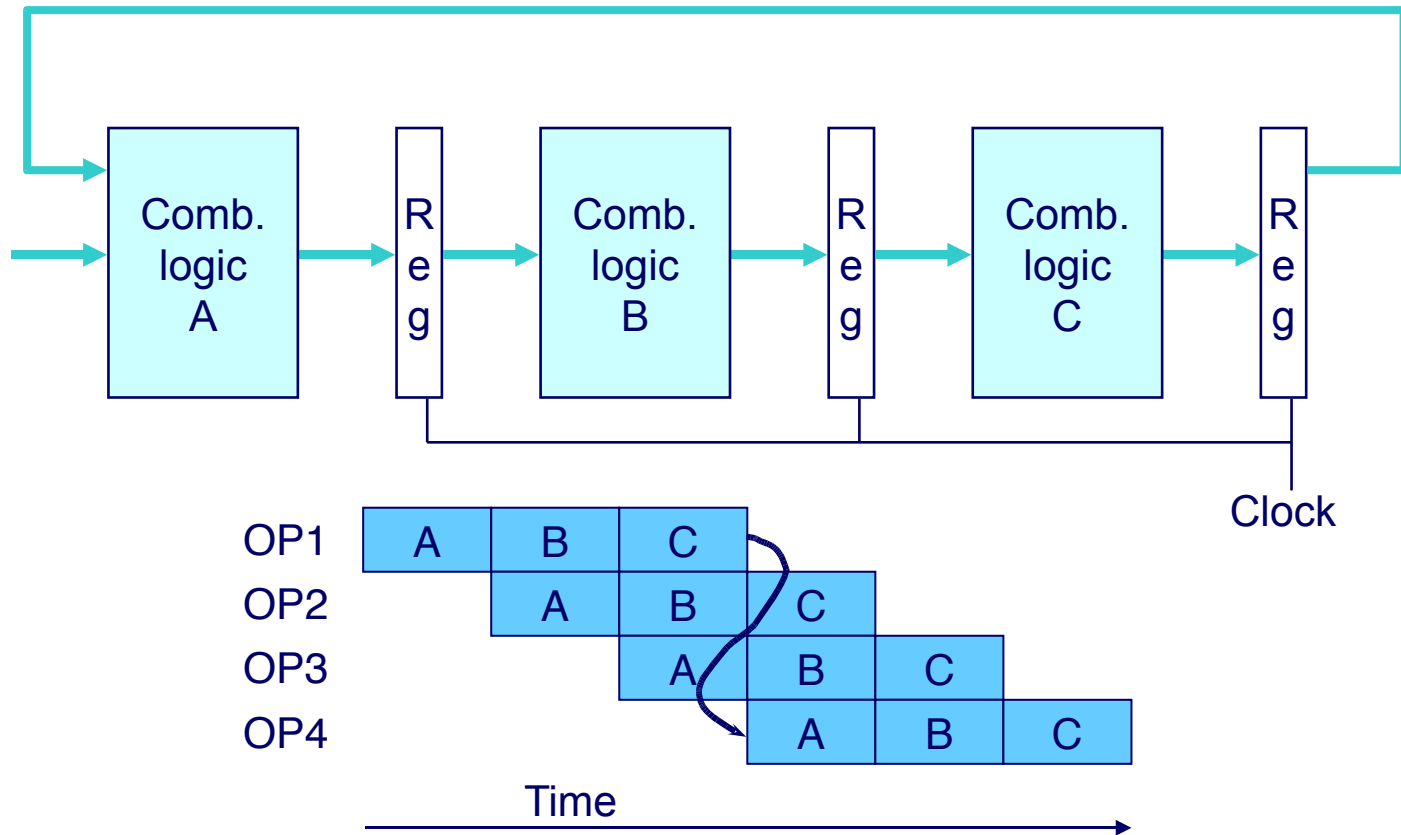
Data Dependencies



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

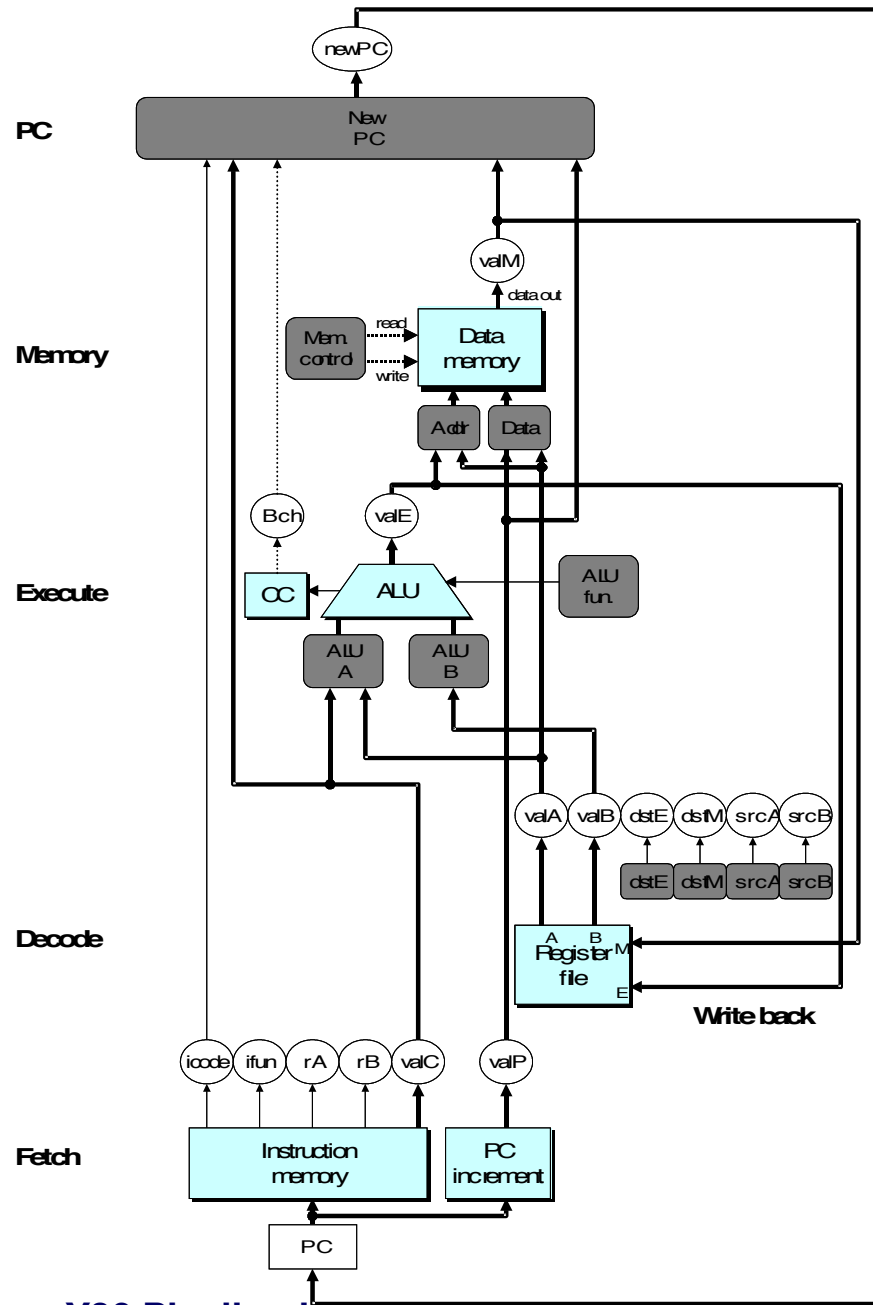
```
1   irmovl $50, %eax
2   addl %eax, %ebx
3   mrmovl 100(%ebx), %edx
```

The diagram illustrates data dependencies between three instructions. Instruction 1 writes to register %eax. Instruction 2 reads %eax and writes to %ebx. Instruction 3 reads %ebx. Arrows indicate the flow of data: 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

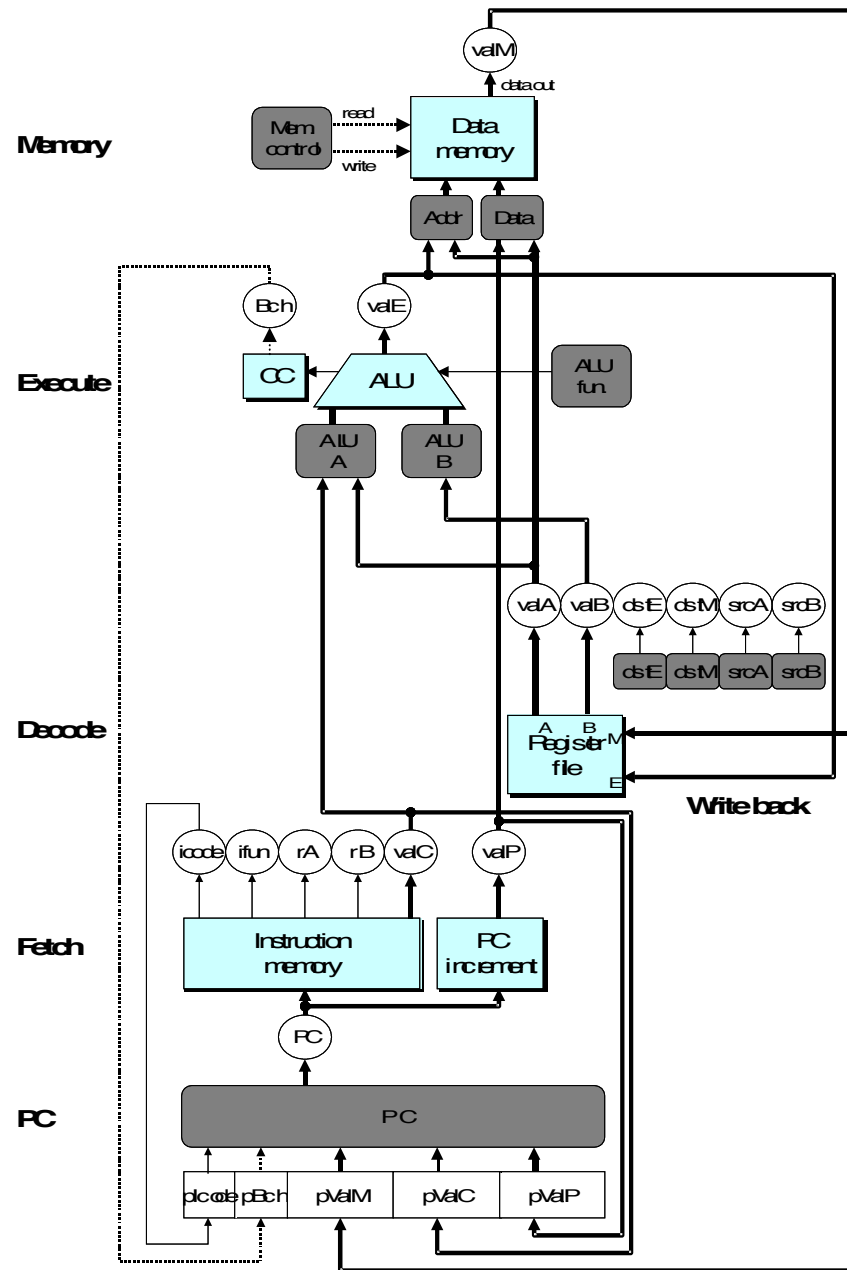
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

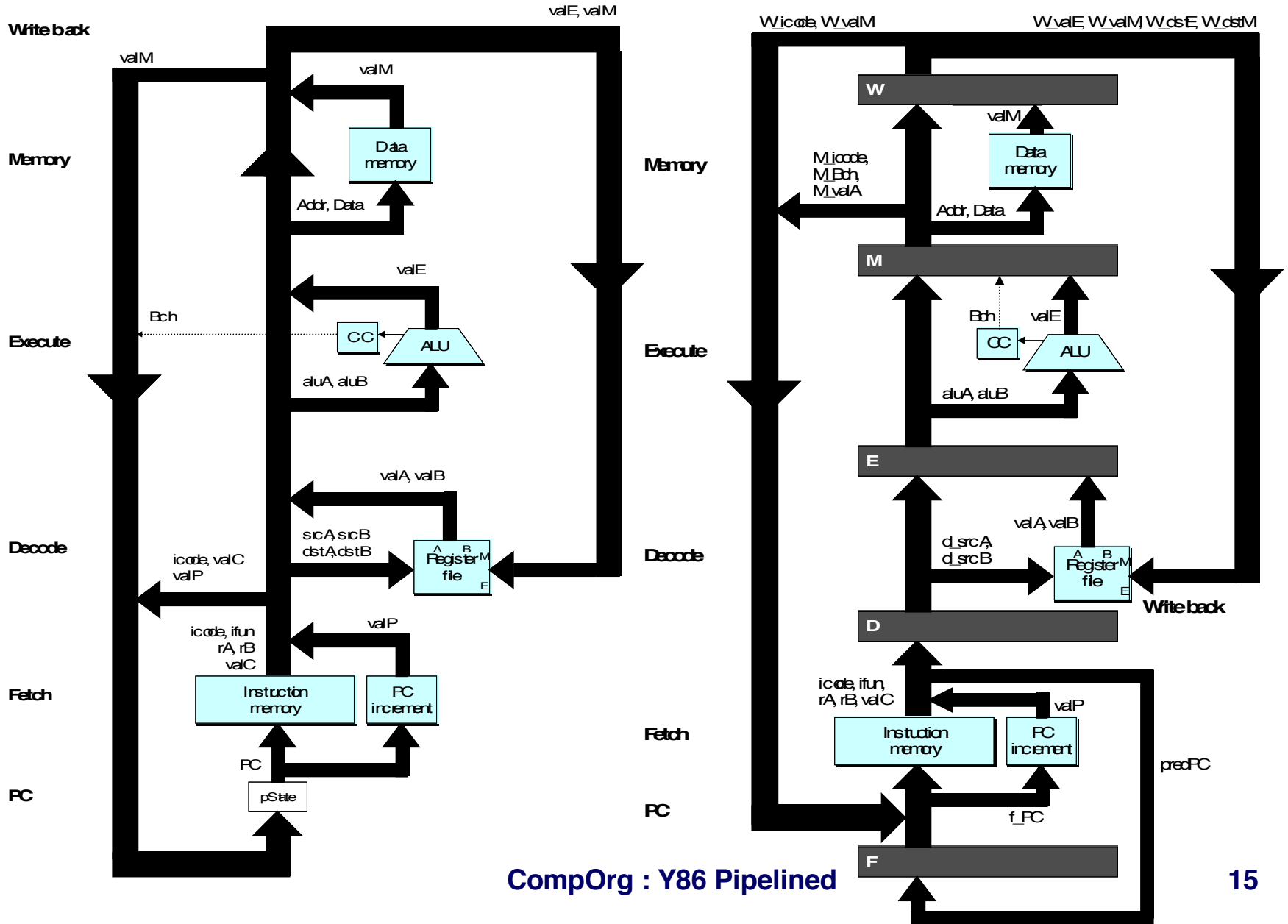
- Task is to select PC for current instruction
- Based on results computed by previous instruction

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

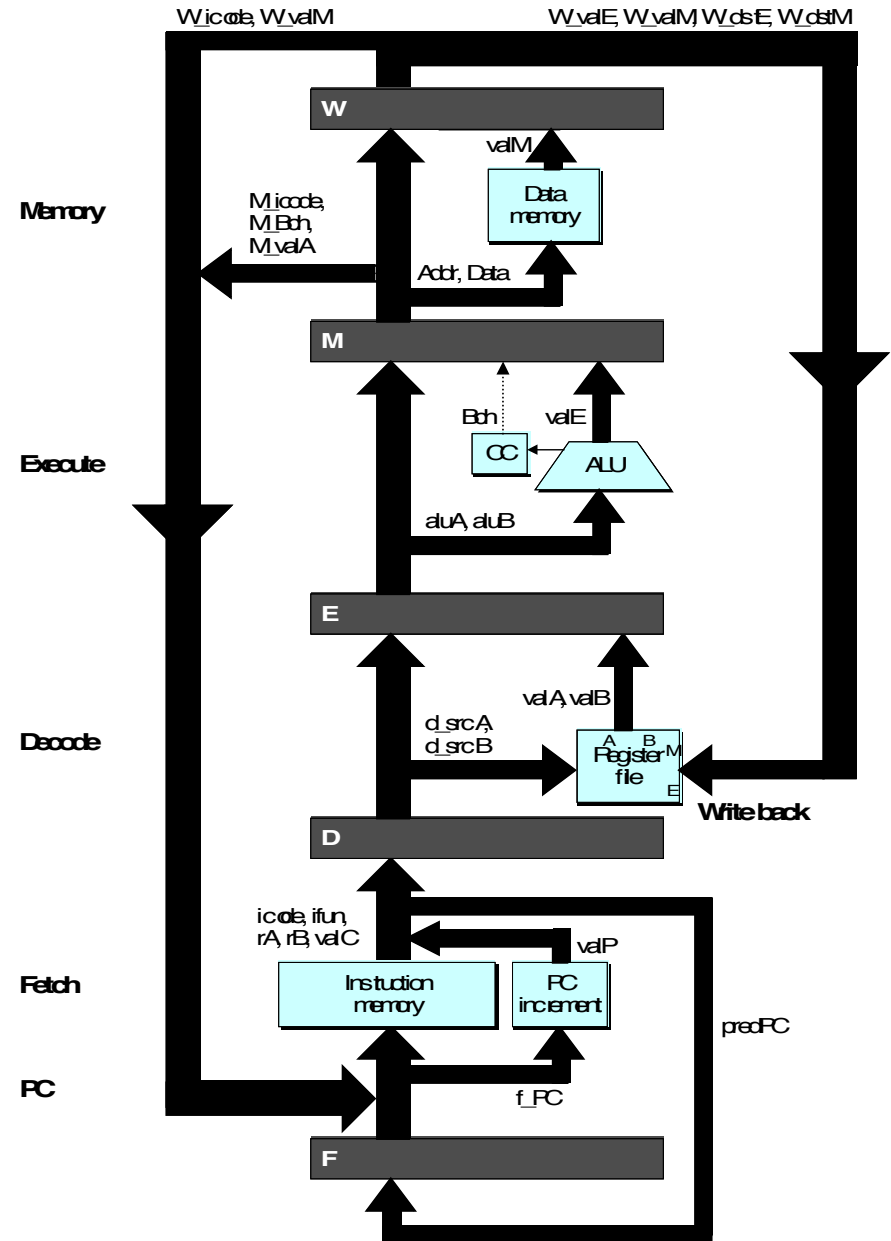
Memory

- Read or write data memory

Write Back

- Update register file

CompOrg : Y86 Pipelined

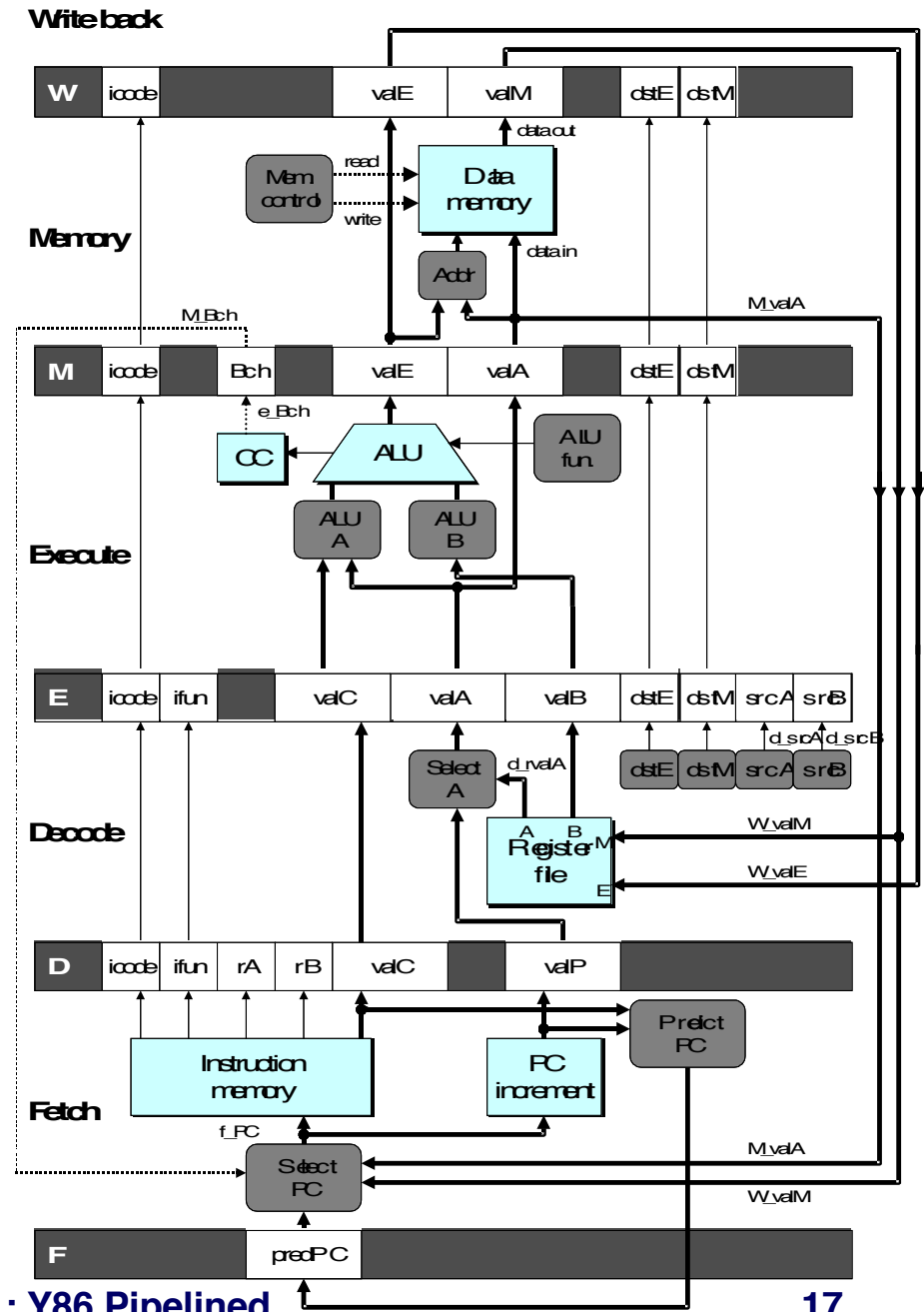


PIPE- Hardware

- 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

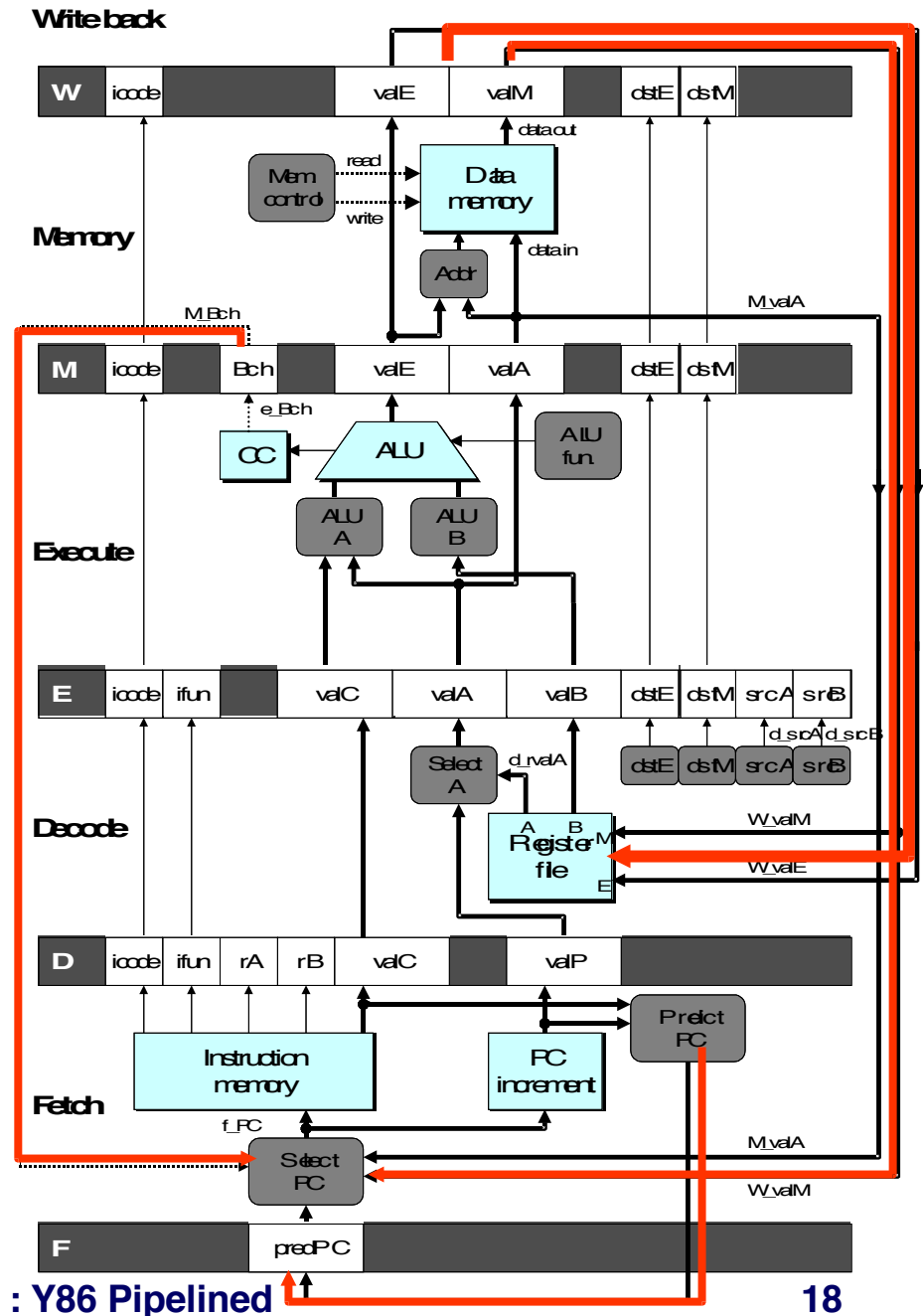
- Jump taken/not-taken
- Fall-through or target address

Return point

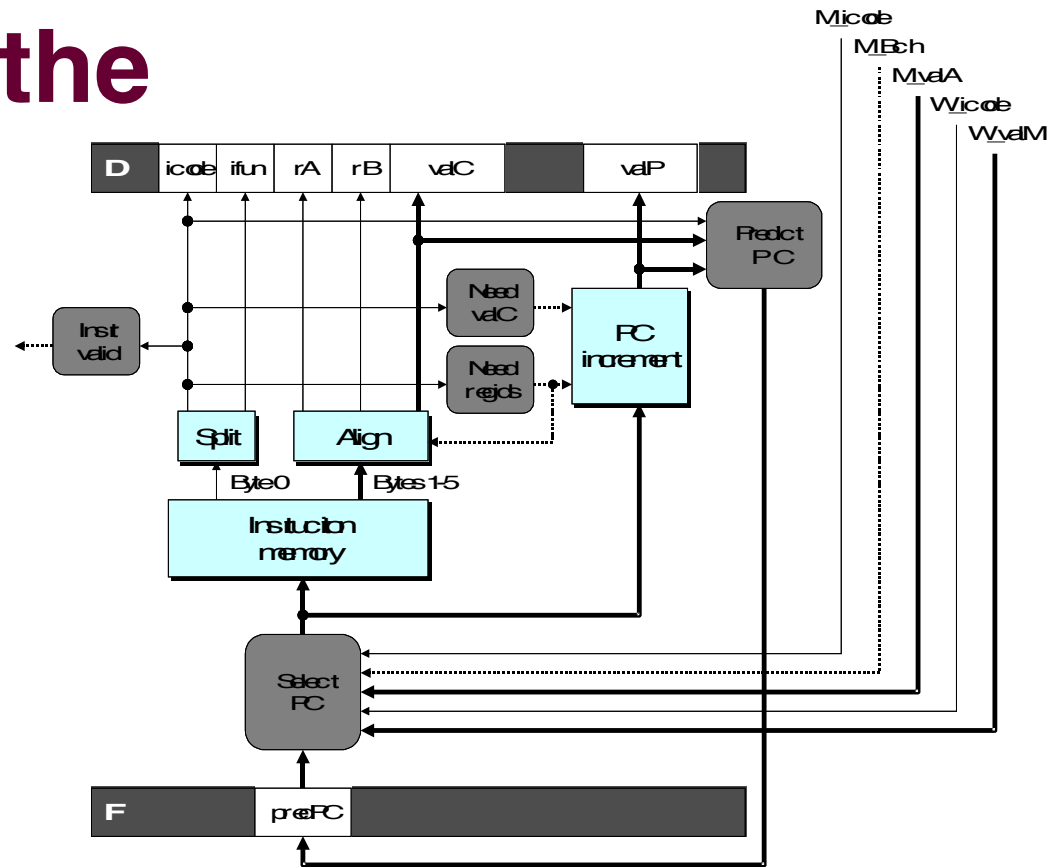
- Read from memory

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

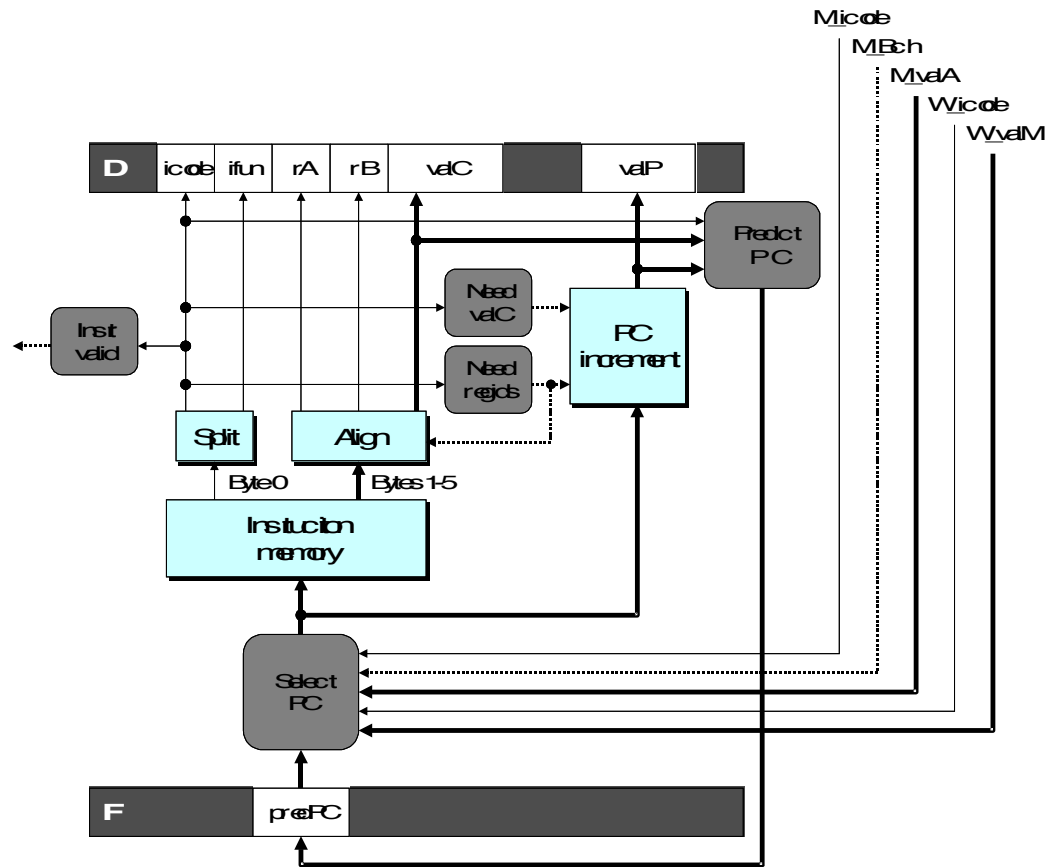
Conditional Jumps

- 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

Recovering from PC Misprediction

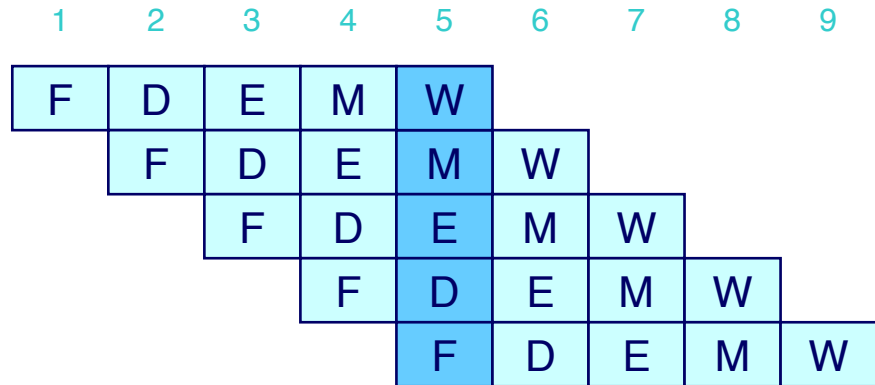


- **Mispredicted Jump**
 - Will see branch flag once instruction reaches memory stage
 - Can get fall-through PC from valA
- **Return Instruction**
 - Will get return PC when ret reaches write-back stage

Pipeline Demonstration

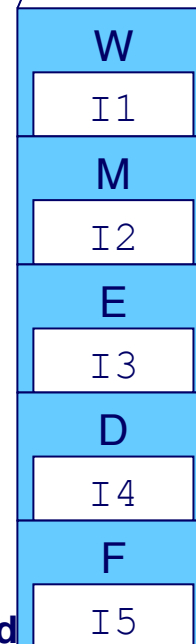
```

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



File: demo-basic.js

Cycle 5



Data Dependencies: 3 Nop's

demo-h3.y8

0x000: irmovl \$10,%edx

0x006: irmovl \$3,%eax

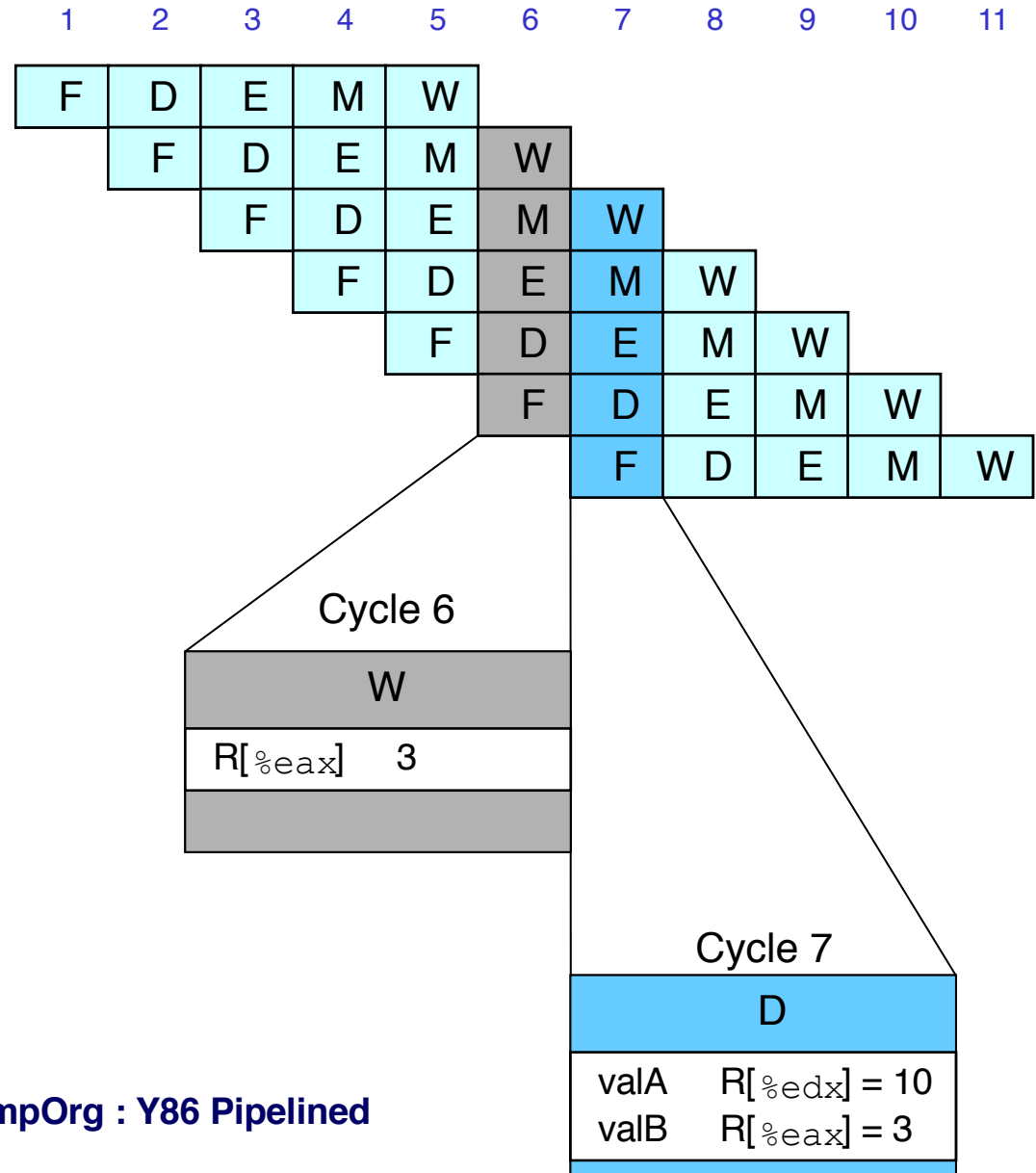
0x00c: nop

0x00d: nop

0x00e: nop

0x00f: addl %edx,%eax

0x011: halt



Data Dependencies: 2 Nop's

demo-h2.y

0x000: irmovl \$10,%edx

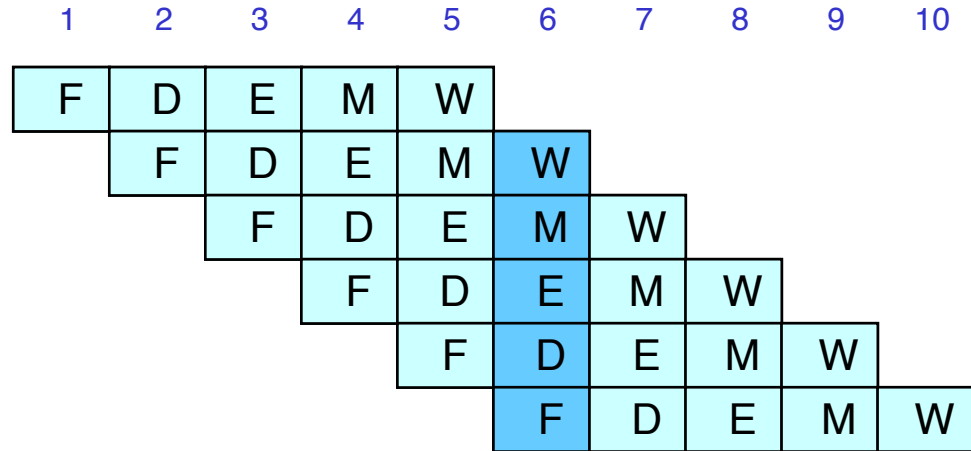
0x006: irmovl \$3,%eax

0x00c: nop

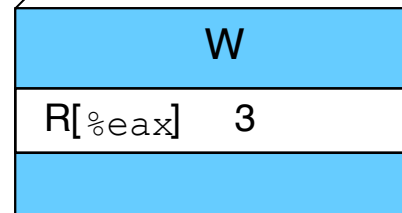
0x00d: nop

0x00e: addl %edx,%eax

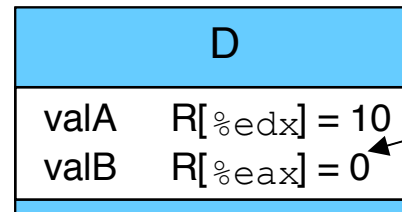
0x010: halt



Cycle 6



⋮



Error

Data Dependencies: 1 Nop

demo-h1.y

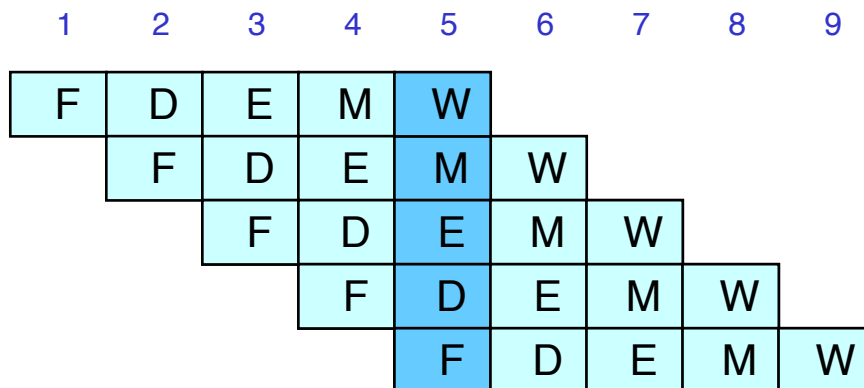
0x000: irmovl \$10,%edx

0x006: irmovl \$3,%eax

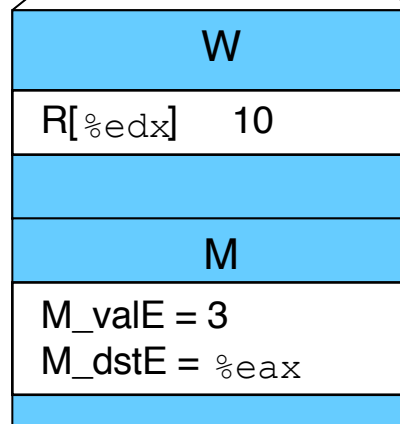
0x00c: nop

0x00d: addl %edx,%eax

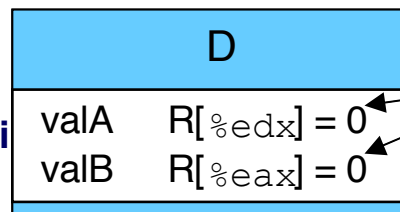
0x00f: halt



Cycle 5



⋮

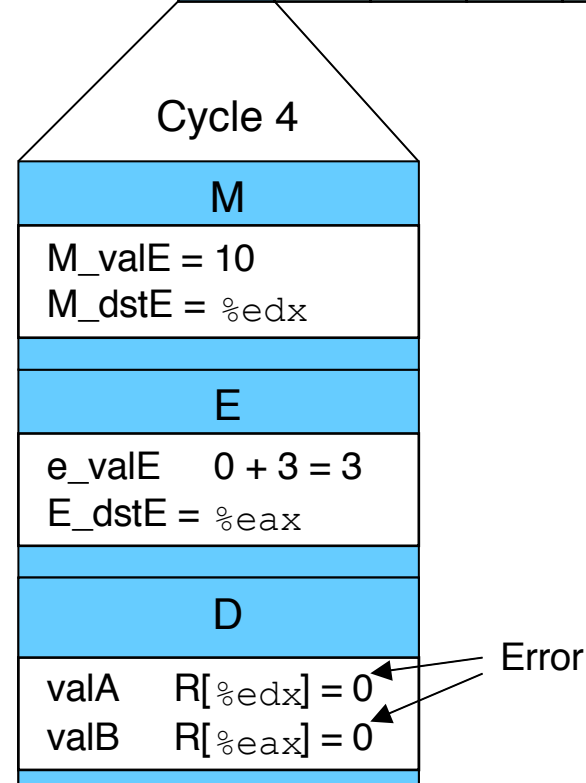
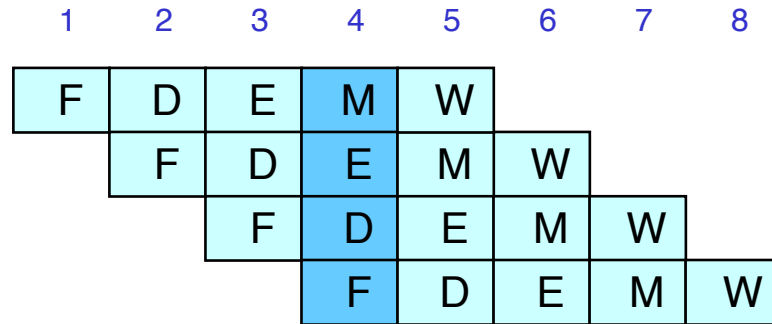


Error

CompOrg : Y86 Pi

Data Dependencies: No Nop

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



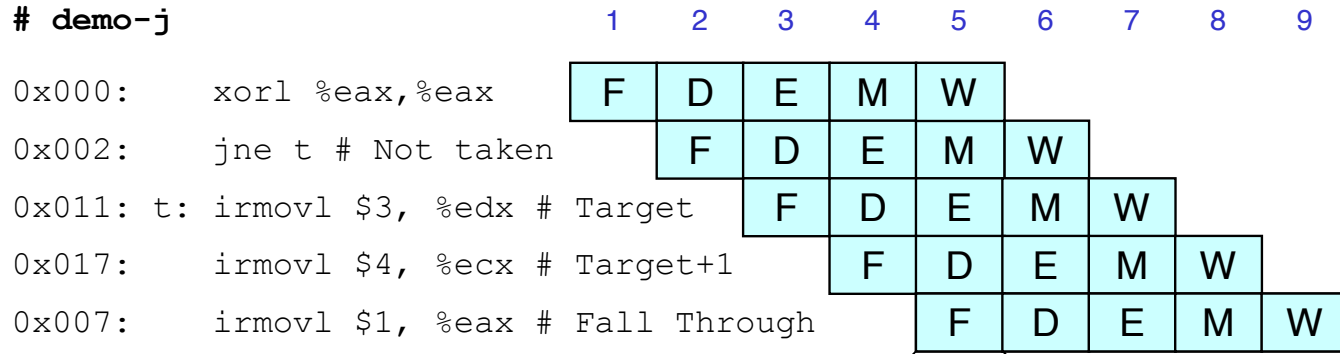
Branch Misprediction Example

demo-j.js

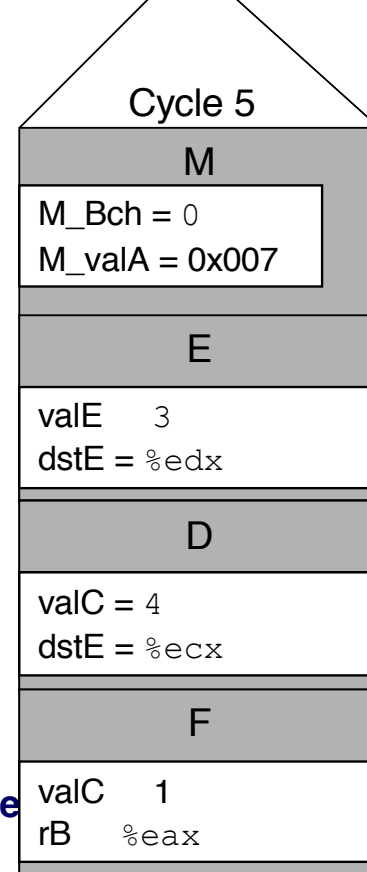
```
0x000:    xorl %eax,%eax
0x002:    jne  t                # Not taken
0x007:    irmovl $1, %eax     # Fall through
0x00d:    nop
0x00e:    nop
0x00f:    nop
0x010:    halt
0x011:  t:  irmovl $3, %edx   # Target (Should not execute)
0x017:    irmovl $4, %ecx     # Should not execute
0x01d:    irmovl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

- Should only execute first 8 instructions

Branch Misprediction Trace



Incorrectly execute two instructions at branch target



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
```

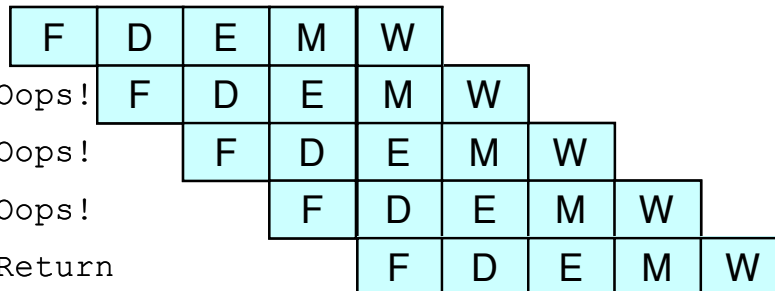
- **Require lots of nops to avoid data hazards**

Incorrect Return Example

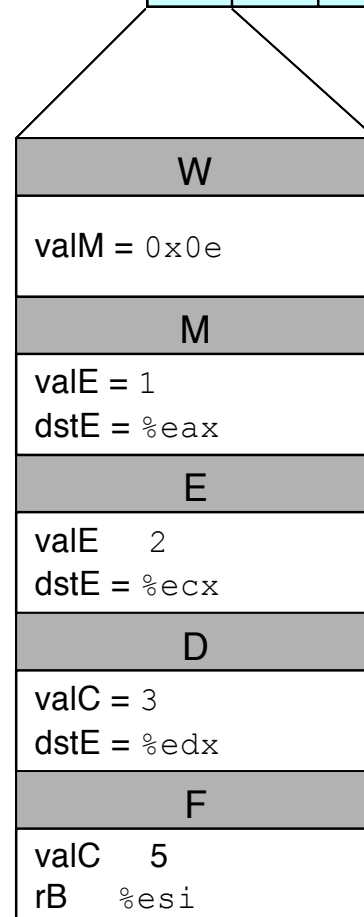
demo-ret

```

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`



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