CS:APP Chapter 4
Computer Architecture

Pipelined Implementation

Randal E. Bryant

adapted by Jason Fritts

`http://csapp.cs.cmu.edu`

Overview

General Principles of Pipelining

- Goal
- Difficulties

Creating a Pipelined Y86 Processor

- Rearranging SEQ to create pipelined datapath, PIPE
- Inserting pipeline registers
- Problems with data and control hazards

Fundamentals of Pipelining

Real-World Pipelines: Car Washes

Sequential

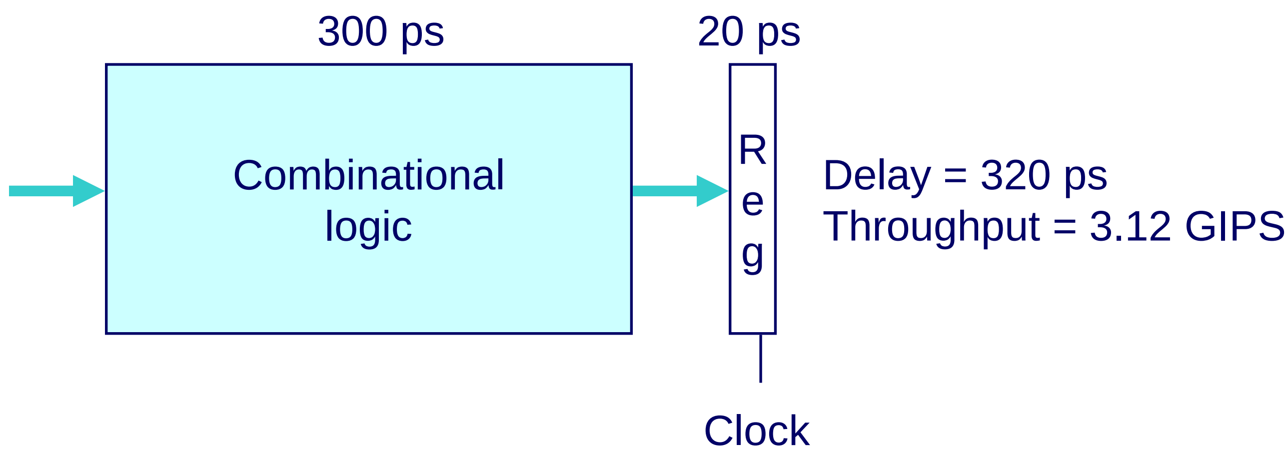
Parallel

Pipelined

Idea

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

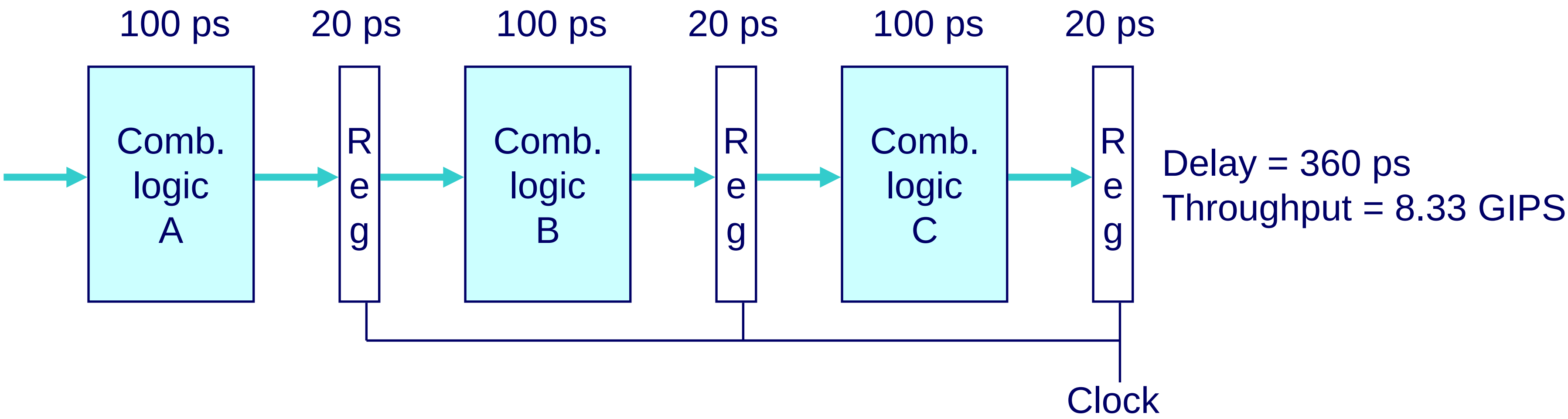
Computational Example



System

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

3-Way Pipelined Version

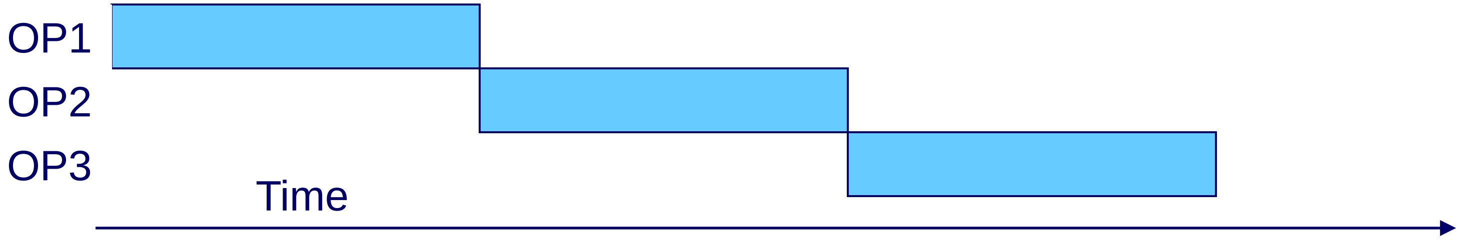


System

- **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**

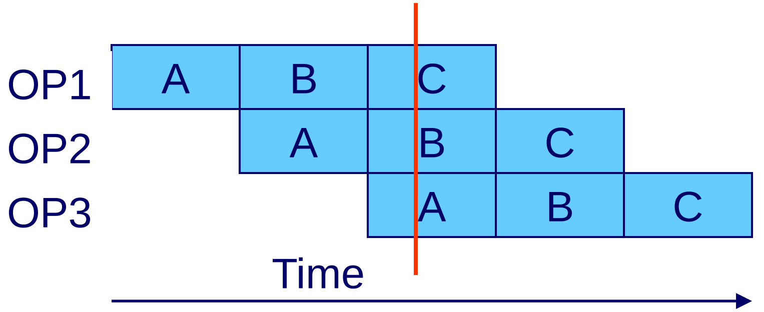
Pipeline Diagrams

Unpipelined



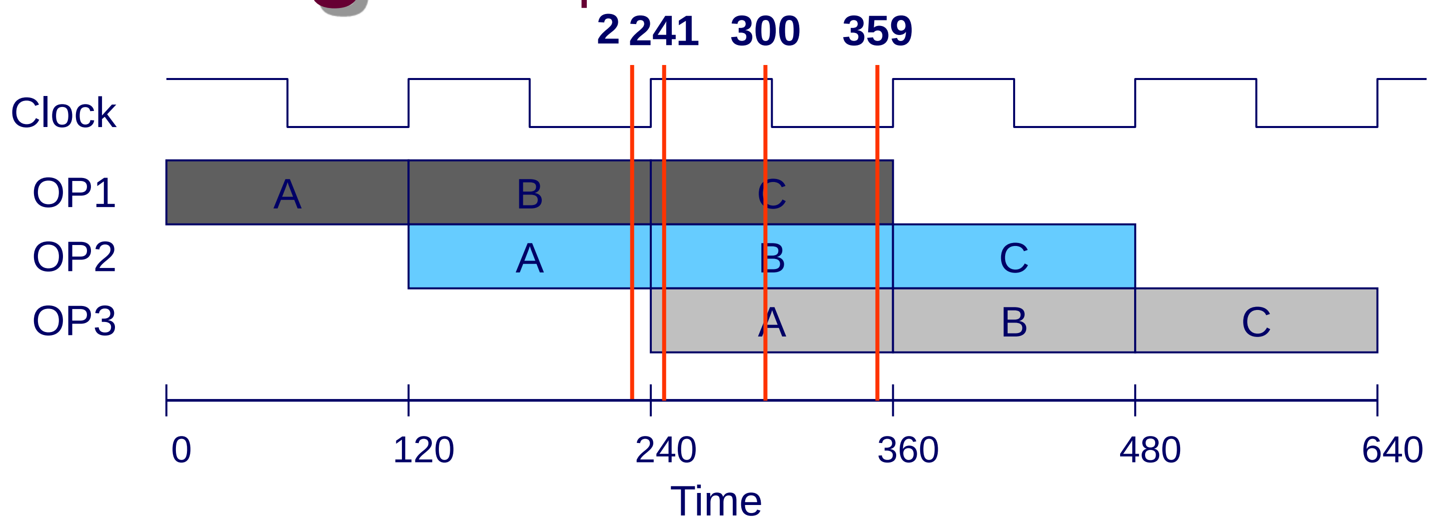
- **Cannot start new operation until previous one completes**

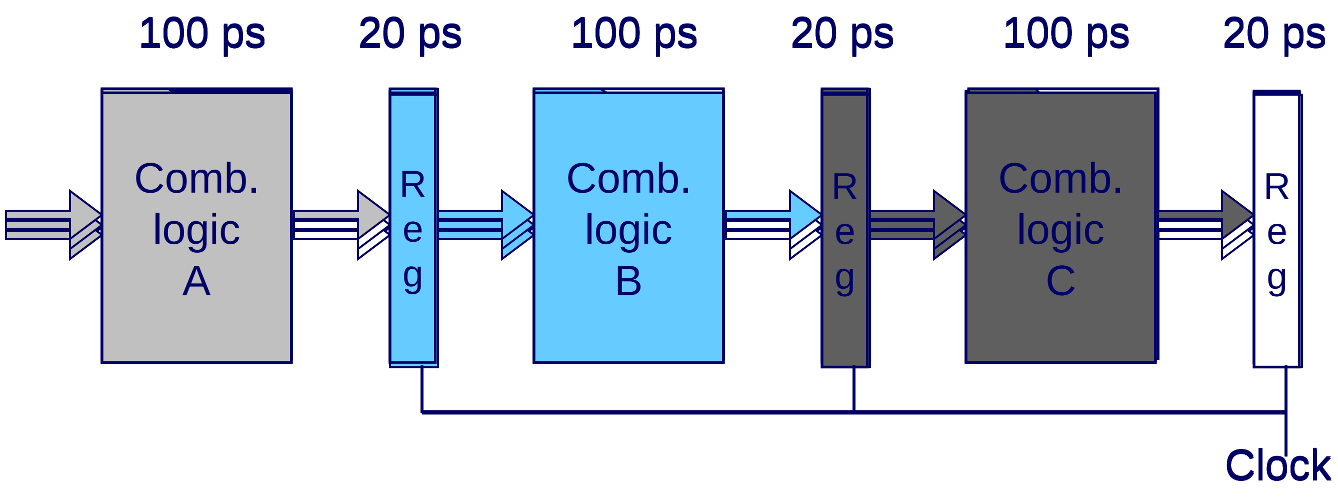
3-Way Pipelined



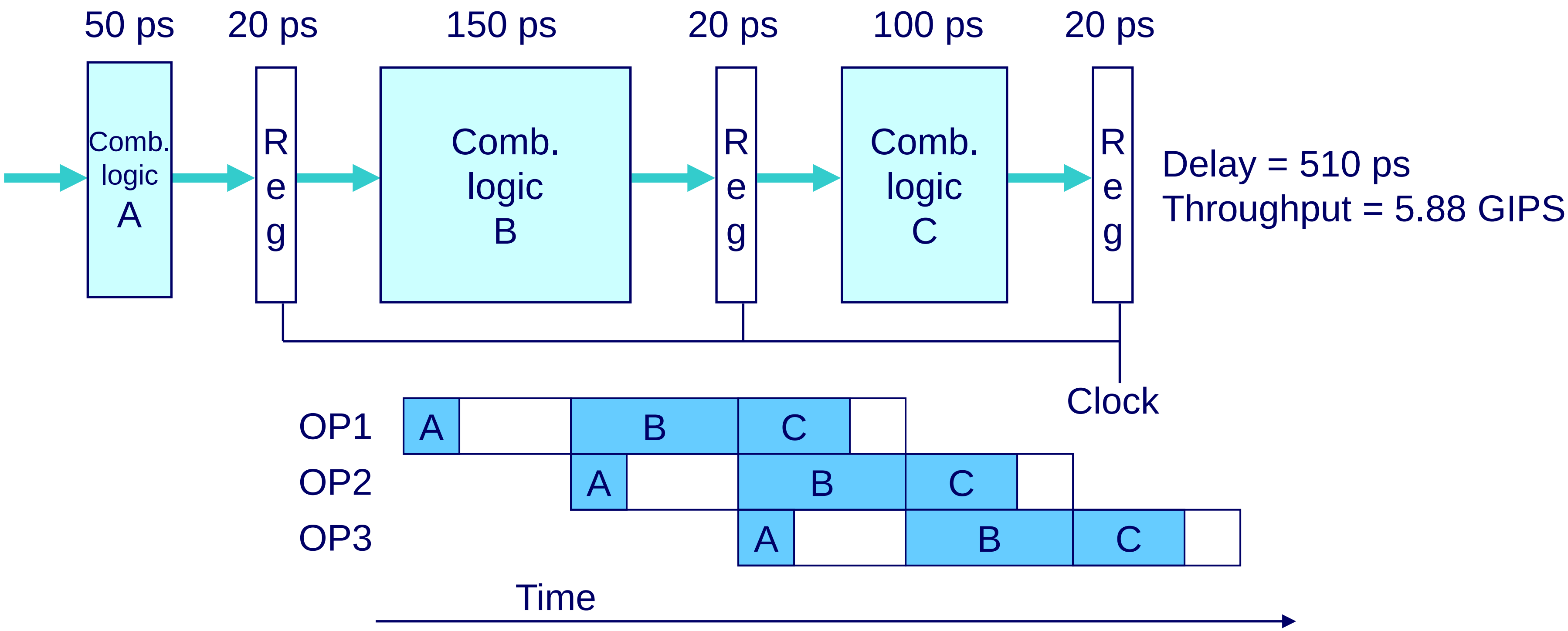
- **Up to 3 operations in process simultaneously**

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**

Sample Circuit Delays & Pipelining

instruction memory	220ps
decode	70ps
register fetch	120ps
ALU	180ps
data memory	260ps
register writeback	120ps

20ps delay for hardware register at end of cycle

Single-cycle processor:

- Clock cycle = 220 + 70 + 120 + 180 + 260 + 120 + 20 = 990ps
- Clock freq = 1 / 990ps = 1 / 990*10^{-12} = 1.01 GHz

Combine and/or split stages for pipelining

- Need to balance time per stage since clock freq determined by slowest time
- Must maintain original order of stages, so can't combine non-neighboring stages (e.g. can't combine *decode & data mem*)

Sample Circuit Delays & Pipelining

instruction memory	220ps
decode	70ps
register fetch	120ps
ALU	180ps
data memory	260ps
register writeback	120ps

20ps delay added for hardware register at end of each cycle

3-stage pipeline:

- **Best combination for minimizing clock cycle time:**
  - **1st stage – *instr mem & decode*: 220 + 70 + 20 = 310ps**
  - **2nd stage – *reg fetch & ALU*: 120 + 180 + 20 = 320ps**
  - **3rd stage – *data mem & reg WB*: 260 + 120 + 20 = 400ps**
- **Slowest stage is 400ps, so clock cycle time is 400ps**
- **Clock freq = 1 / 400ps = 1 / 400*10^{-12} = 2.5 GHz**

Sample Circuit Delays & Pipelining

instruction memory	220ps
decode	70ps
register fetch	120ps
ALU	180ps
data memory	260ps
register writeback	120ps

20ps delay added for hardware register at end of each cycle

5-stage pipeline:

- **Best combination for minimizing clock cycle time:**
  - **1st stage – *instr mem*:** **220 + 20ps** **= 240ps**
  - **2nd stage – *decode* & *reg fetch*:** **70 + 120 + 20ps** **= 210ps**
  - **3rd stage – *ALU*:** **180 + 20ps** **= 200ps**
  - **4th stage – *data mem*:** **260 + 20ps** **= 280ps**
  - **5th stage – *reg WB*:** **120 + 20ps** **= 140ps**
- **Slowest stage is 280ps, so clock cycle time is 280ps**
- **Clock freq = 1 / 280ps = 1 / 280*10^{-12} = 3.57 GHz**

Sample Circuit Delays & Pipelining

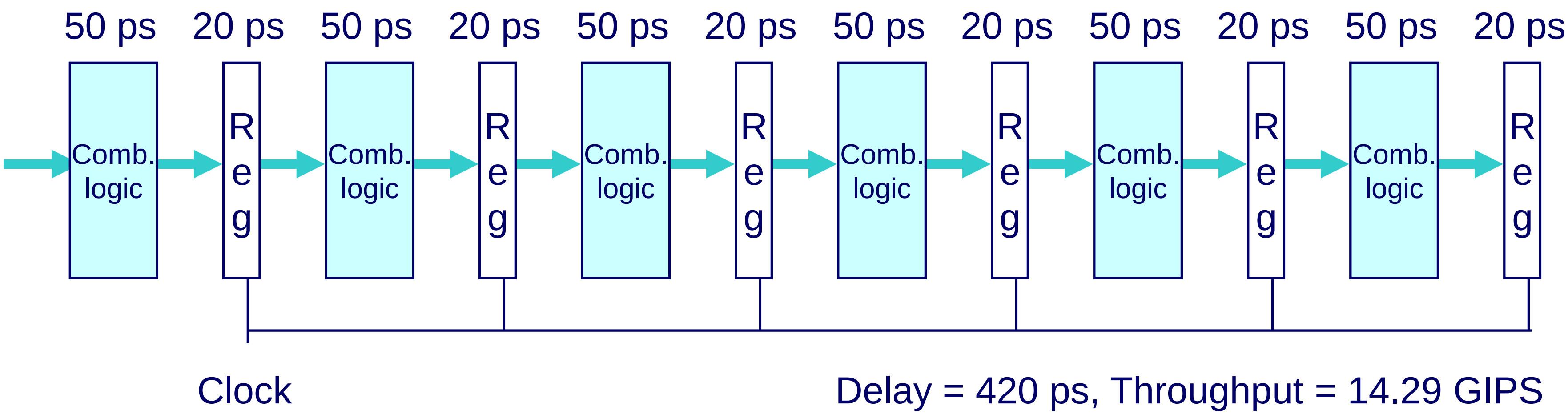
instruction memory	220ps
decode	70ps
register fetch	120ps
ALU	180ps
data memory	260ps
register writeback	120ps

20ps delay added for hardware register at end of each cycle

9-stage pipeline:

- **Assuming can split stages evenly into halves, thirds, or quarters**
  - **not a valid assumption, but useful for simplifying problem**
- **Best combination for minimizing clock cycle time:**
  - **Each circuit is its own stage, with 20ps added delay for reg**
  - **Split *instr mem* circuit into two stages, each 110+20ps**
  - **Split *data mem* circuit into two stages, each 130+20ps**
  - **Split ALU circuit into two stages, each 90+20ps**
- **Slowest stage is 150ps, so clock cycle time is 150ps**
- **Clock freq = 1 / 150ps = 1 / 150*10^{-12} = 6.67 GHz**

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**

In Practice

- **i386 – 3 stage pipeline**
- **I486 – 5 stages**
- **Pentium 3 – 11 stages**
- **Pentium 4 (willamette)**
  - **20 stages**
- **Pentium 4 (prescott)**
  - **31 stages!**
  - **Up to 3.8GHz**
  - **Severe heat problems**
  - **Long pipeline actually hurt some application's performance**
  - **115 Watts dissipation**

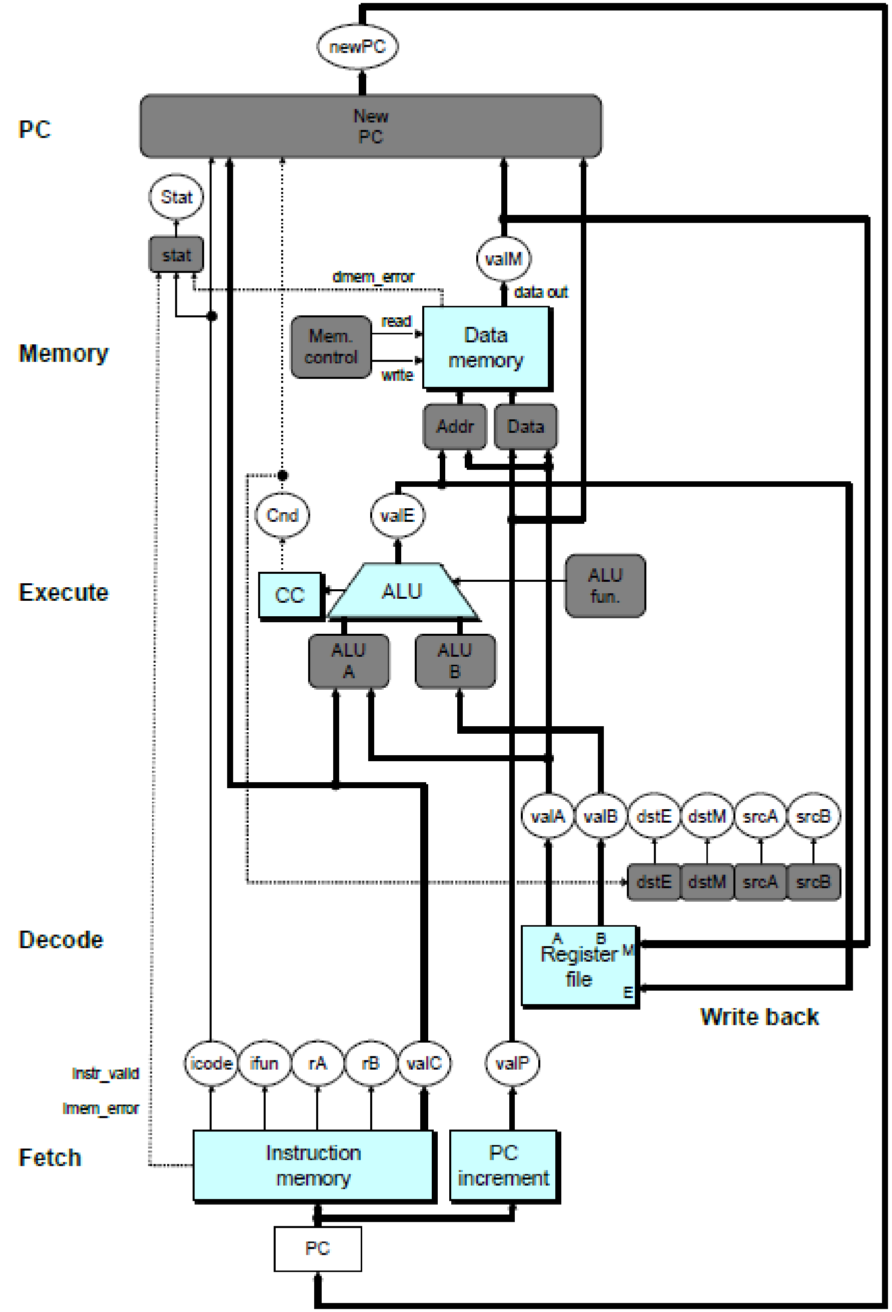
Converting SEQ to PIPE, a pipelined datapath

SEQ Hardware

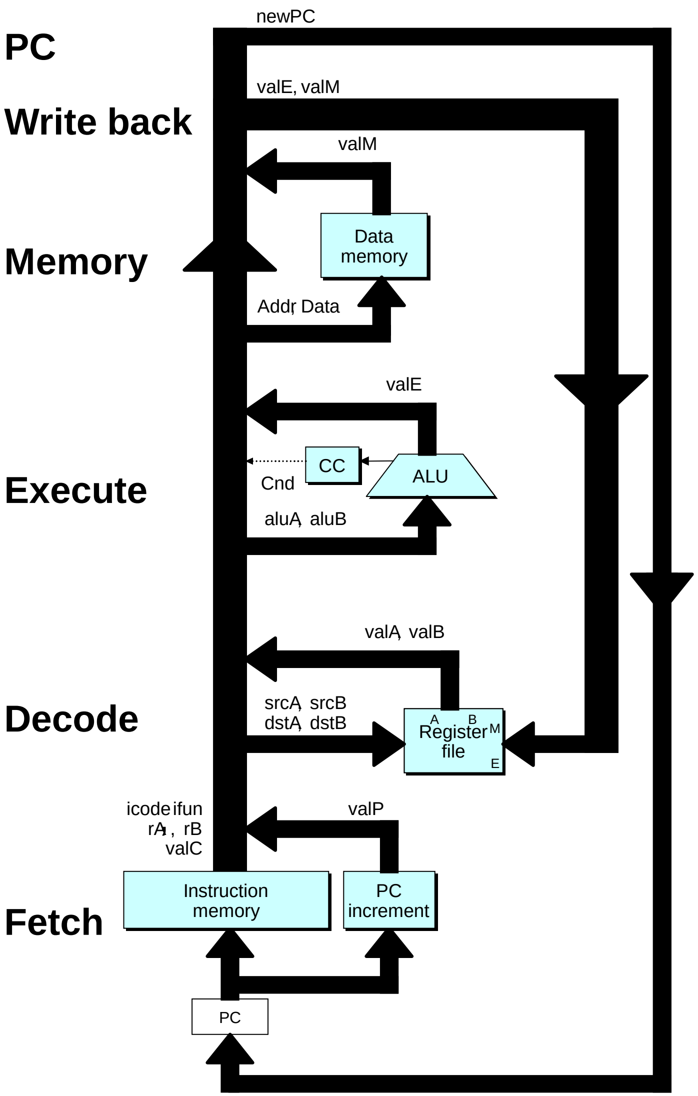
- Stages occur in sequence
- One operation in process at a time

To convert to pipelined datapath, start by adding registers between stages, resulting in 5 pipeline stages:

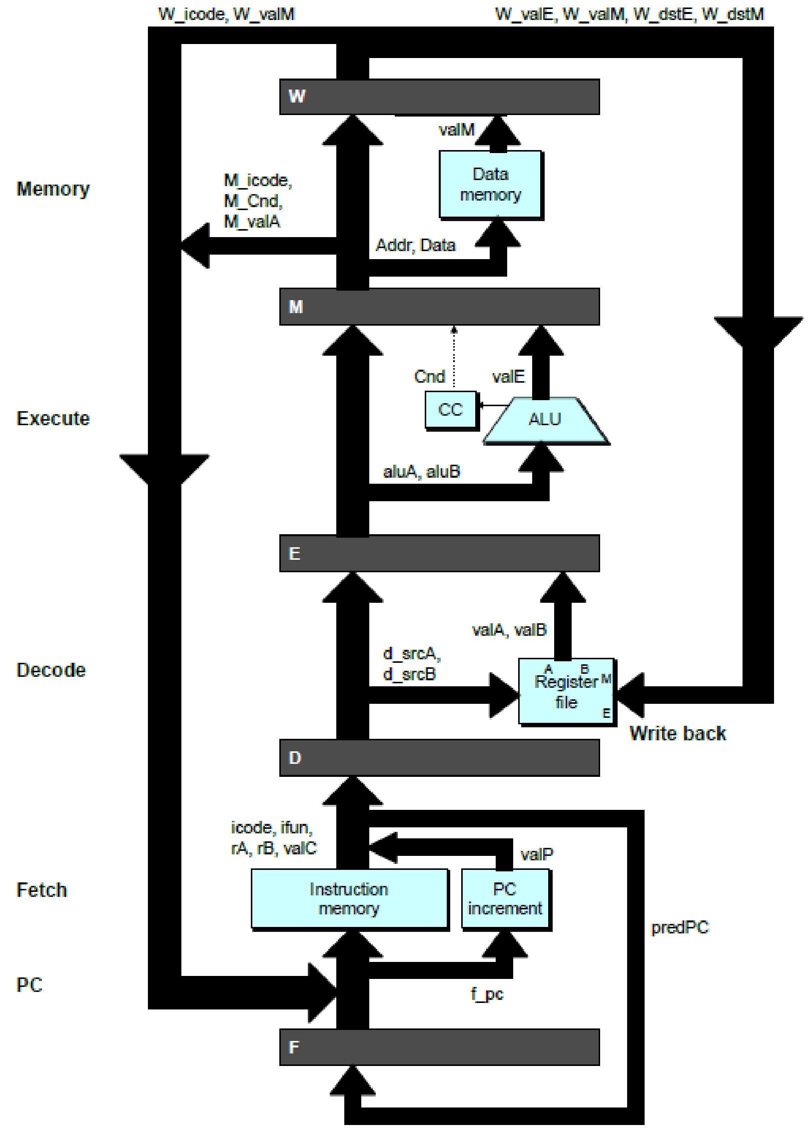
- Fetch
- Decode
- Execute
- Memory
- Writeback



Converting to pipelined datapath



Add pipeline registers between stages



Problem: Fetching a new instruction each cycle

Two problems

- PC generated in last stage of SEQ datapath
- PC sometimes not available until end of Execute or Memory stage

PC needs to be computed early

- In order to fetch a new instruction every cycle, PC generation must be moved to first stage of datapath
- Solve first problem by moving PC generation from end of SEQ to beginning of SEQ

Use prediction to select PC early

- Solve second problem by <u>predicting</u> next instruction from current instruction
- If prediction is wrong, squash (kill) predicted instructions

SEQ+ Hardware

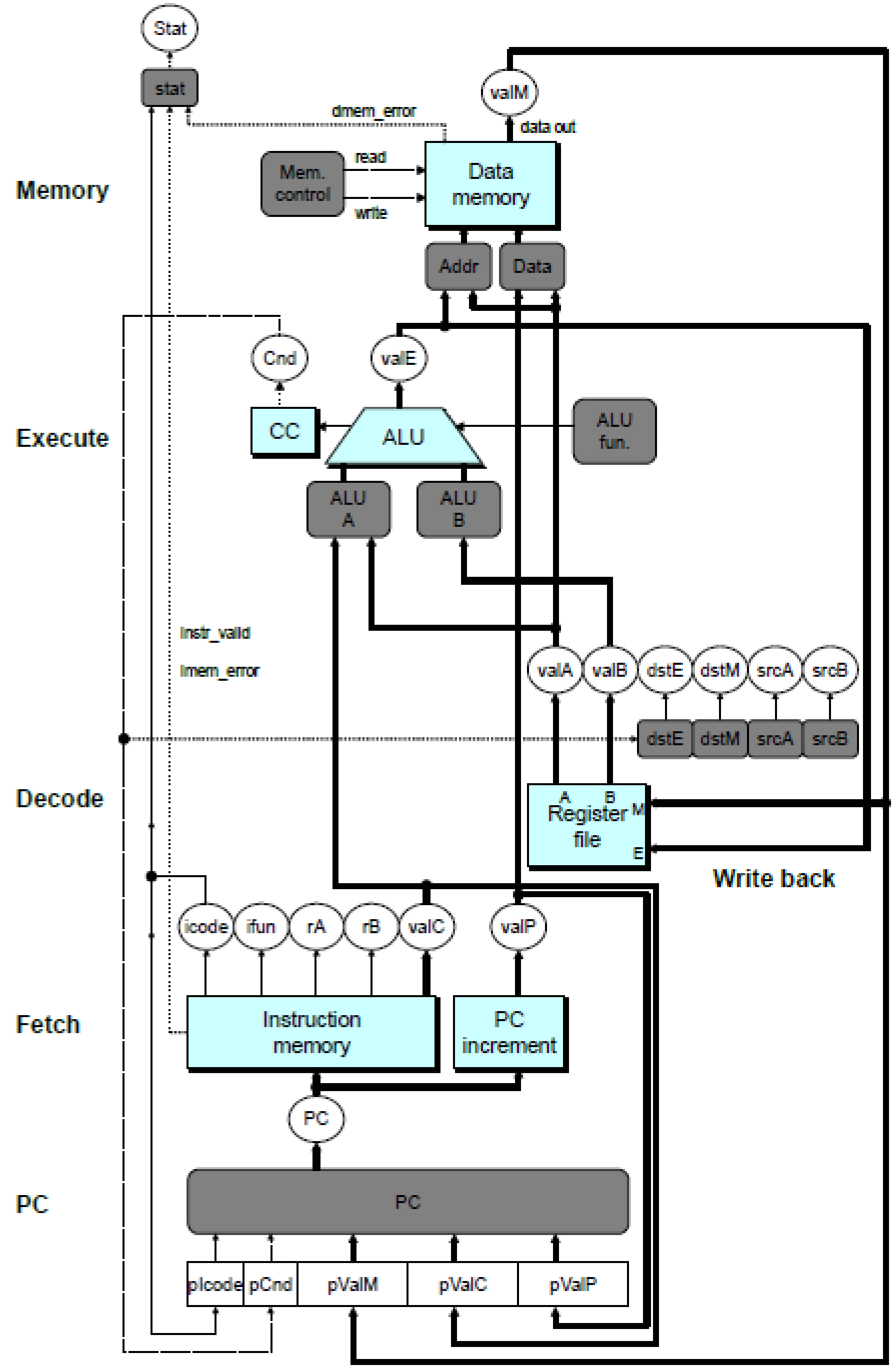
- Still sequential implementation
- Reorder PC stage to put at beginning

PC Stage

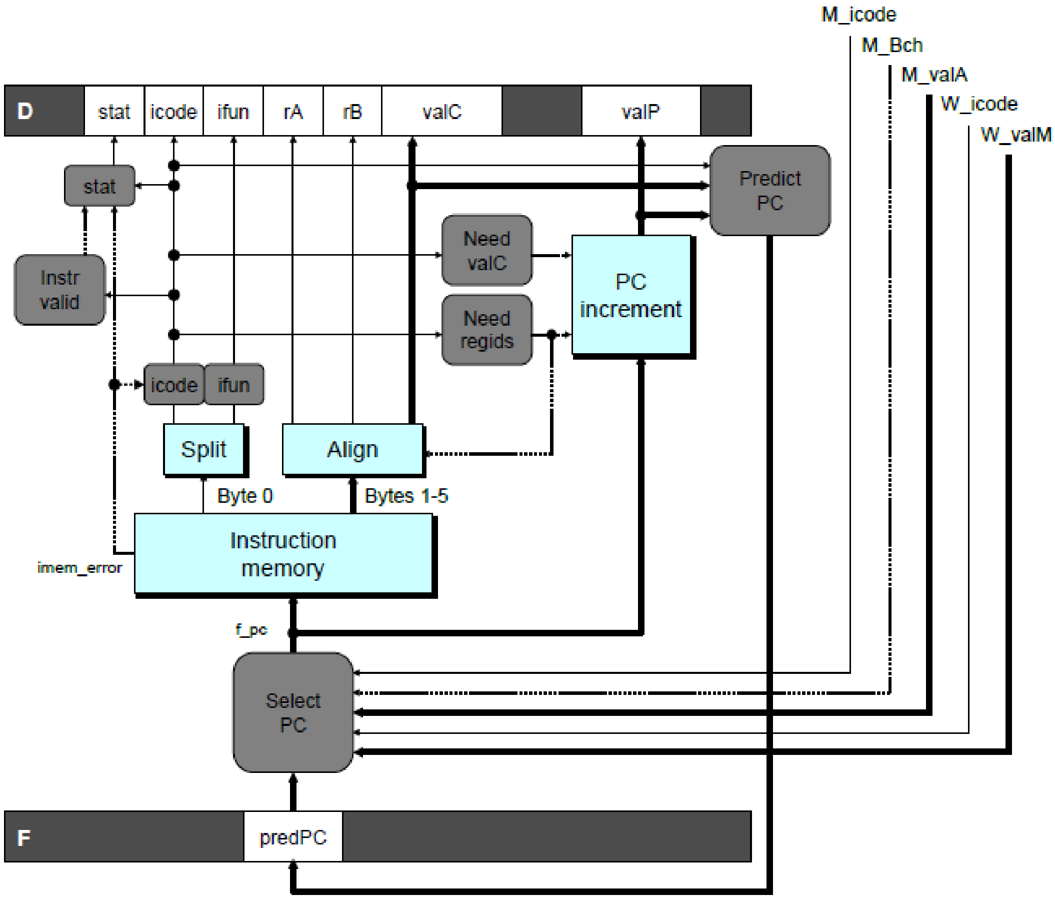
- Task is to select PC for current instruction
- Based on results computed by previous instruction

Processor State

- PC is no longer stored in register
- But, can determine PC based on other stored information



Predicting the PC



- Start fetch of new instruction after current has been fetched
  - Not enough time to fully determine next instruction
- Attempt to predict which instruction will be next
  - Recover if prediction was incorrect

Our Prediction Strategy

Predict next instruction from current instruction

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

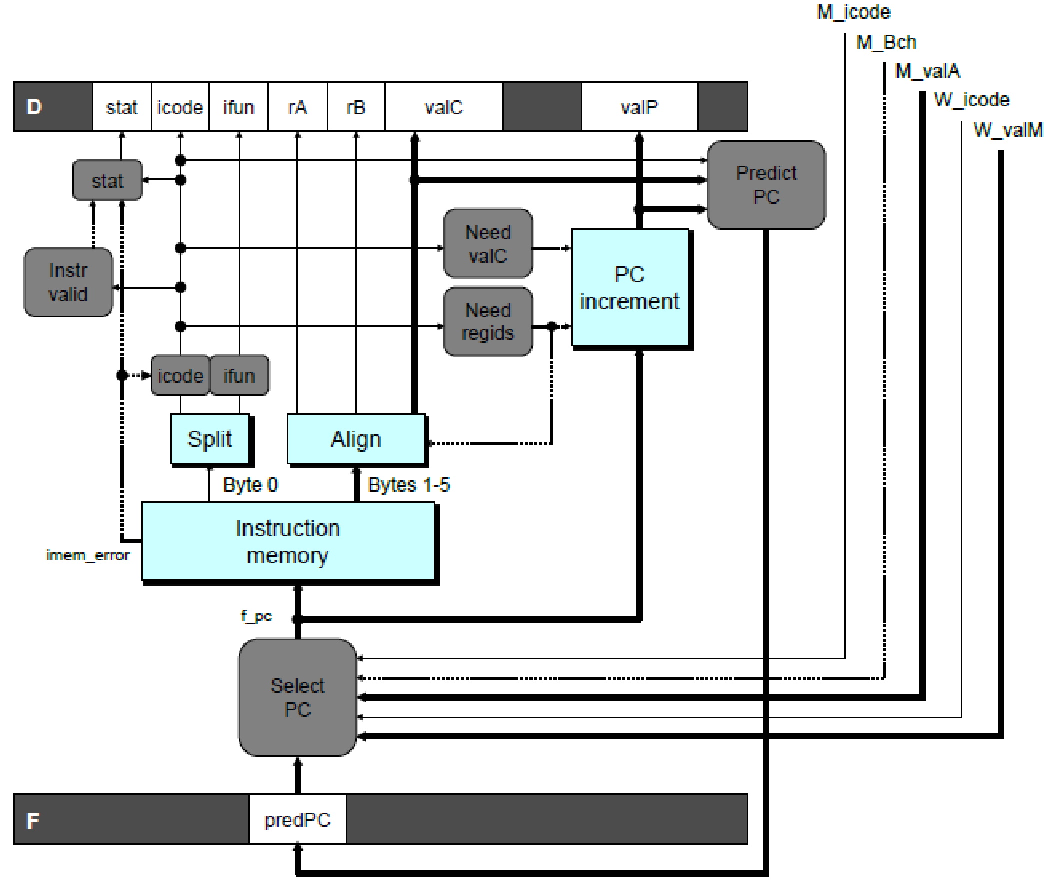
Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

Return Instruction

- Don’t predict, just stall

Recovering from PC Misprediction



Mispredicted Jump

- Will see branch condition flag once instruction reaches memory stage
- Can get fall-through PC from valA (value M_valA)

Return Instruction

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

Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

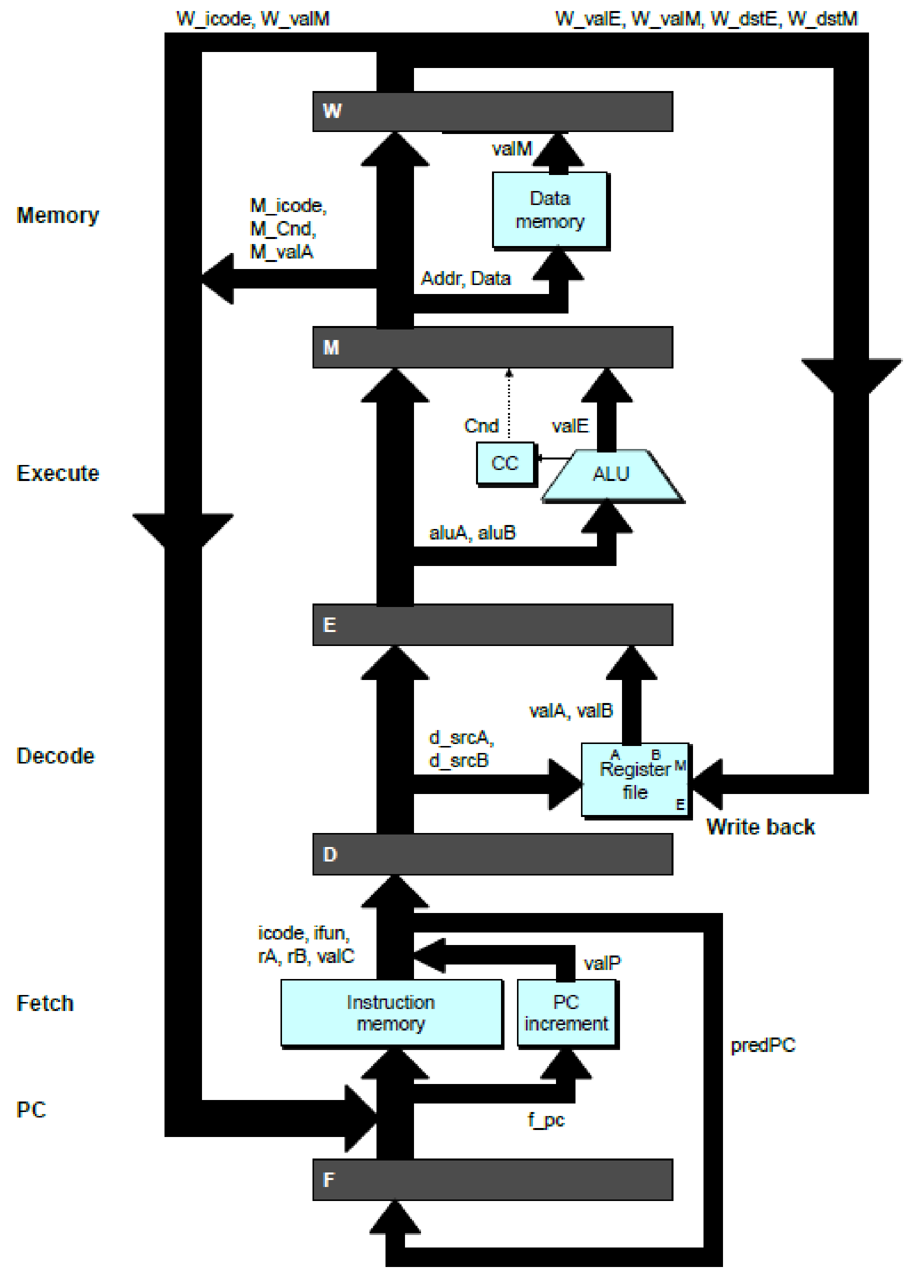
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file

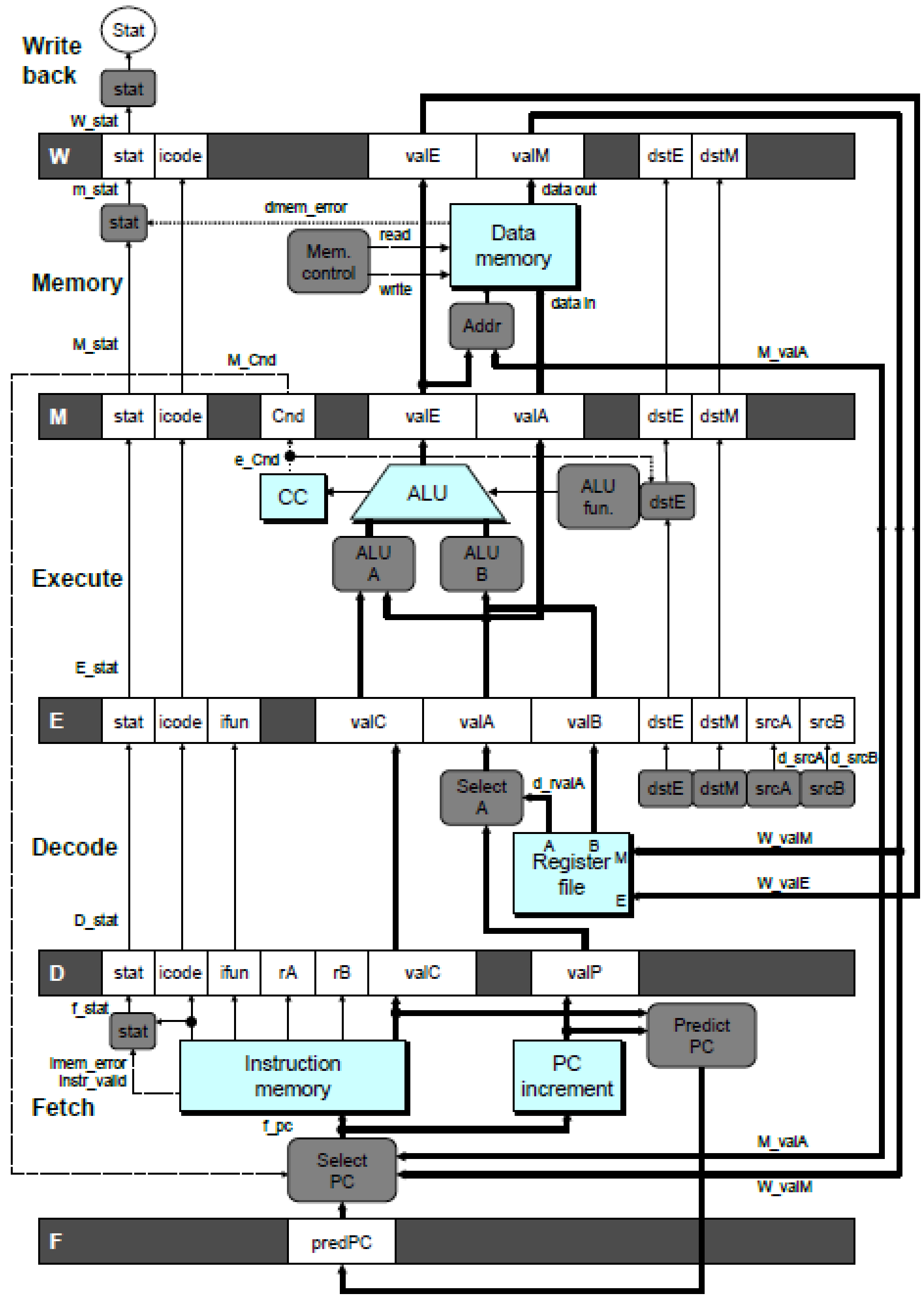


PIPE- Hardware

- 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

Important for distinguishing dependencies between pipeline stages

Predicted PC

- Guess value of next PC

Branch information

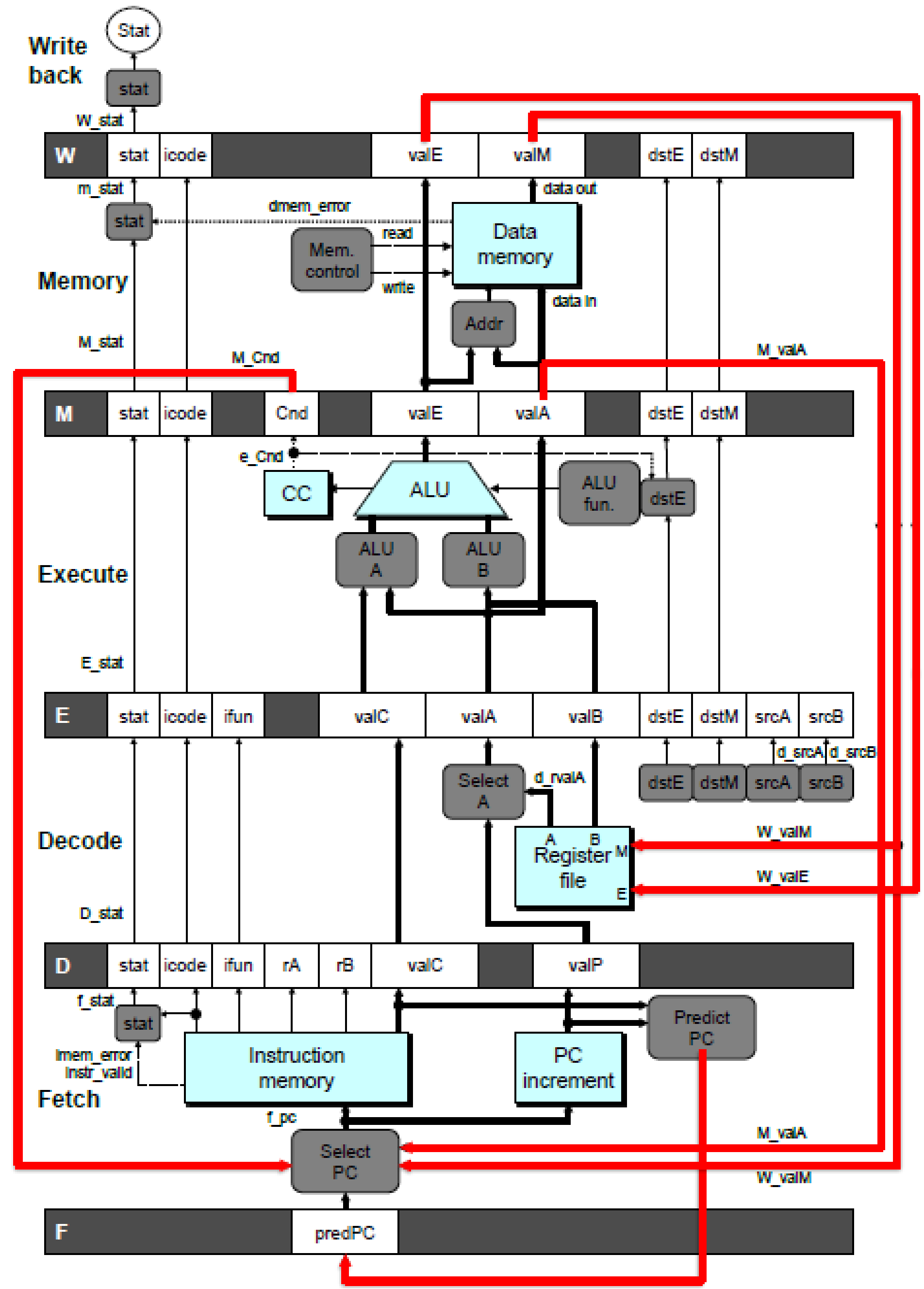
- Jump taken/not-taken
- Fall-through or target address

Return point

- Read from memory

Register updates

- To register file write ports



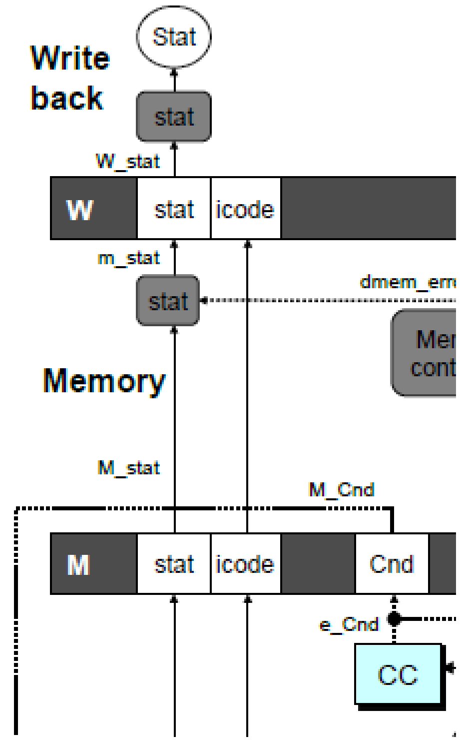
Signal Naming Conventions

S_Field

- Value of Field held in stage S pipeline register

s_Field

- Value of Field computed in stage S



Dealing with Dependencies between Instructions

Hazards

Hazards

- Problems caused by dependencies between separate instructions in the pipeline

Data Hazards

- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

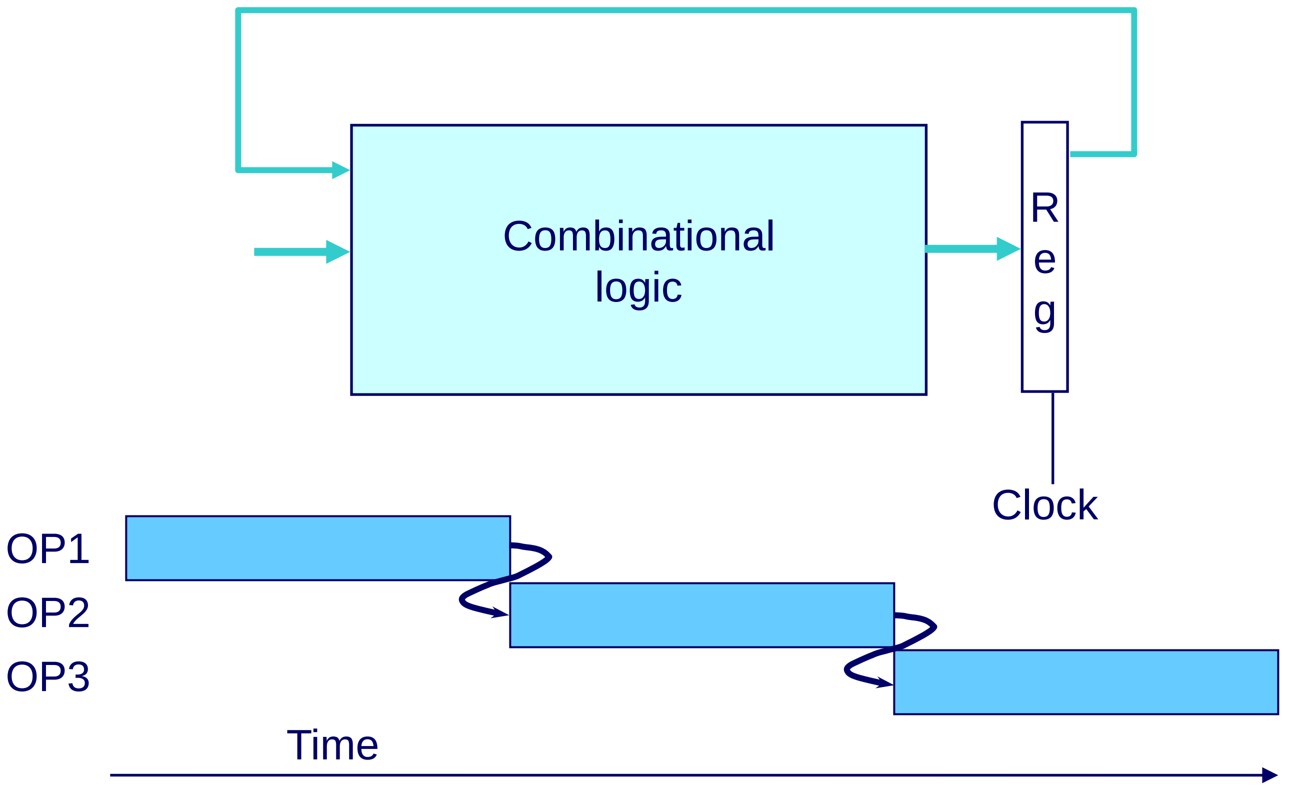
Control Hazards

- Mispredict conditional branch
  - Our design predicts all branches as being taken
  - Naïve pipeline executes two extra instructions
- Getting return address for `ret` instruction
  - Naïve pipeline executes three extra instructions

Dealing with Dependencies between Instructions

Data Hazards

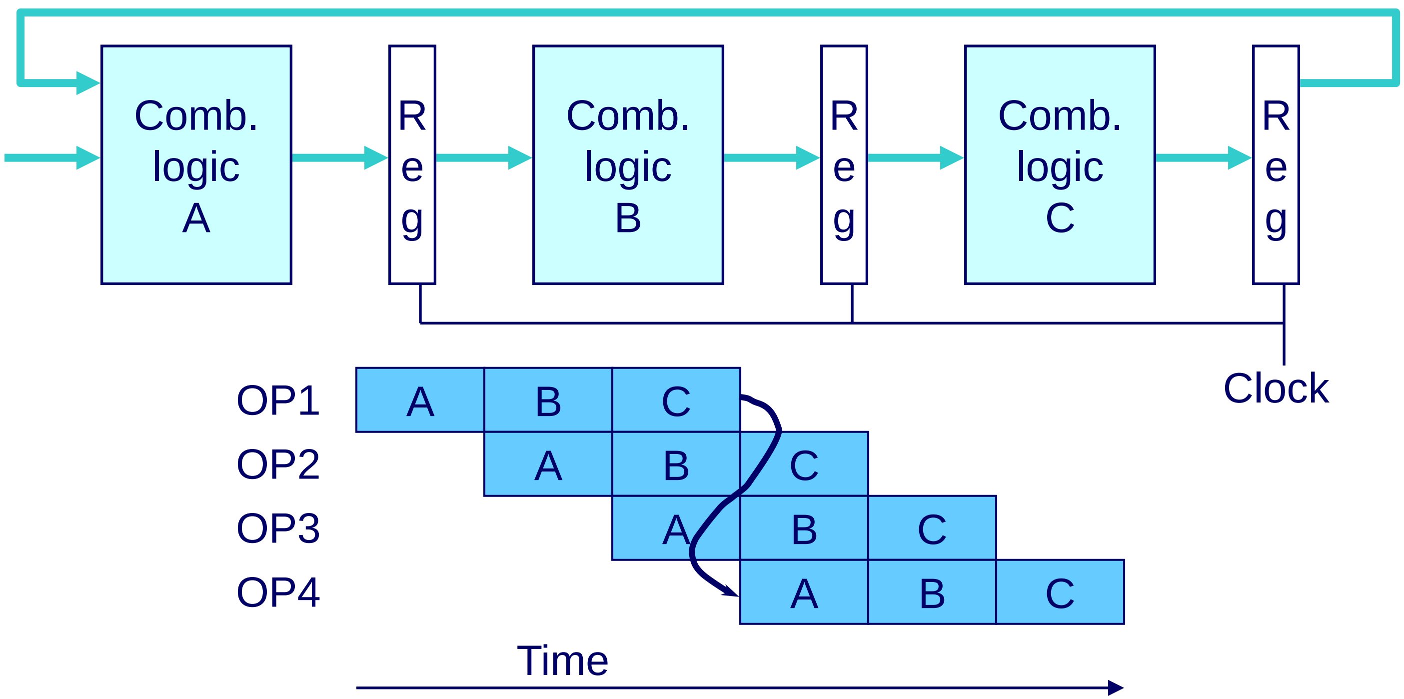
Data Dependencies
- not a problem in SEQ



System

- Each operation depends on result from preceding one

Data Hazards
- the problems caused by data dependences in pipelined datapaths



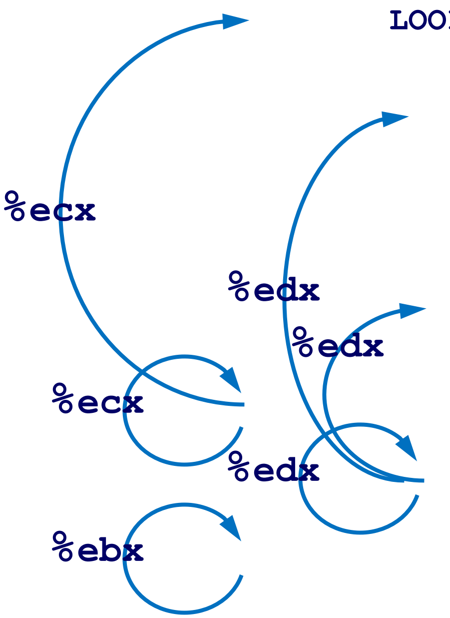
- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

Data Dependencies between Instructions

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

- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
  - Dependency is between writeback stage of earlier instruction and decode stage of later instruction
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

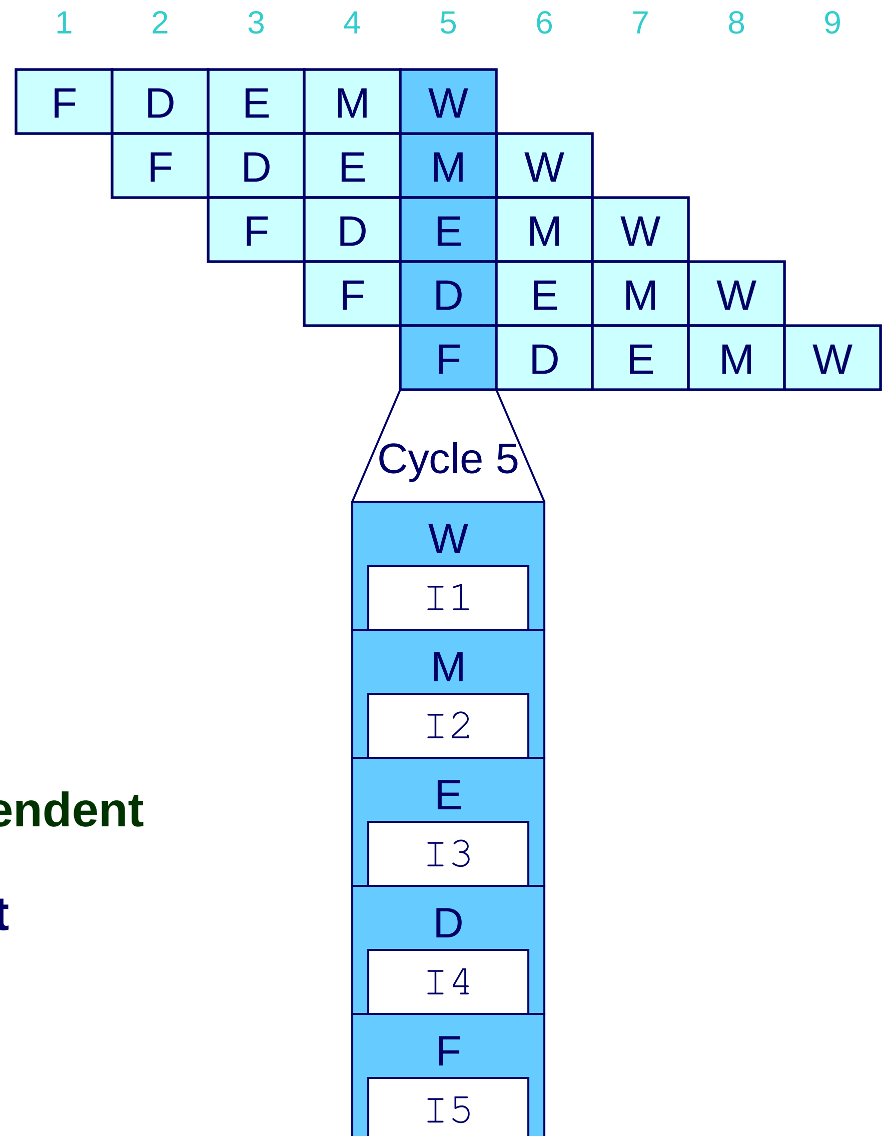
Data Dependencies – Loop-Carried Dependencies




```
        mrmovl  4(%esp), %ecx
        mrmovl  8(%esp), %edx
        mrmovl  12(%esp), %ebx
LOOP:   mrmovl  0(%ecx), %edi
        mrmovl  0(%edx), %eax
        addl    %eax, %edi
        rmmovl  %edi, 0(%edx)
        iaddl   $4, %ecx
        iaddl   $4, %edx
        iaddl   $-1, %ebx
        jne     LOOP
```

Pipeline Demonstration

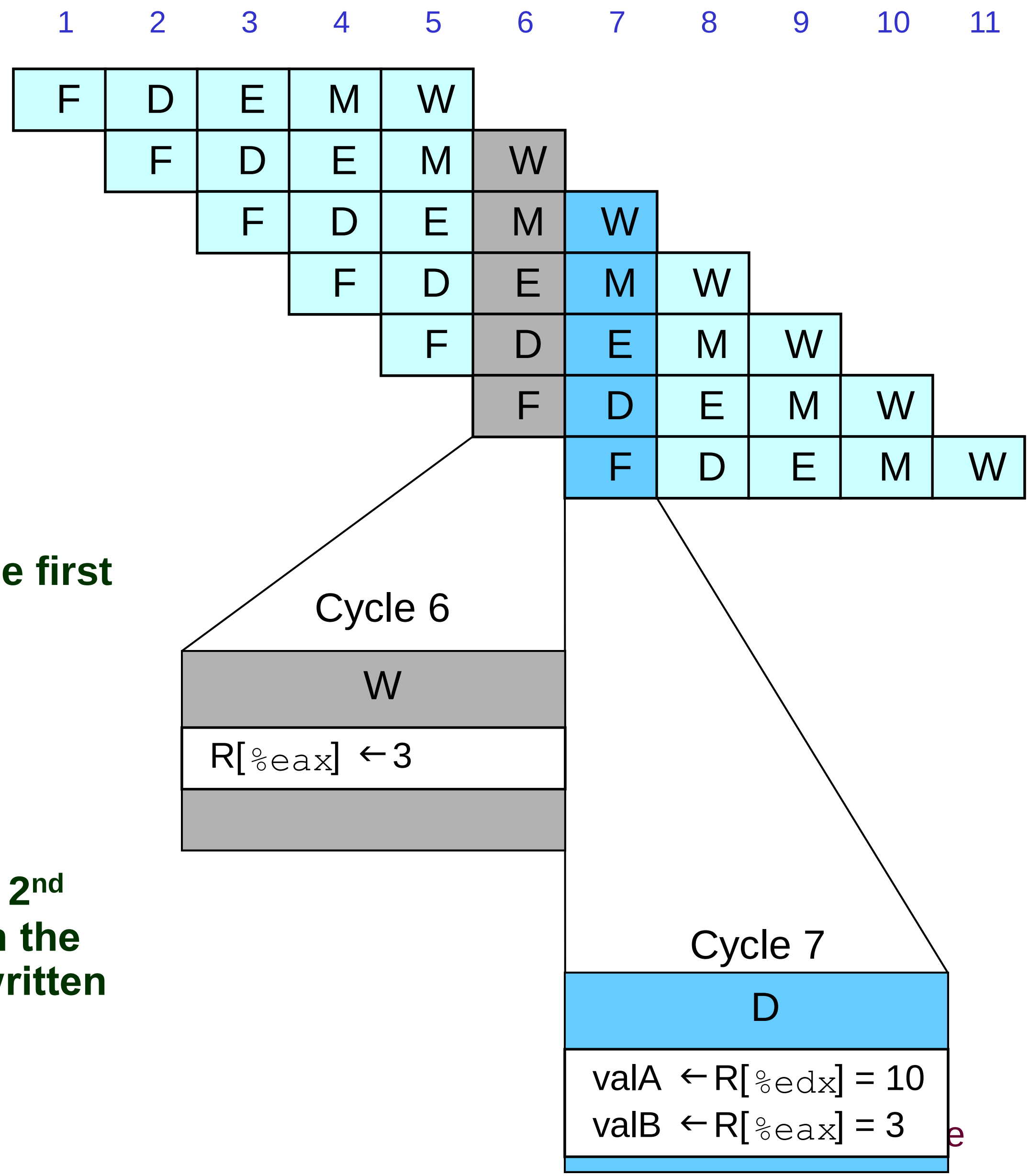
```
irmovl  $1,%eax  #I1
irmovl  $2,%ecx  #I2
irmovl  $3,%edx  #I3
irmovl  $4,%ebx  #I4
halt             #I5
```



All the instructions are independent of each other
- No dependencies exist

Data Dependencies: 3 Nop's

```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```



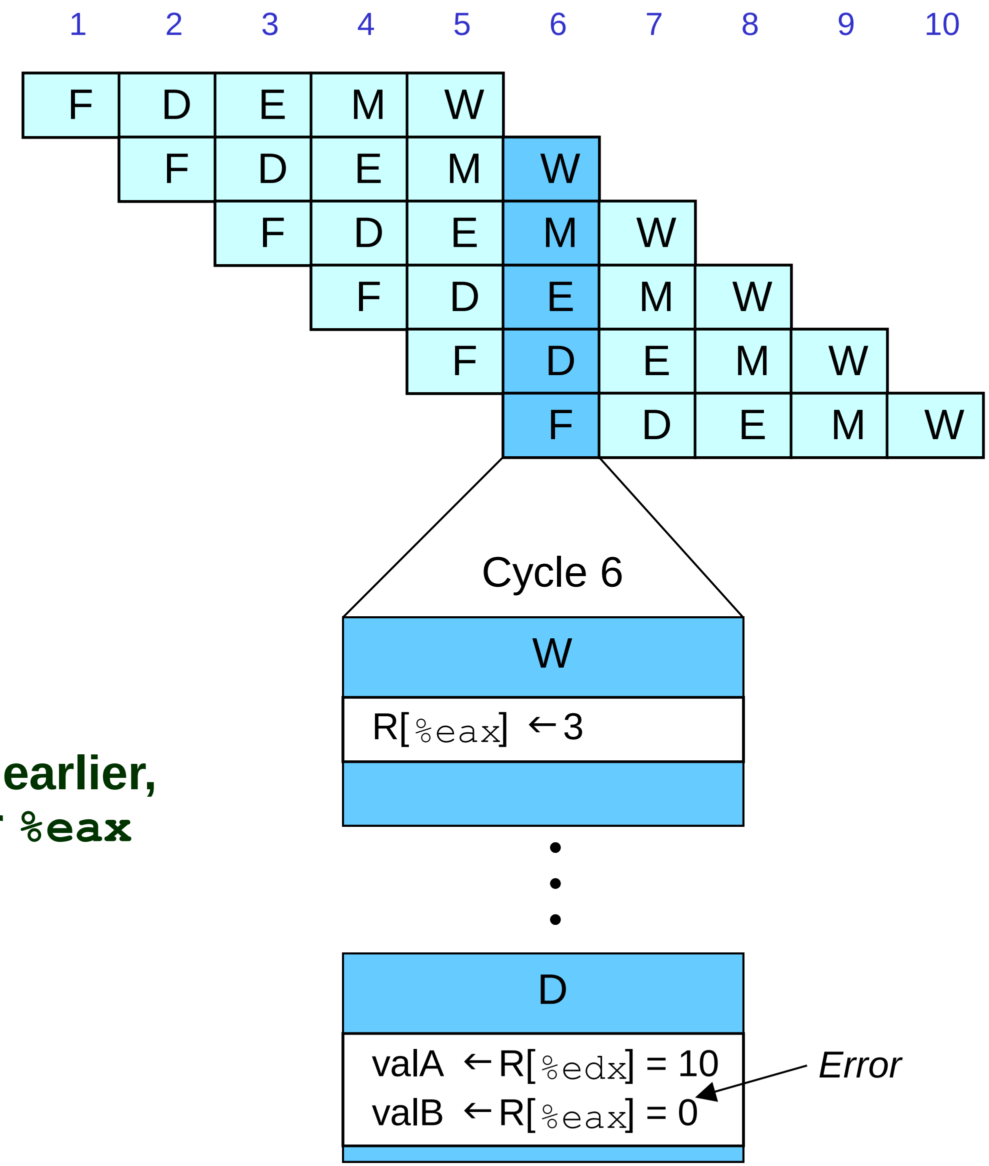
The `addl` instruction depends on the first two instructions

- **`addl` depends upon `%edx` from the 1st instr**
- **`addl` depends upon `%eax` from the 2nd instr**

`addl` must wait 3 cycles after the 2nd instruction, so that it doesn't fetch the two registers before they've been written to the register file

Data Dependencies: 2 Nop's

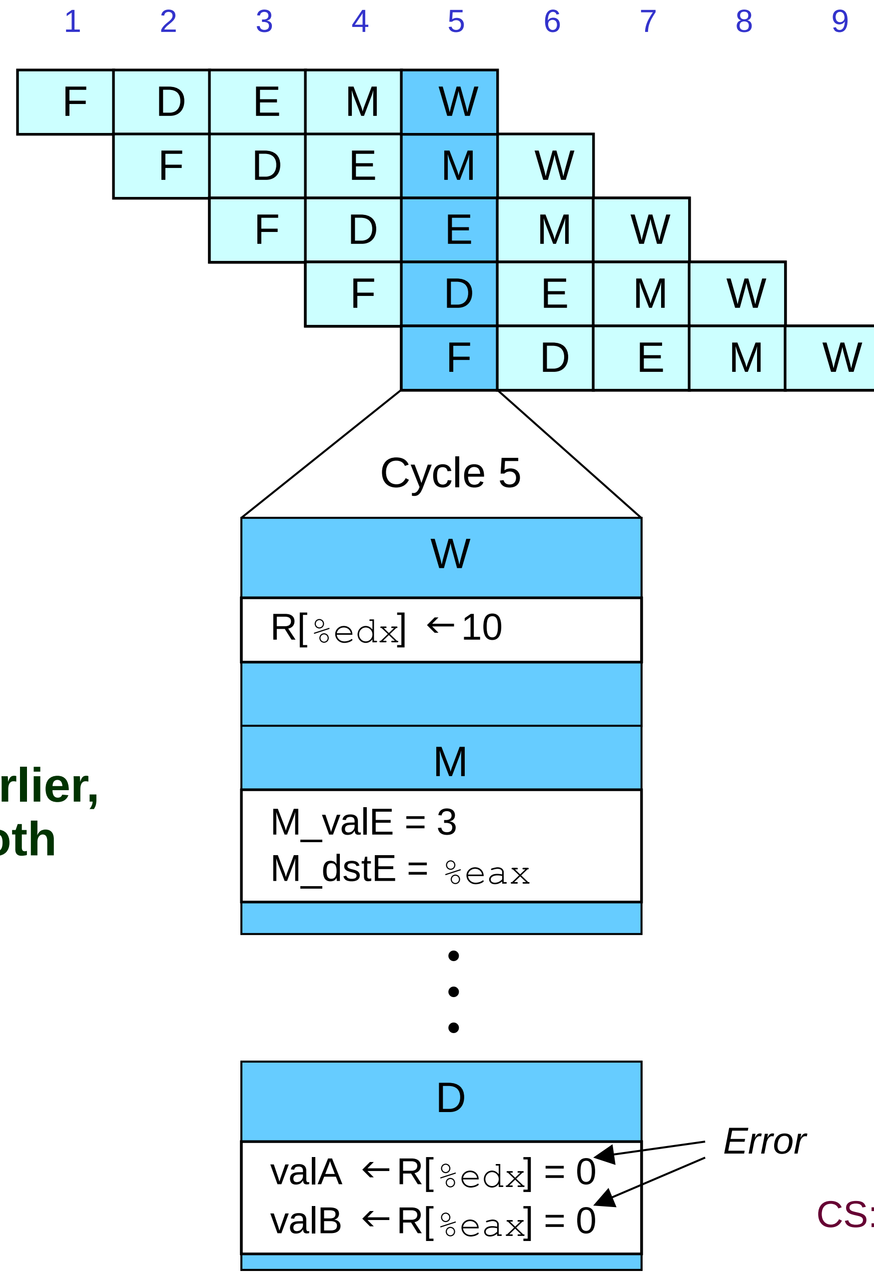
```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



If `addl` executes one cycle earlier, it gets the wrong value for `%eax`

Data Dependencies: 1 Nop

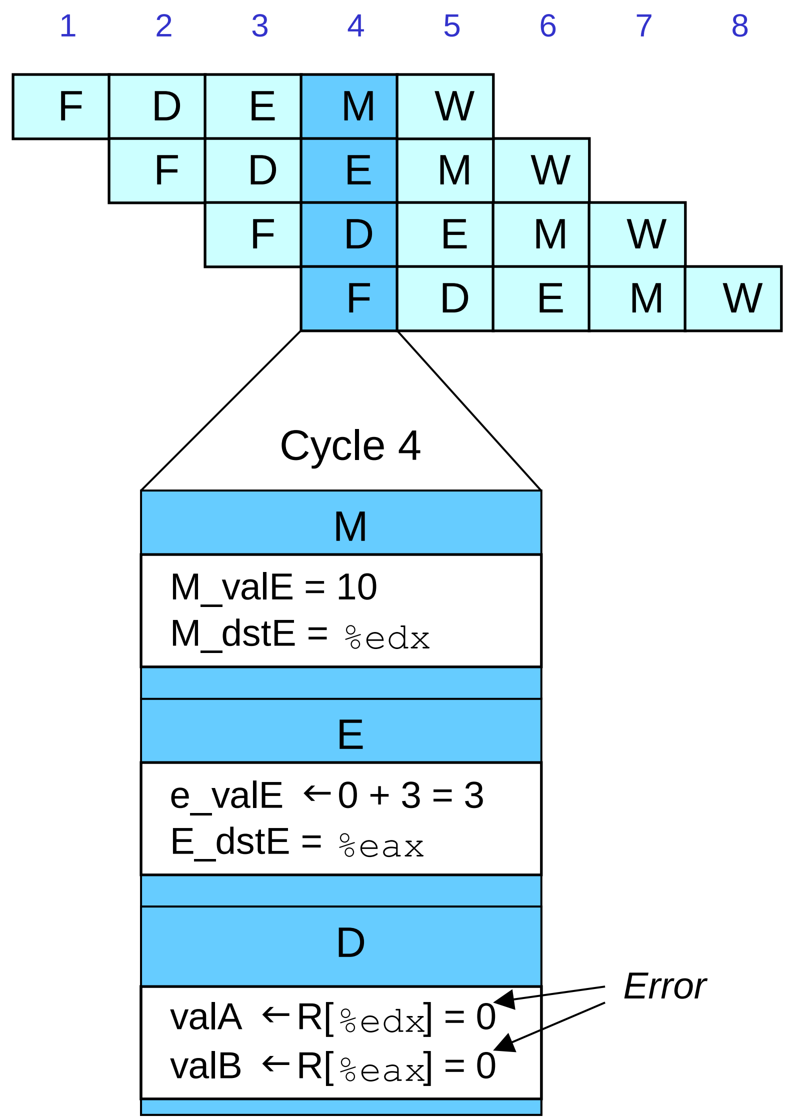
```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl %edx,%eax
0x00f: halt
```



If `addl` executes two cycles earlier, it gets the wrong value for both `%eax` and `%edx`

Data Dependencies: No Nop

```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```



Like the prior case, if `addl` executes three cycles earlier, it gets the wrong value for both `%eax` and `%edx`

Stalling for Data Dependencies

	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, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

Stall Condition

Source Registers

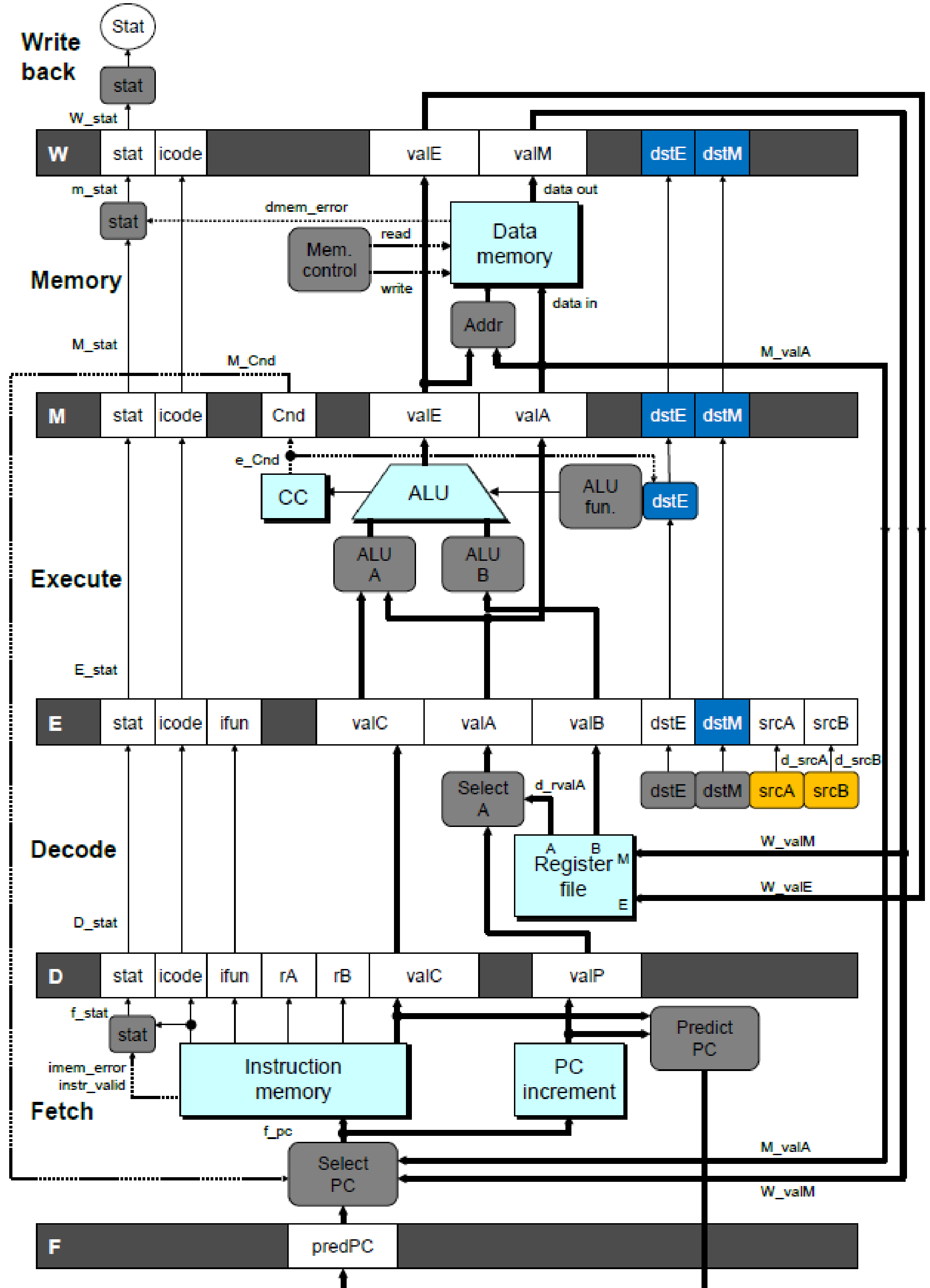
- srcA and srcB of current instruction in decode stage

Destination Registers

- dstE and dstM fields
- Instructions in execute, memory, and write-back stages

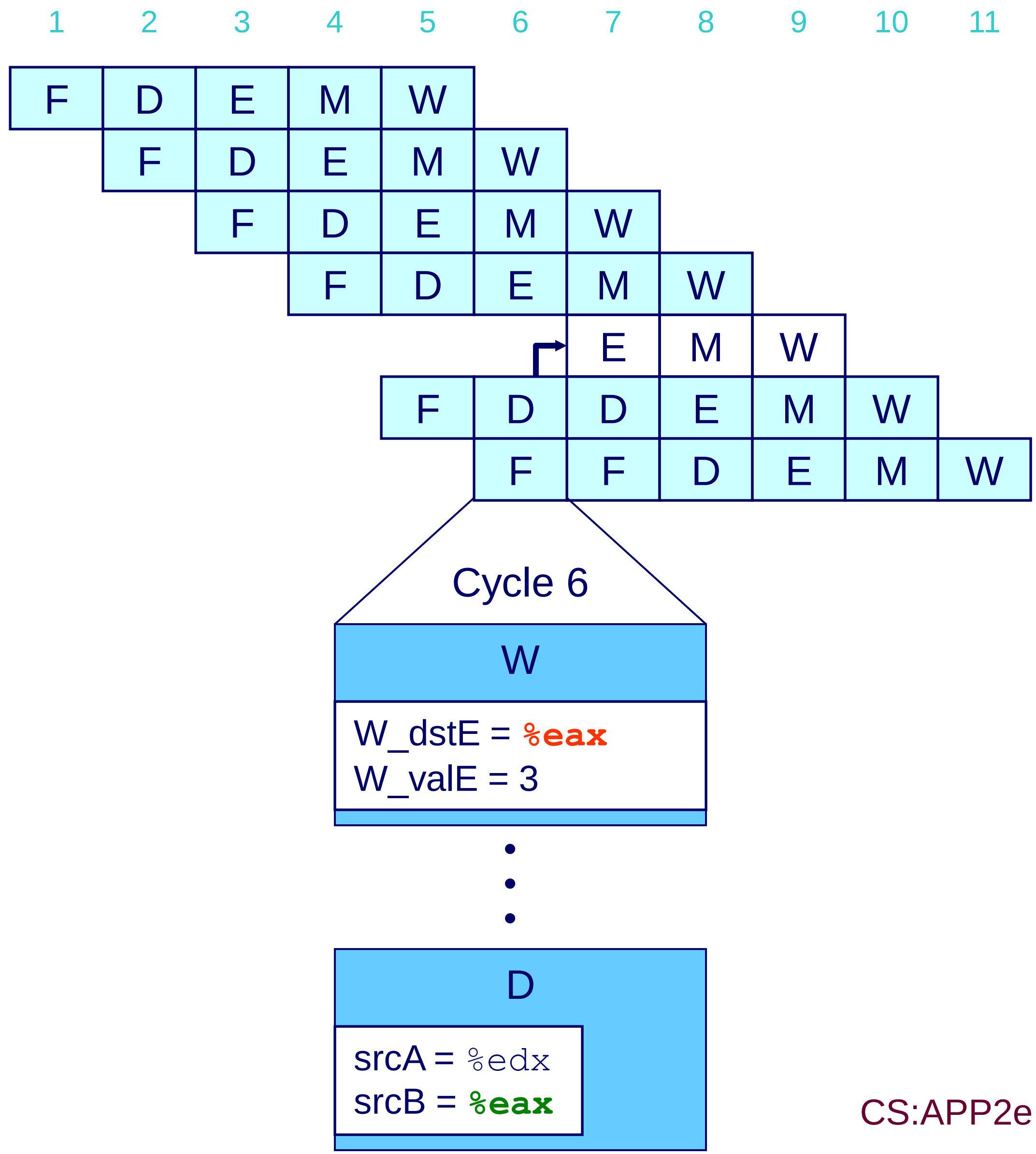
Special Case

- Don’t stall for register ID 15 (0xF)
  - Indicates absence of register operand
- Don’t stall for failed conditional move



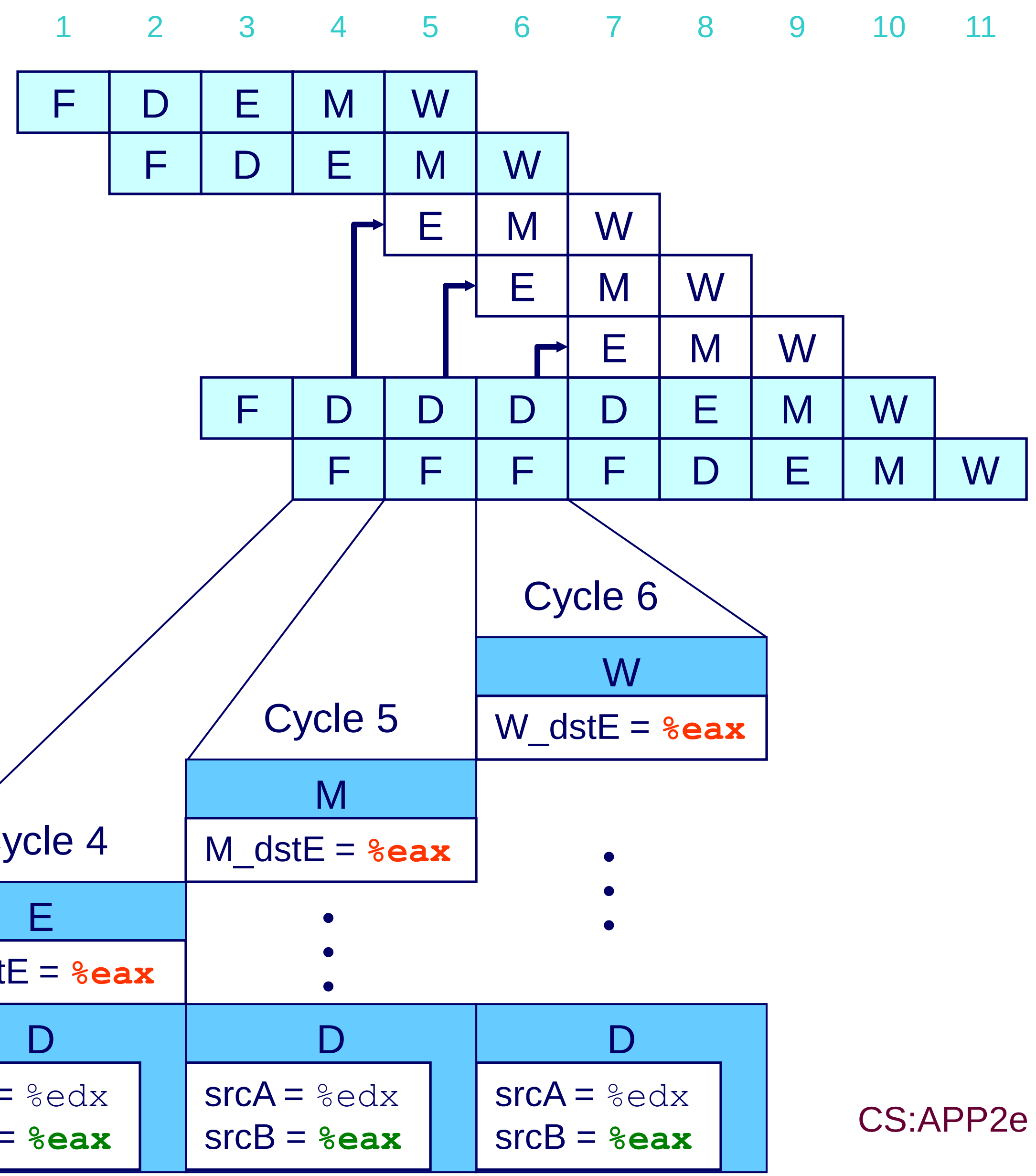
Detecting Stall Condition

```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```



Stalling X3

```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```



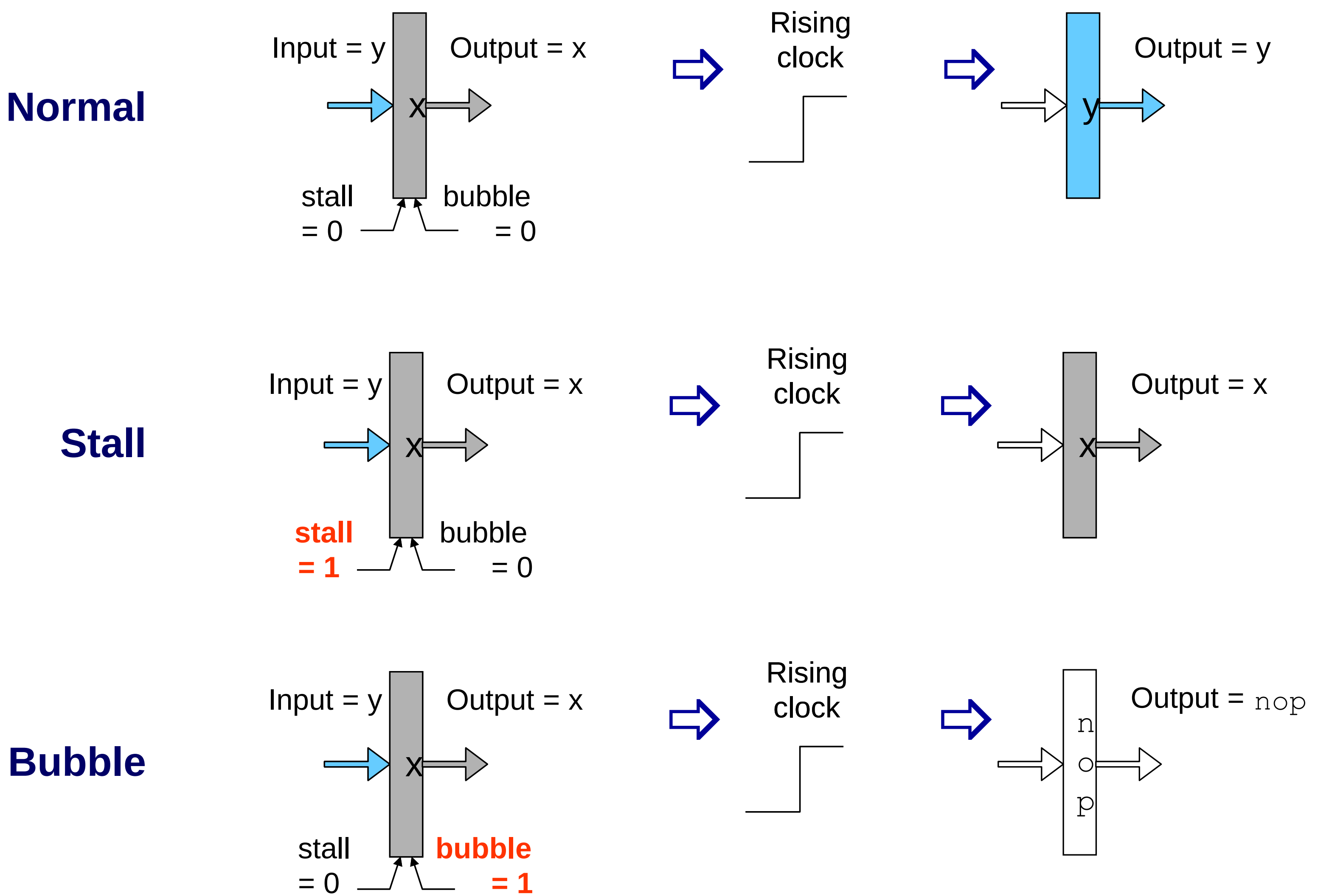
What Happens When Stalling?

```
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

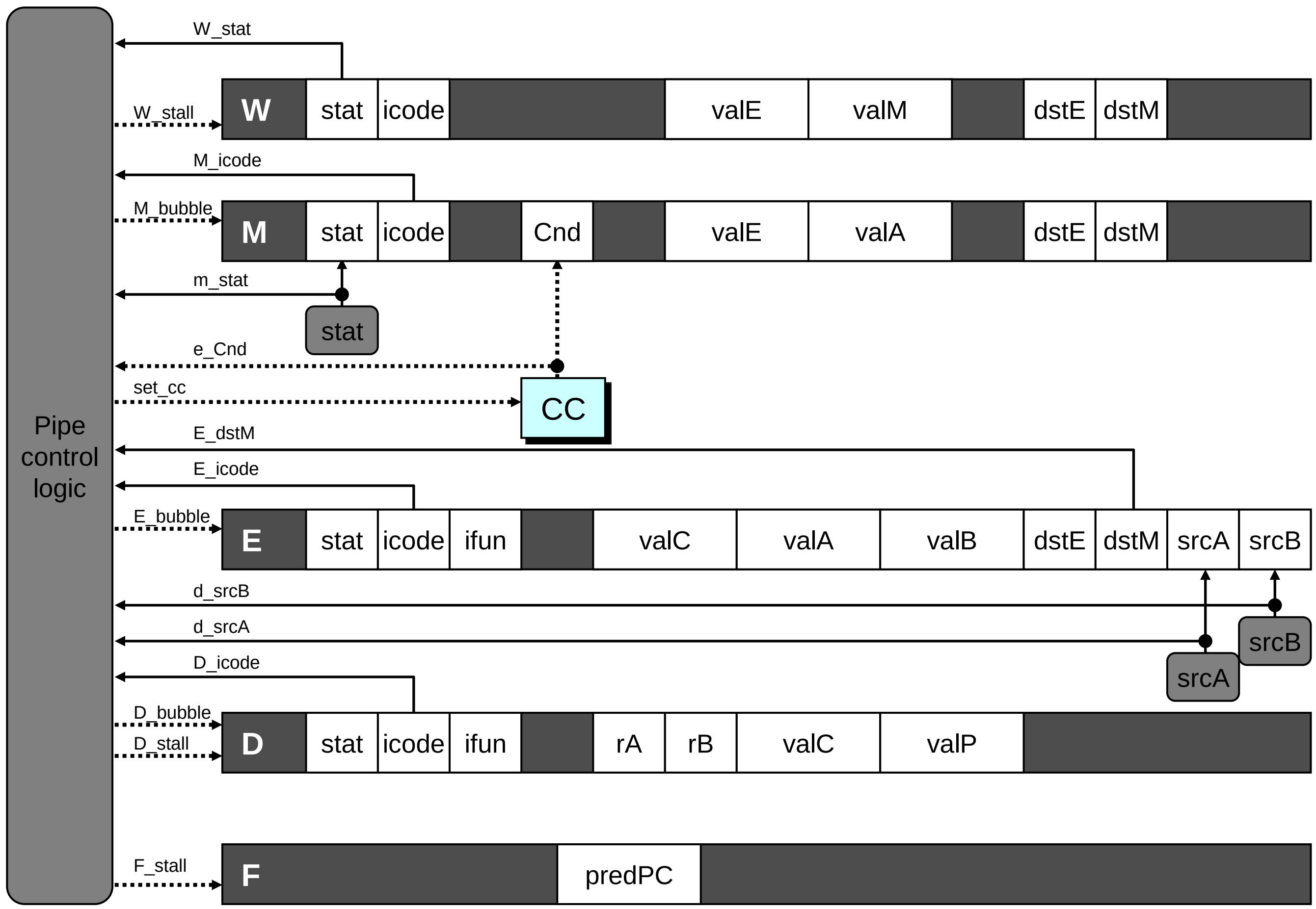
	Cycle 8
Write Back	***bubble***
Memory	***bubble***
Execute	`0x00c: addl %edx,%eax`
Decode	`0x00e: halt`
Fetch	

- **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**

Pipeline Register Modes



Implementing Stalling



Pipeline Control

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

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

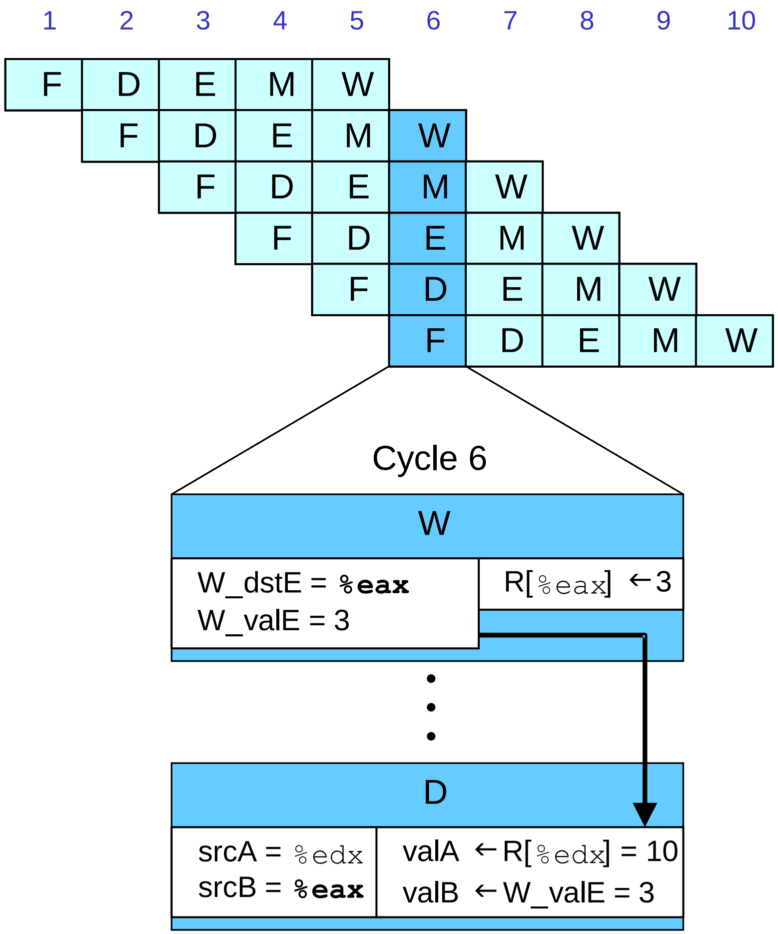
- Value generated in execute or memory stage

Trick

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

Data Forwarding Example

```
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**

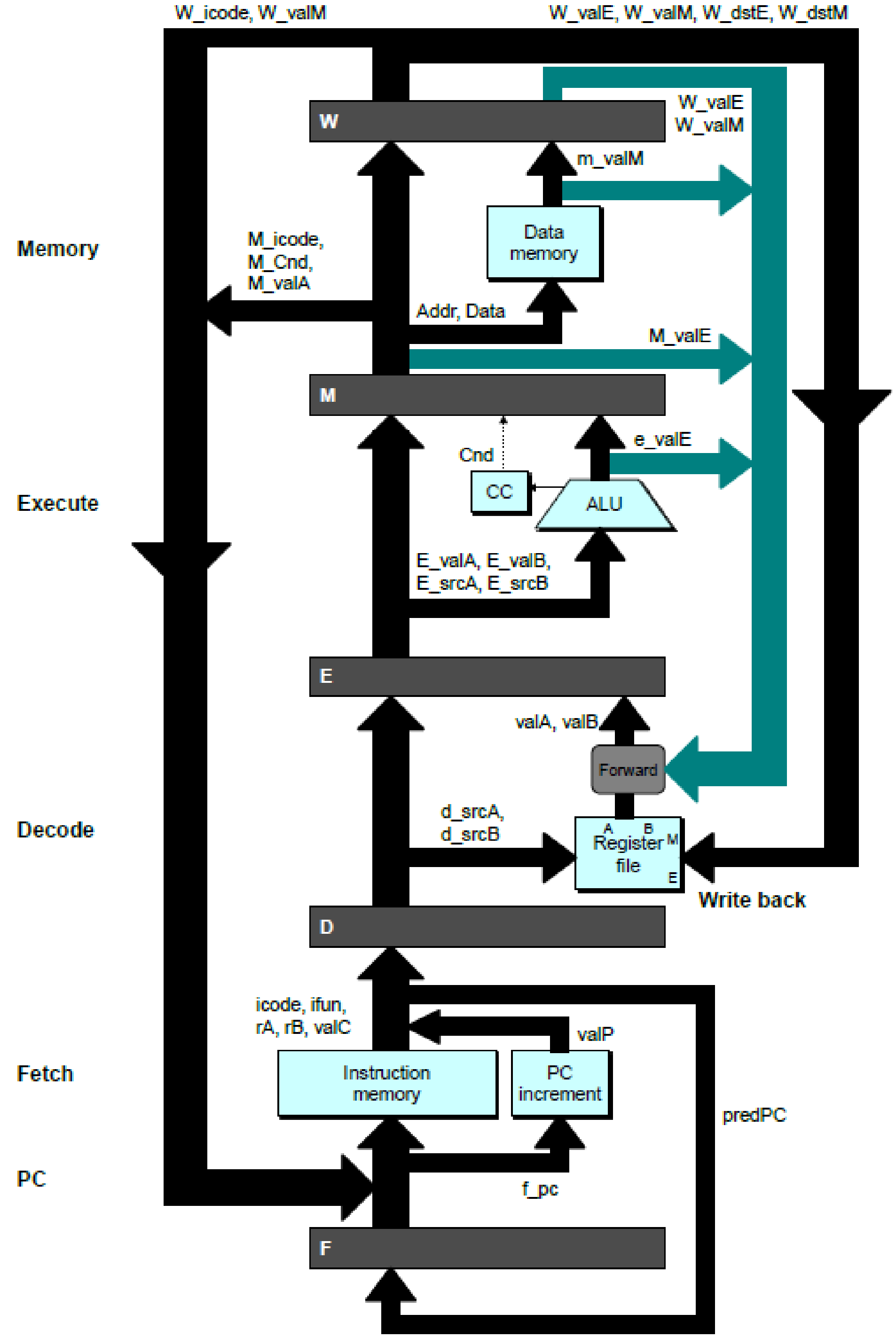
Bypass Paths

Decode Stage

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

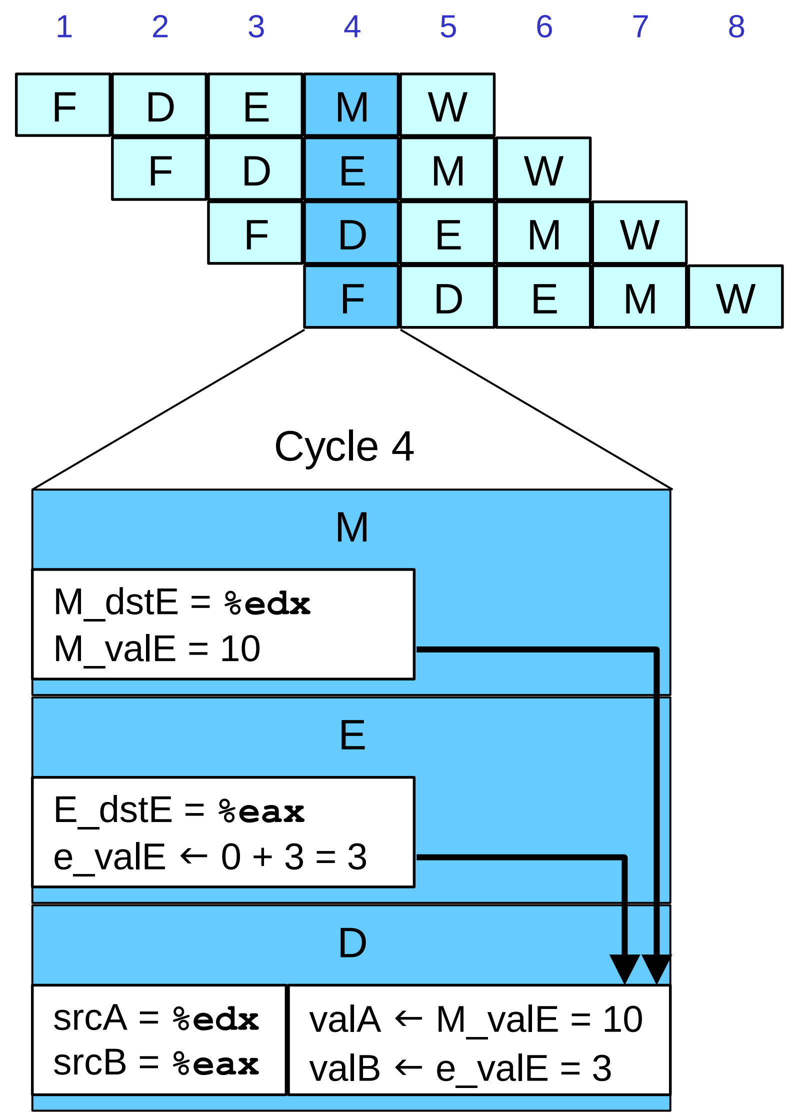
Forwarding Sources

- 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`

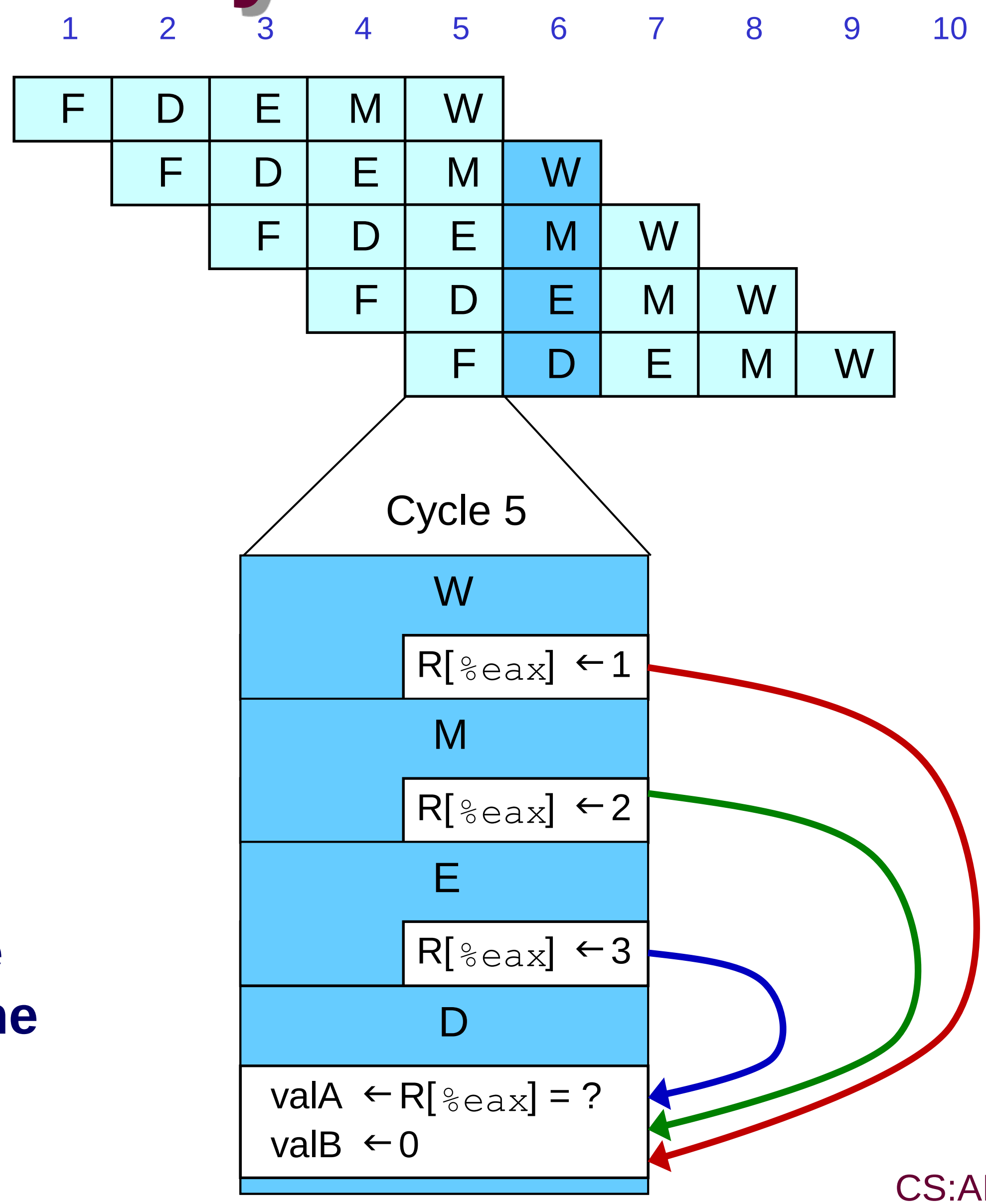
- Generated by ALU during previous cycle
- Forward from memory as valA

Register `%eax`

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

Forwarding Priority

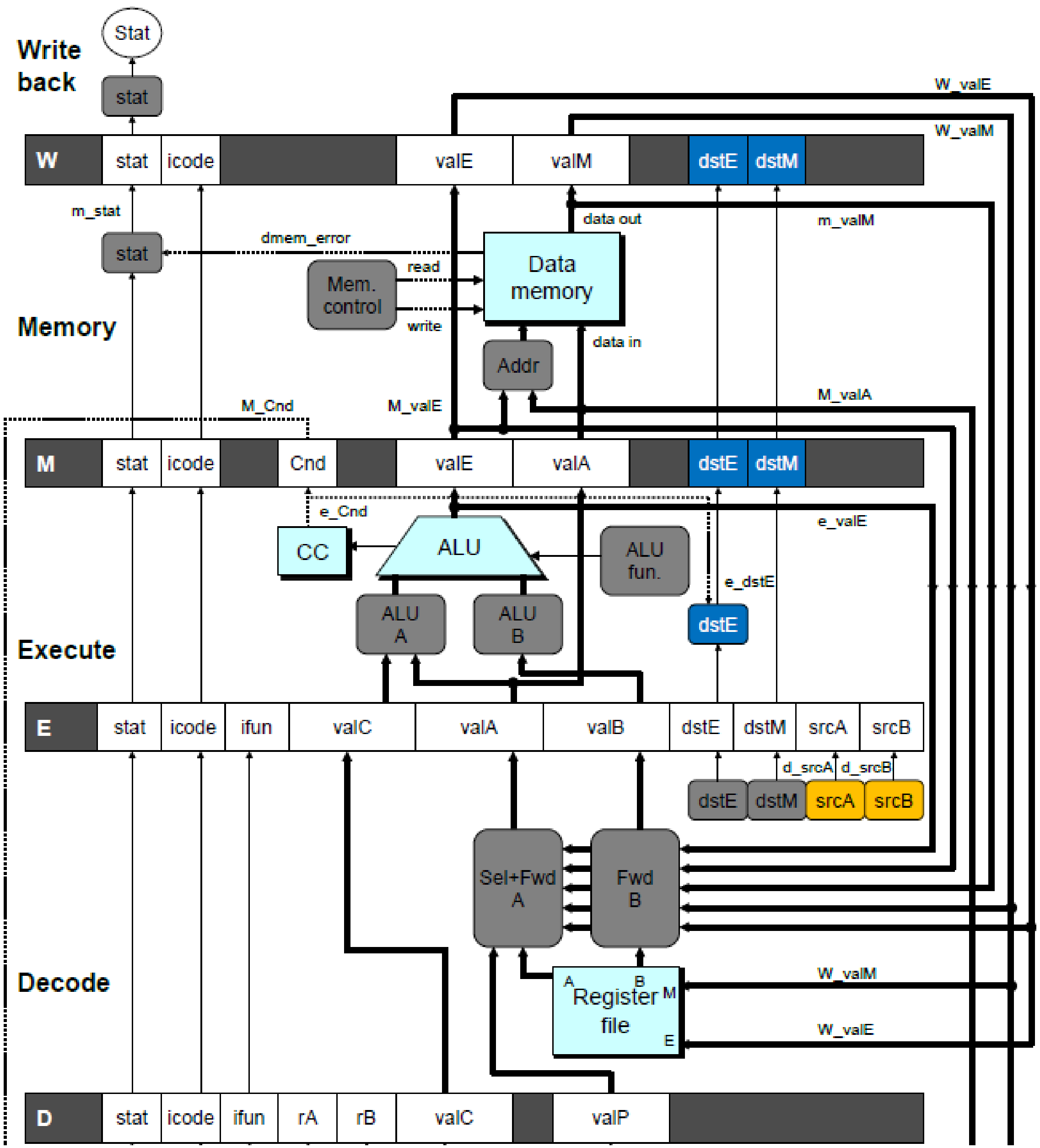
```
0x000: irmovl $1, %eax
0x006: irmovl $2, %eax
0x00c: irmovl $3, %eax
0x012: rrmovl %eax, %edx
0x014: halt
```



Multiple Forwarding Choices

- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage

Implementing Forwarding

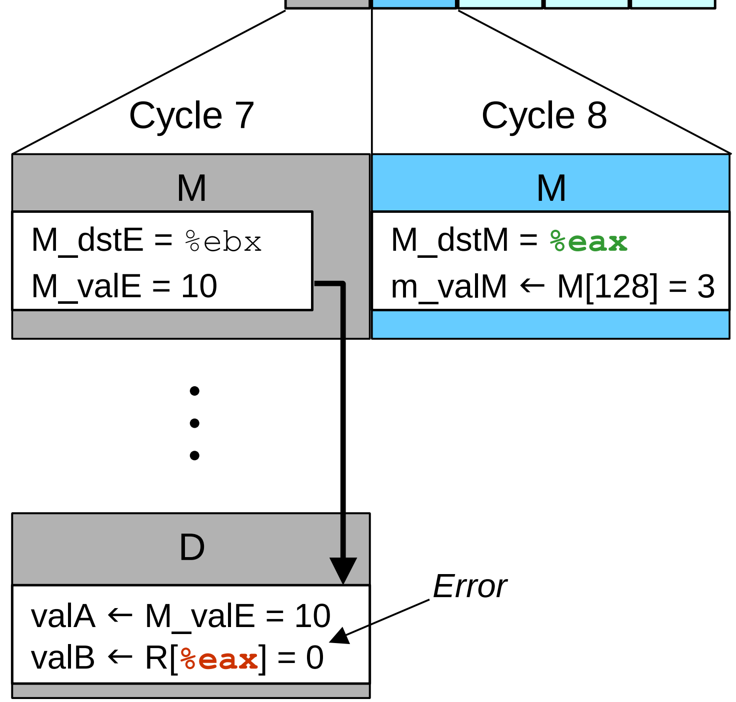


- **Add additional feedback paths from E, M, and W pipeline registers into decode stage**
- **Create logic blocks to select from multiple sources for valA and valB in decode stage**

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



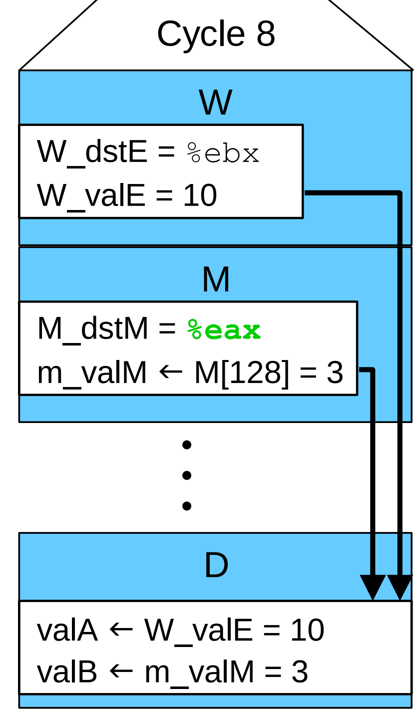
Load-use dependency

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

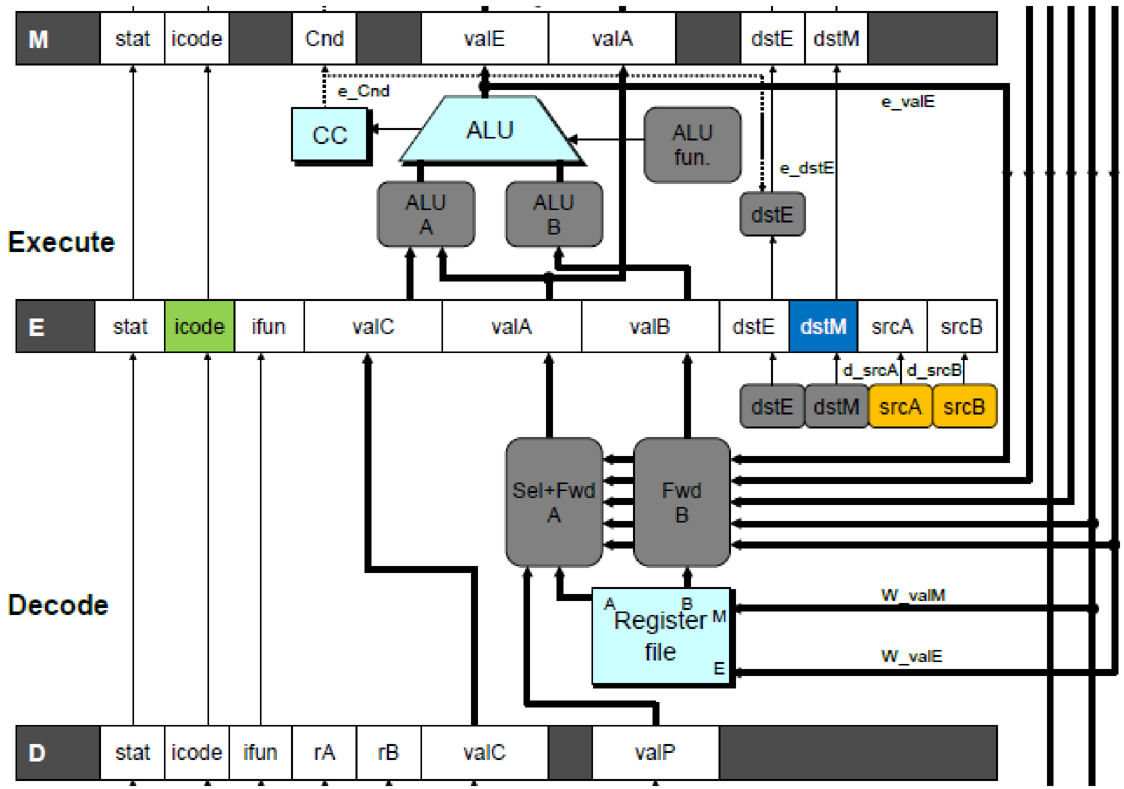
Avoiding Load/Use Hazard

# demo-luh.ys	1	2	3	4	5	6	7	8	9	10	11	12
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			
bubble								E	M	W		
0x01e: addl %ebx,%eax # Use %eax						F	D	D	E	M	W	
0x020: halt							F	F	D	E	M	W

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



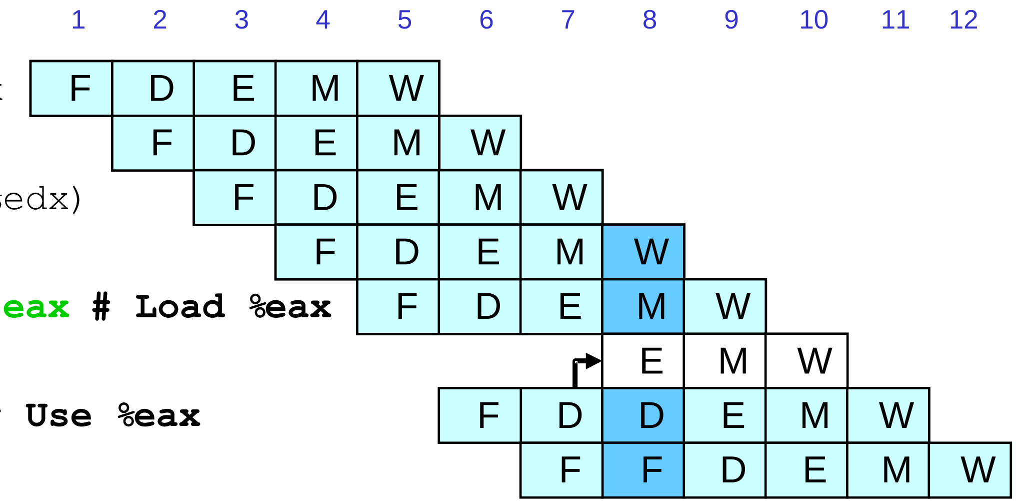
Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	**E_icode in { MRMOVL, POPL } &&** **E_dstM in { d_srcA, d_srcB }**

Control for Load/Use Hazard

```
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 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

Dealing with Dependencies between Instructions

Control Hazards

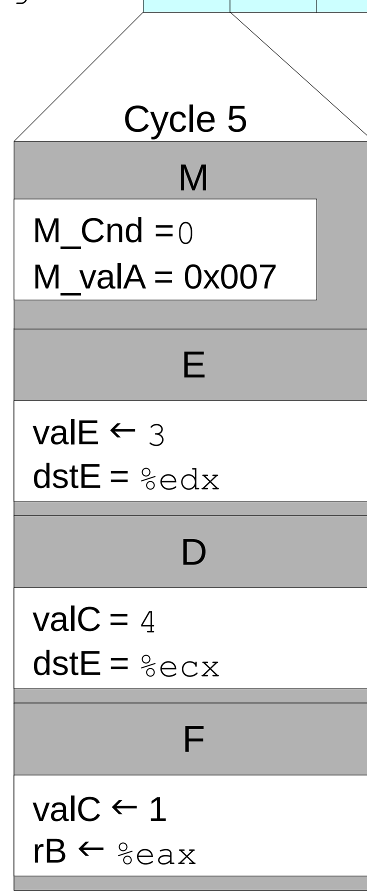
Branch Misprediction Example

```
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 7 instructions

Branch Misprediction Trace

	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

Handling Misprediction

# demo-j.ys	1	2	3	4	5	6	7	8	9	10
`0x000:    xorl %eax,%eax`	F	D	E	M	W					
`0x002:    jne target # Not taken`		F	D	E	M	W				
`0x011: t: irmovl $2,%edx # Target`			F	D						
bubble					E	M	W			
`0x017:    irmovl $3,%ebx # Target+1`				F						
bubble					D	E	M	W		
`0x007:    irmovl $1,%eax # Fall through`					F	D	E	M	W	
`0x00d:    nop`						F	D	E	M	W

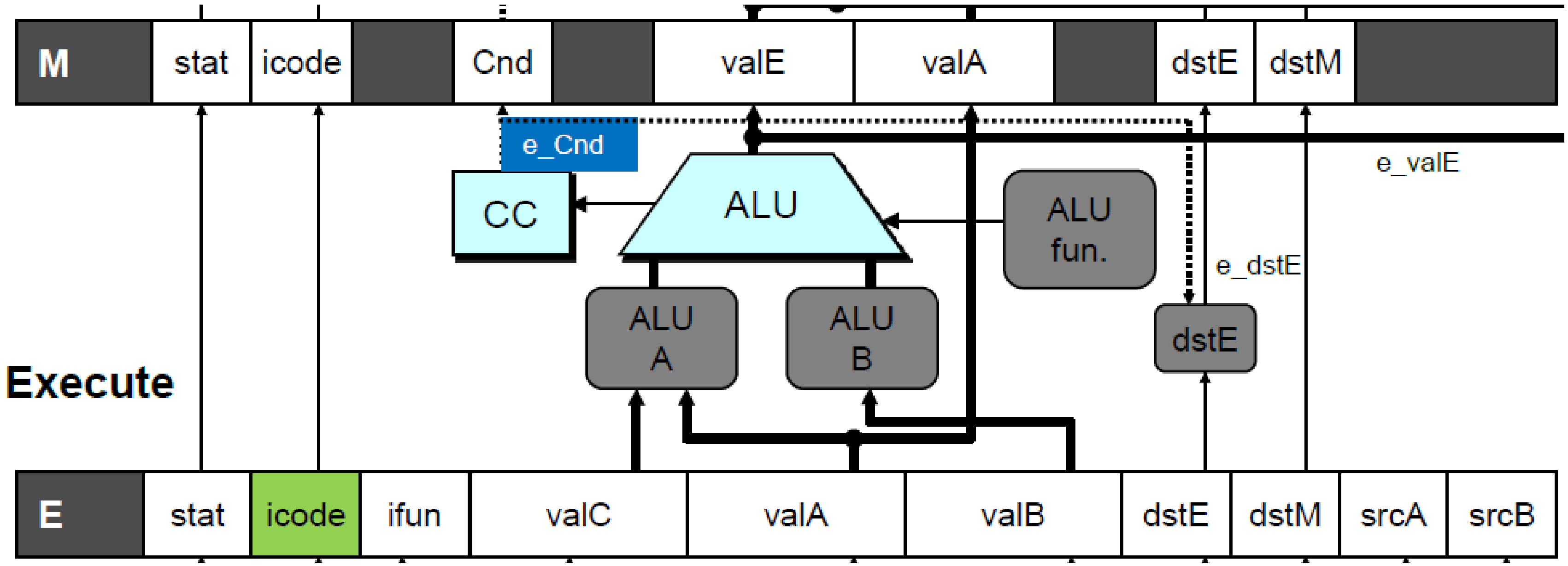
Predict branch as taken

- 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 == JXX & !e_Cnd

Control for Misprediction

# demo-j.ys	1	2	3	4	5	6	7	8	9	10
`0x000:     xorl %eax,%eax`	F	D	E	M	W					
`0x002:     jne target # Not taken`		F	D	E	M	W				
`0x011: t: irmovl $2,%edx # Target`			F	D						
bubble					E	M	W			
`0x017:     irmovl $3,%ebx # Target+1`				F						
bubble					D	E	M	W		
`0x007:     irmovl $1,%eax # Fall through`					F	D	E	M	W	
`0x00d:     nop`						F	D	E	M	W

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

Return Example

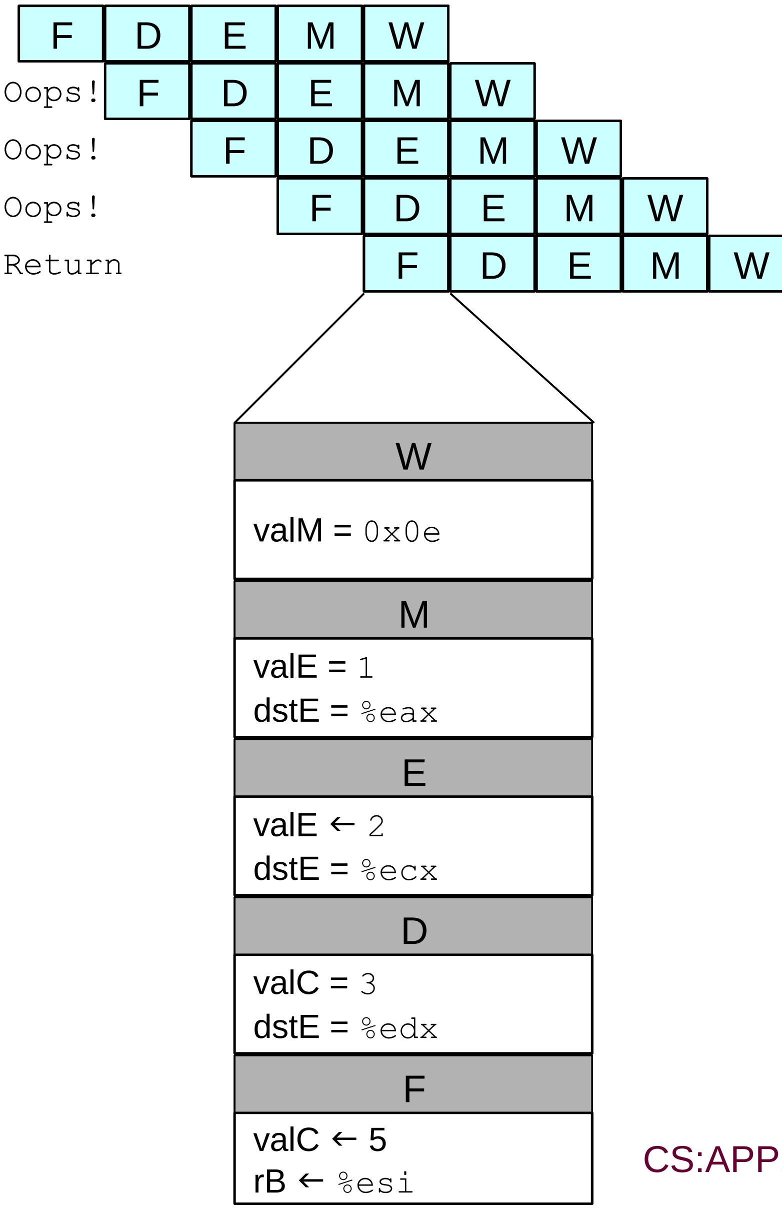
`demo-retb.ys`

```
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

Incorrect Return Example

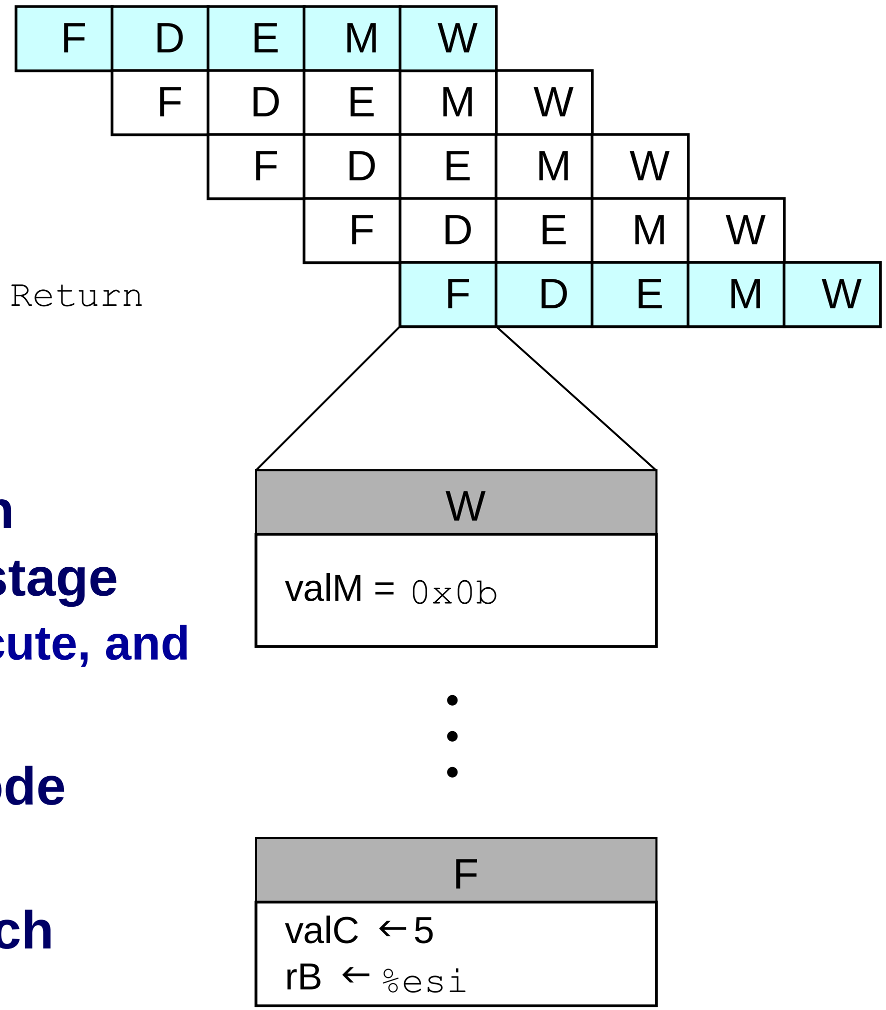
```
# demo-ret

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`**

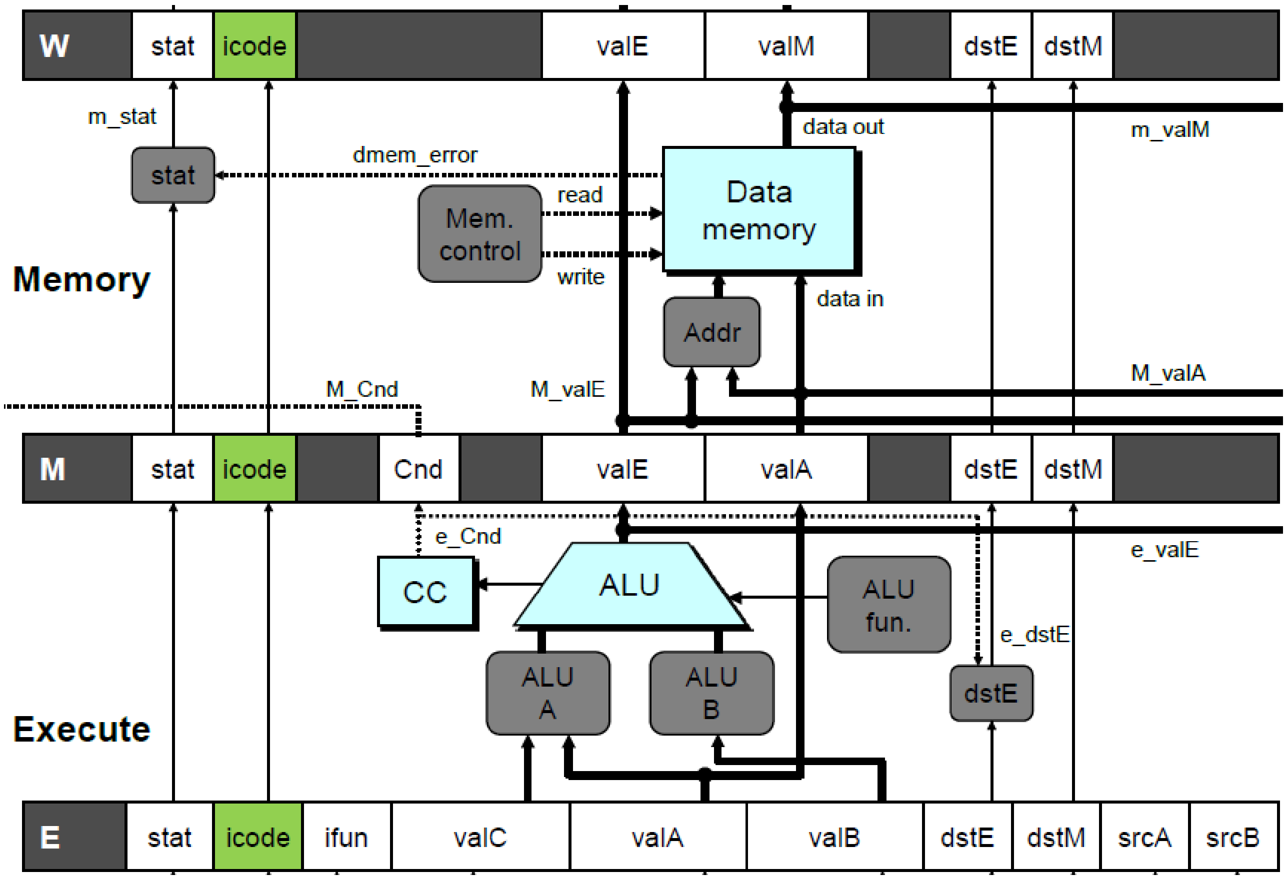
Correct Return Example

```
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	F	D	E	M	W				
	bubble		F	D	E	M	W			
	bubble			F	D	E	M	W		
	bubble				F	D	E	M	W	
0x00b:	irmovl $5,%esi # Return					F	D	E	M	W

Condition	F	D	E	M	W
Processing ret	**stall**	**bubble**	**normal**	**normal**	**normal**

Special Control Cases

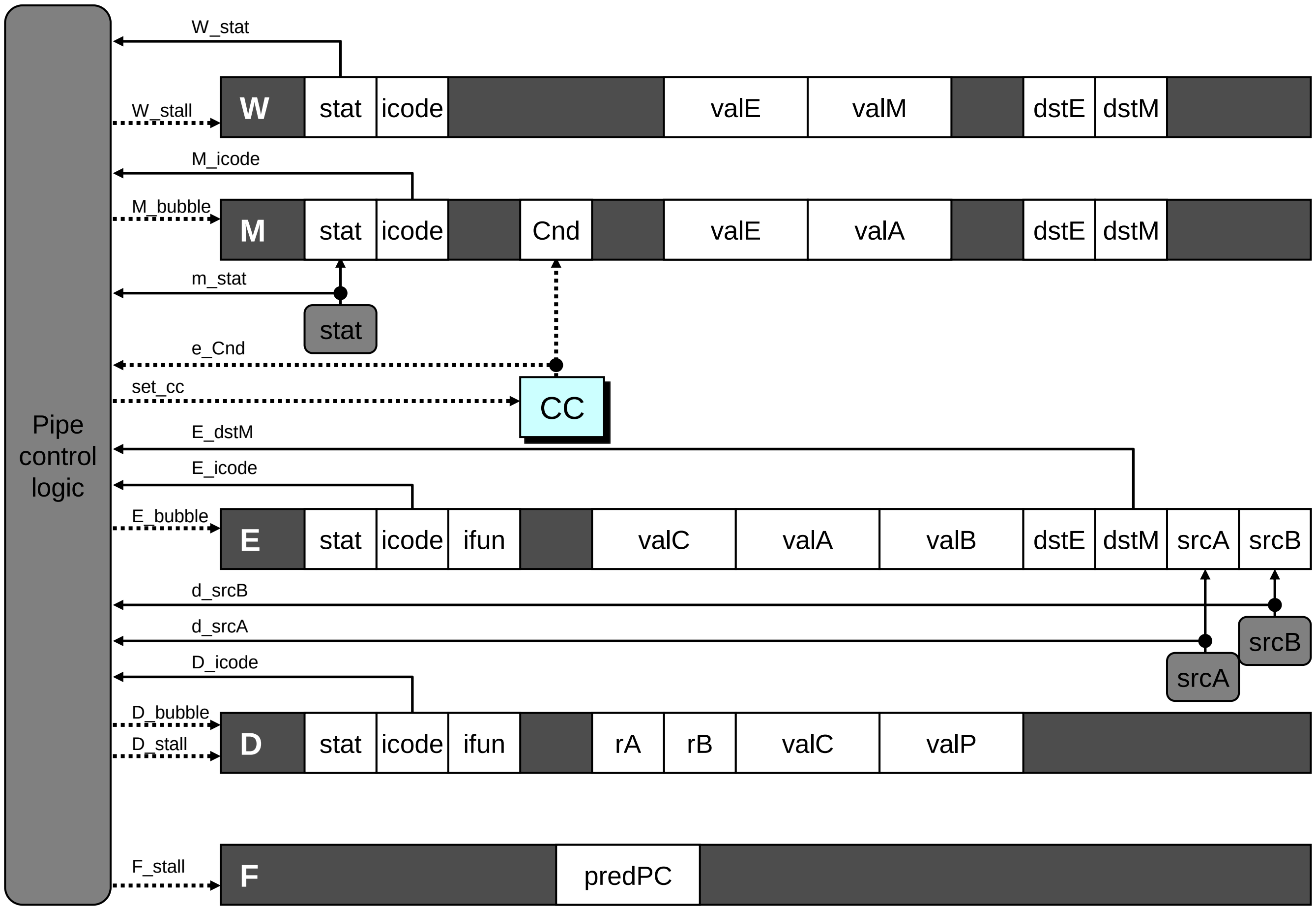
Detection

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_Cnd

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

Pipeline Control Logic

- A sequence of control instructions complicates the control logic
  - in particular, should stall in Decode stage (instead of bubble, as an initial inspection suggests)
- Load/use hazard should get priority
- `ret` instruction should be held in decode stage for additional cycle

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*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

- Read-after-write dependencies 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