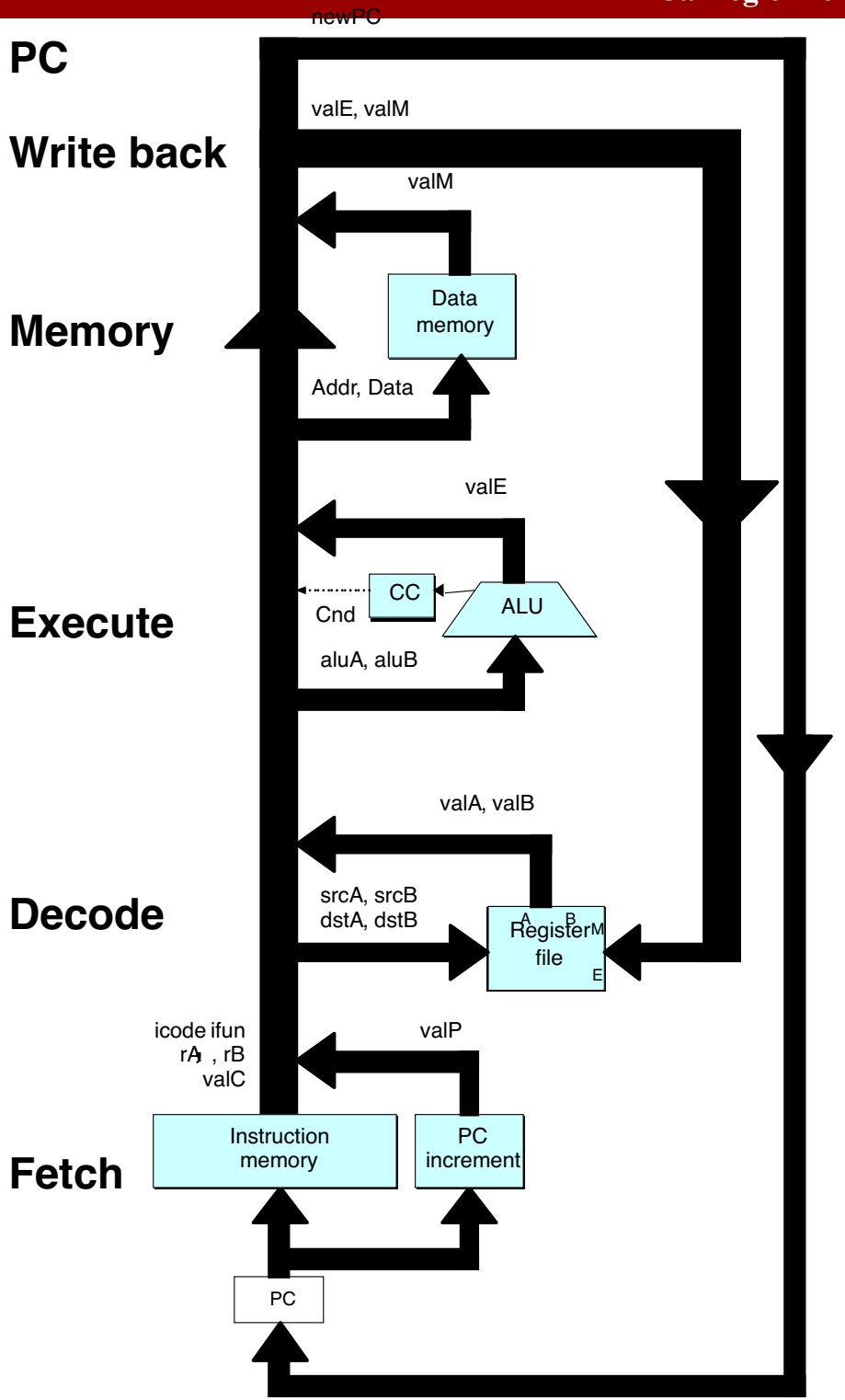
Processor Architecture: Pipelined Implementation

SEQ Stages

- **Fetch**
  - Read instruction from instruction memory
- **Decode**
  - Read program registers
- **Execute**
  - Compute value or address
- **Memory**
  - Read or write data
- **Write Back**
  - Write program registers
- **PC**
  - Update program counter



Overview

- **General Principles of Pipelining**
  - Goal
  - Difficulties
- **Creating a Pipelined Y86-64 Processor**
  - Rearranging SEQ
  - Inserting pipeline registers
  - Problems with data and control hazards

Real-World Pipelines: Car Washes

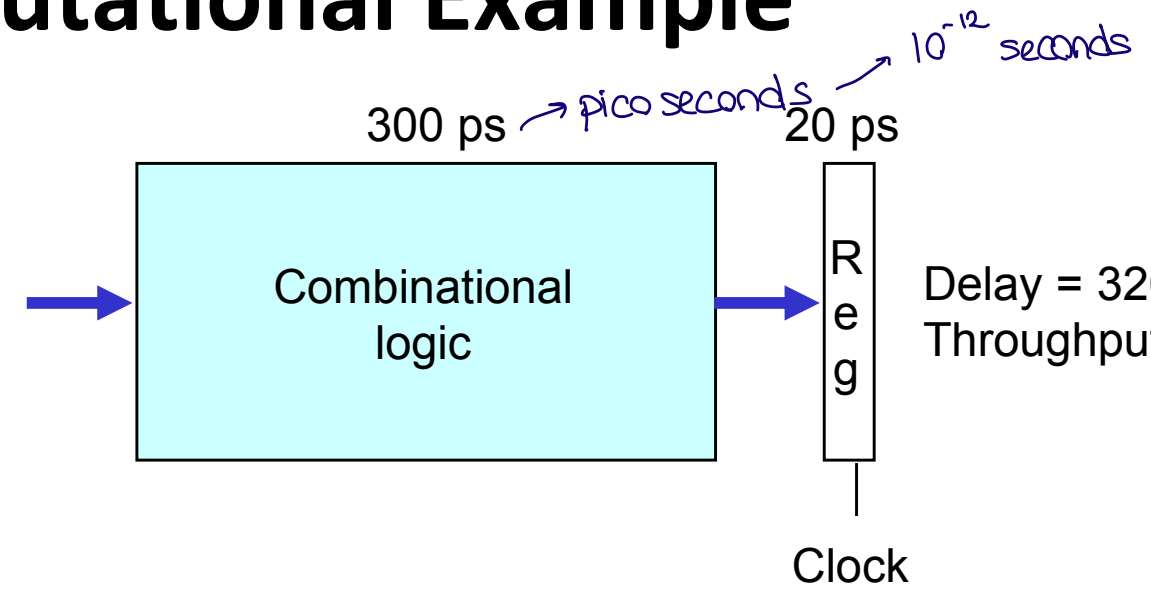
Sequential

Parallel

Pipelined

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

Computational Example



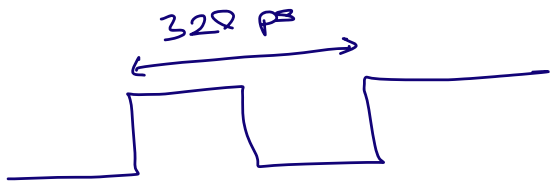
- **System**
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

time per instance = 320×10^{-12}

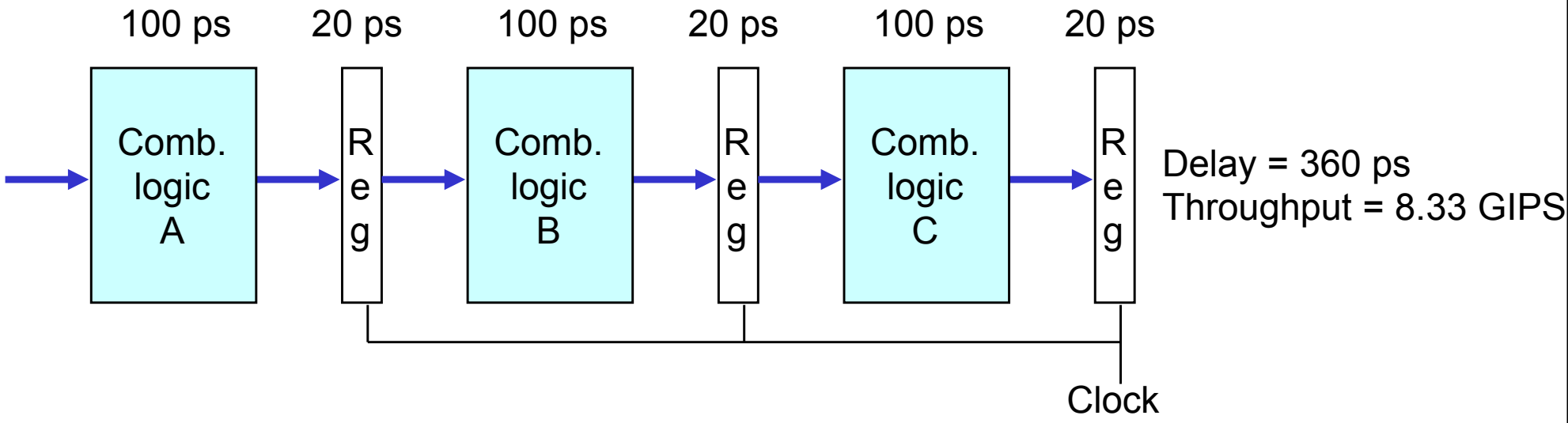
instruction per time/s $\frac{1}{320 \times 10^{-12}}$

$= 3.12 \times 10^{9}$ IPS

$= 3.12$ GIPS



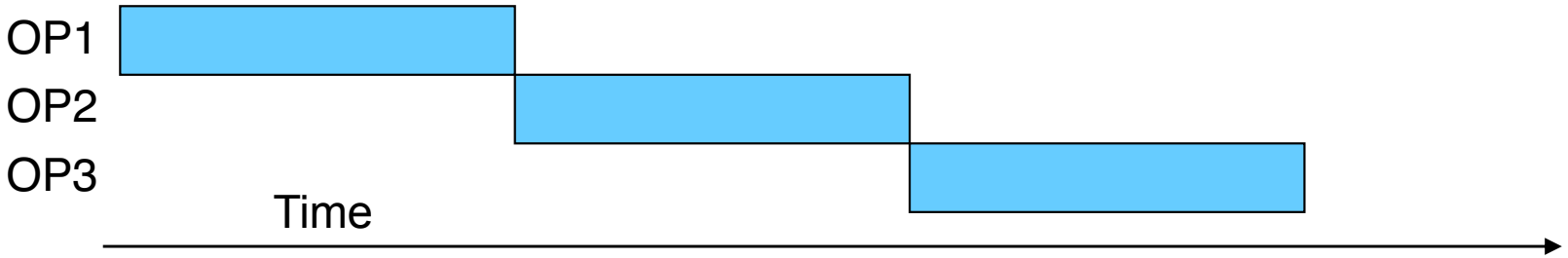
3-Way Pipelined Version



- **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

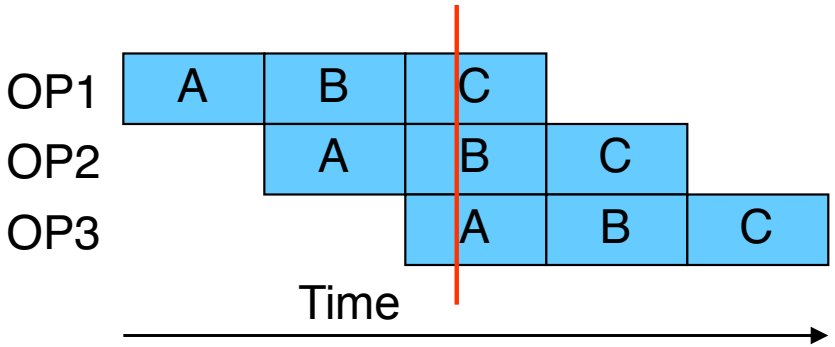
Pipeline Diagrams

- **Unpipelined**



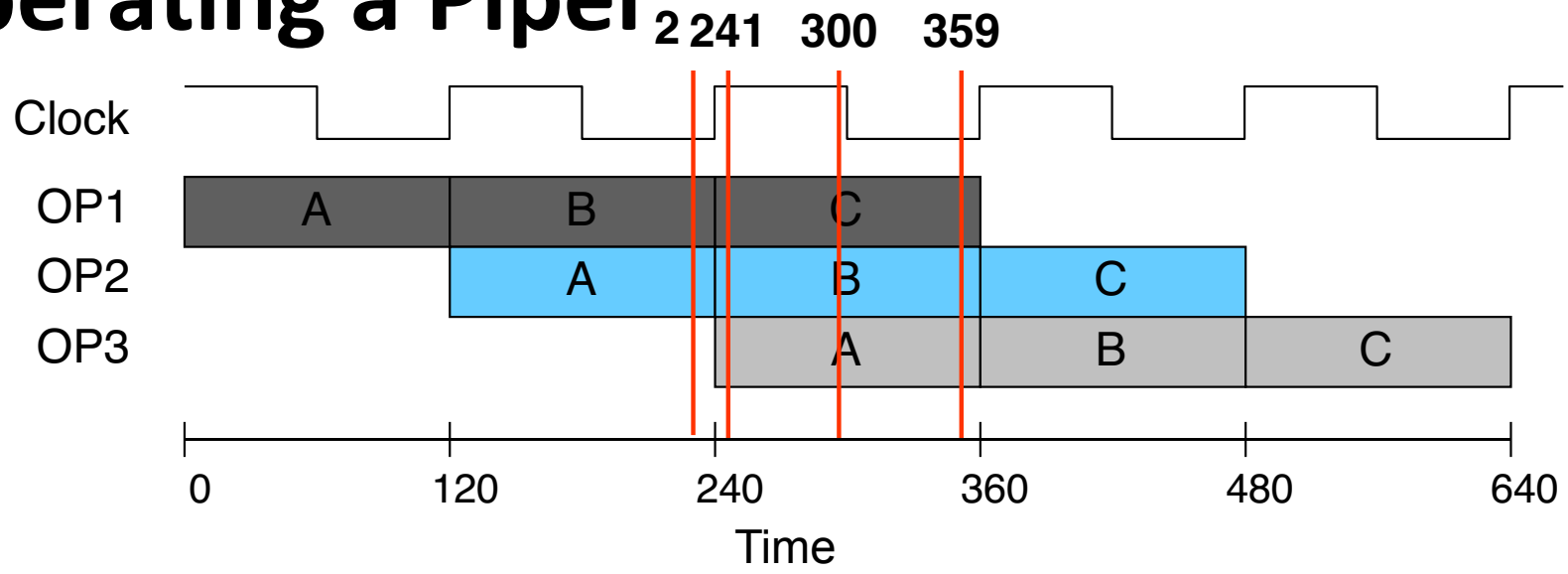
  - Cannot start new operation until previous one completes

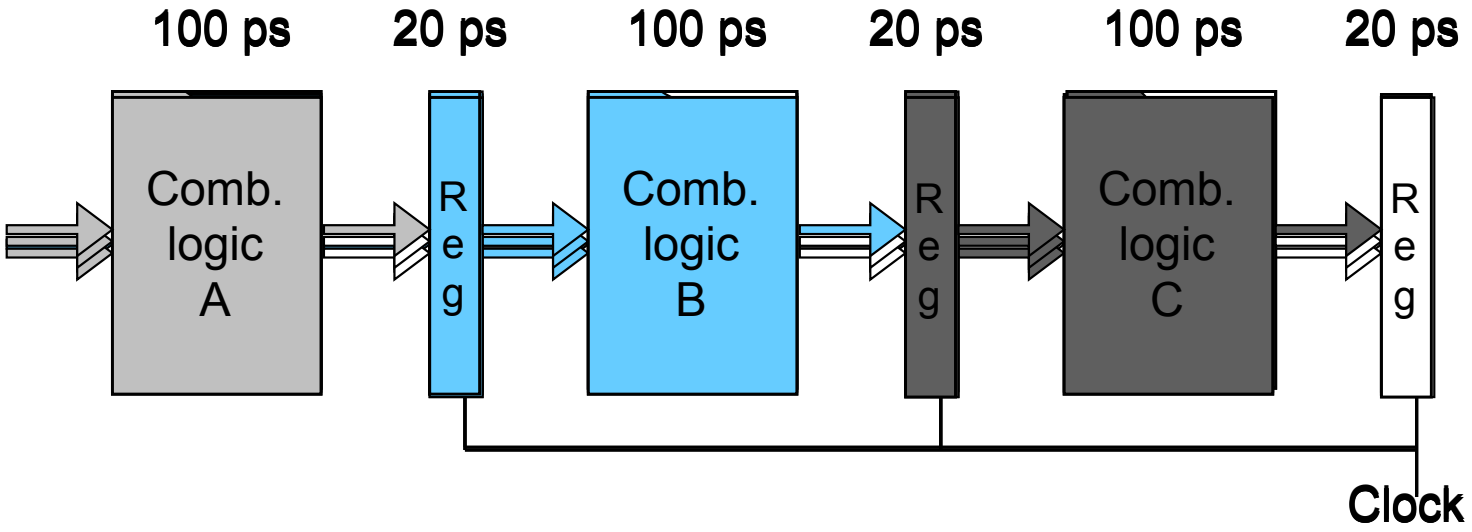
- **3-Way Pipelined**



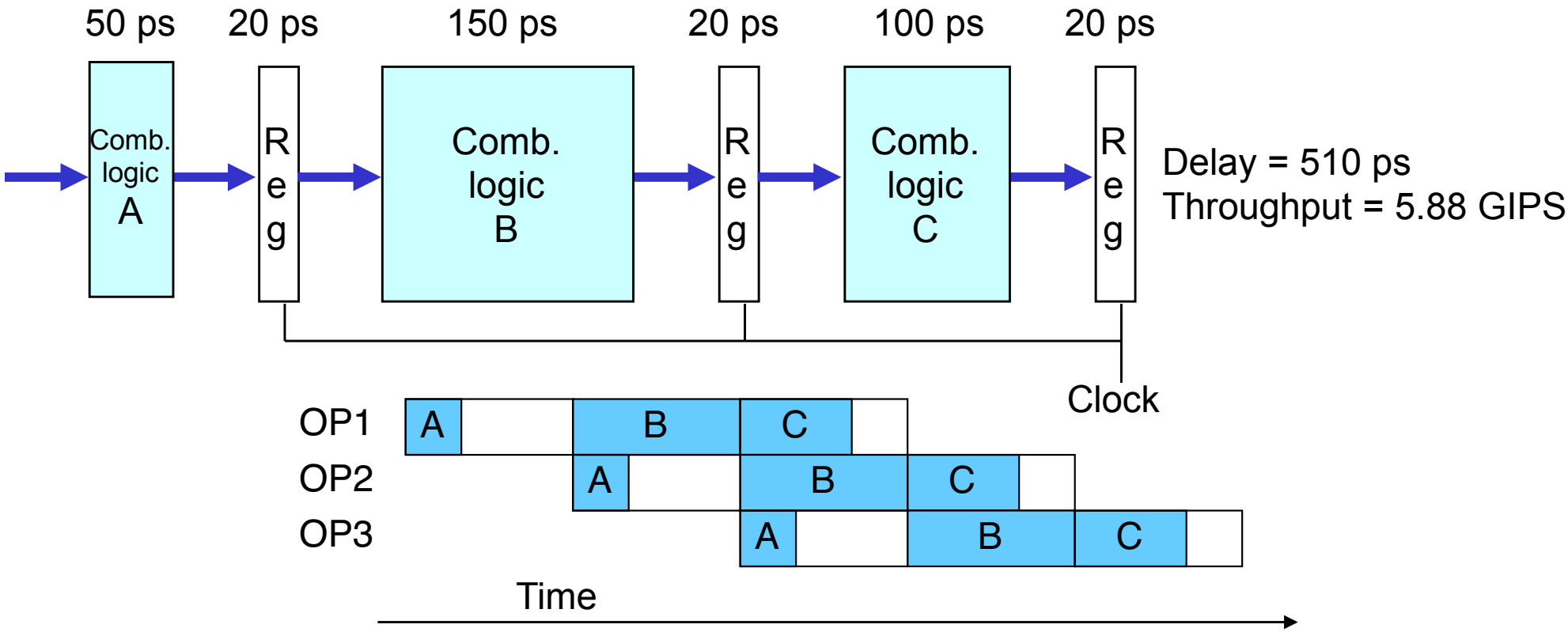
  - Up to 3 operations in process simultaneously

Operating a Pipeline



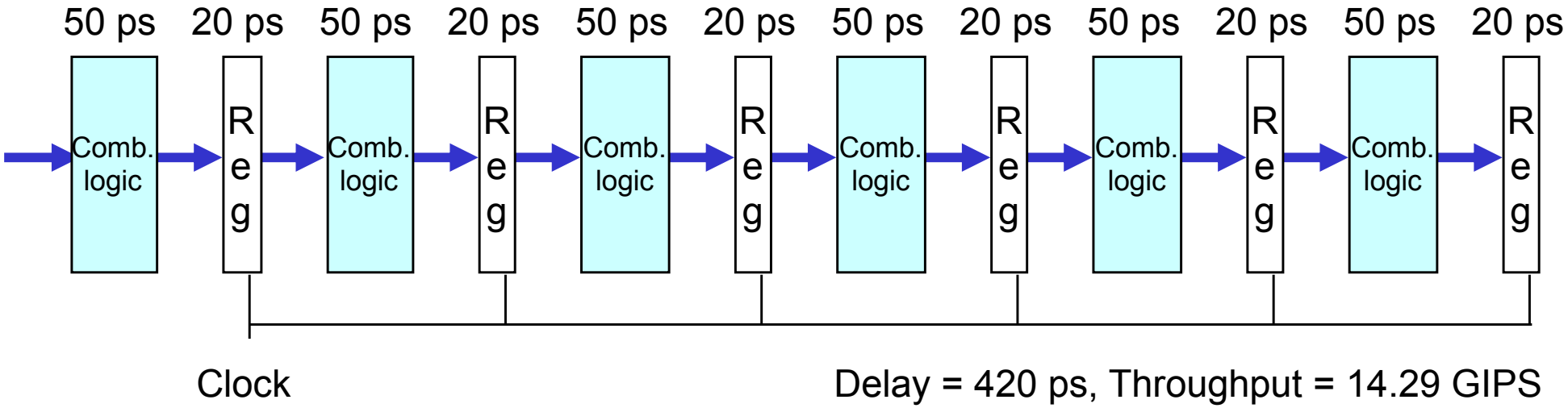


Limitations: Nonuniform Delays



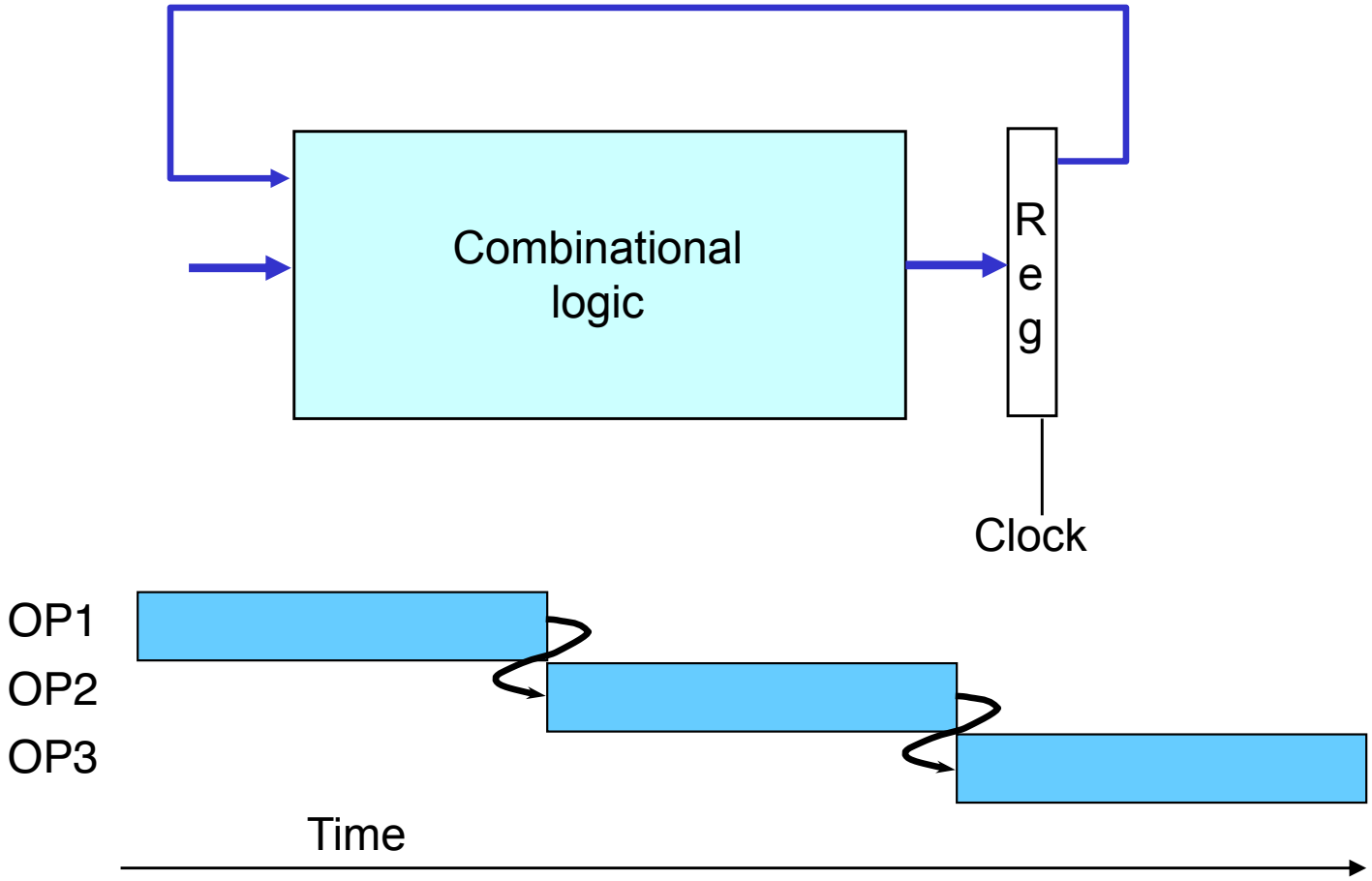
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



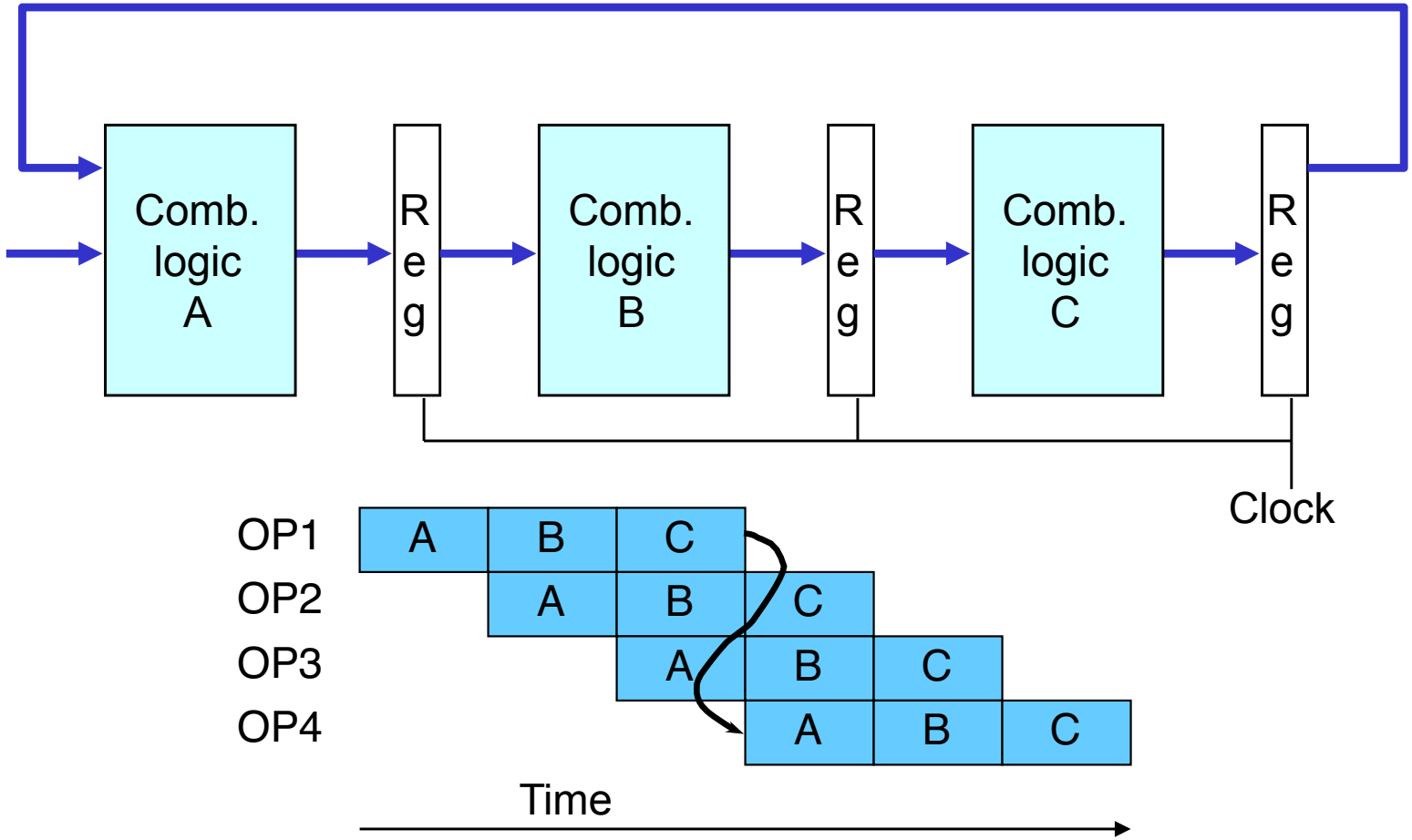
- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Data Dependencies



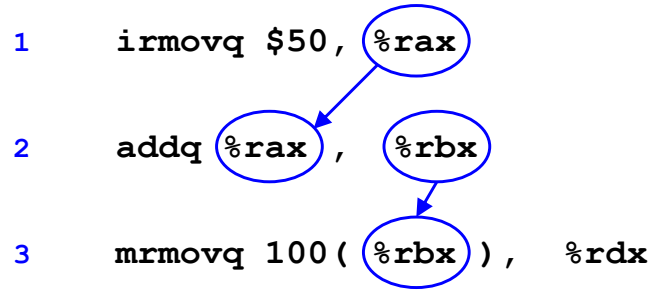
- **System**
  - Each operation depends on result from preceding one

Data Hazards



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

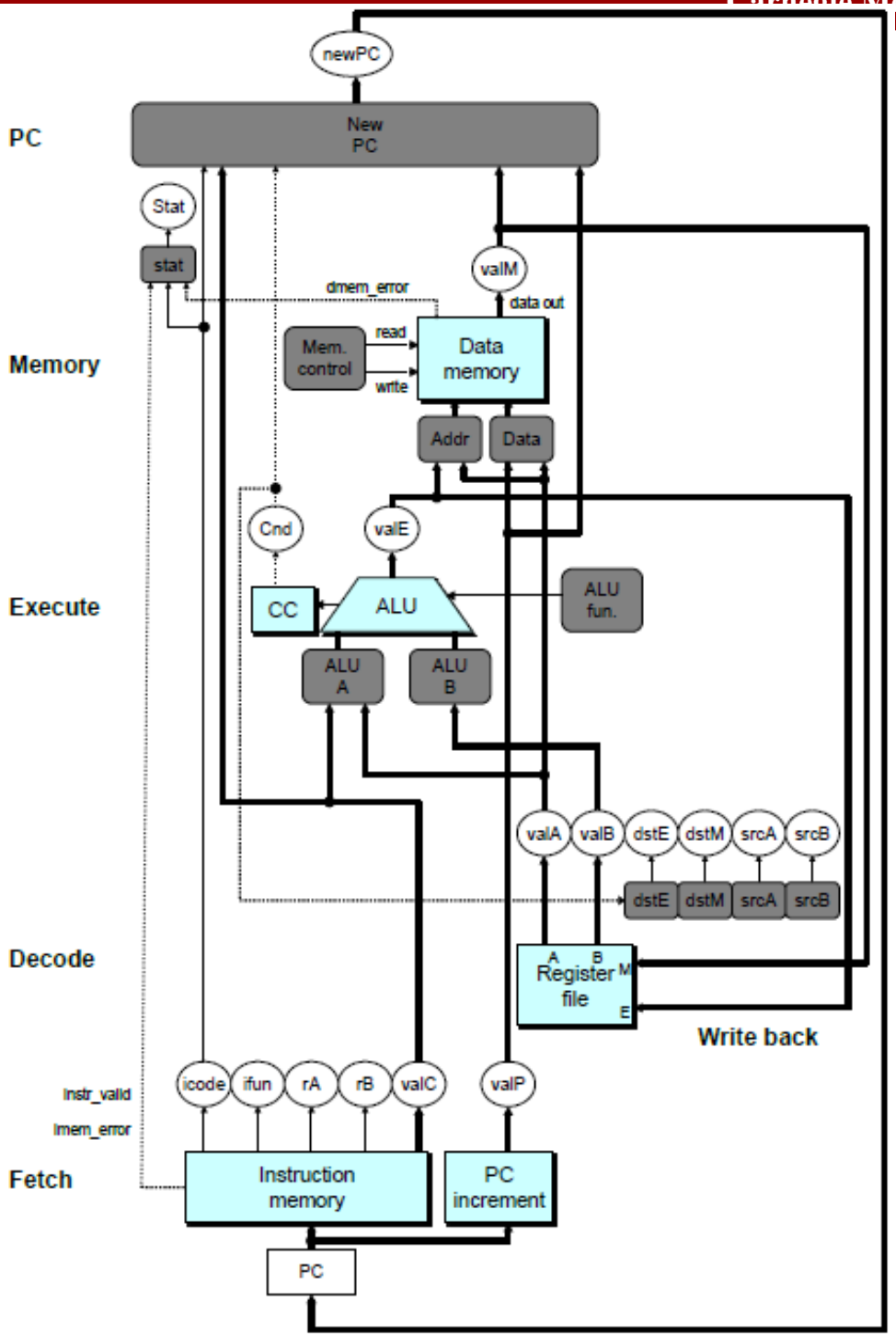
Data Dependencies in Processors



- 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

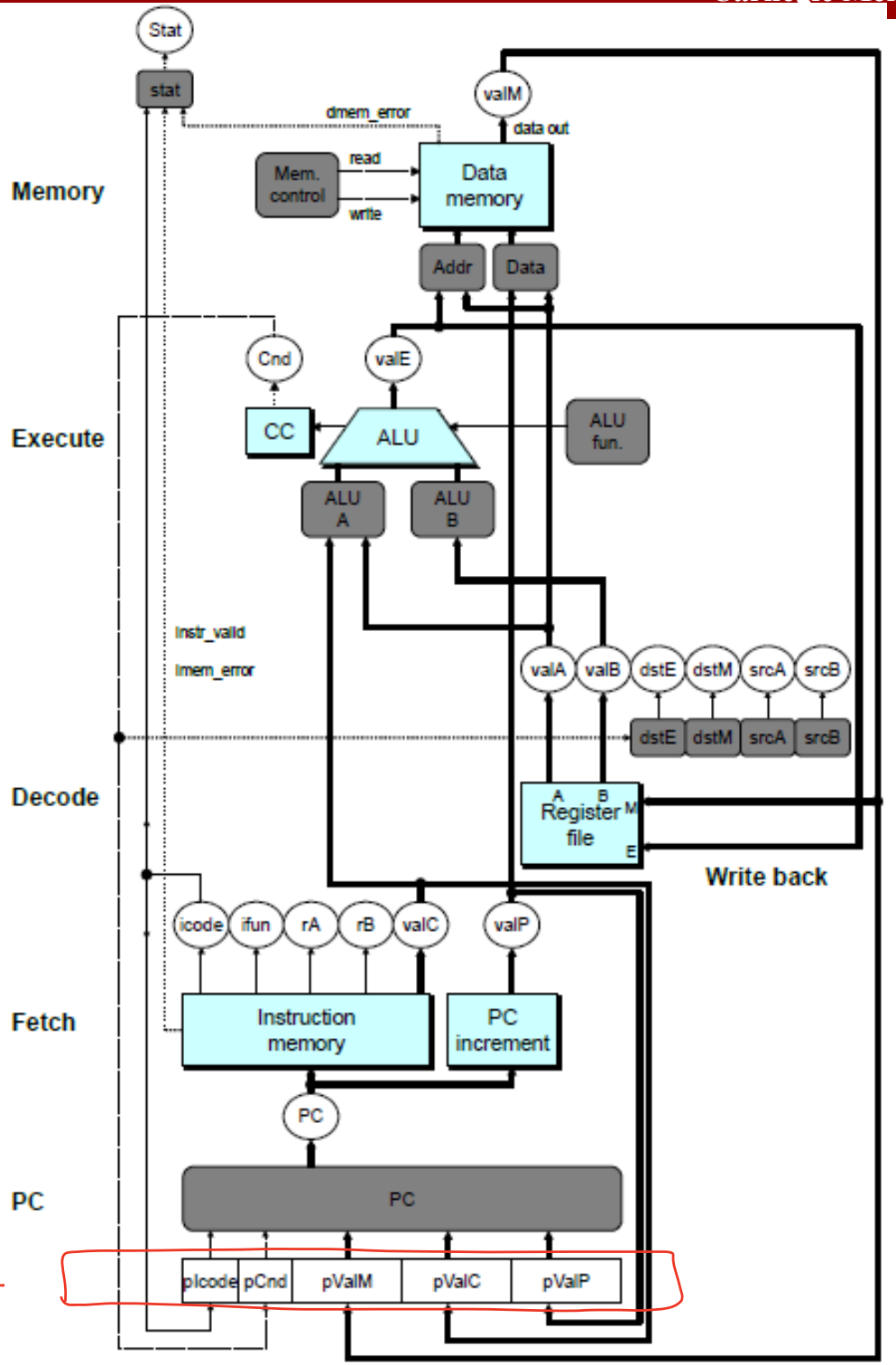
- Stages occur in sequence
- One operation in process at a time



SEQ+ Hardware

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

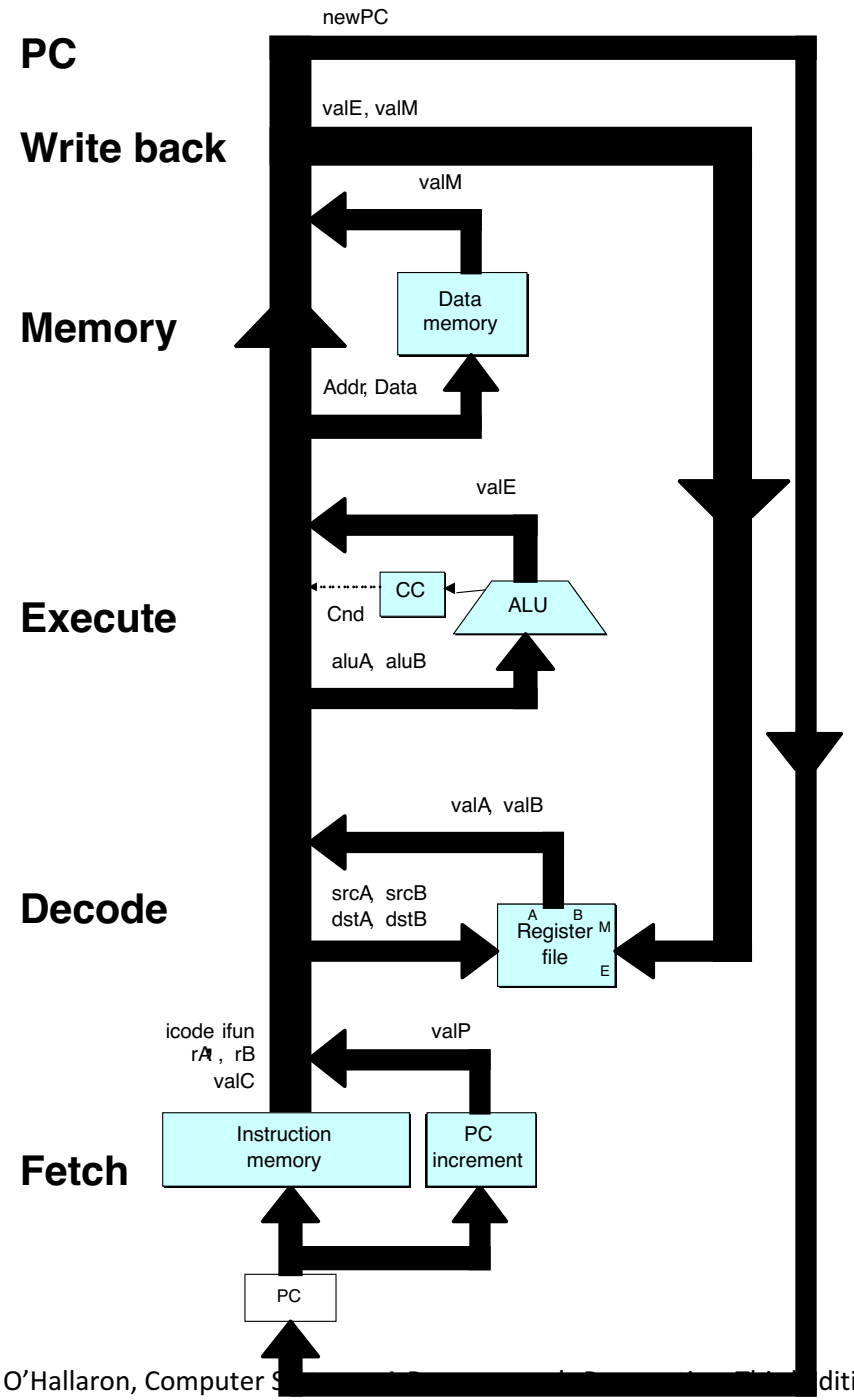
- **PC Stage**
  - Task is to select PC for current instruction
  - Based on results computed by previous instruction
- **Processor State**
  - PC is no longer stored in register
  - But, can determine PC based on other stored information

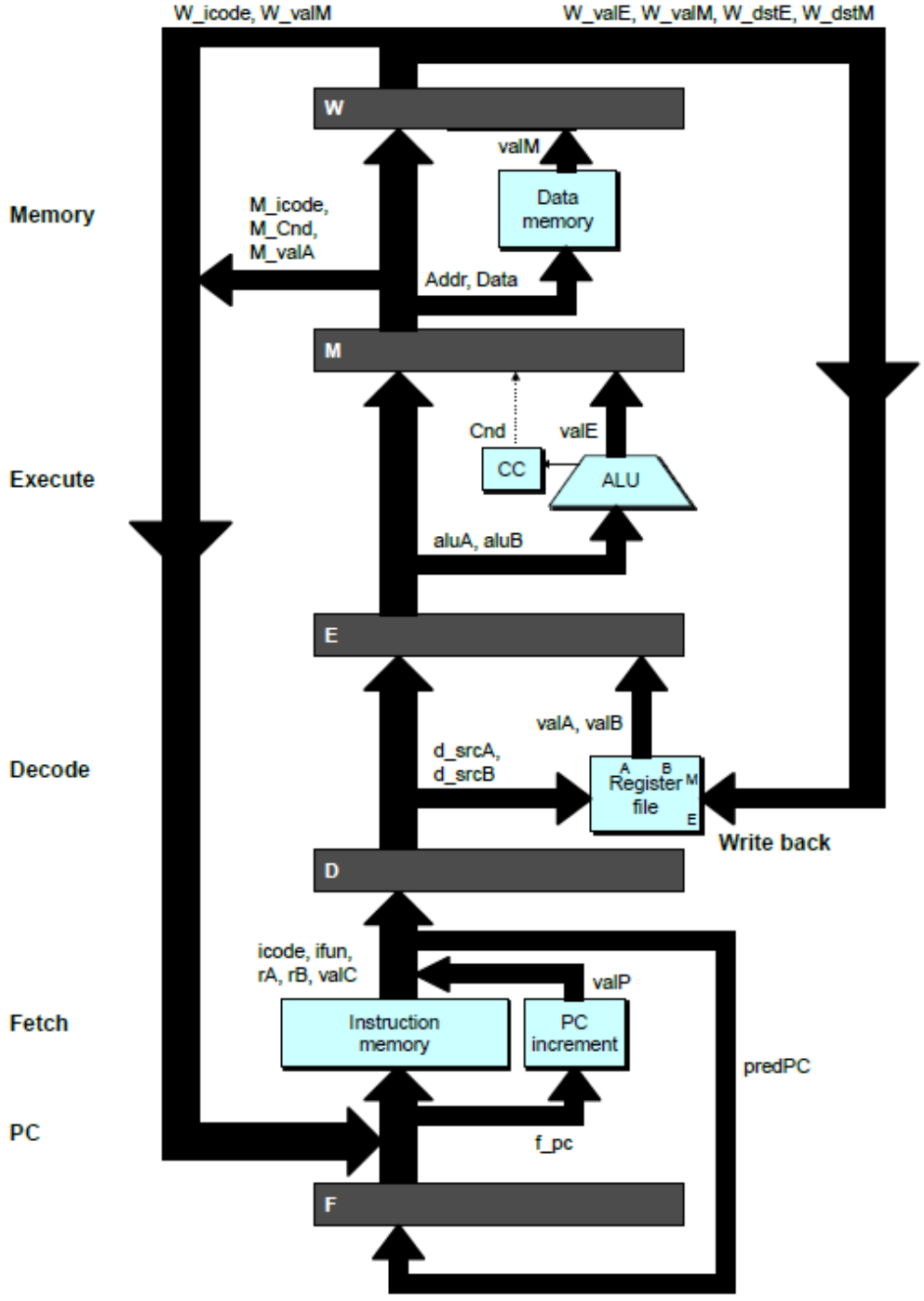


There is no PC register here

computed from the prev. instr.

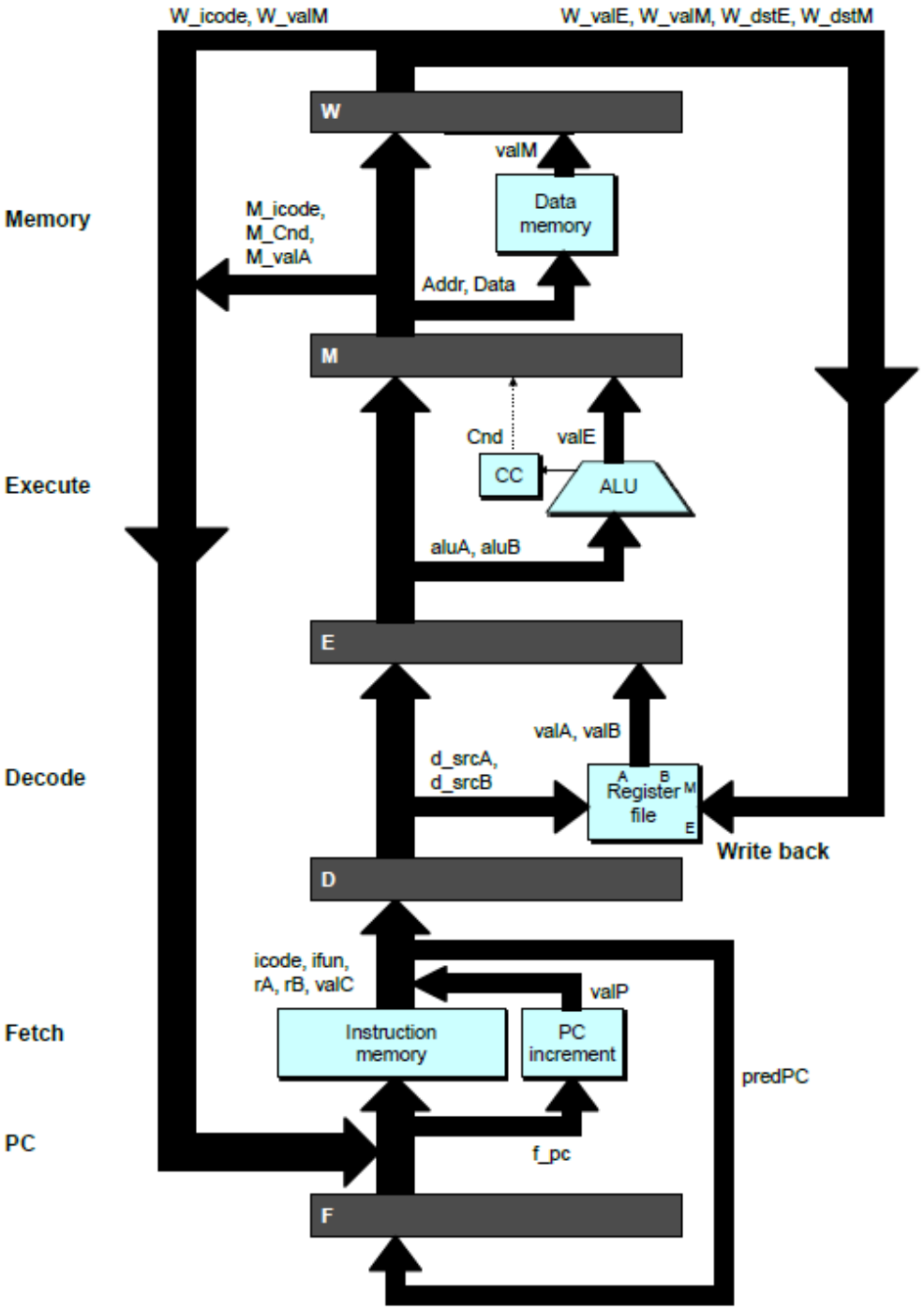
Adding Pipeline Registers





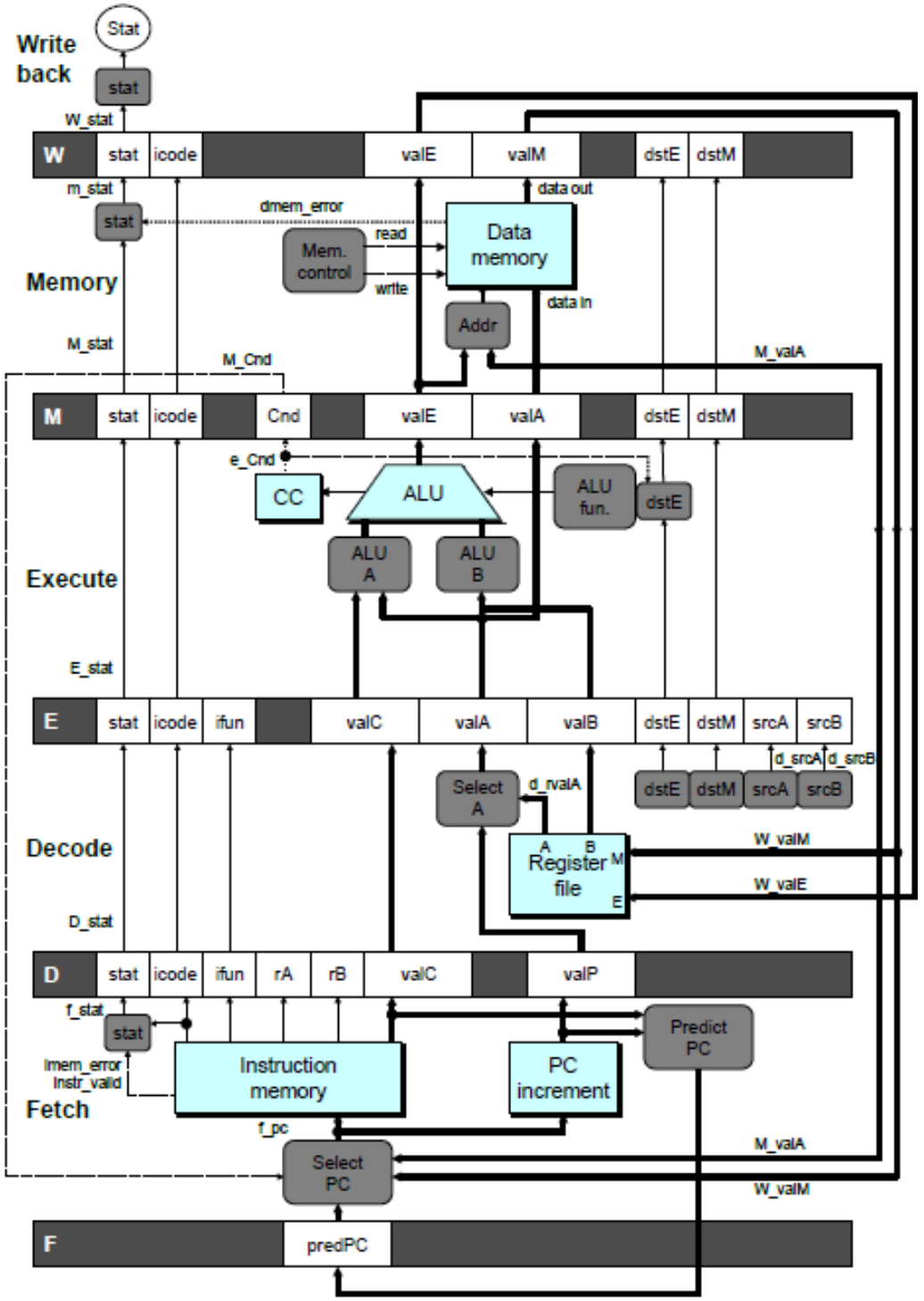
Pipeline Stages

- **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



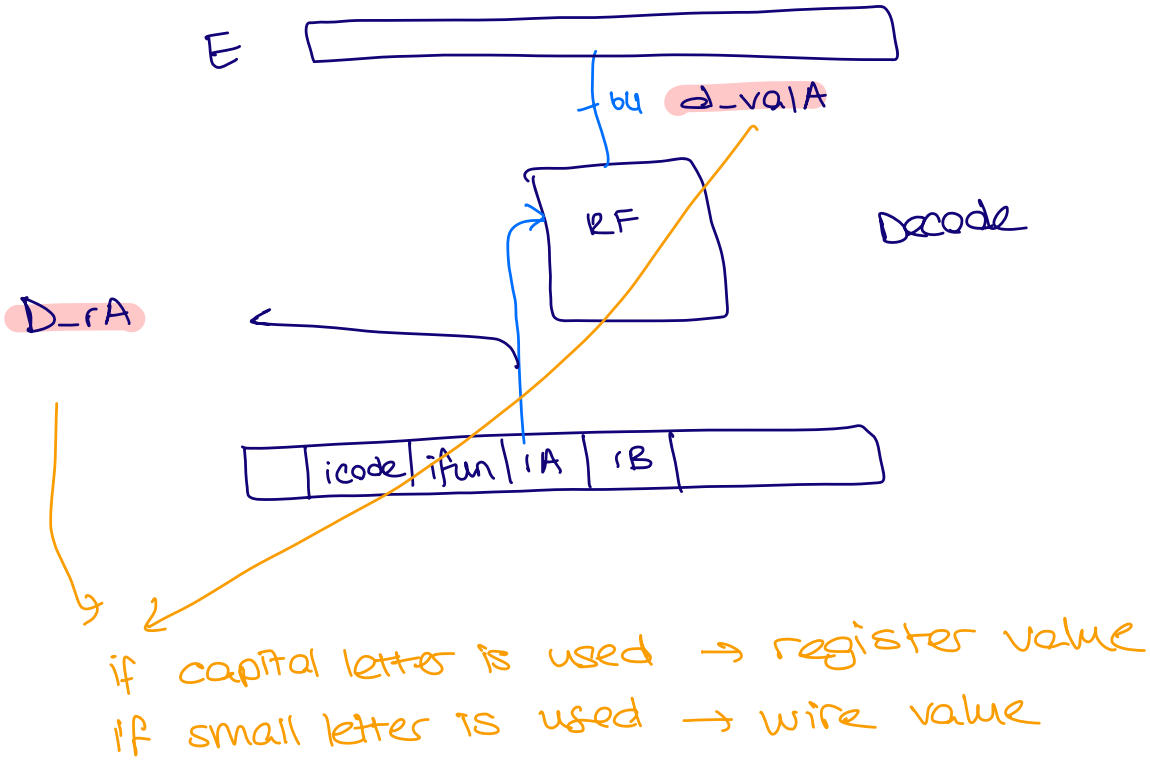
PIPE- Hardware

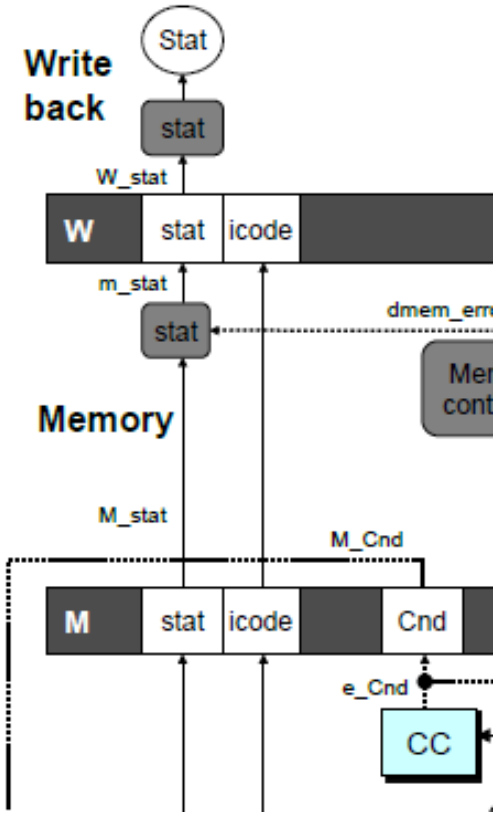
- Pipeline registers hold intermediate values from instruction execution
- **Forward (Upward) Paths**
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., valC passes through decode



Signal Naming Conventions

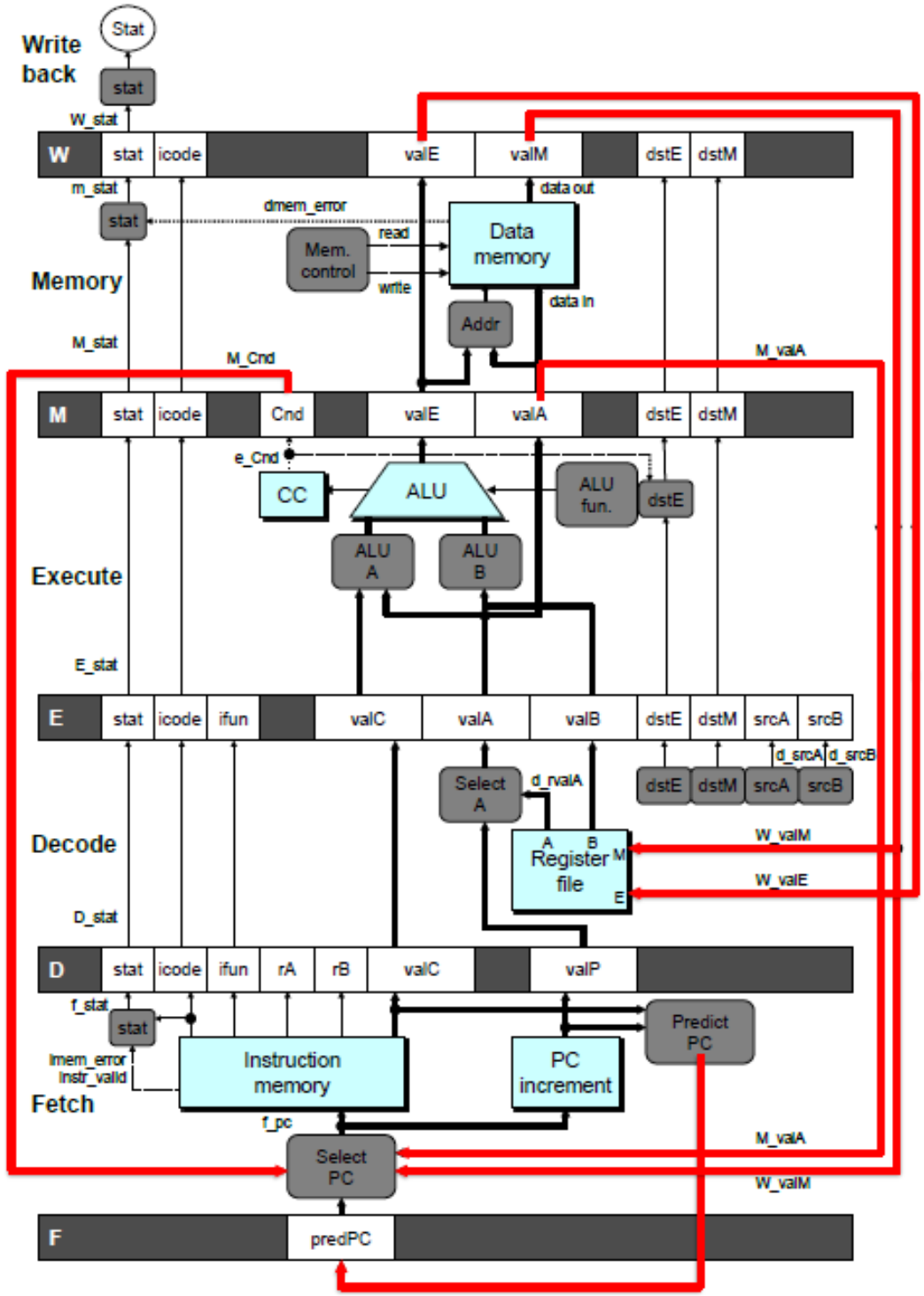
- **S_Field**
  - Value of Field held in stage S pipeline register
- **s_Field**
  - Value of Field computed in stage S





Feedback Paths

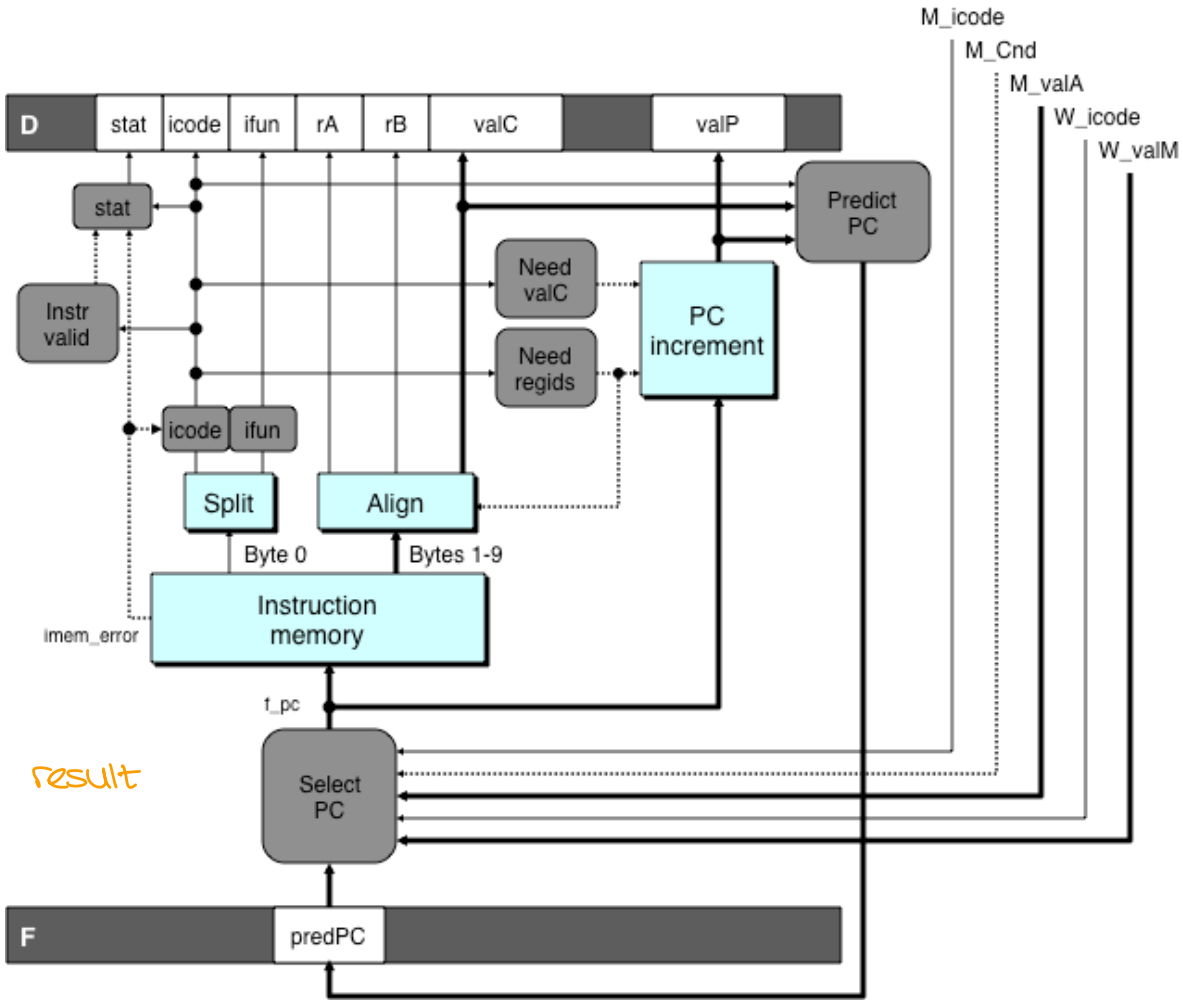
- **Predicted PC**
  - Guess value of next PC
- **Branch information**
  - Jump taken/not-taken
  - Fall-through or target address
- **Return point**
  - Read from memory
- **Register updates**
  - To register file write ports



Predicting the PC

PIPE –

↳ Hazard
- control Hazards
  - ↳ j XX
  - ↳ ret
- Data Hazards
  - mrmq 0x5, %rax
  - addq %rax, %rbx
  - ↳ will yield a wrong result

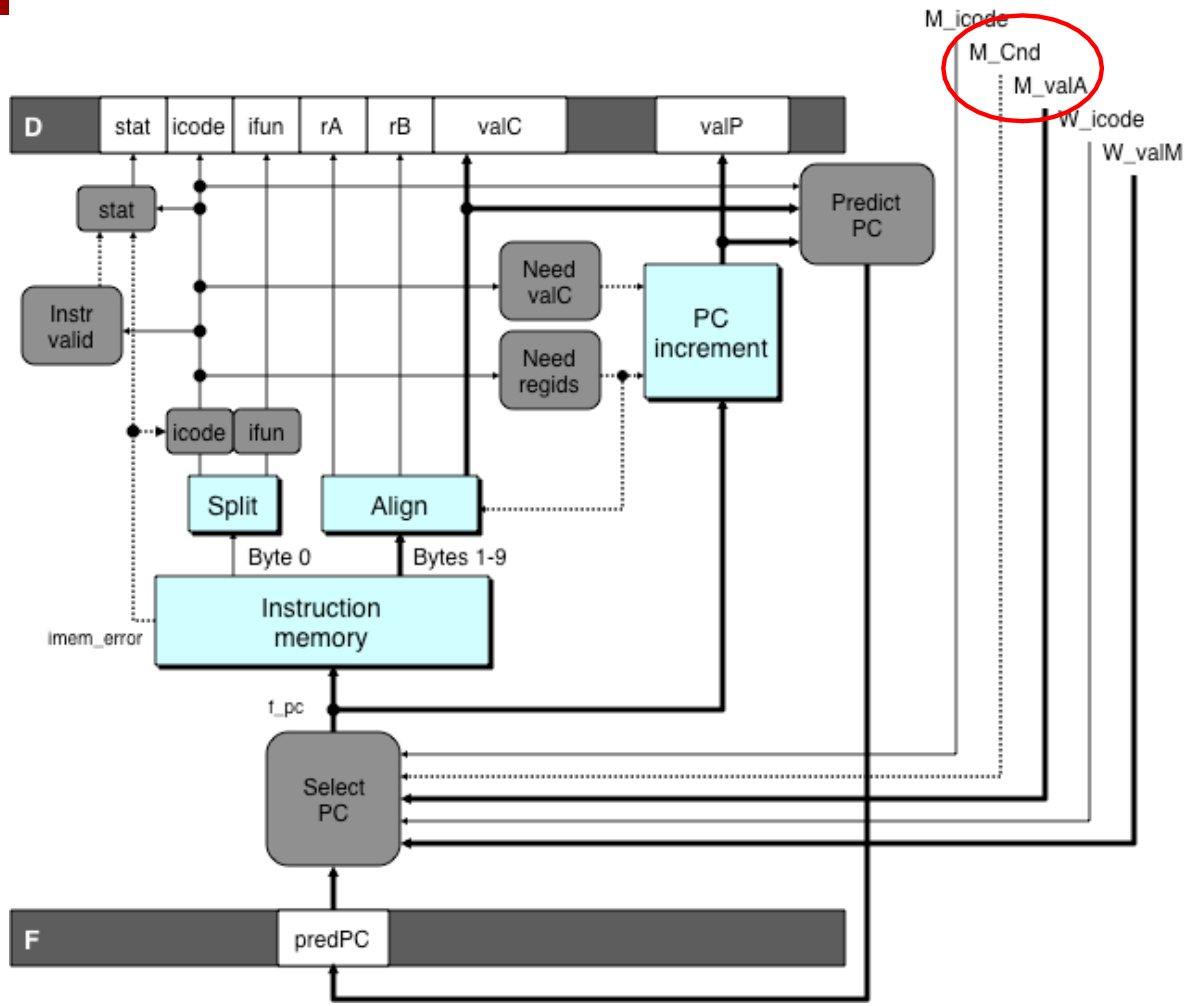


- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

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

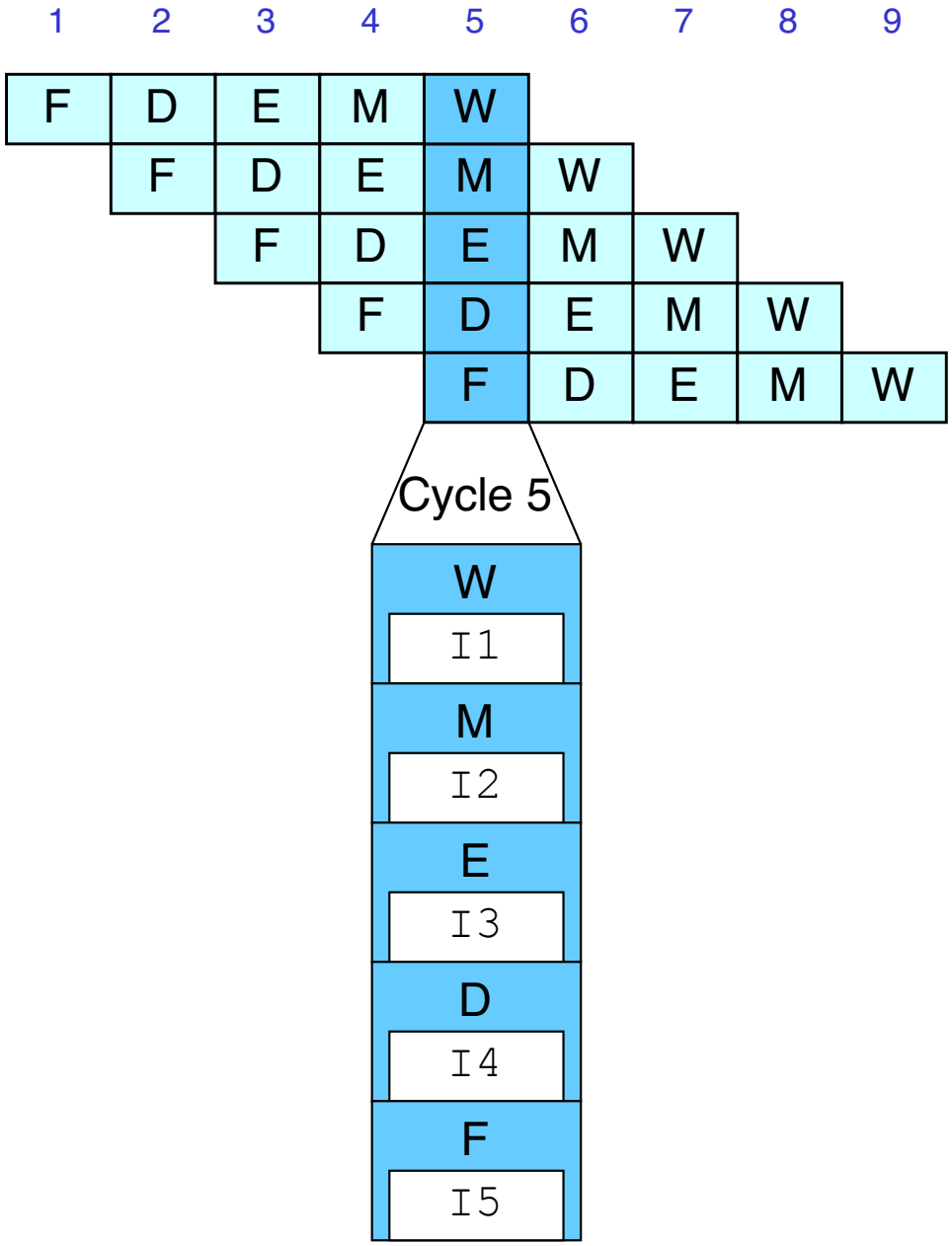
Recovering from PC Misprediction



- Mispredicted Jump
  - Will see branch 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 Demonstration

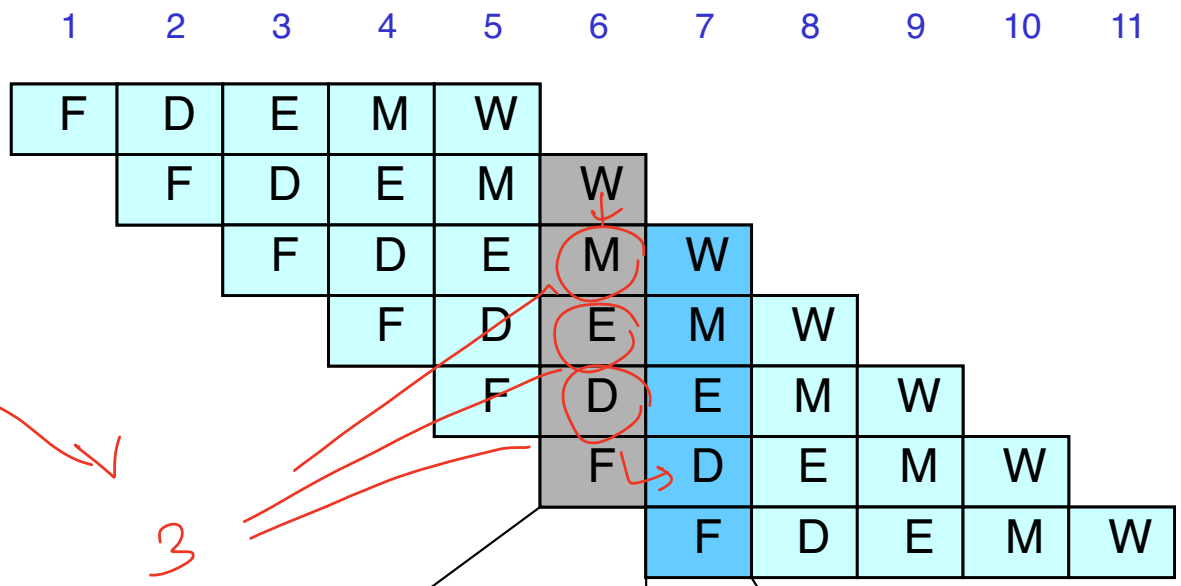
```
irmovq    $1,%rax   #I1
irmovq    $2,%rcx   #I2
irmovq    $3,%rdx   #I3
irmovq    $4,%rbx   #I4
halt                #I5
```

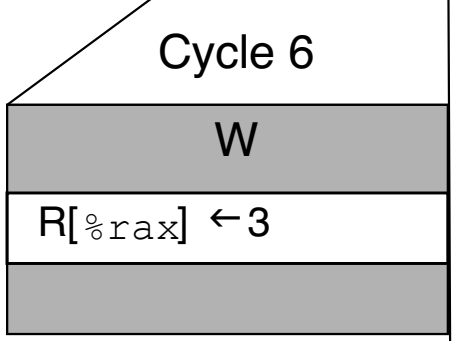


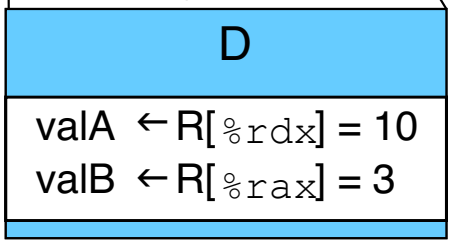
- File: **`demo-basic.ys`**

Data Dependencies: 3 Nop's

```
# demo-h3.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```

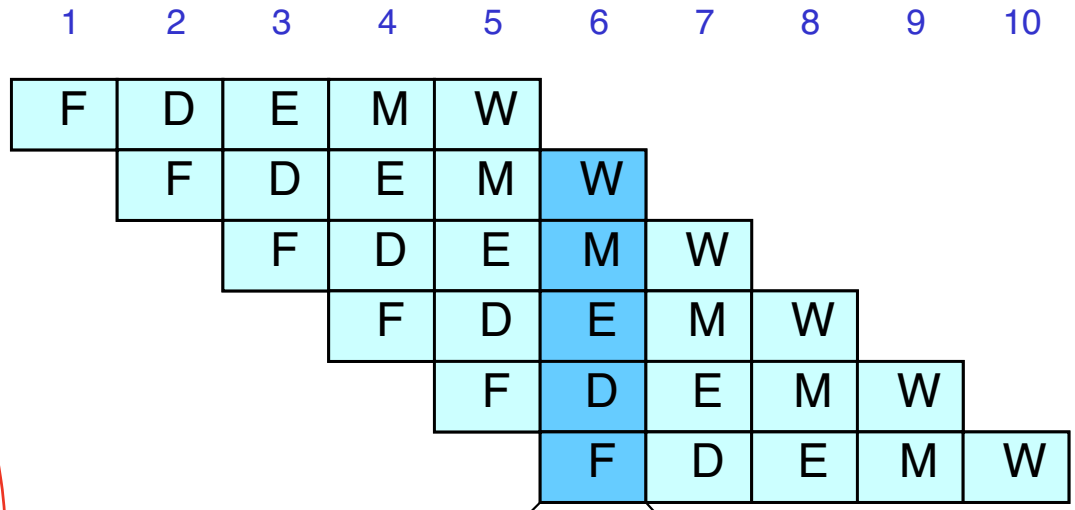


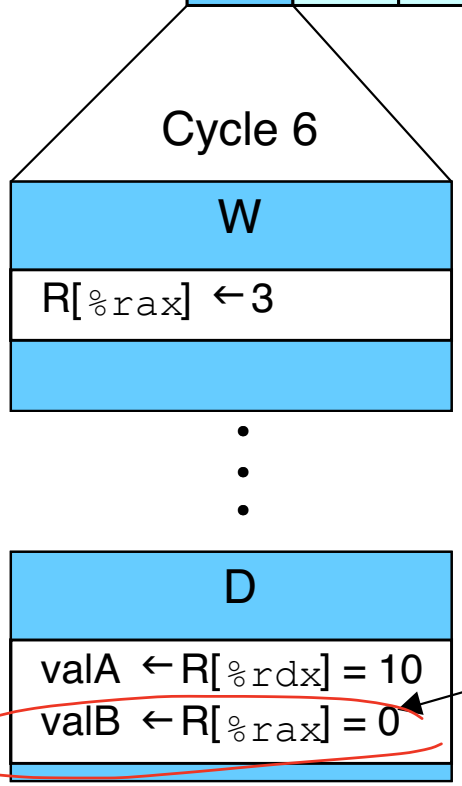




Data Dependencies: 2 Nop's

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```





try to write 3

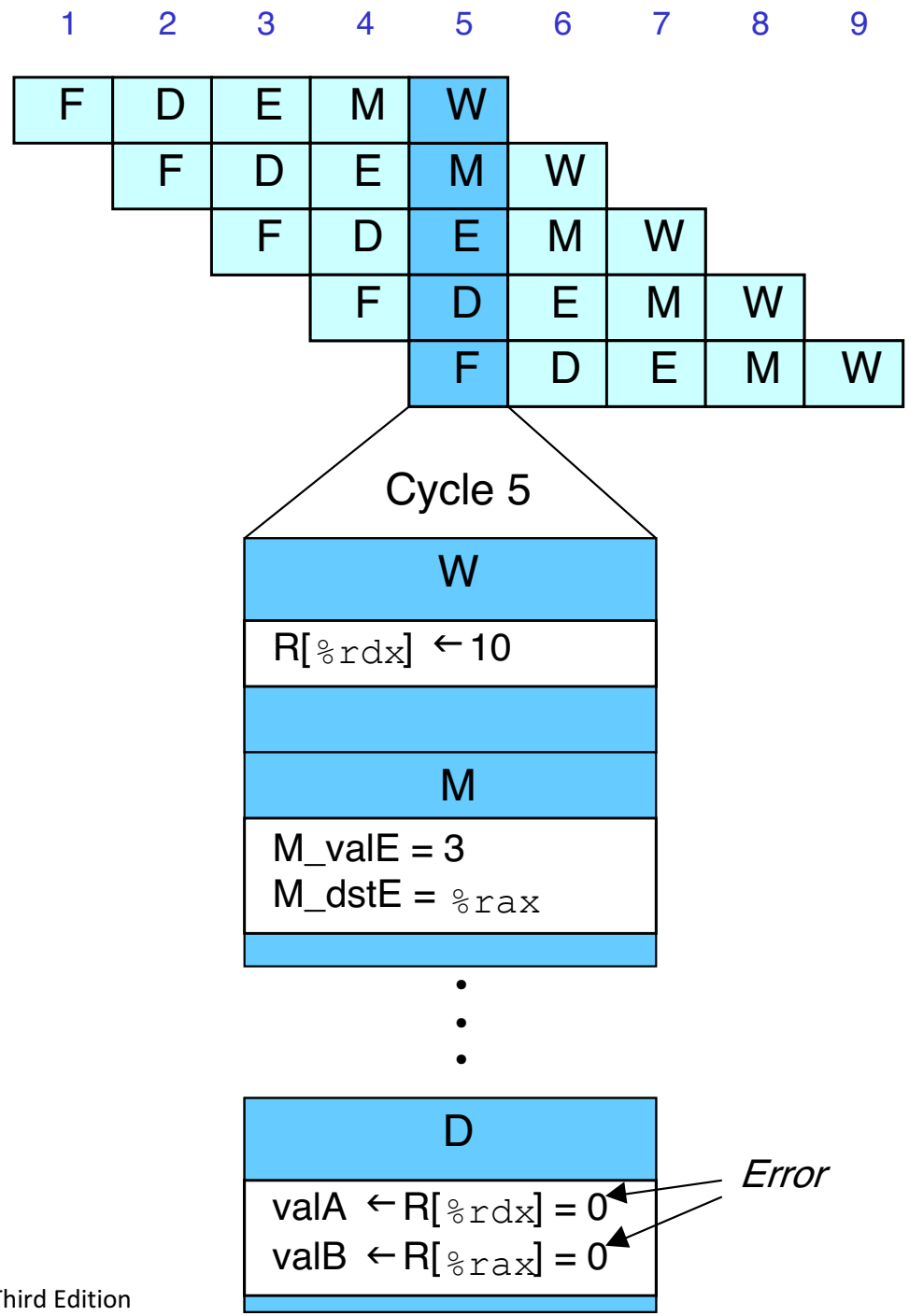
it happens in the next rising edge

try to read

this happens first

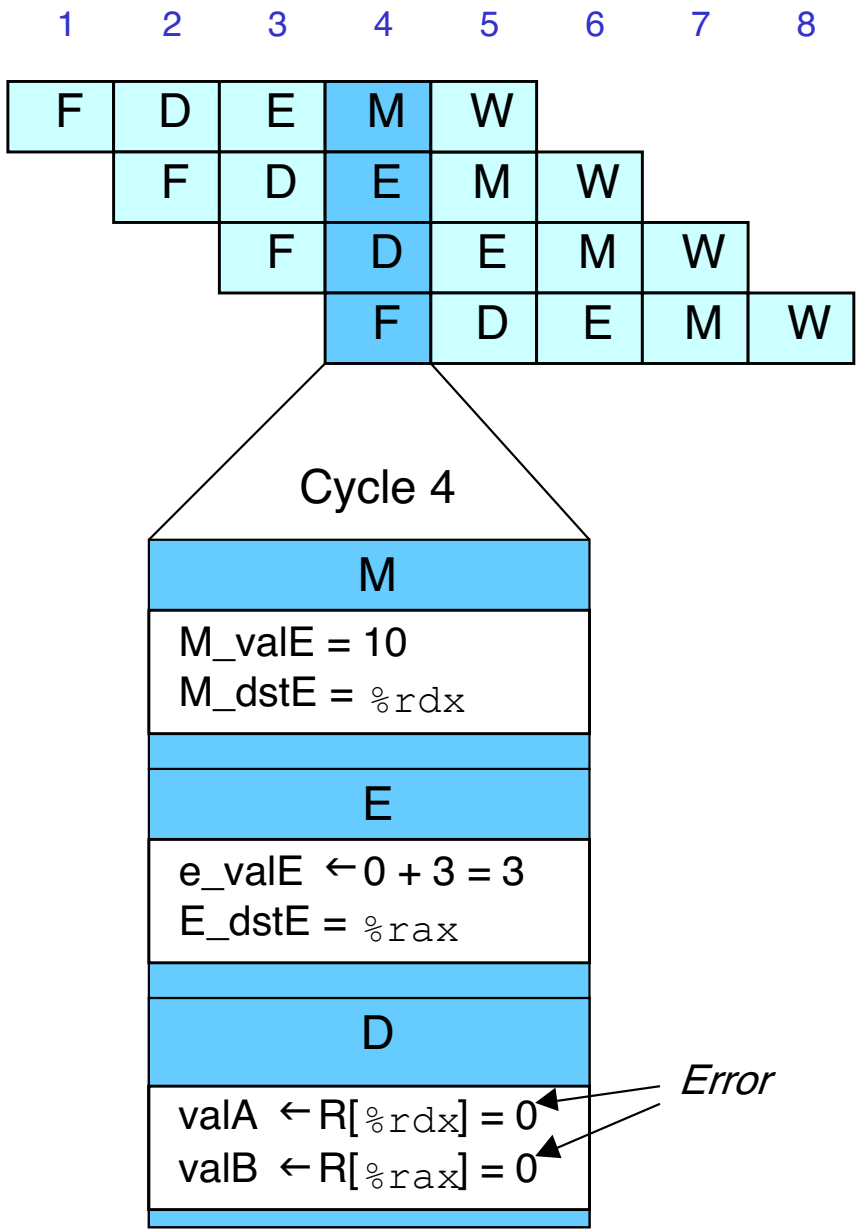
Data Dependencies: 1 Nop

```
# demo-h1.ys

0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Branch Misprediction Example

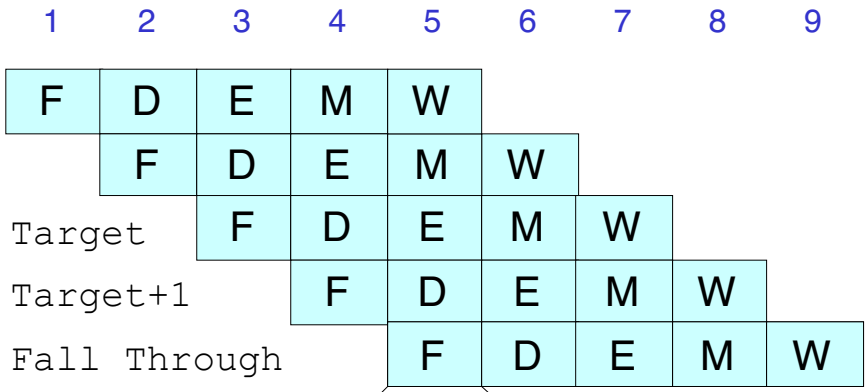
demo-j.ys

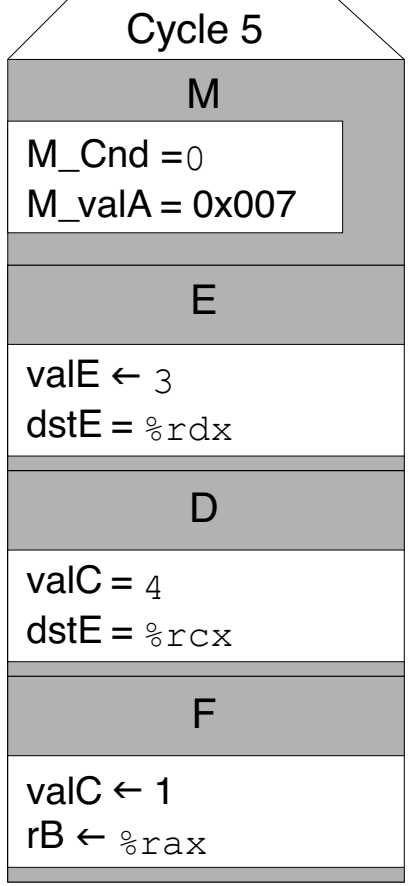
```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target (Should not execute)
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

- Should only execute first 8 instructions

Branch Misprediction Trace

```
# demo-j
0x000:     xorq %rax,%rax
0x002:     jne t # Not taken
0x019: t:  irmovq $3, %rdx # Target
0x023:     irmovq $4, %rcx # Target+1
0x00b:     irmovq $1, %rax # Fall Through
```



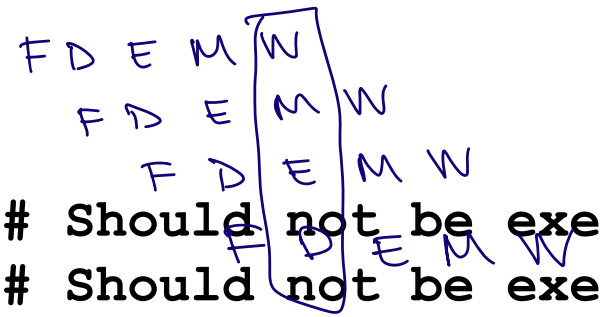


- **Incorrectly execute two instructions at branch target**

demo-ret.ys

Return Example

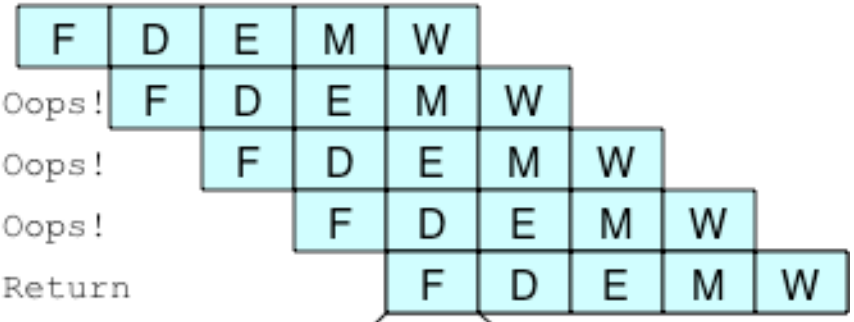
```
 0x000:    irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     nop                # Avoid hazard on %rsp
0x00b:     nop
0x00c:     nop
0x00d:     call p             # Procedure call
0x016:     irmovq $5,%rsi     # Return point
0x020:     halt
0x020: .pos 0x20
0x020: p:  nop                # procedure
0x021:     nop
0x022:     nop
0x023:     ret
0x024:     irmovq $1,%rax     # Should not be executed
0x02e:     irmovq $2,%rcx     # Should not be executed
0x038:     irmovq $3,%rdx     # Should not be executed
0x042:     irmovq $4,%rbx     # Should not be executed
0x100: .pos 0x100
0x100: Stack:                 # Initial stack pointer
```



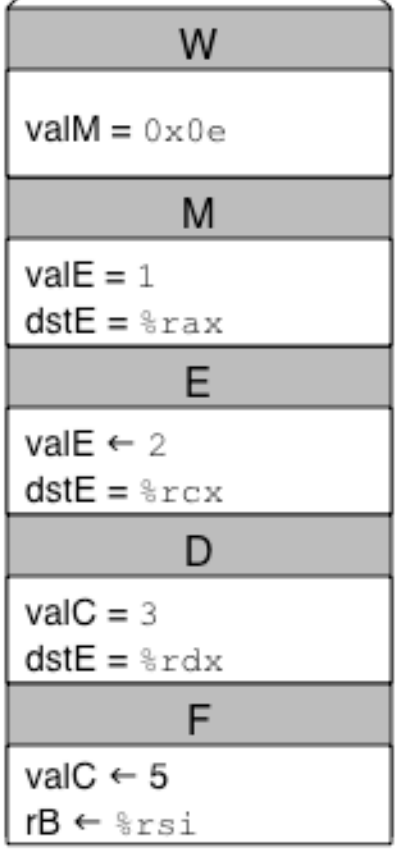
- Require lots of nops to avoid data hazards

Incorrect Return Example

```
# demo-ret
0x033:     ret
0x034:     irmovq $1,%rax # Oops!
0x03e:     irmovq $2,%rcx # Oops!
0x048:     irmovq $3,%rdx # Oops!
0x052:     irmovq $5,%rsi # Return
```



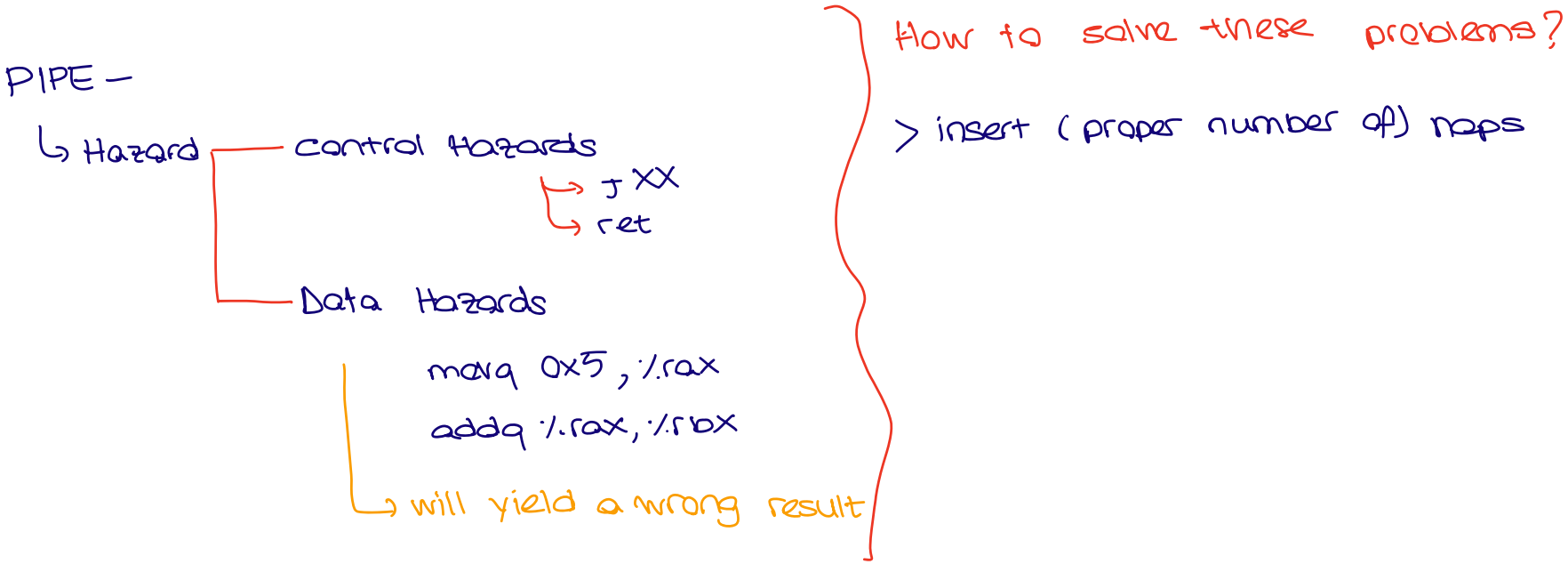
- **Incorrectly execute 3 instructions following `ret`**



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

Processor Architecture: Pipelined Implementation: 2



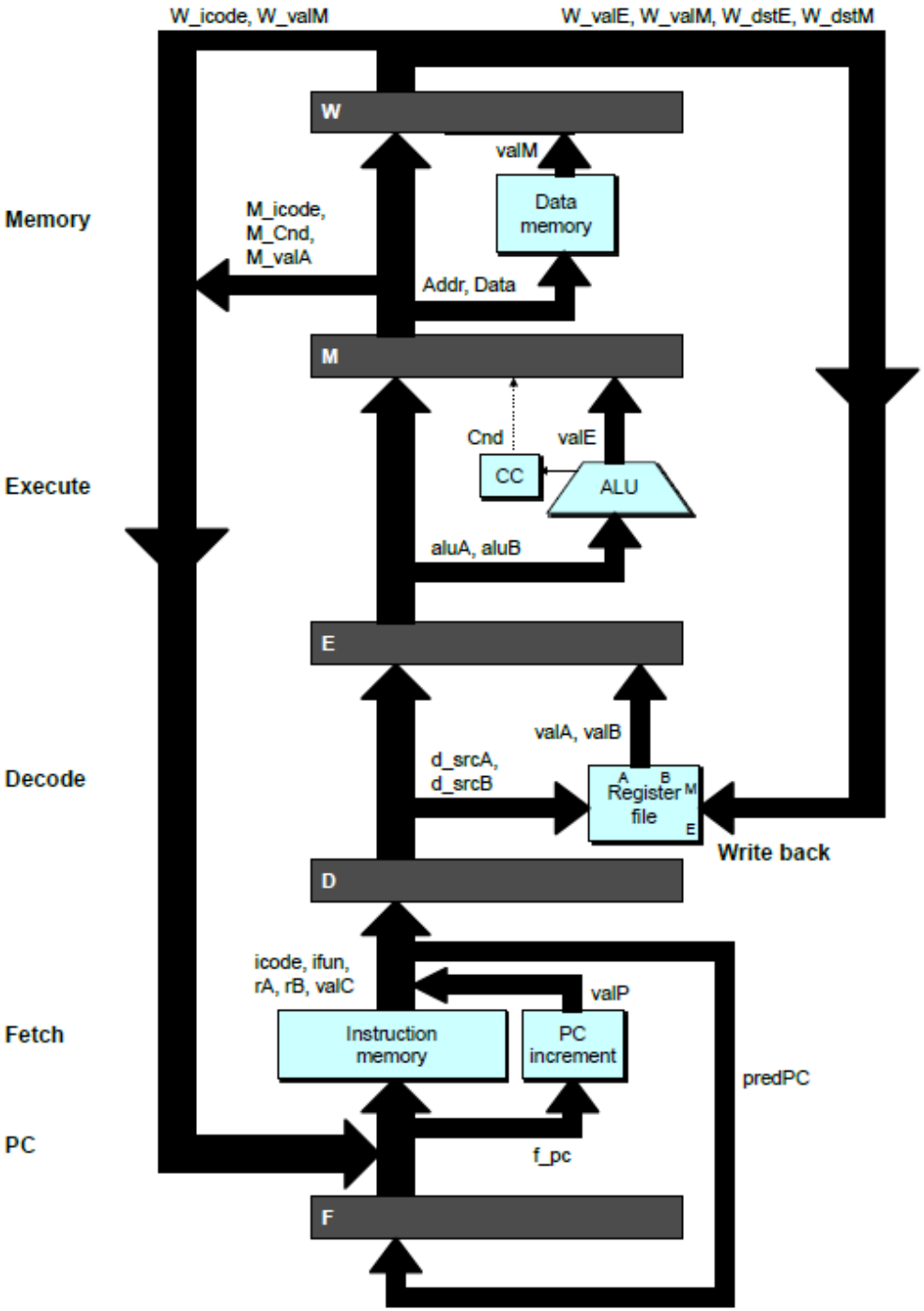
Overview

Make the pipelined processor work!

- **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
- **Making Sure It Really Works**
  - What if multiple special cases happen simultaneously?

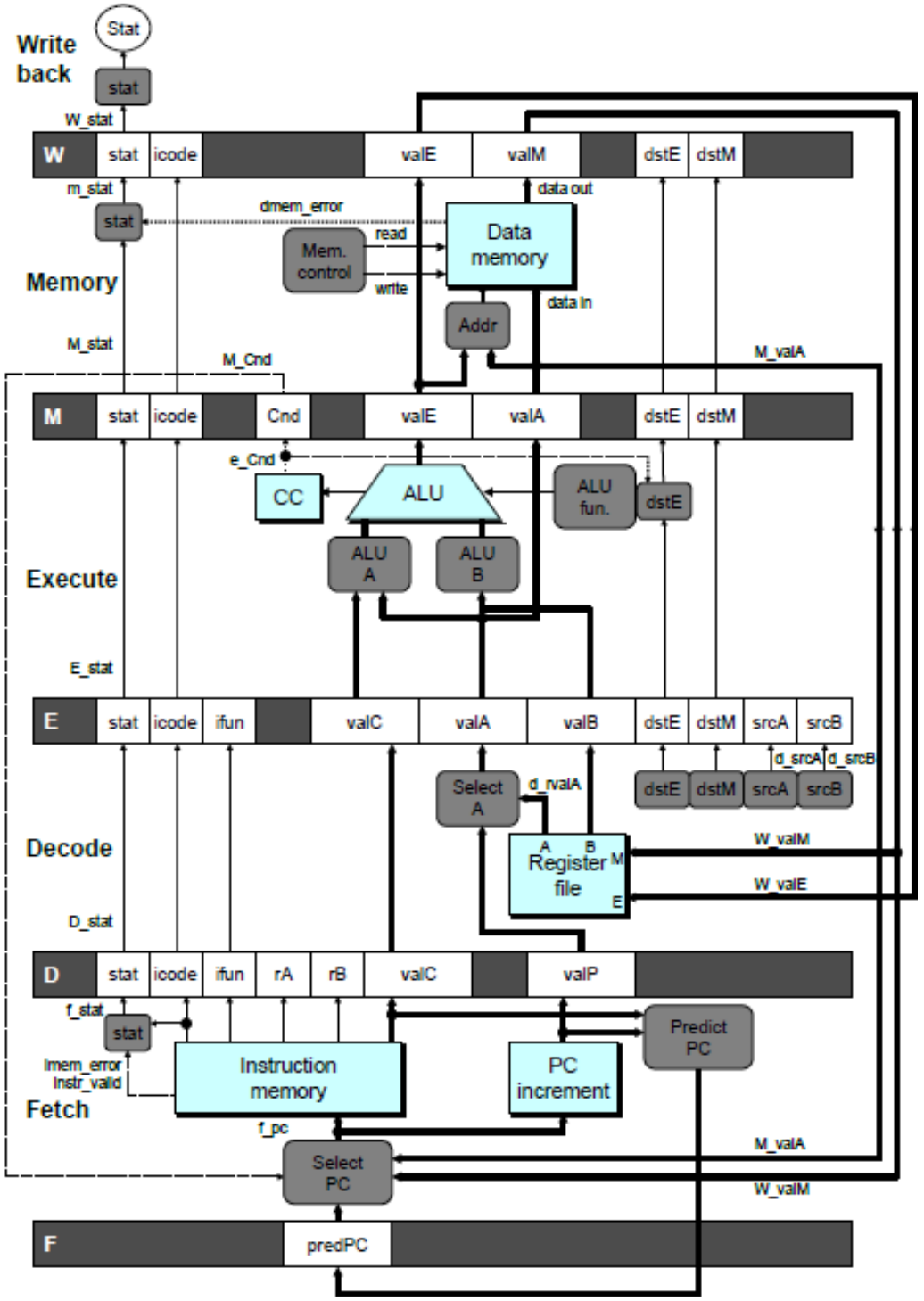
Pipeline Stages

- **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



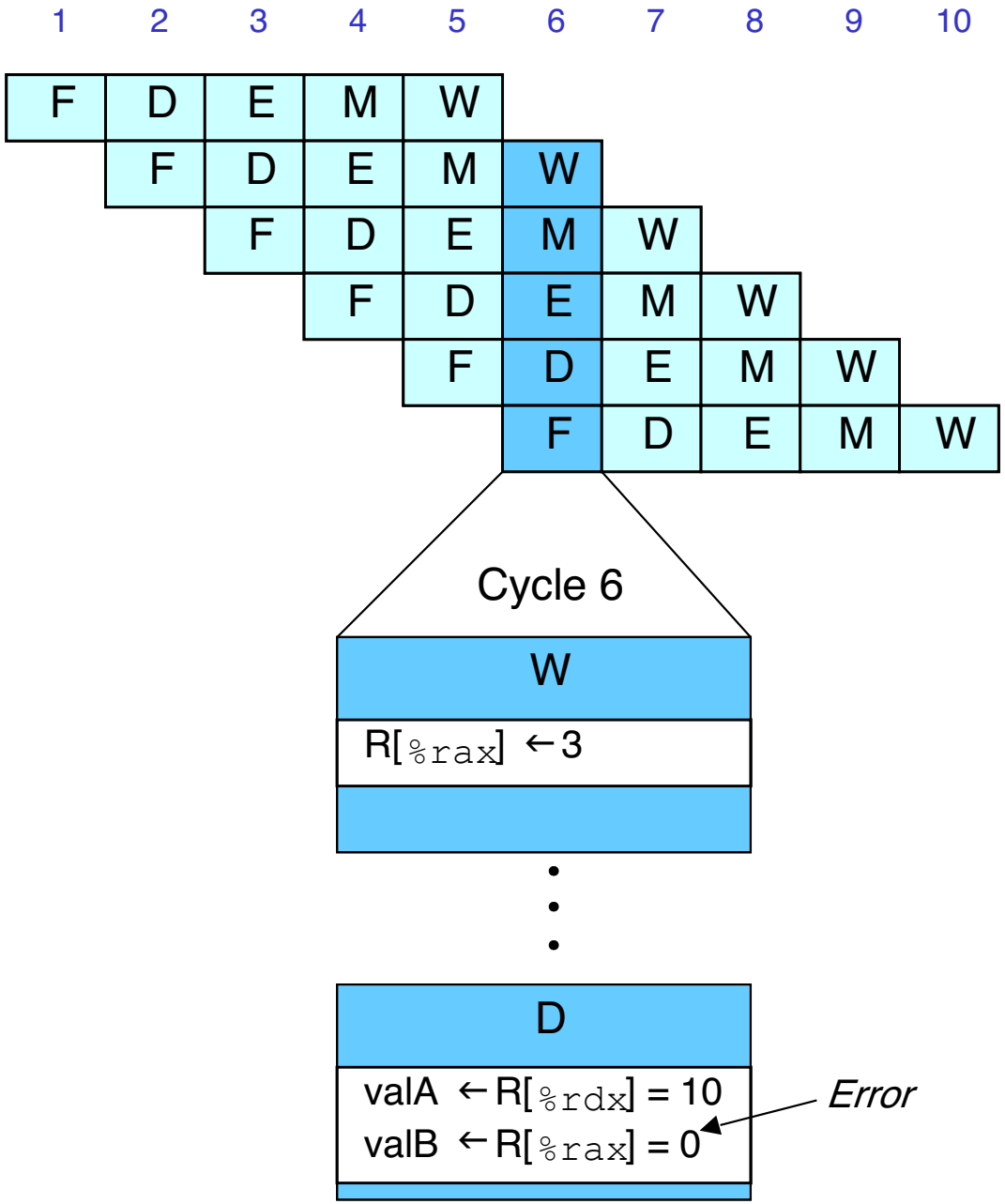
PIPE- Hardware

- 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



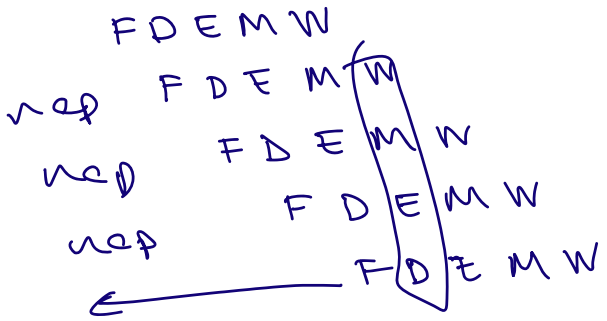
Data Dependencies: 2 Nop's

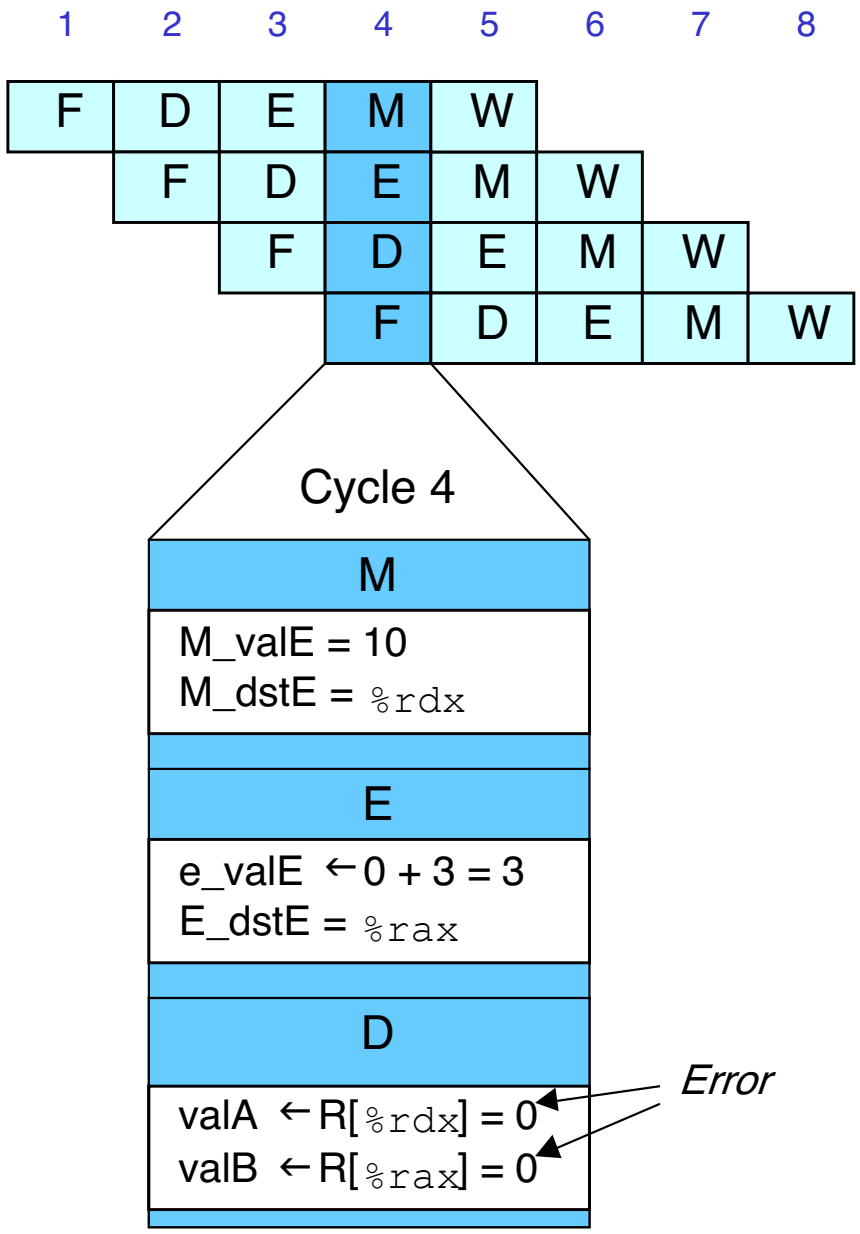
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



Data Dependencies: No Nop

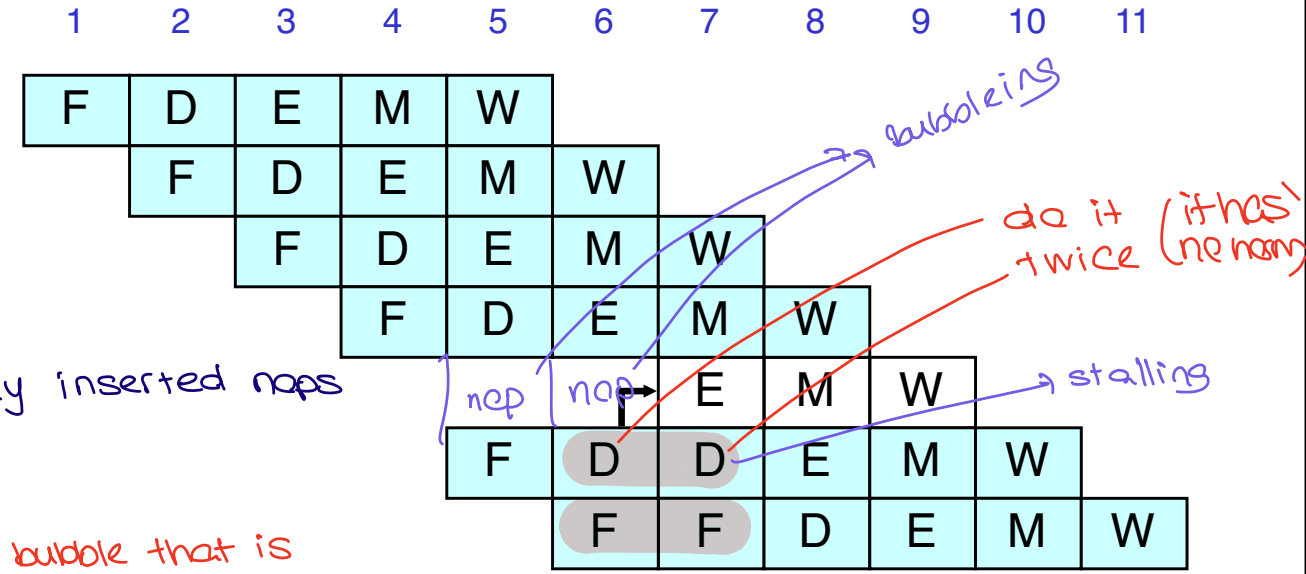
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```





Stalling for Data Dependencies

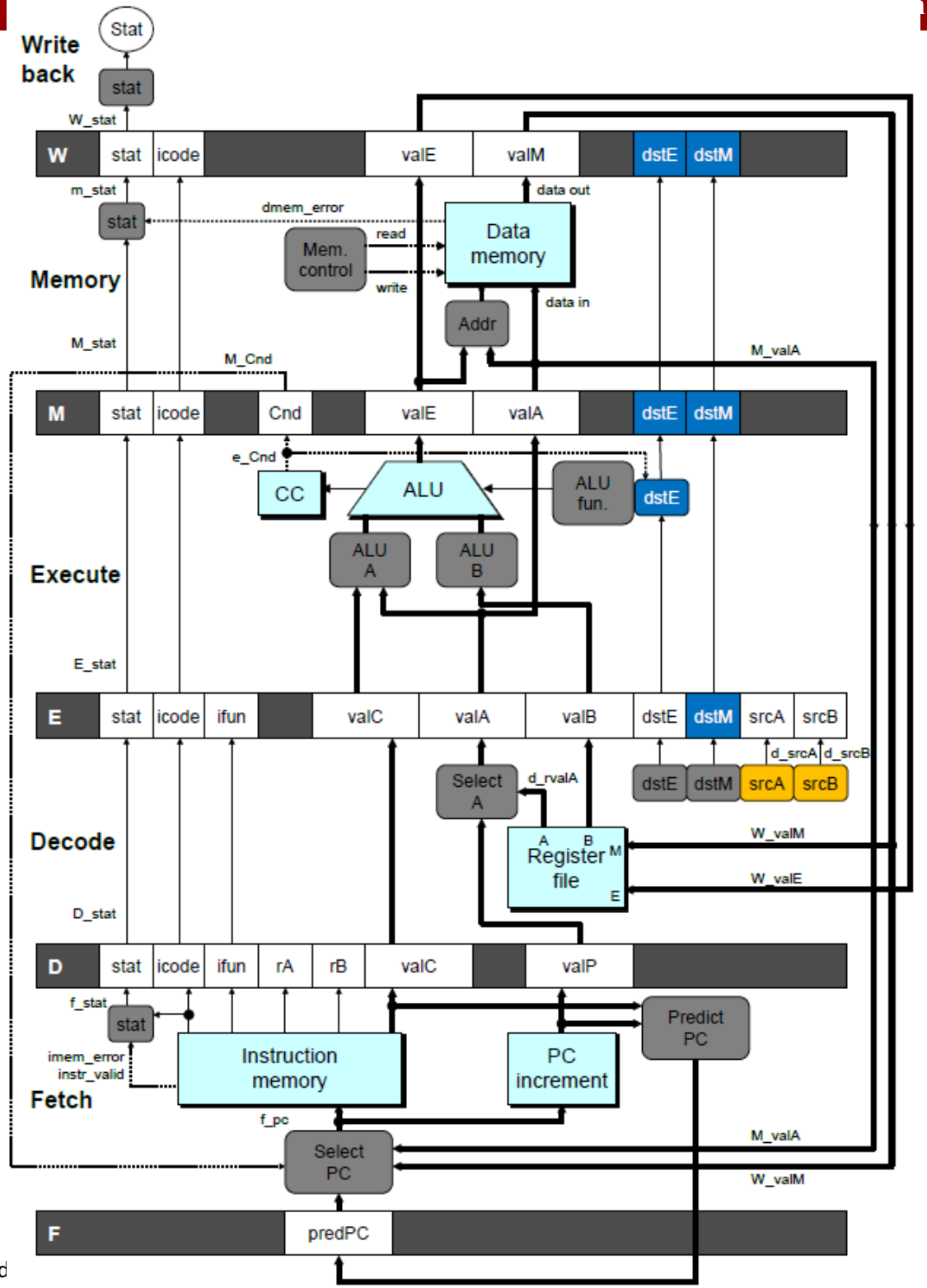
```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

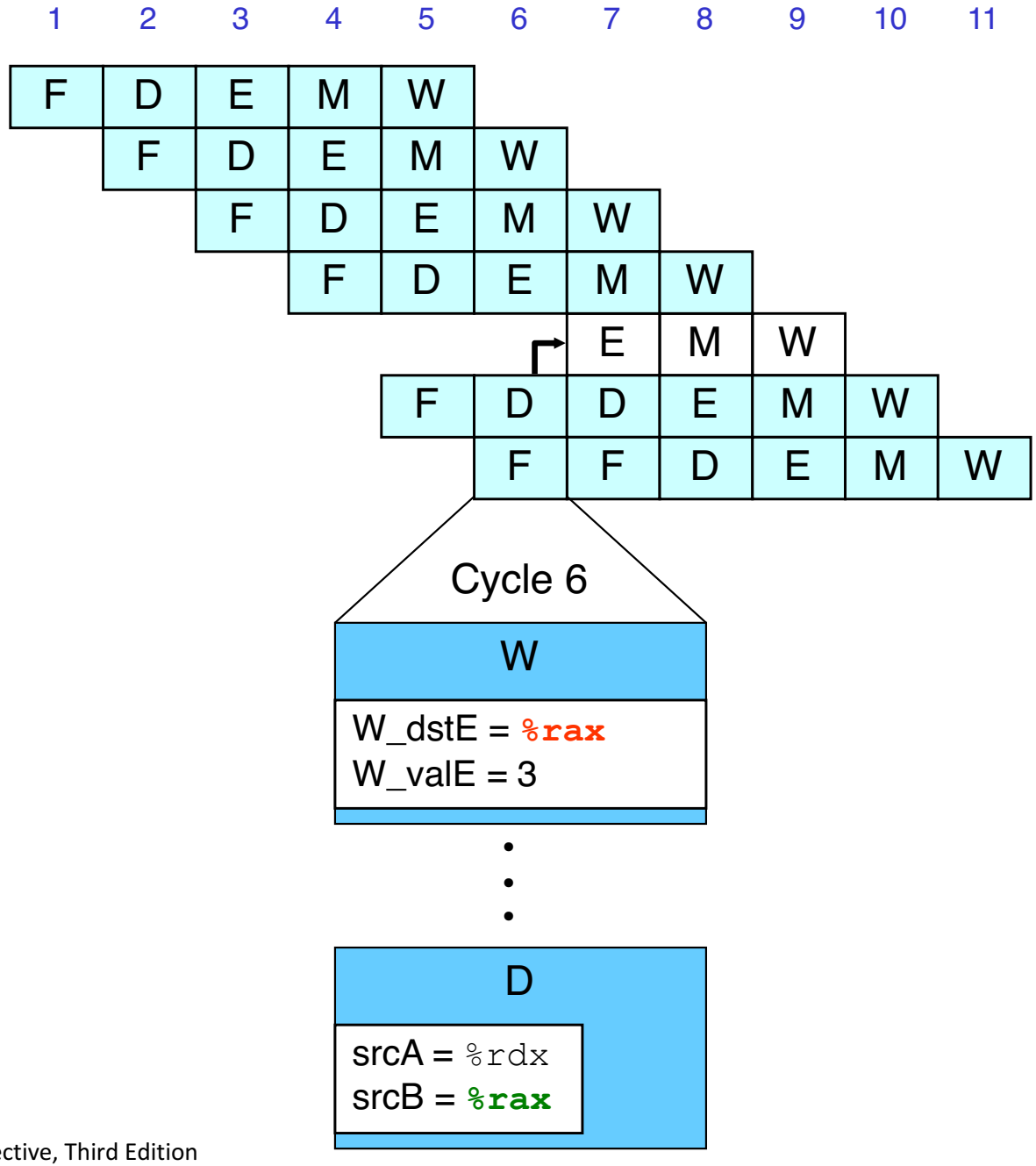
Stall Condition

- **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 15 (0xF)
    - Indicates absence of register operand
    - Or failed cond. move



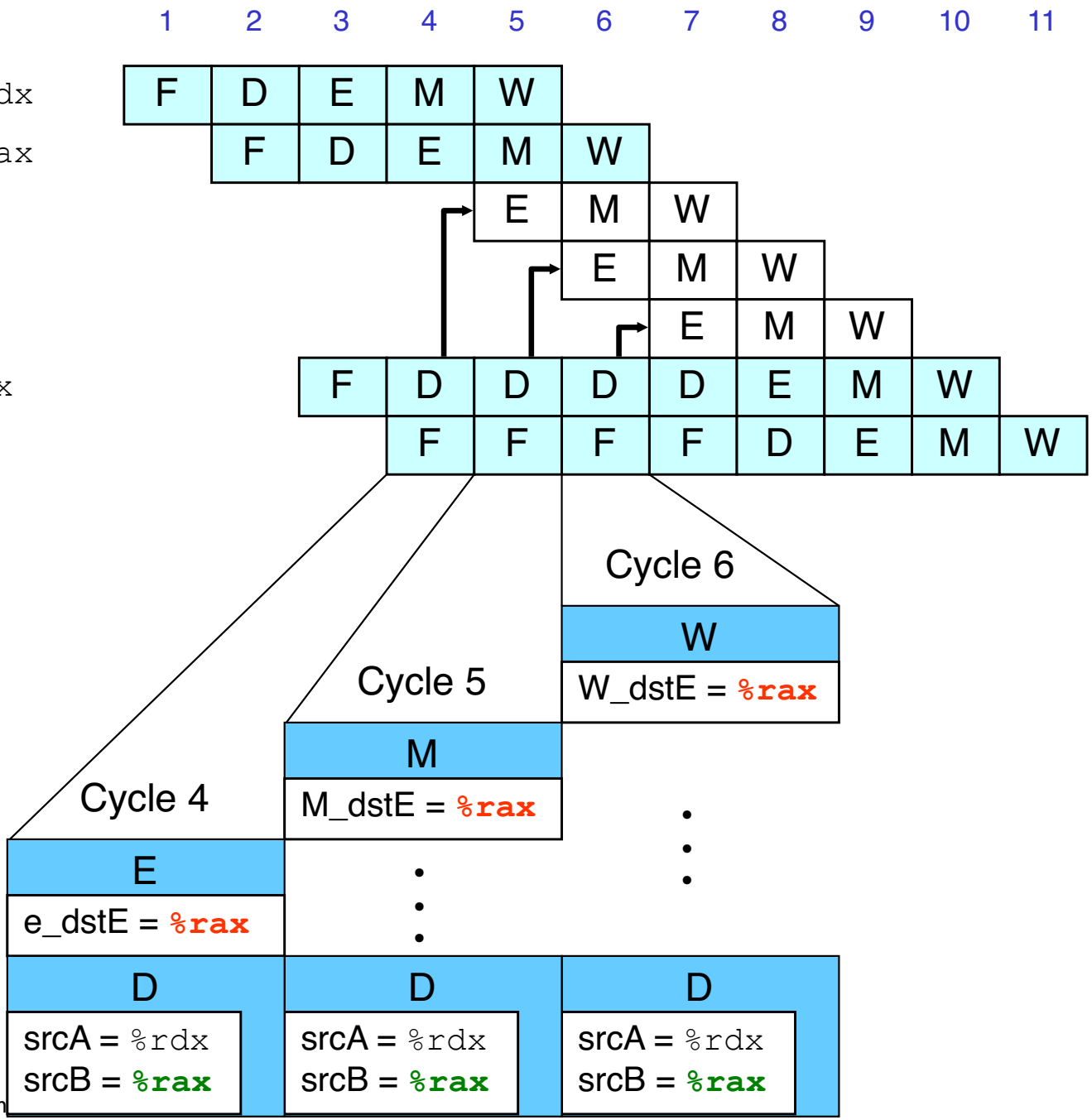
Detecting Stall Condition

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



Stalling X3

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



What Happens When Stalling?

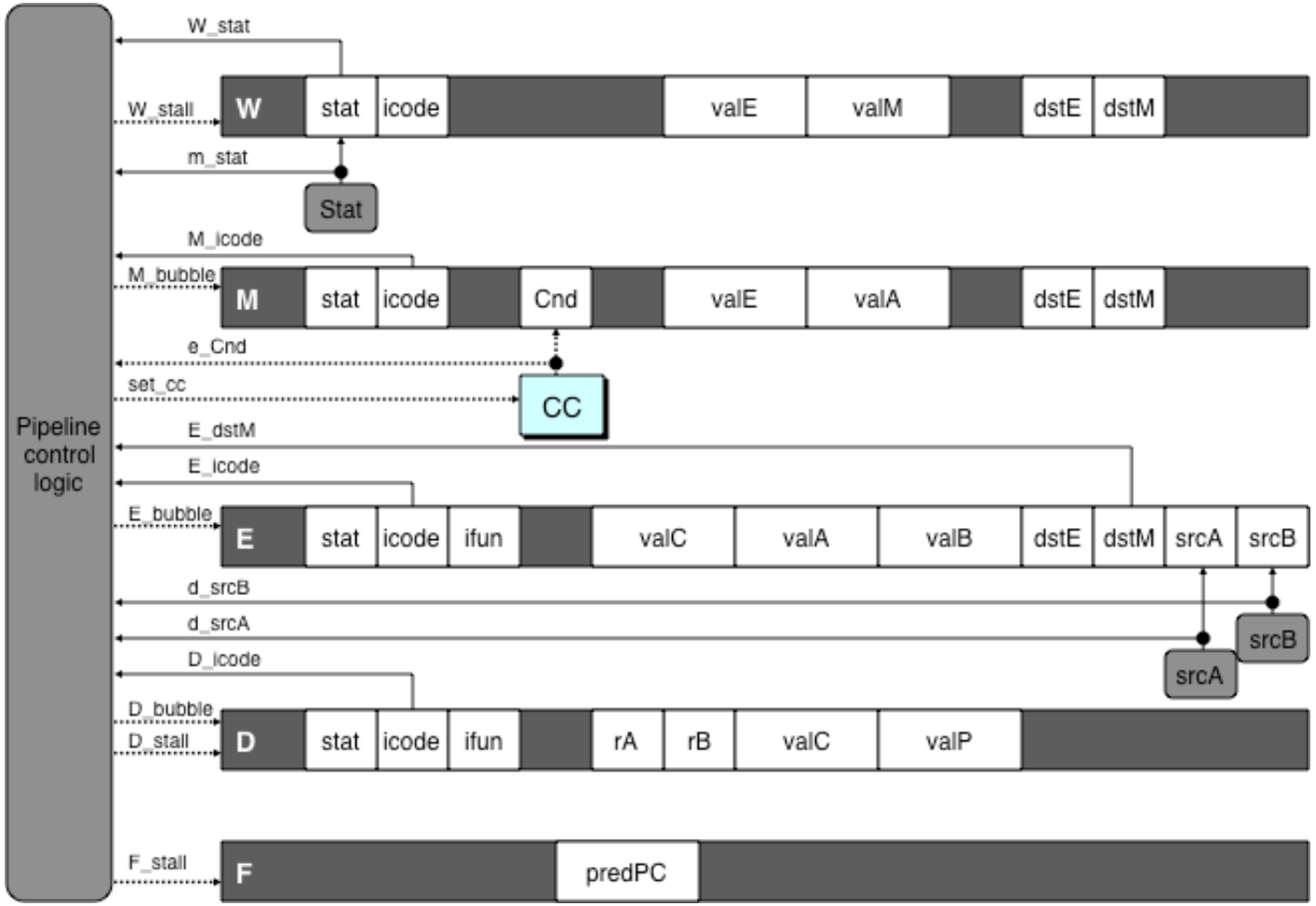
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8

Write Back	***bubble***
Memory	***bubble***
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

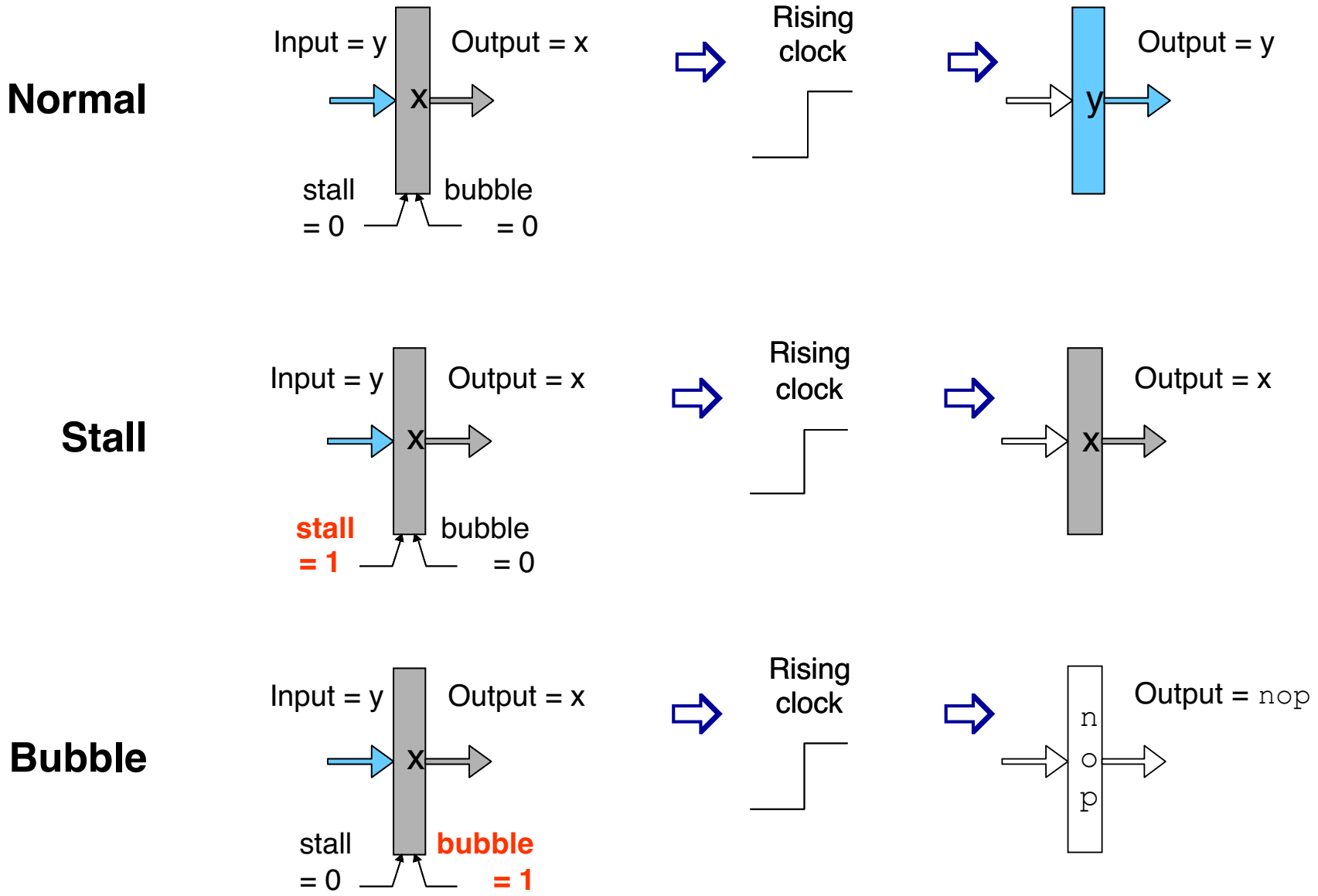
- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

Implementing Stalling



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

Pipeline Register Modes

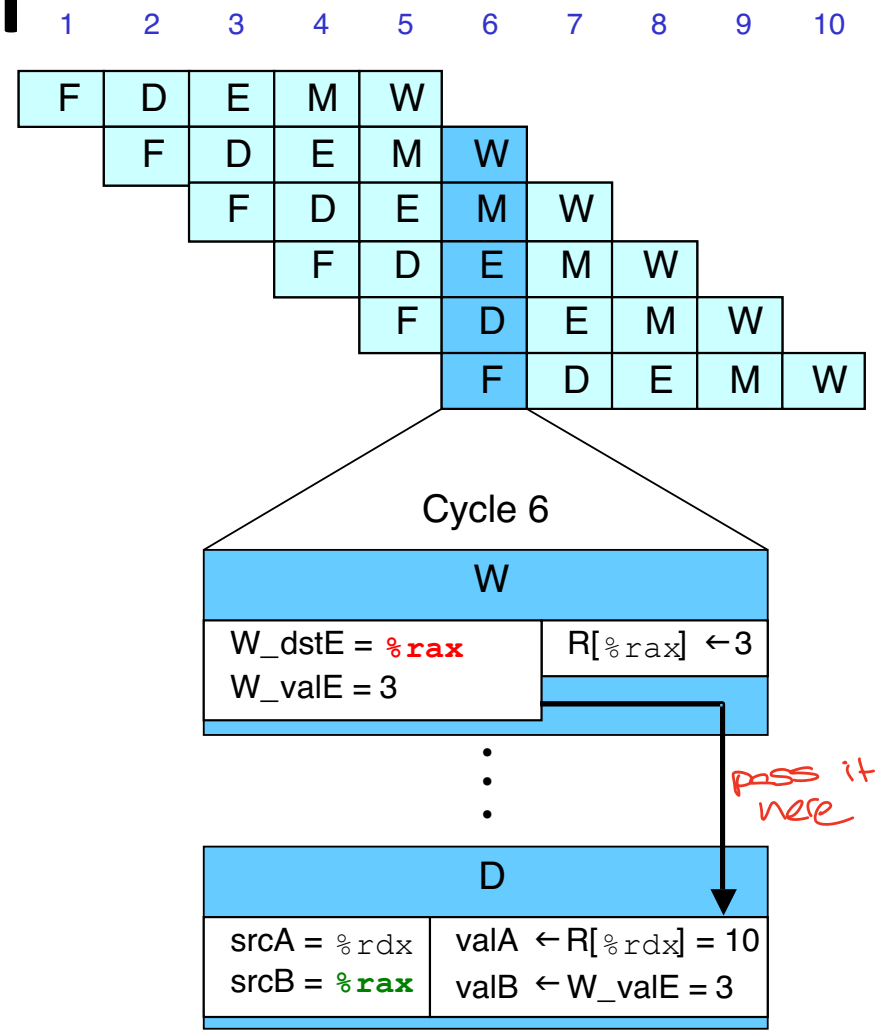


Data Forwarding

- **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.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

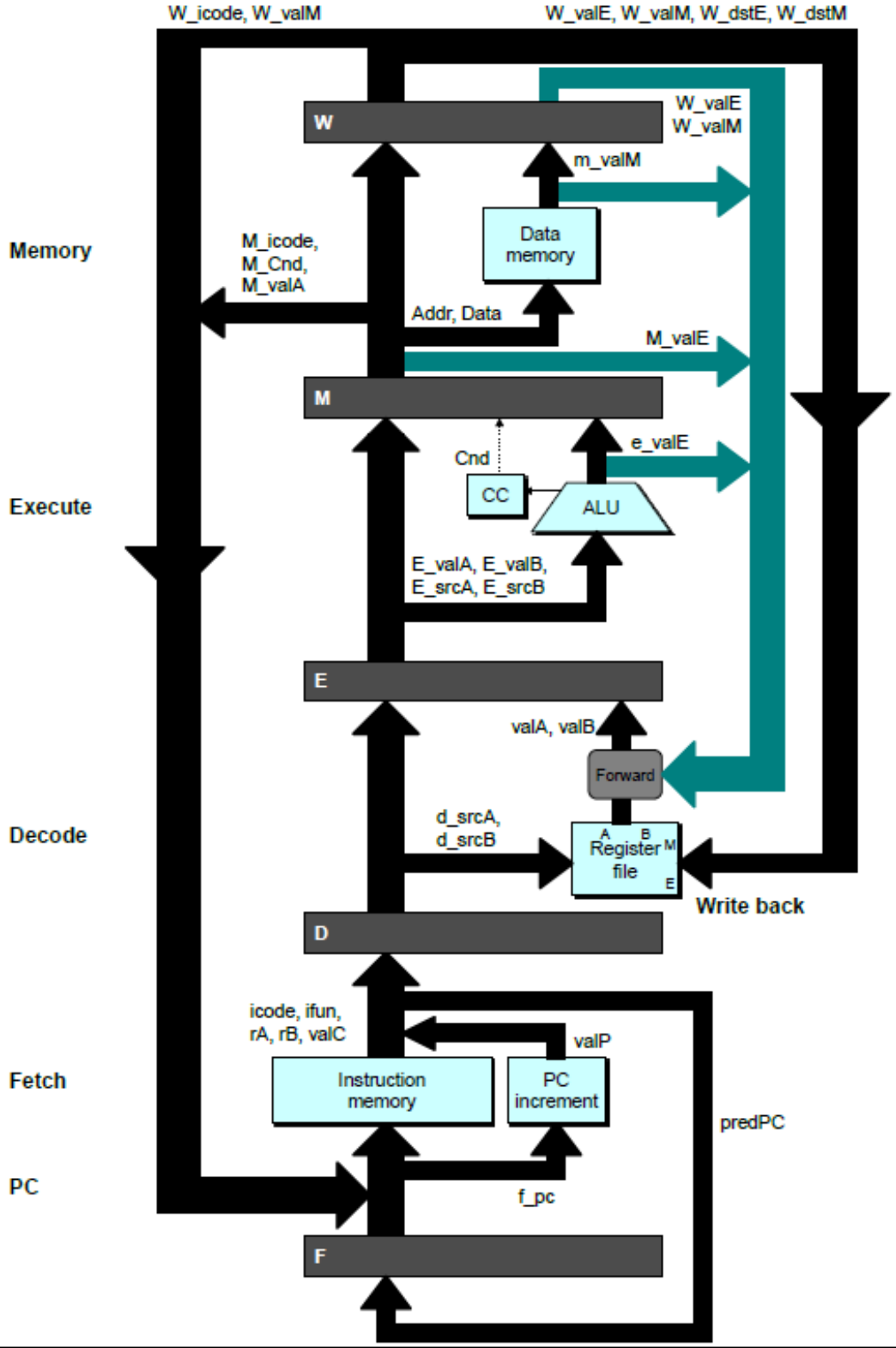


- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

Bypass Paths

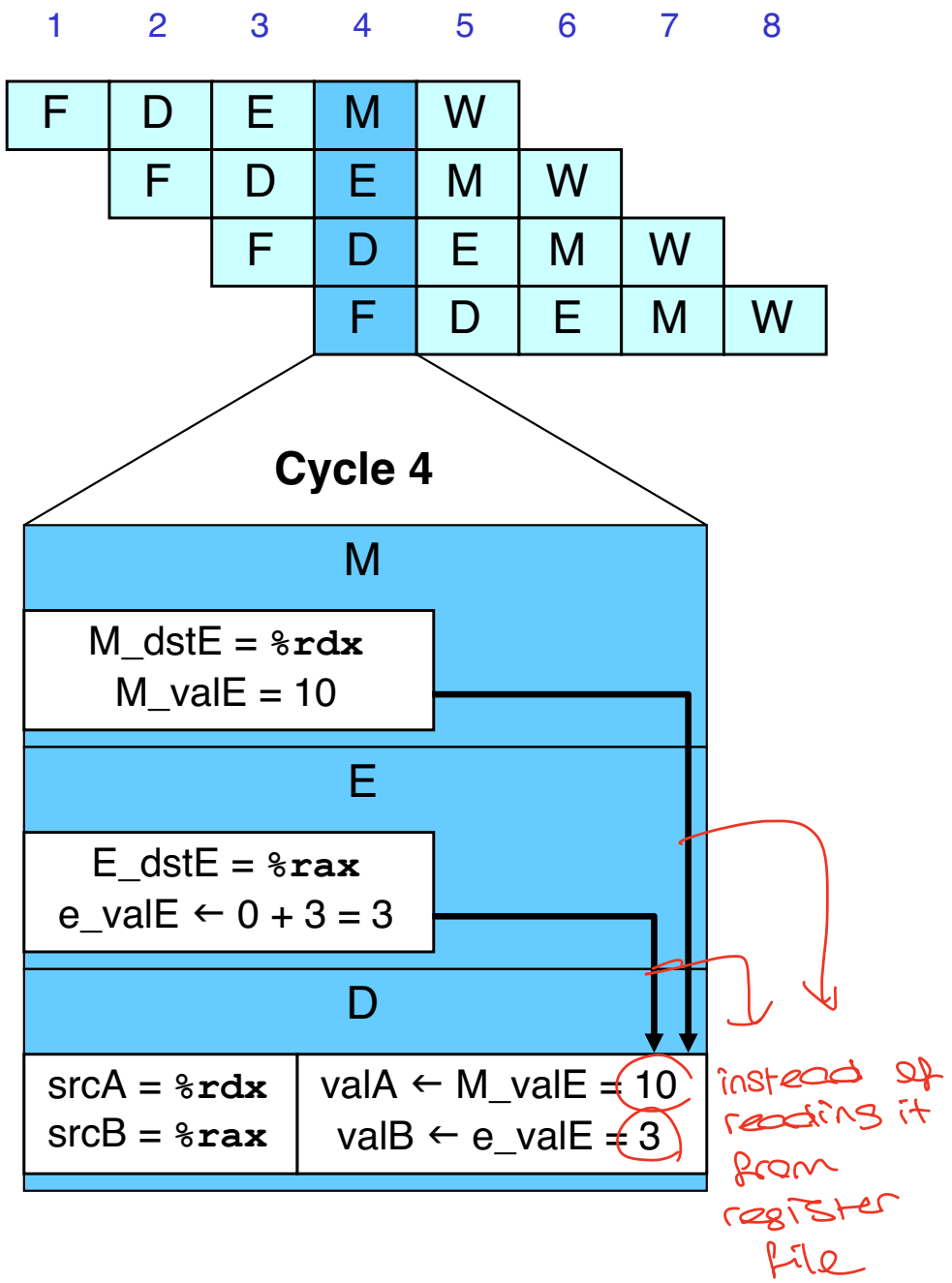
- **Decode Stage**
  - Forwarding logic selects valA and valB
  - Normally from register file
  - Forwarding: get valA or valB from later pipeline stage
- **Forwarding Sources**
  - Execute: valE
  - Memory: valE, valM
  - Write back: valE, valM

no stalling or bubbleing



Data Forwarding Example #2

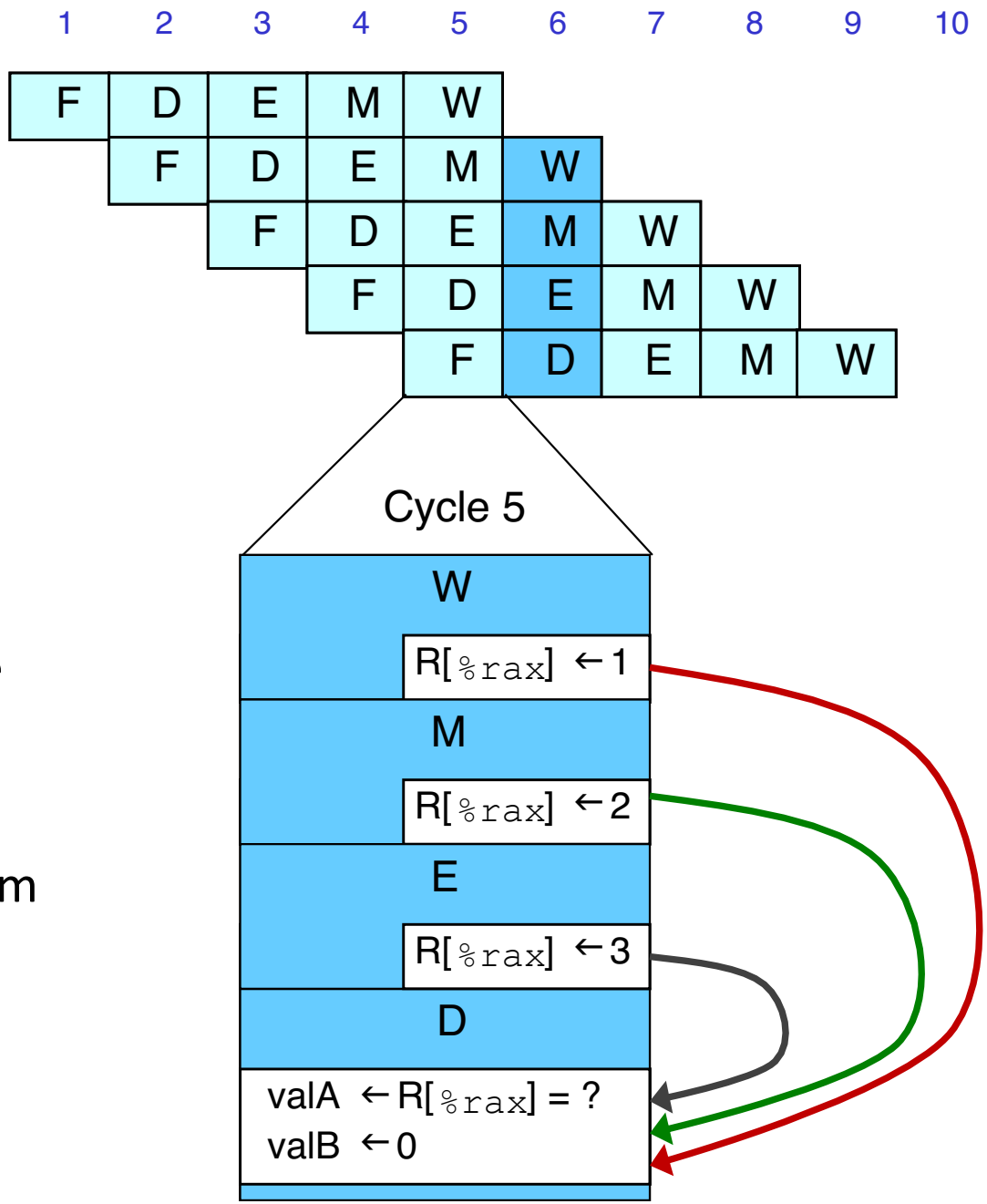
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



- **Register %rdx**
  - Generated by ALU during previous cycle
  - Forward from memory as valA
- **Register %rax**
  - Value just generated by ALU
  - Forward from execute as valB

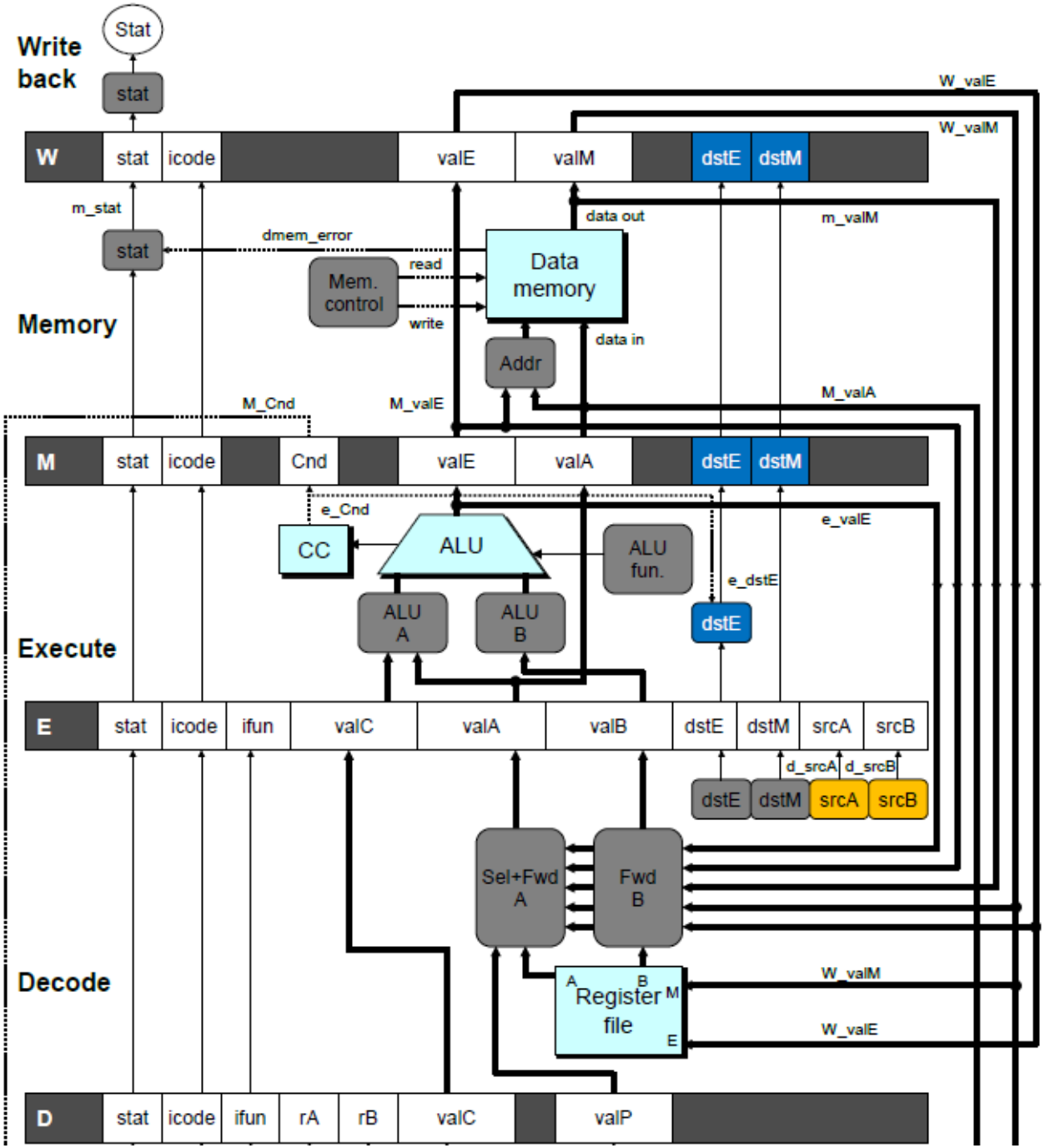
Forwarding Priority

```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



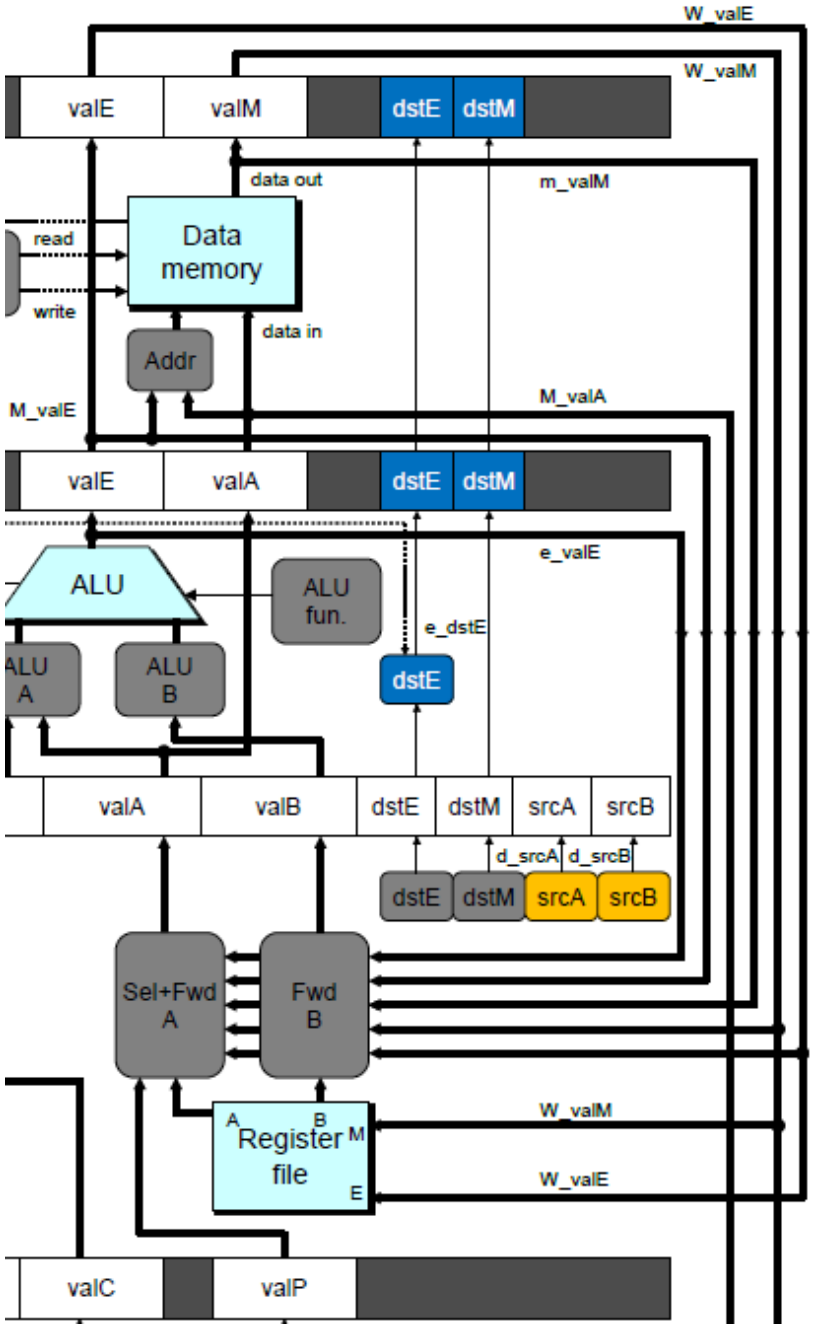
- **Multiple Forwarding Choices**
  - Which one should have priority
  - Match serial semantics
  - Use matching value from earliest pipeline stage

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

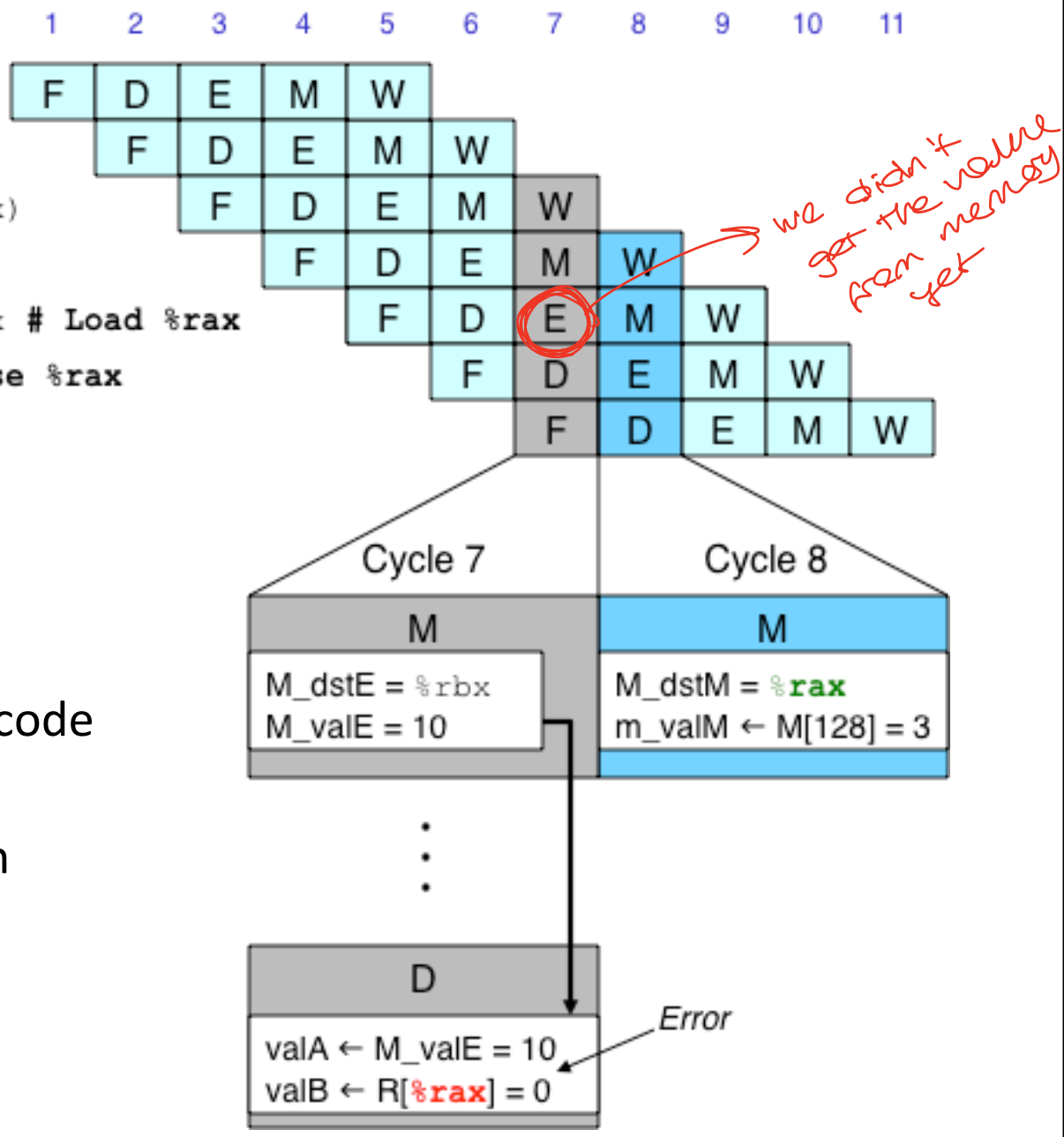
Implementing Forwarding




```
## What should be the A value?
int d_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

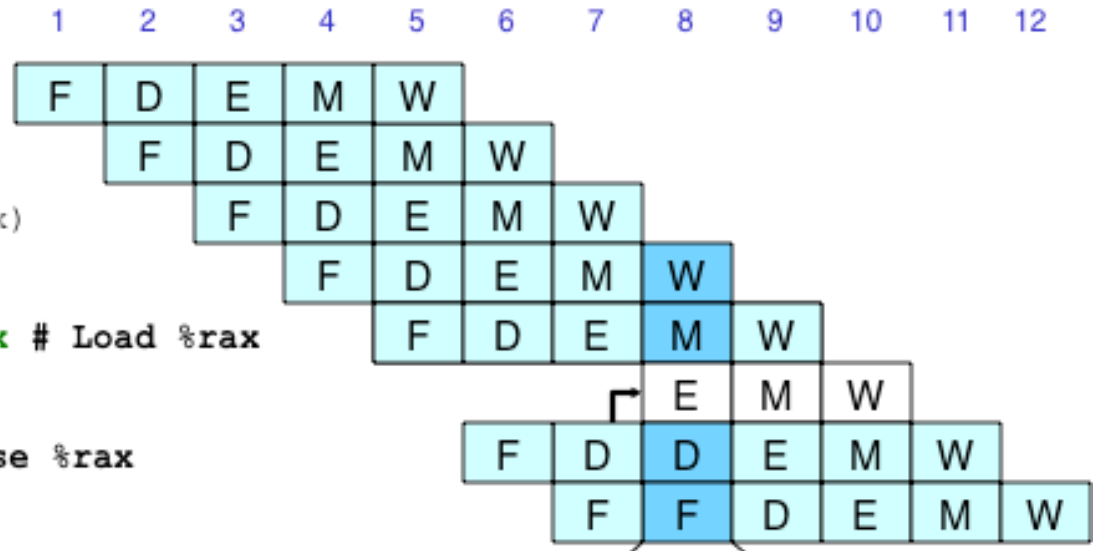
```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

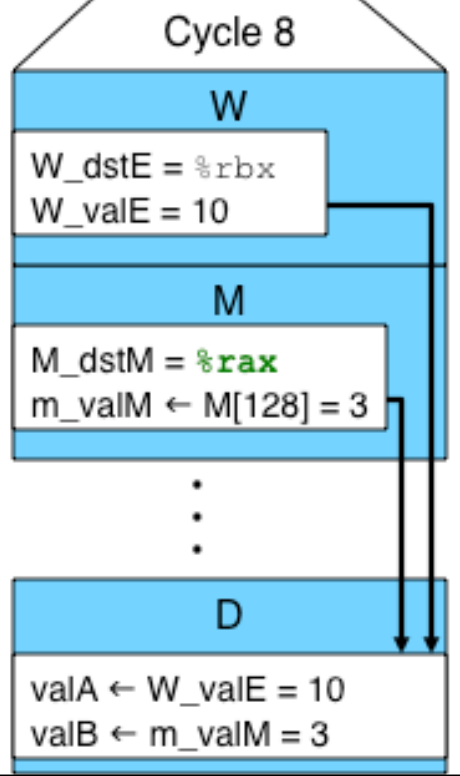


- **Load-use dependency**
  - Value needed by end of decode stage in cycle 7
  - Value read from memory in memory stage of cycle 8

Avoiding Load/Use Hazard

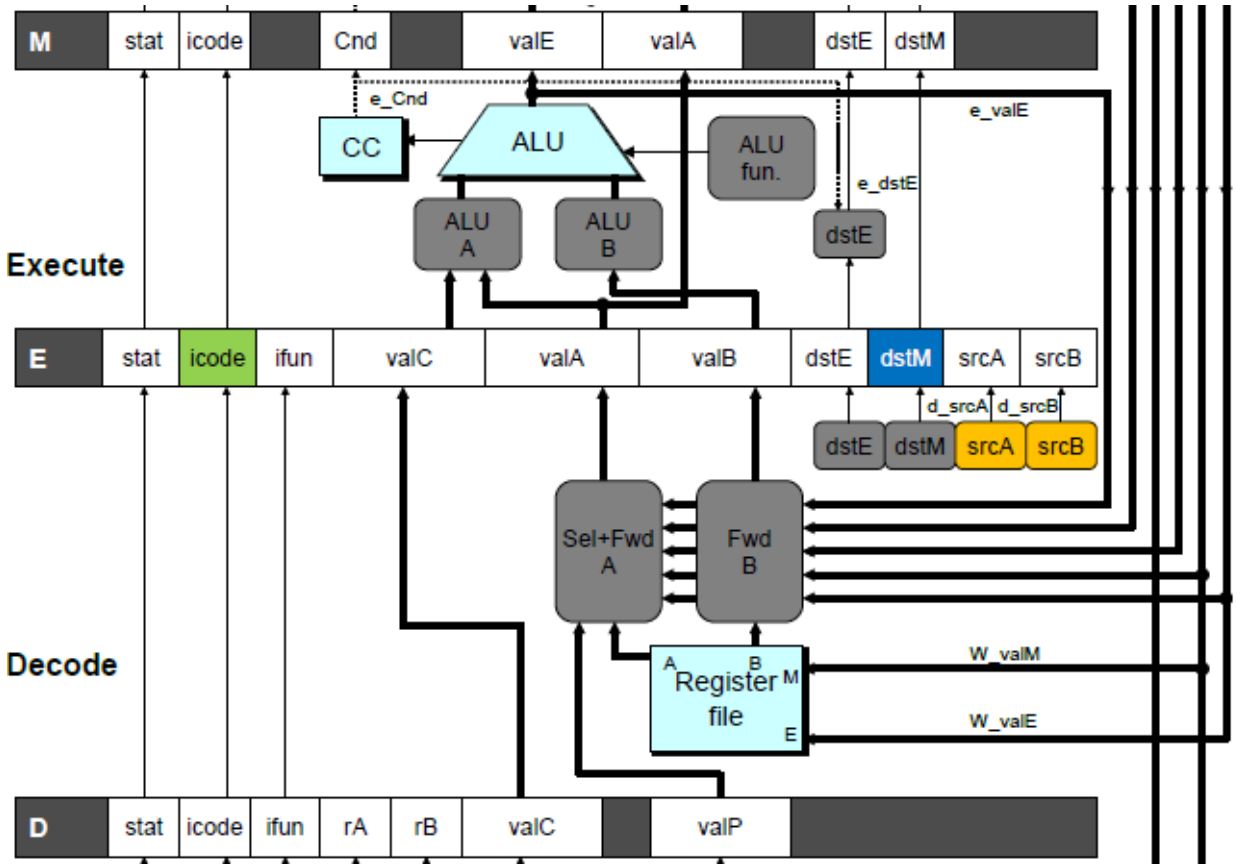
```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
     bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```





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

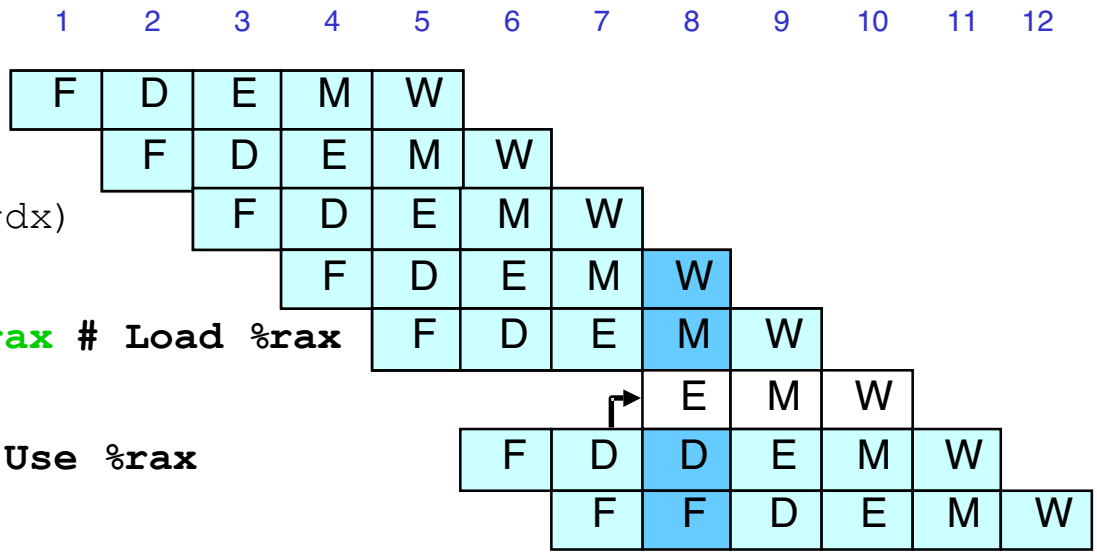
Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

Control for Load/Use Hazard

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%ebx
0x028: mrmovq 0(%rdx),%rax # Load %rax
       bubble
0x032: addq %ebx,%rax # Use %rax
0x034: 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

Branch Misprediction Example

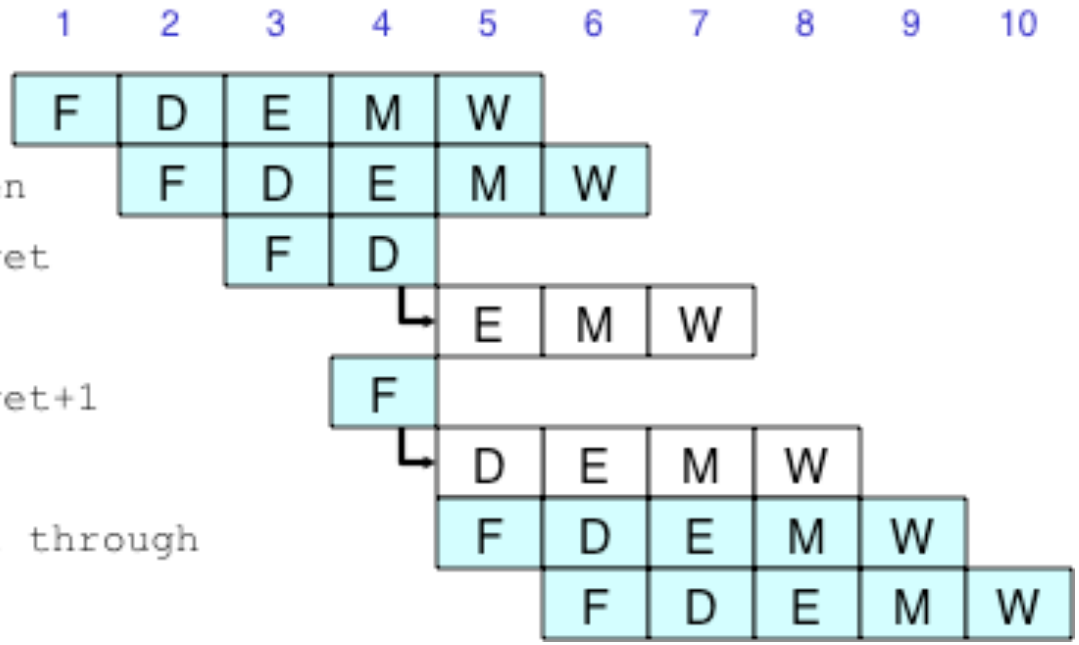
demo-j.ys

```
0x000:     xorq %rax,%rax
0x002:     jne  t                  # Not taken
0x00b:     irmovq $1, %rax         # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx         # Target
0x023:     irmovq $4, %rcx         # Should not execute
0x02d:     irmovq $5, %rdx         # Should not execute
```

- Should only execute first 8 instructions

Handling Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

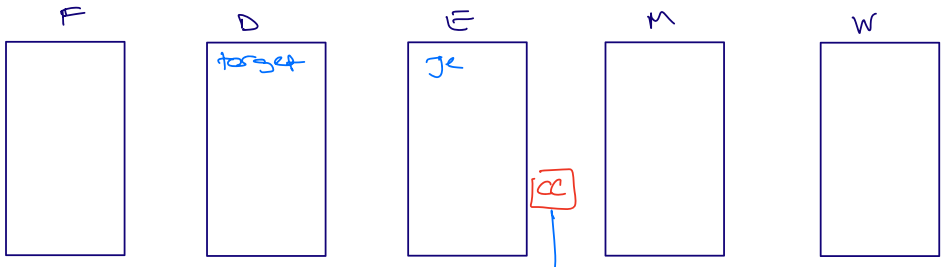


Predict branch as taken

- Fetch 2 instructions at target

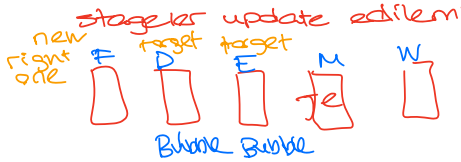
Cancel when mispredicted

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



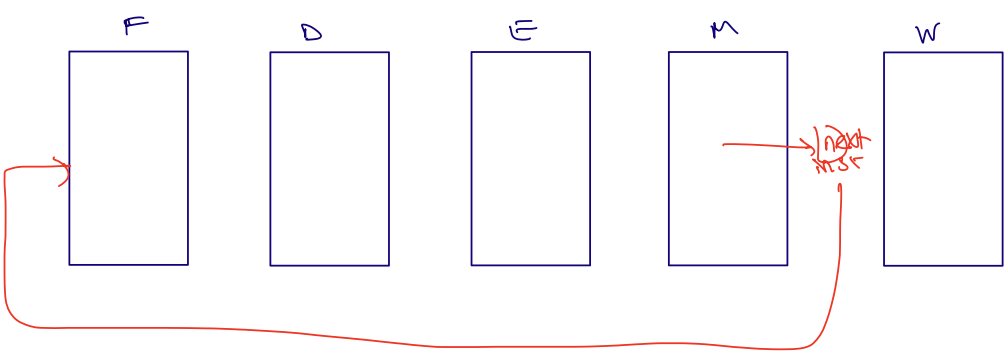
jne target

sanki condition doğruymuş gibi davranıp bir sonraki instructionu targetten fetchleyerek devam ediyorsun. → current cycle'da misprediction bulunuyor ama rising edge olmadan stageler update edilemiyor, yani



ret → bir sonraki instruction'ın ne olduğu Memory stage'de belli olacak (%rsp deki return address almak lazım)

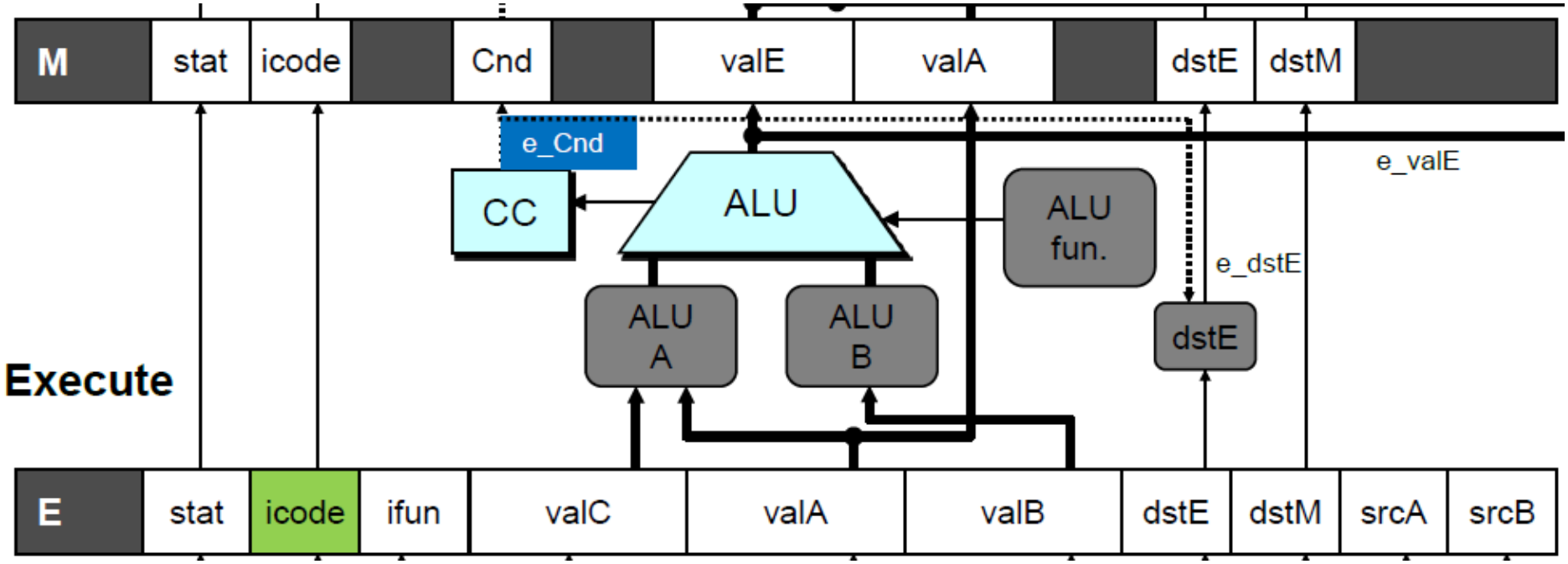
↳ insert 3 nops after return



IRET in {D_icode, E_icode, M_icode} → this way it will be triggered 3 times

↳ and it will generate 3 nops

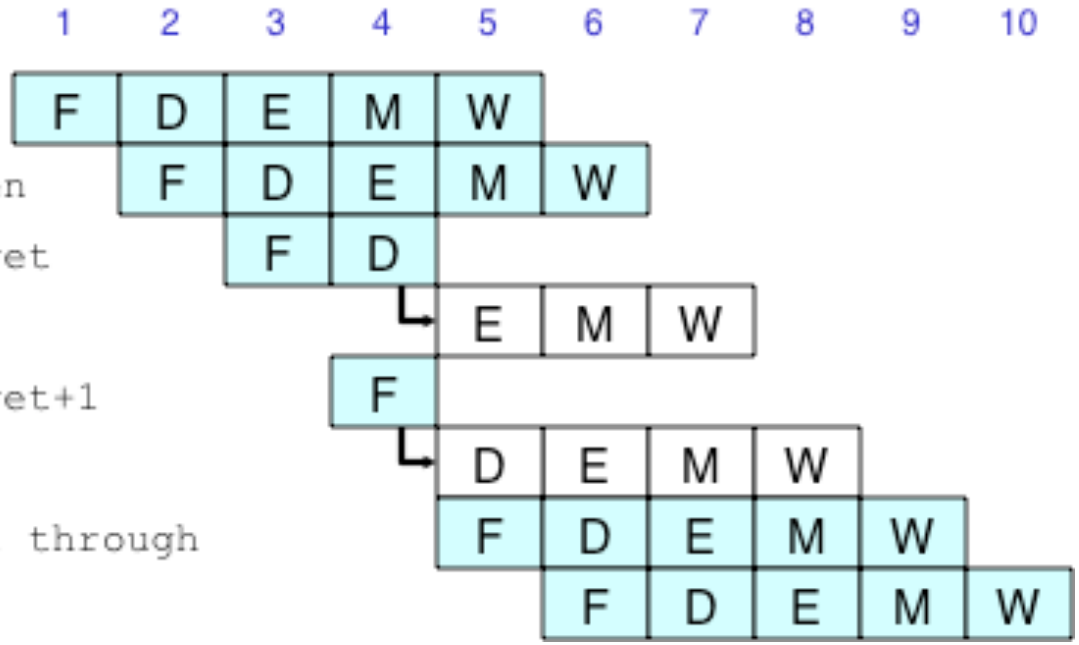
Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

Control for Misprediction

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



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

demo-retb.ys

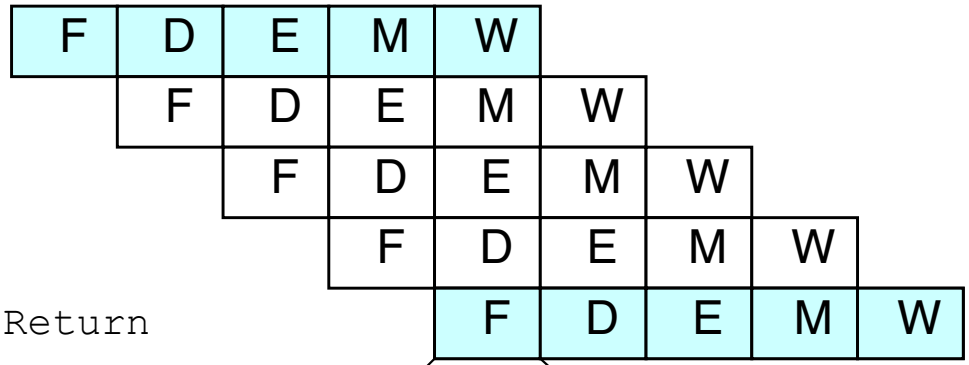
Return Example

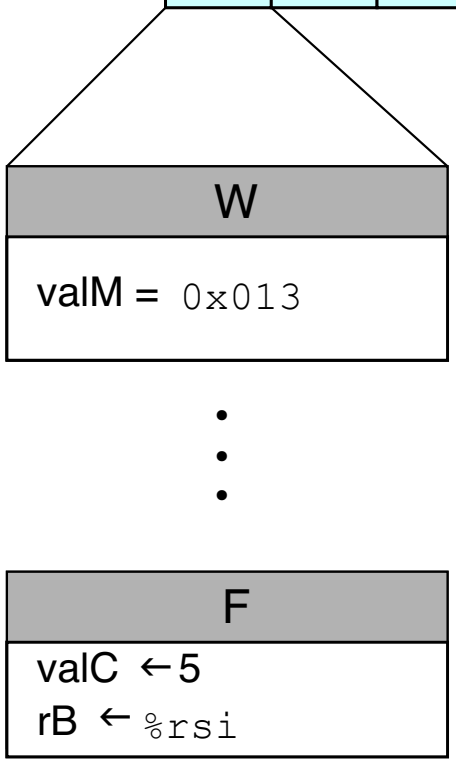
```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020:  .pos 0x20
0x020: p:  irmovq $-1,%rdi    # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100:  .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

- Previously executed three additional instructions

Correct Return Example

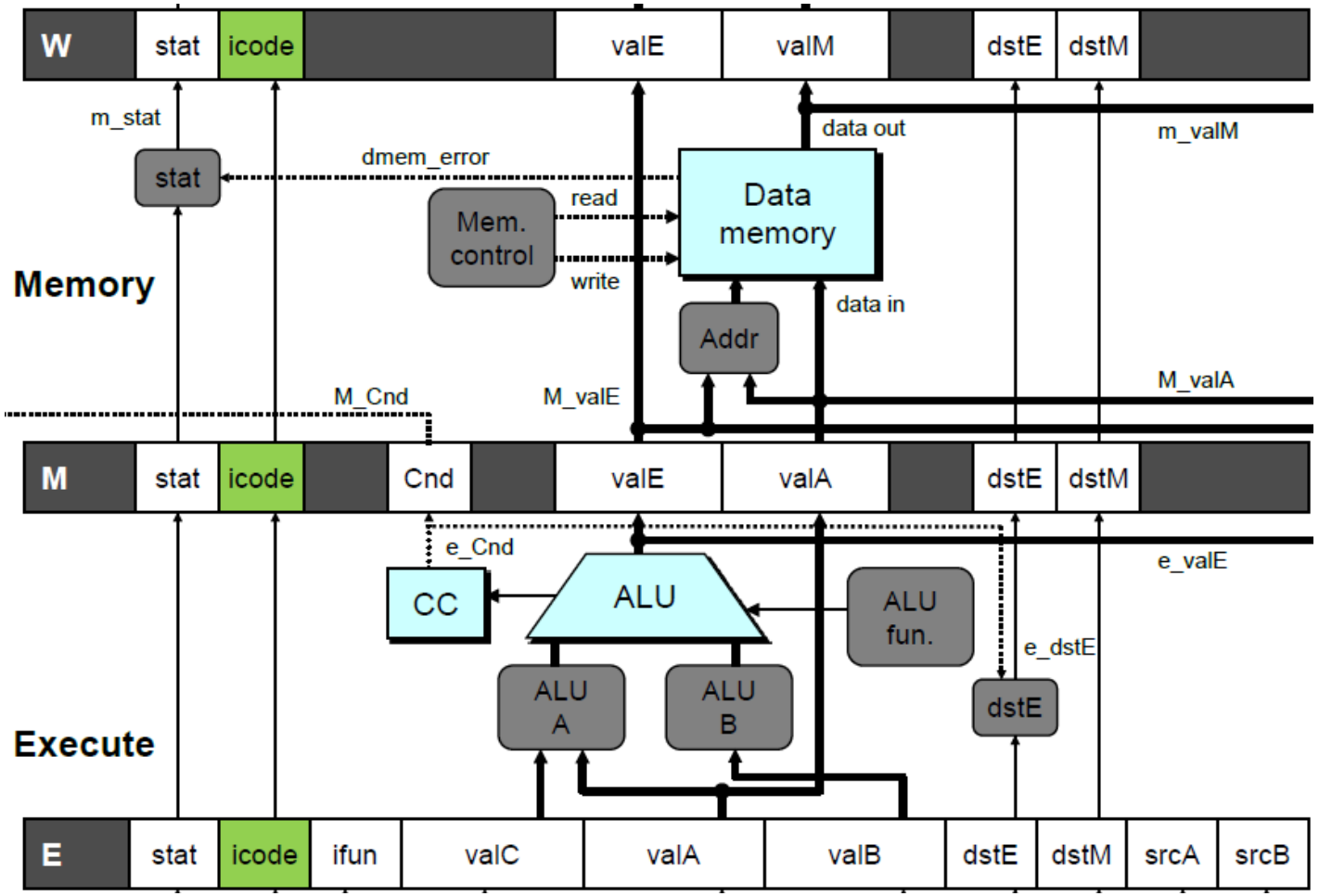
```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x013:      irmovq $5,%rsi # Return
```





- 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

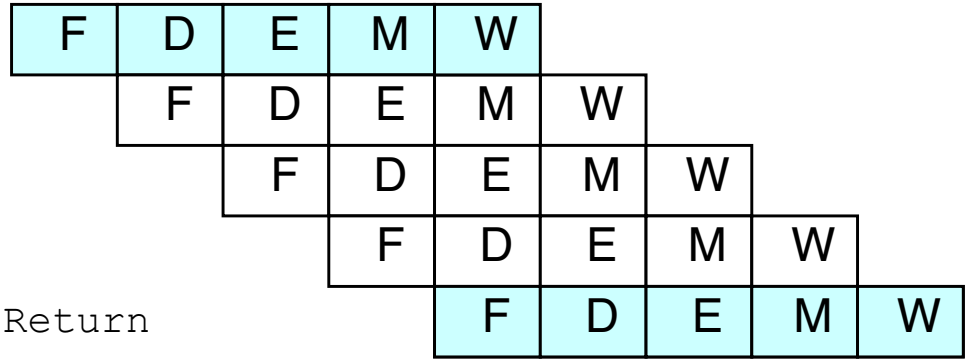
Detecting Return



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

Control for Return

```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x014:      irmovq $5,%rsi # Return
```



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

Special Control Cases

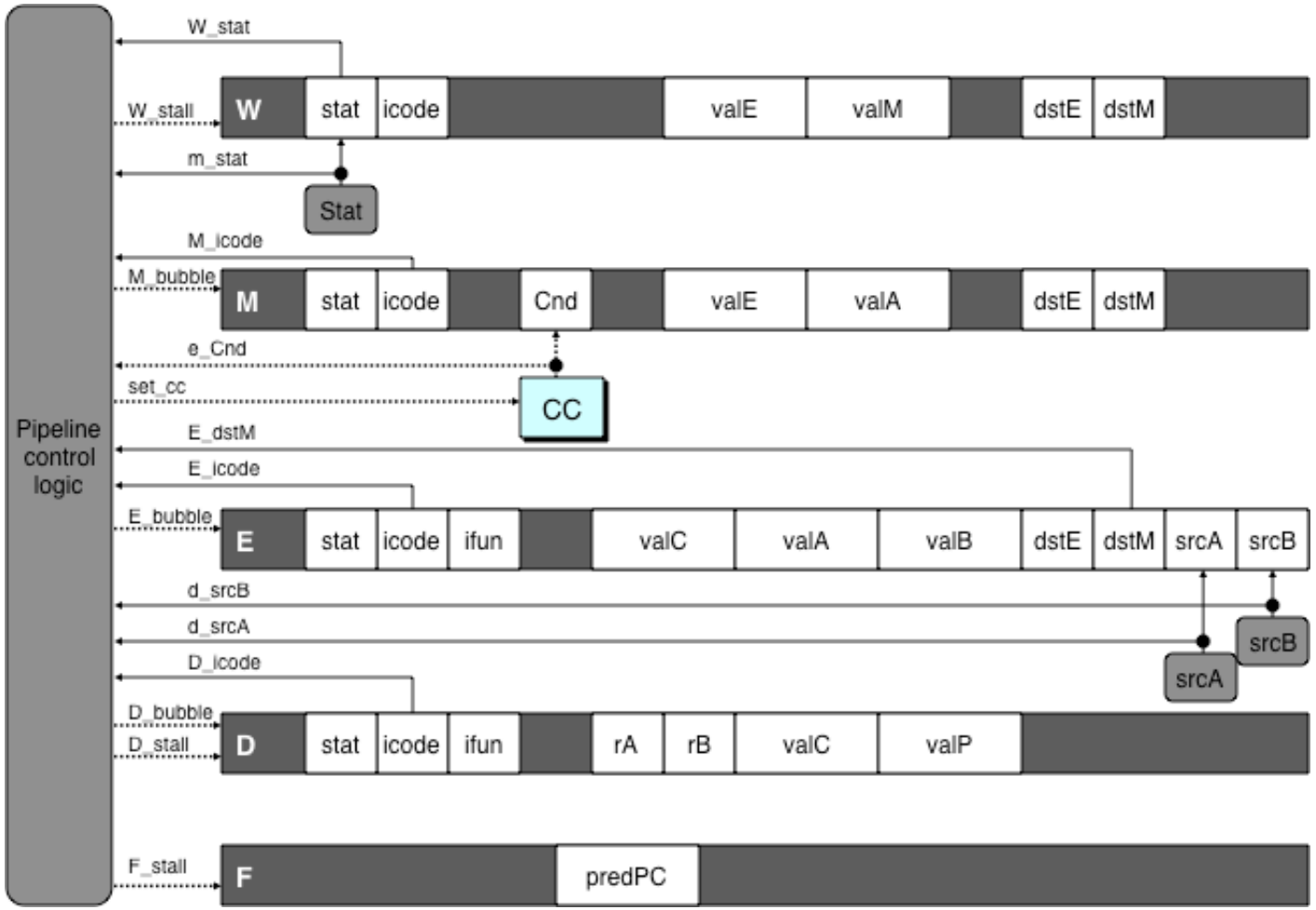
- Detection

Condition	Trigger
Processing `ret`	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && 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

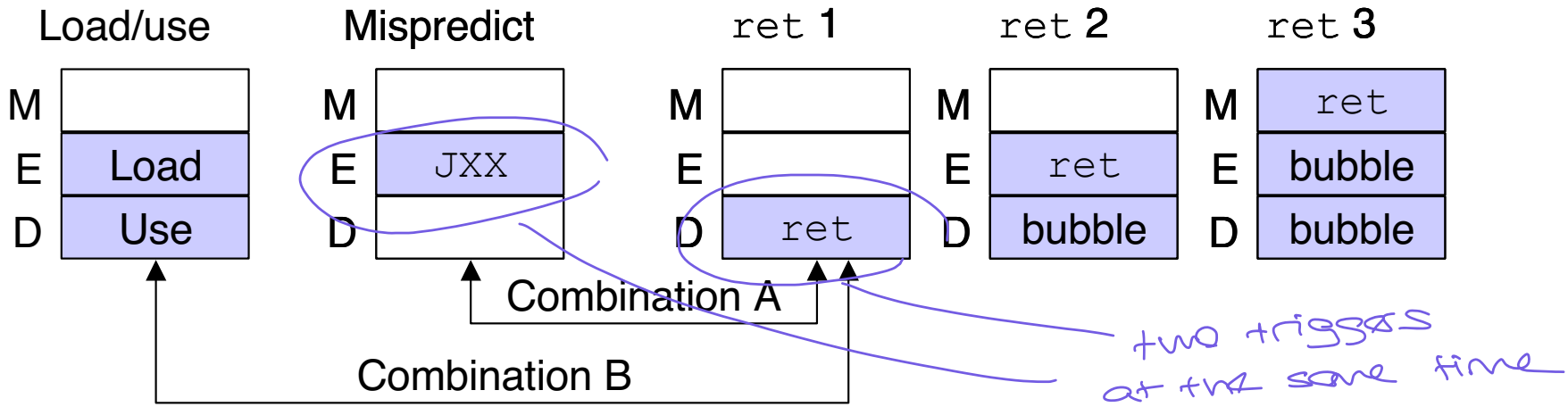


- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

Initial Version of Pipeline Control

```
bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && 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 { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # 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_Cnd) ||
    # Load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
```

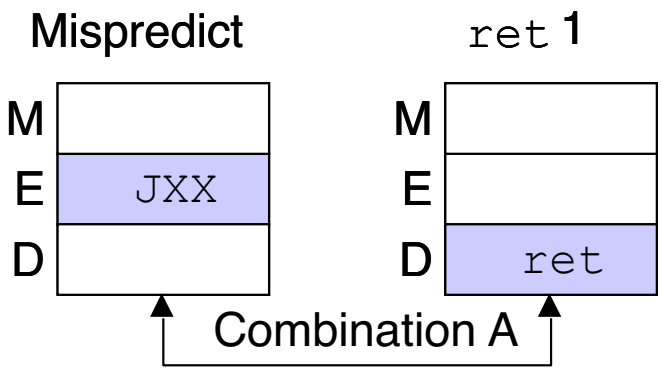
Control Combinations

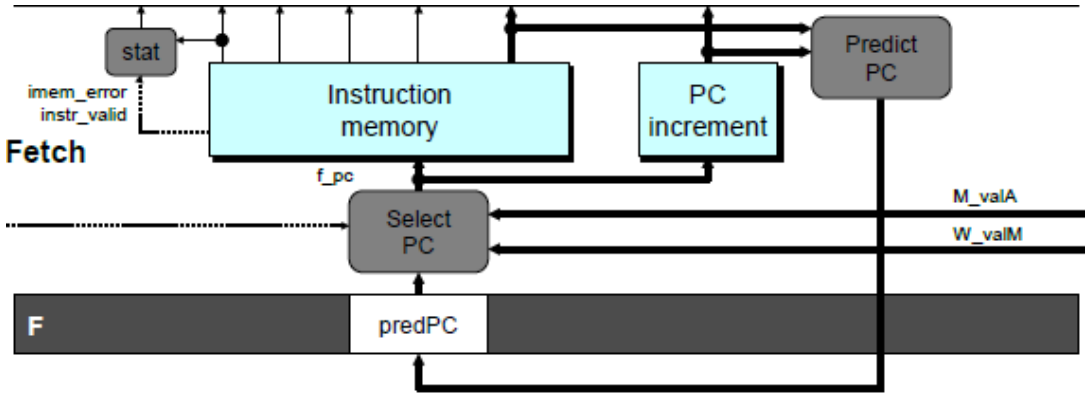


- Special cases that can arise on same clock cycle

- **Combination A**
  - Not-taken branch
  - `ret` instruction at branch target
- **Combination B**
  - Instruction that reads from memory to `%rsp`
  - Followed by `ret` instruction

Control Combination A

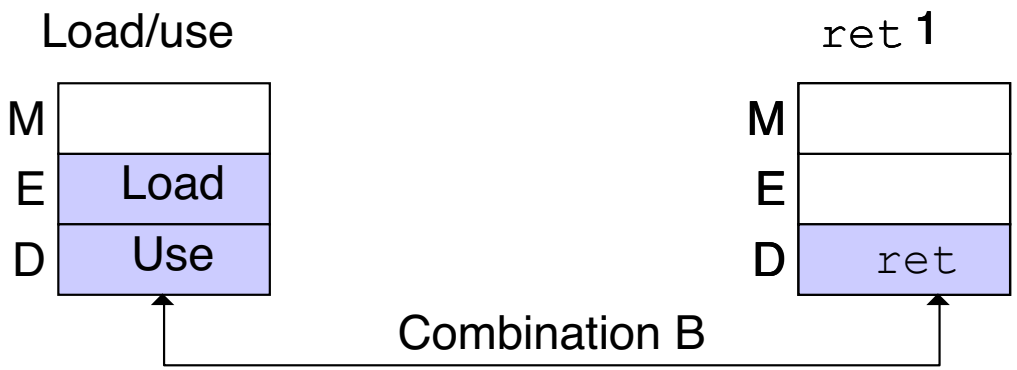




Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	*stall*	*bubble*	*bubble*	*normal*	*normal*

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

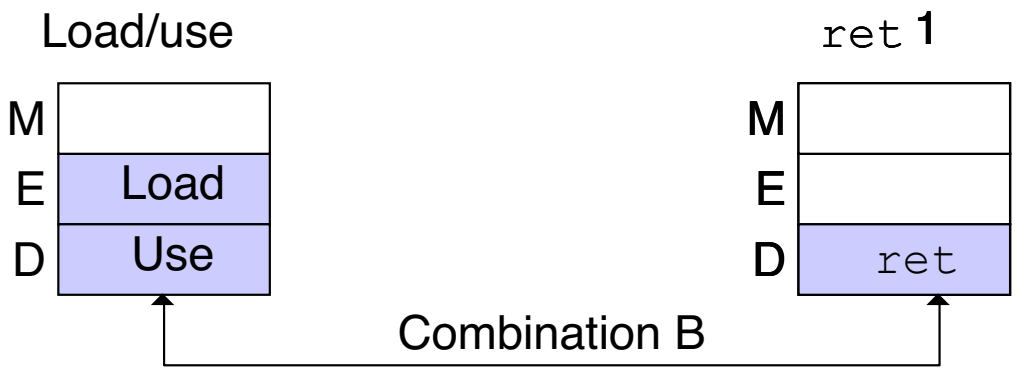
Control Combination B



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

- 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**
Combination	***stall***	***stall***	***bubble***	***normal***	***normal***

- 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_Cnd) ||
     # 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 { IMRMOVQ, IPOPQ }
            && 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
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

- 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

SEQ → PIPE –

↳ Hazards

↳ Data hazards

↳ Reading wrong register value due to late update

irmovq $5, %rax
addq %rax, %rbx

↳ Recompile, insert nops (compiler does this)

↳ Performance penalty

↳ Control Hazards

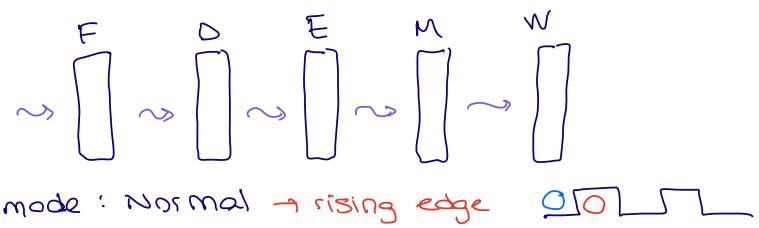
PIPE

↳ stall

↳ bubble

↓

dynamically injected nop



mode : Normal → rising edge

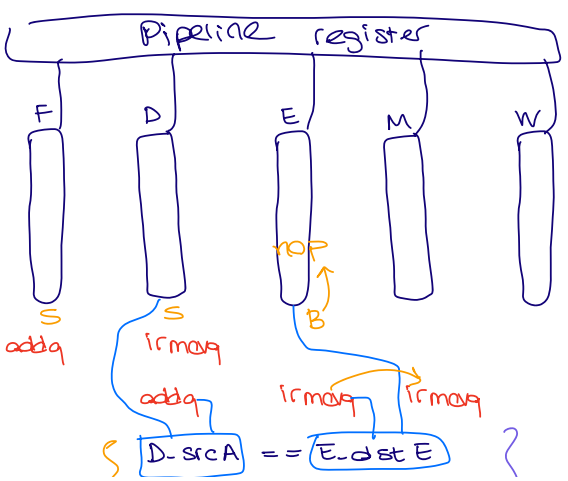
mode : Stall → Fetch the same inst. one more

↳ Data forwarding

Bubble → after the rising edge, it becomes

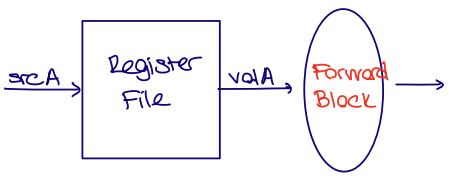
↳ It gives the current stages value to the next but it changes the current one to nop

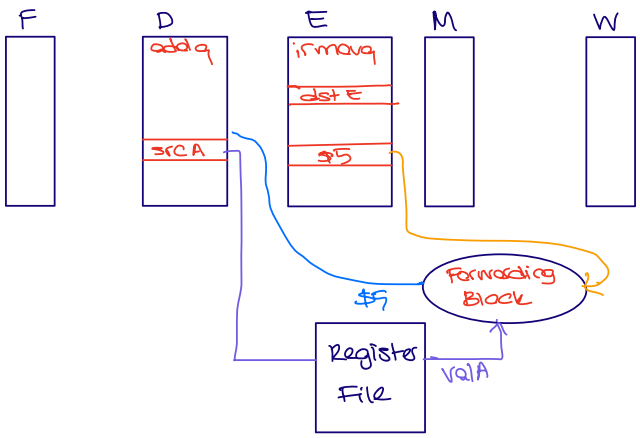
irmovq $5, %rax
addq %rax, %rbx



D_srcA == E_dstM → in pipelined register

↳ Eğer durum böyleyse

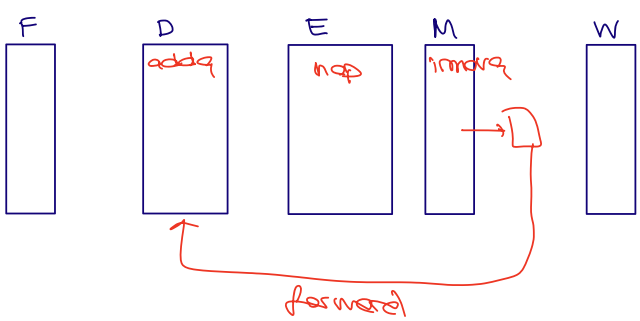




If I'm reading from the memory data forwarding won't work

↳ Load/use hazard → insert one bubble

```
mrmovq   0(%rbx), %rax
addq     %rax, %rsi
```



Processor Architecture: Wrap-up

Overview

- **Wrap-Up of PIPE Design**
  - Exceptional conditions
  - Performance analysis
  - Fetch stage design
- **Modern High-Performance Processors**
  - Out-of-order execution

Exceptions

- Conditions under which processor cannot continue normal operation

- **Causes**
  - Halt instruction (Current)
  - Bad address for instruction or data (Previous)
  - Invalid instruction (Previous)
- **Typical Desired Action**
  - Complete some instructions
    - Either current or previous (depends on exception type)
  - Discard others
  - Call exception handler
    - Like an unexpected procedure call
- **Our Implementation**
  - Halt when instruction causes exception

Exception Examples

- Detect in Fetch Stage

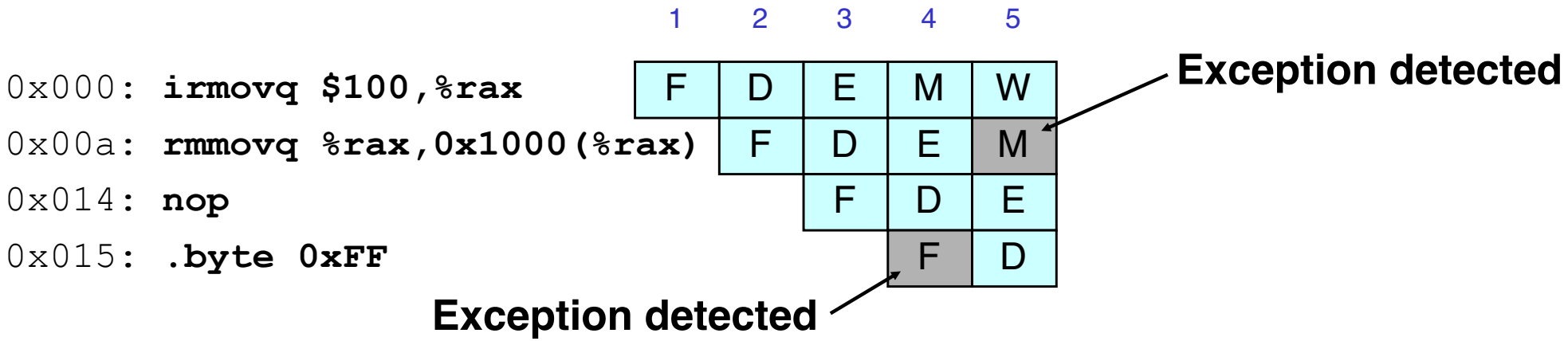
```
jmp $-1                    # Invalid jump target

.byte 0xFF                 # Invalid instruction code

halt                       # Halt instruction
```

Detect in Memory Stage

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
```

Exceptions in Pipeline Processor #1

```
# demo-exc1.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```

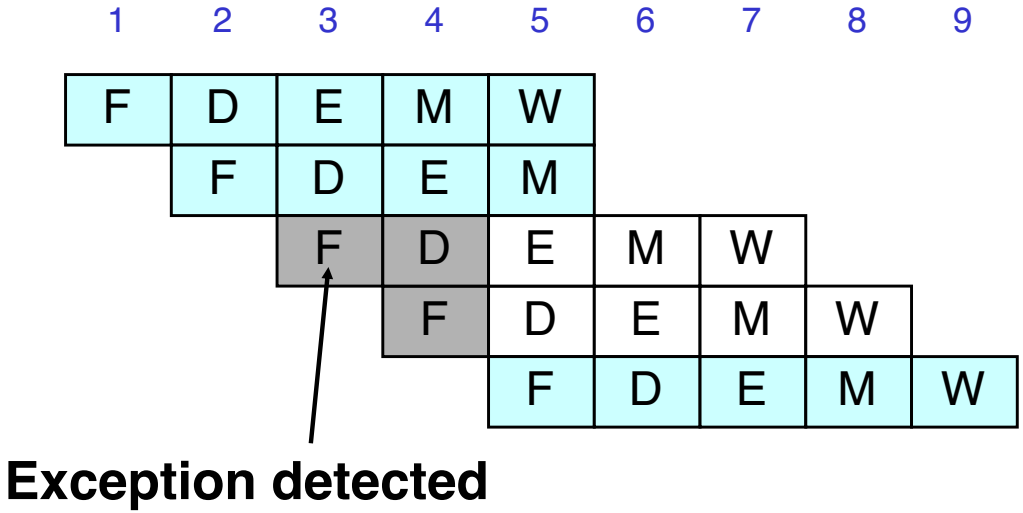


- **Desired Behavior**
  - `rmmovq` should cause exception
  - Following instructions should have no effect on processor state

Exceptions in Pipeline Processor #2

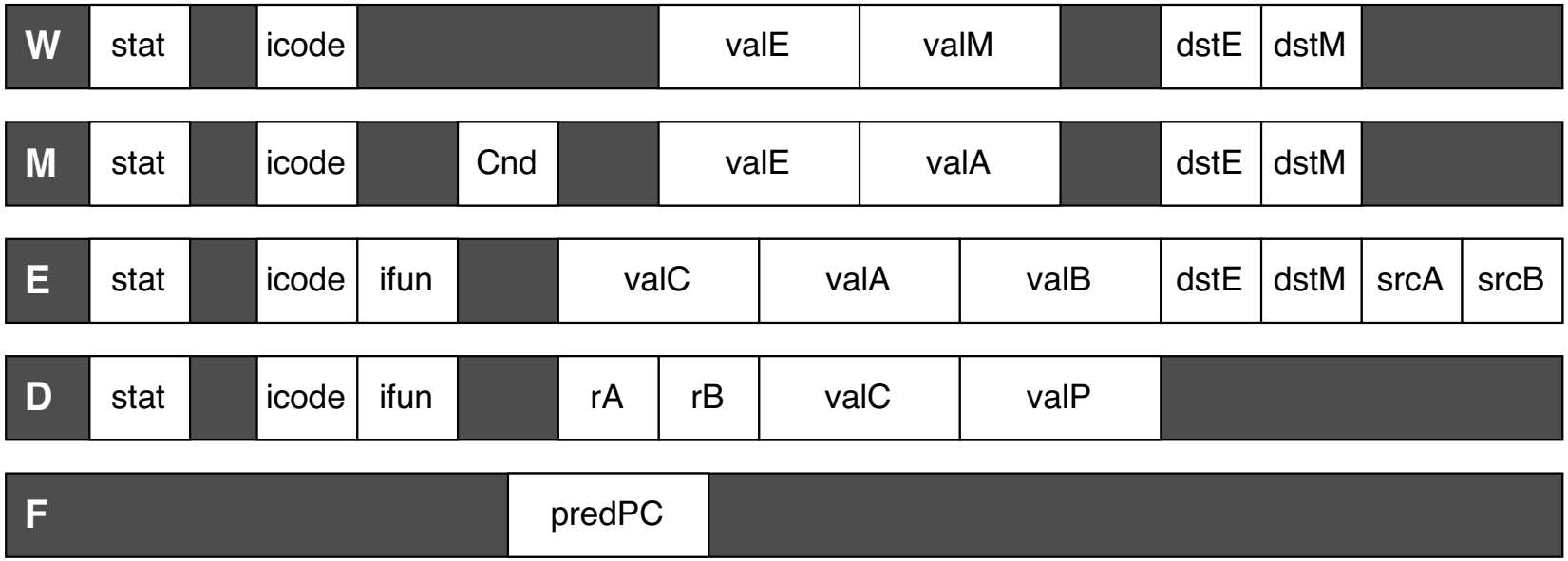
```
# demo-exc2.ys
0x000:     xorq %rax,%rax    # Set condition codes
0x002:     jne t             # Not taken
0x00b:     irmovq $1,%rax
0x015:     irmovq $2,%rdx
0x01f:     halt
0x020: t:  .byte 0xFF        # Target
```

```
0x000:     xorq %rax,%rax
0x002:     jne t
0x020: t:  .byte 0xFF
0x???: (I'm lost!)
0x00b:     irmovq $1,%rax
```



- **Desired Behavior**
  - No exception should occur

Maintaining Exception Ordering



- Add status field to pipeline registers
- Fetch stage sets to either “AOK,” “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
- Decode & execute pass values through
- Memory either passes through or sets to “ADR”
- Exception triggered only when instruction hits write back

Exception Handling Logic

- Fetch Stage

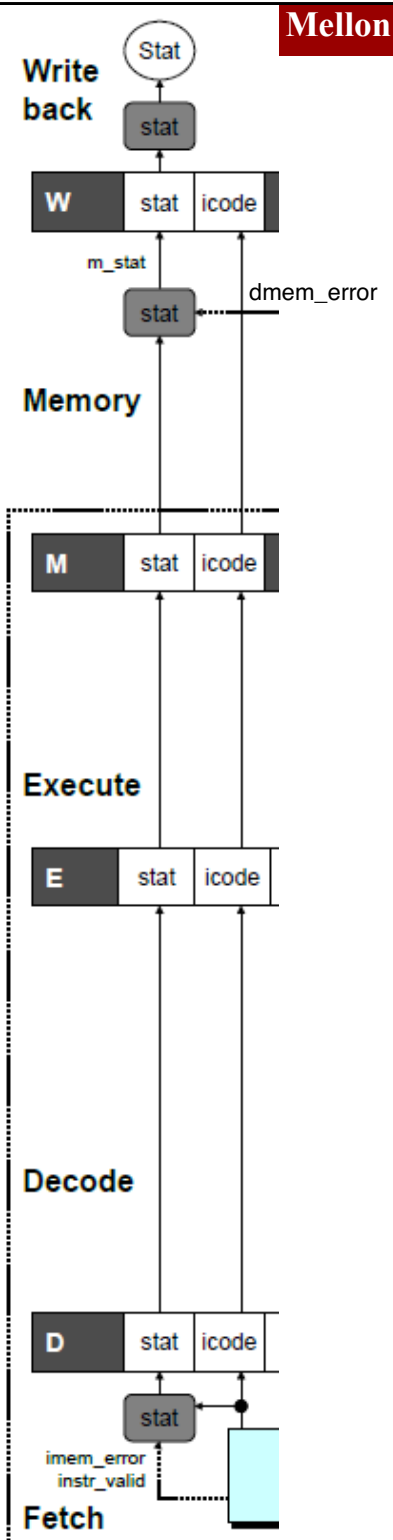
```
# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

- Memory Stage

```
# Update the status
int m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];
```

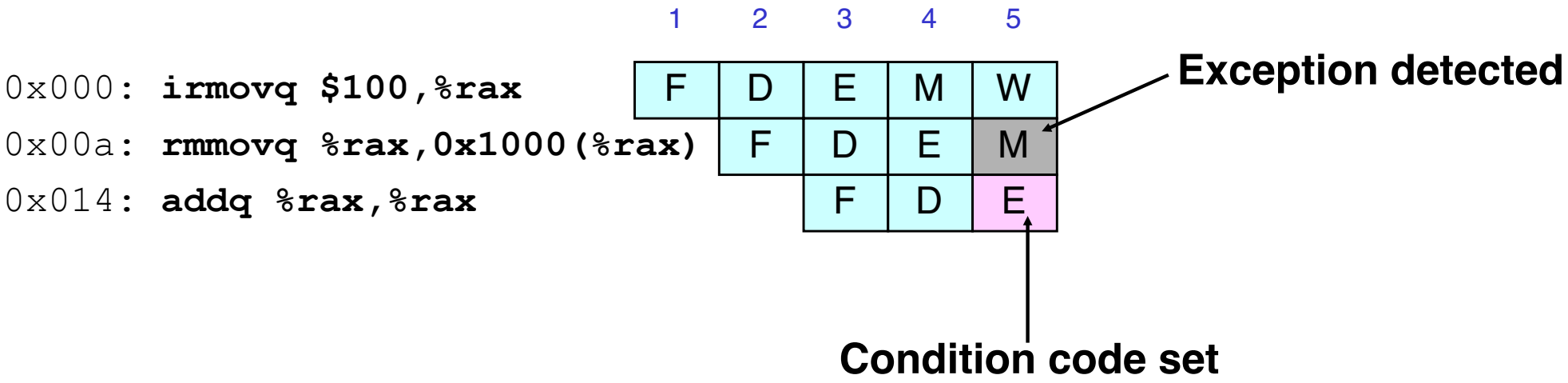
- Writeback Stage

```
int Stat = [
        # SBUB in earlier stages indicates bubble
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```



Side Effects in Pipeline Processor

```
# demo-exc3.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
addq %rax,%rax            # Sets condition codes
```



- **Desired Behavior**
  - `rmmovq` should cause exception
  - No following instruction should have any effect

Avoiding Side Effects

- **Presence of Exception Should Disable State Update**
  - Invalid instructions are converted to pipeline bubbles
    - Except have stat indicating exception status
  - Data memory will not write to invalid address
  - Prevent invalid update of condition codes
    - Detect exception in memory stage
    - Disable condition code setting in execute
    - Must happen in same clock cycle
  - Handling exception in final stages
    - When detect exception in memory stage
      - Start injecting bubbles into memory stage on next cycle
    - When detect exception in write-back stage
      - Stall excepting instruction
  - Included in HCL code

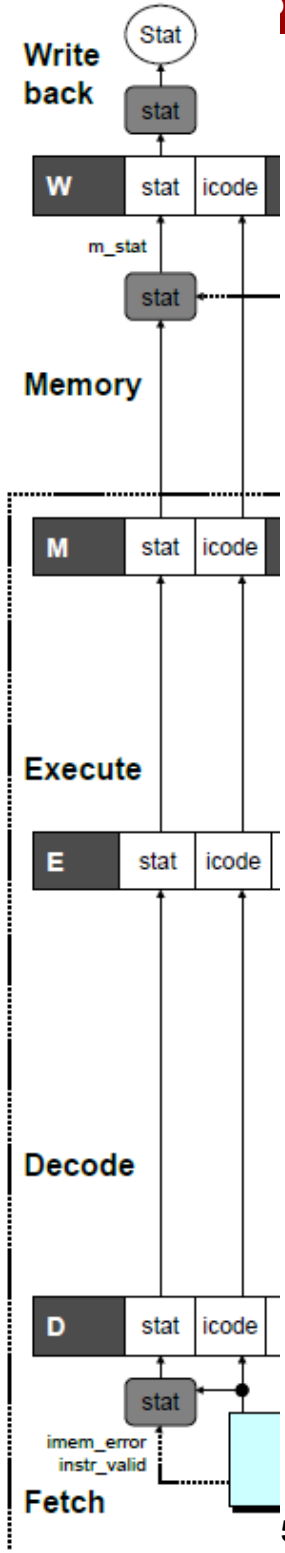
Control Logic for State Changes

- **Setting Condition Codes**

```
# Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
        # State changes only during normal operation
        !m_stat in { SADR, SINS, SHLT }
        && !W_stat in { SADR, SINS, SHLT };
```

- **Stage Control**
  - Also controls updating of memory

```
# Start injecting bubbles as soon as exception passes
through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT }
        || W_stat in { SADR, SINS, SHLT };

# Stall pipeline register W when exception encountered
bool W_stall = W_stat in { SADR, SINS, SHLT };
```

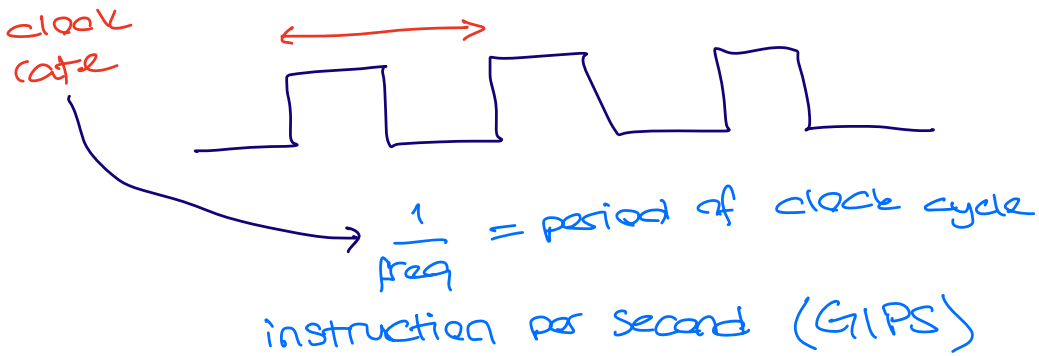


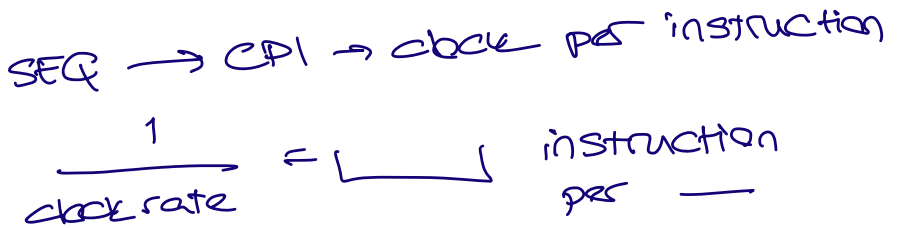
Rest of Real-Life Exception Handling

- **Call Exception Handler**
  - Push PC onto stack
    - Either PC of faulting instruction or of next instruction
    - Usually pass through pipeline along with exception status
  - Jump to handler address
    - Usually fixed address
    - Defined as part of ISA
- **Implementation**
  - Haven't tried it yet!

Performance Metrics

- **Clock rate**
  - Measured in Gigahertz
  - Function of stage partitioning and circuit design
    - Keep amount of work per stage small
- **Rate at which instructions executed**
  - CPI: cycles per instruction
  - On average, how many clock cycles does each instruction require?
  - Function of pipeline design and benchmark programs
    - E.g., how frequently are branches mispredicted?





CPI for PIPE

- **CPI ≈ 1.0**
  - Fetch instruction each clock cycle
  - Effectively process new instruction almost every cycle
    - Although each individual instruction has latency of 5 cycles
- **CPI > 1.0**
  - Sometimes must stall or cancel branches
- **Computing CPI**
  - C clock cycles
  - I instructions executed to completion
  - B bubbles injected (C = I + B)

number of cycles / # of instructions

of cycles

# of instructions

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

  - Factor B/I represents average penalty due to bubbles

CPI for PIPE (Cont.)

B/I = LP + MP + RP

Typical Values

- LP: Penalty due to load/use hazard stalling
  - Fraction of instructions that are loads — 0.25
  - Fraction of load instructions requiring stall — 0.20
  - Number of bubbles injected each time — 1

  $\Rightarrow$ LP = 0.25 * 0.20 * 1 = 0.05

- MP: Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps — 0.20
  - Fraction of cond. jumps mispredicted — 0.40
  - Number of bubbles injected each time — 2

  $\Rightarrow$ MP = 0.20 * 0.40 * 2 = 0.16

- RP: Penalty due to `ret` instructions
  - Fraction of instructions that are returns — 0.02
  - Number of bubbles injected each time — 3

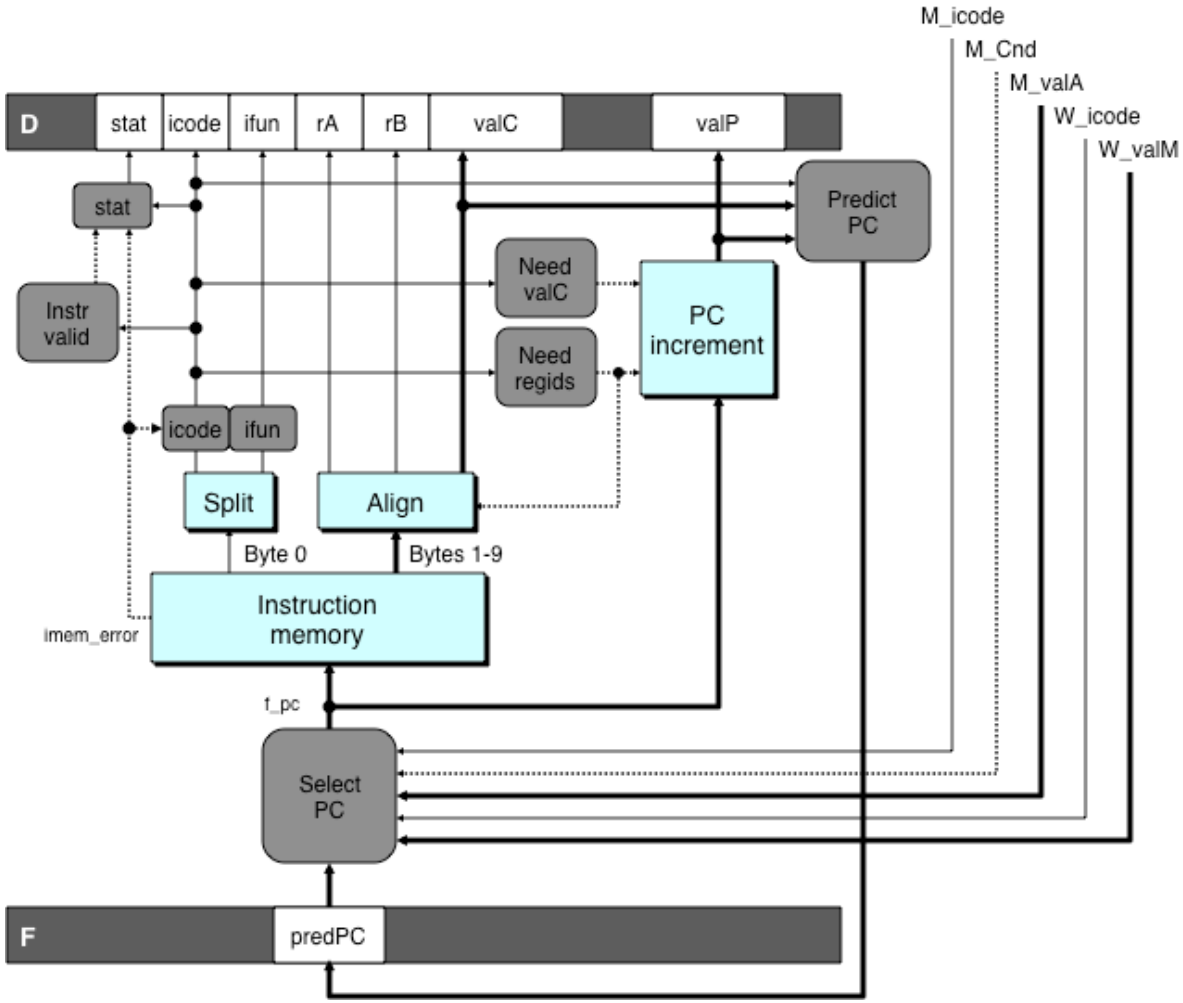
  $\Rightarrow$ RP = 0.02 * 3 = 0.06

- Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

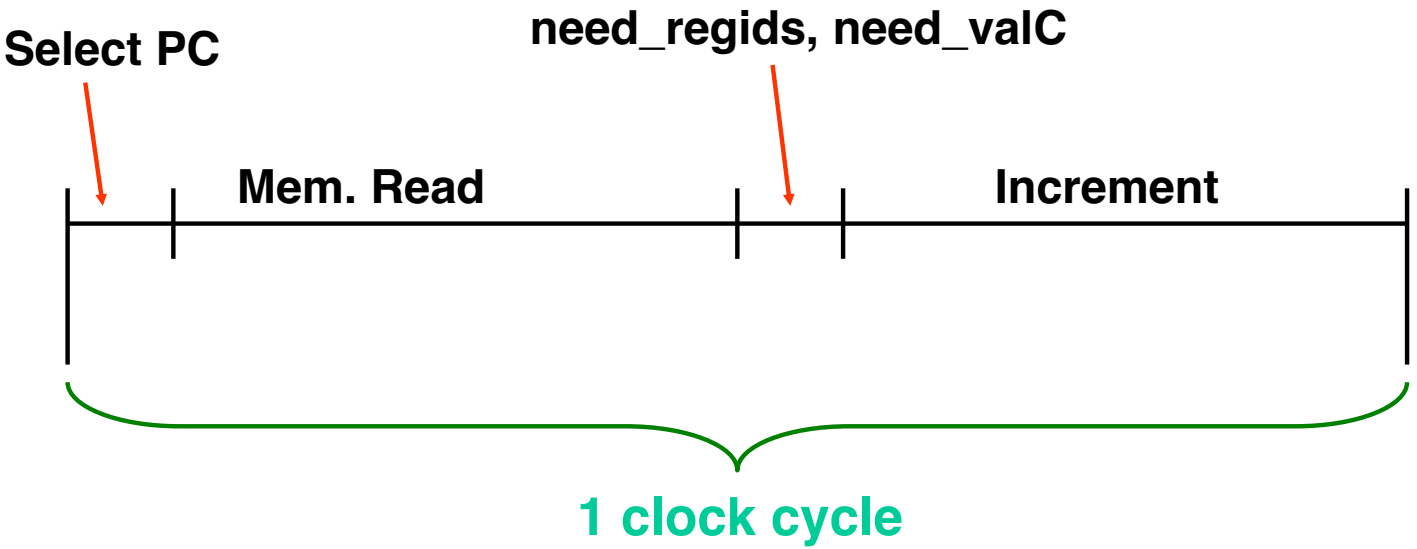
  $\Rightarrow$ CPI = 1.27 (Not bad!)

Fetch Logic Revisited

- **During Fetch Cycle**
  1. Select PC
  2. Read bytes from instruction memory
  3. Examine icode to determine instruction length
  4. Increment PC
- **Timing**
  - Steps 2 & 4 require significant amount of time

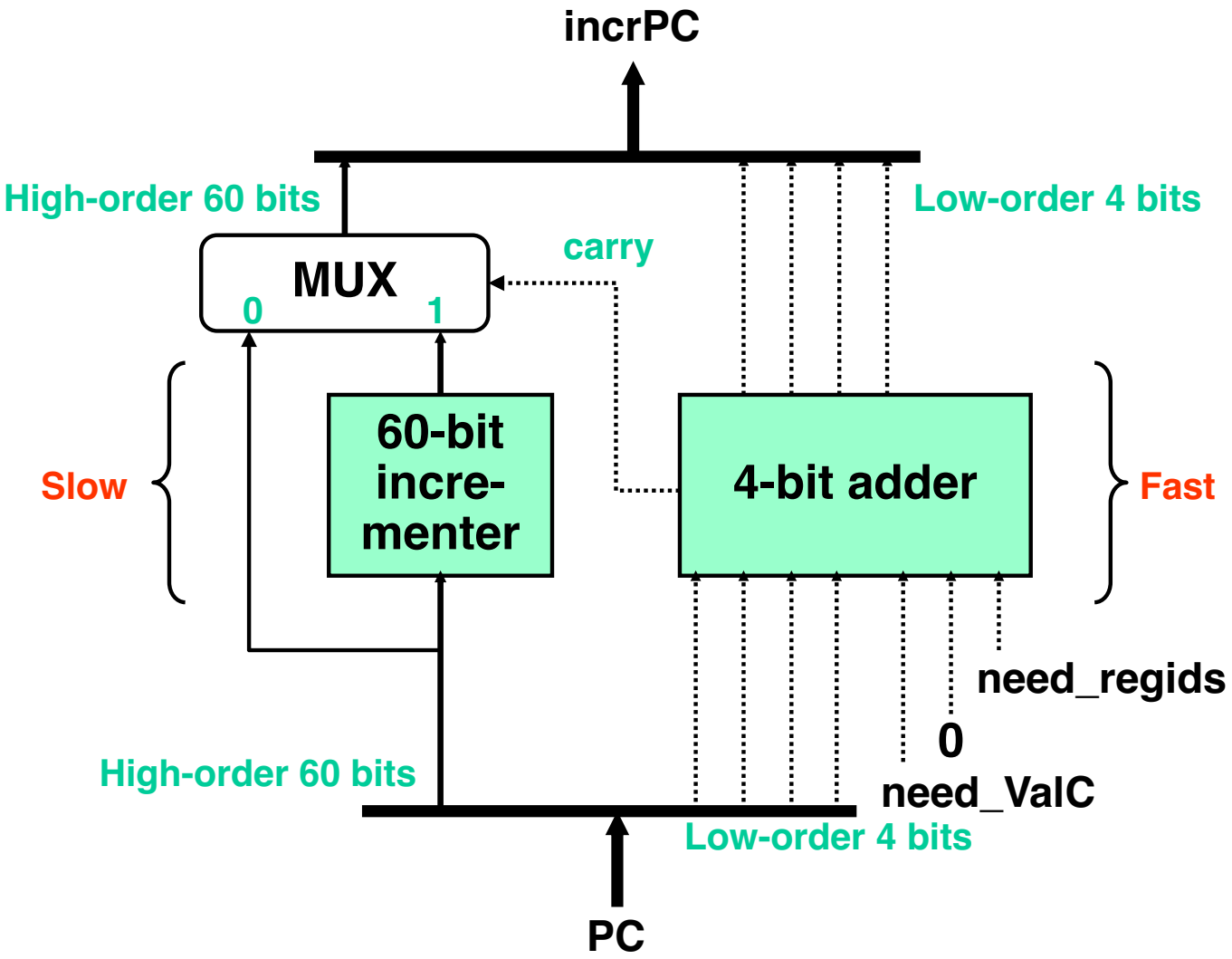


Standard Fetch Timing

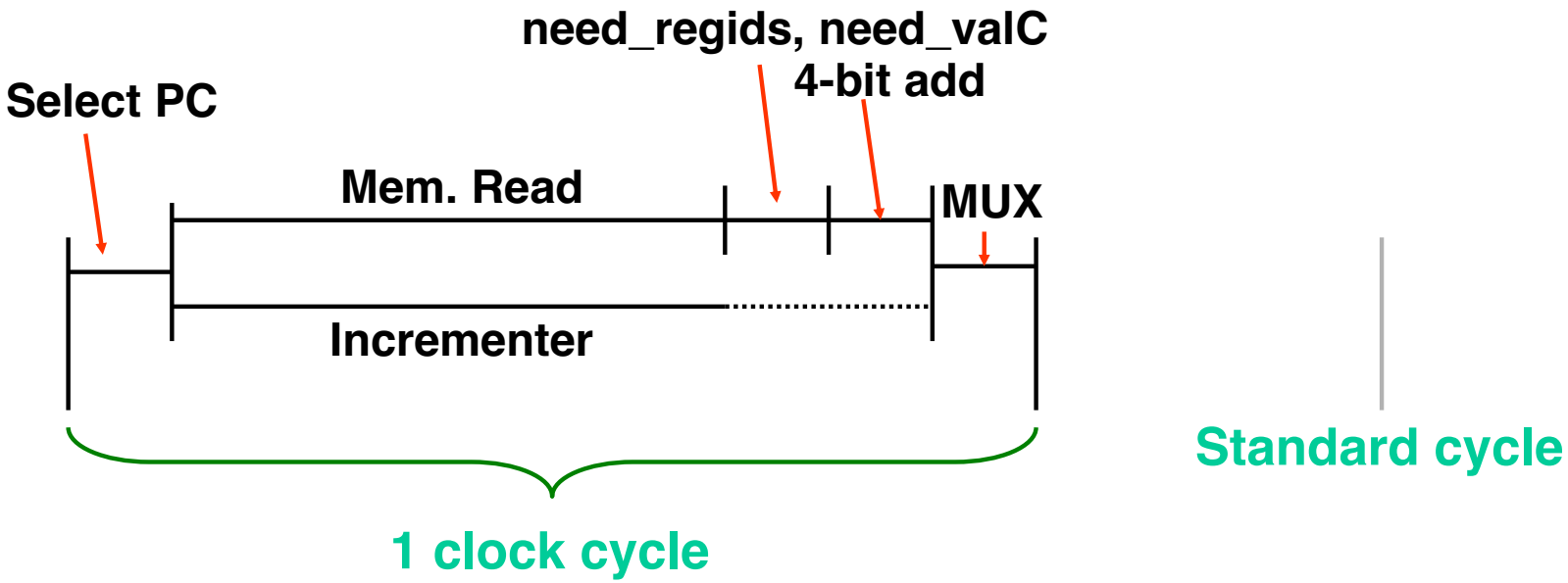


- Must Perform Everything in Sequence
- Can't compute incremented PC until know how much to increment it by

A Fast PC Increment Circuit

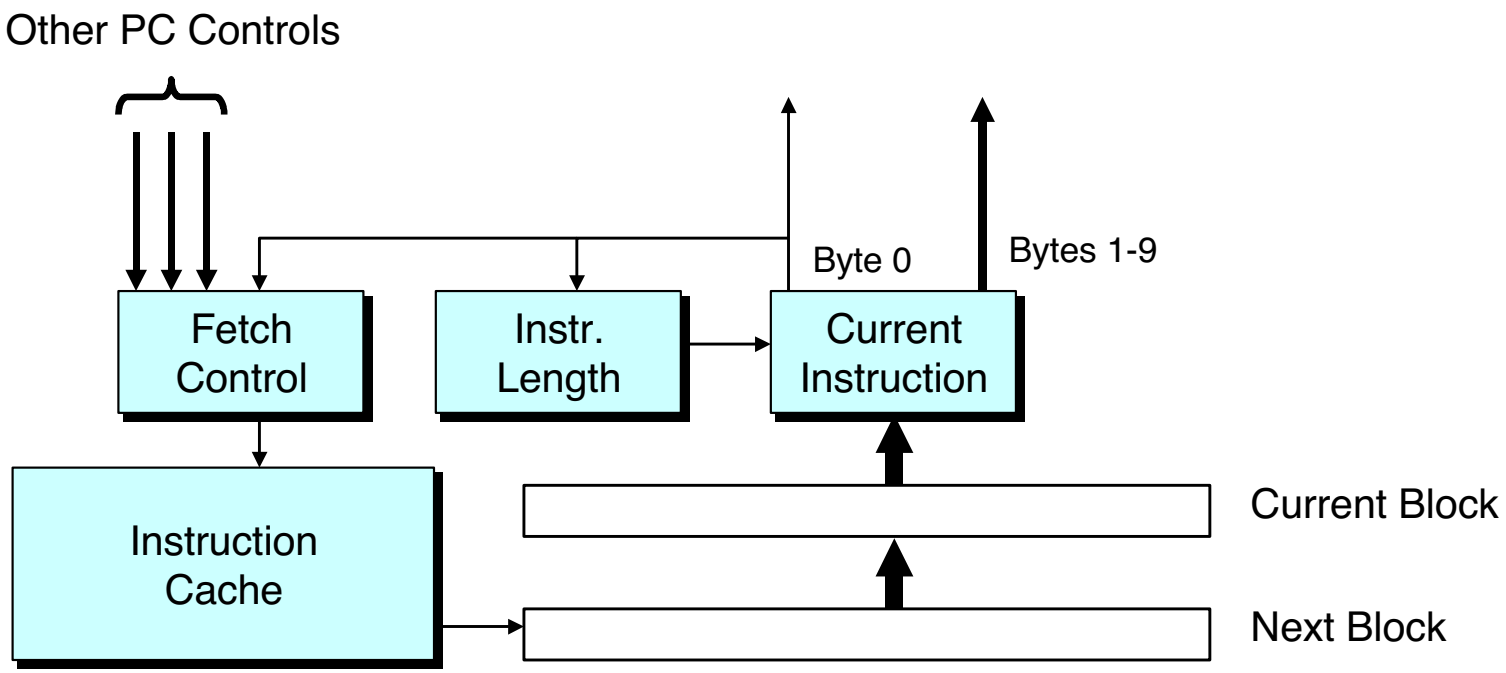


Modified Fetch Timing



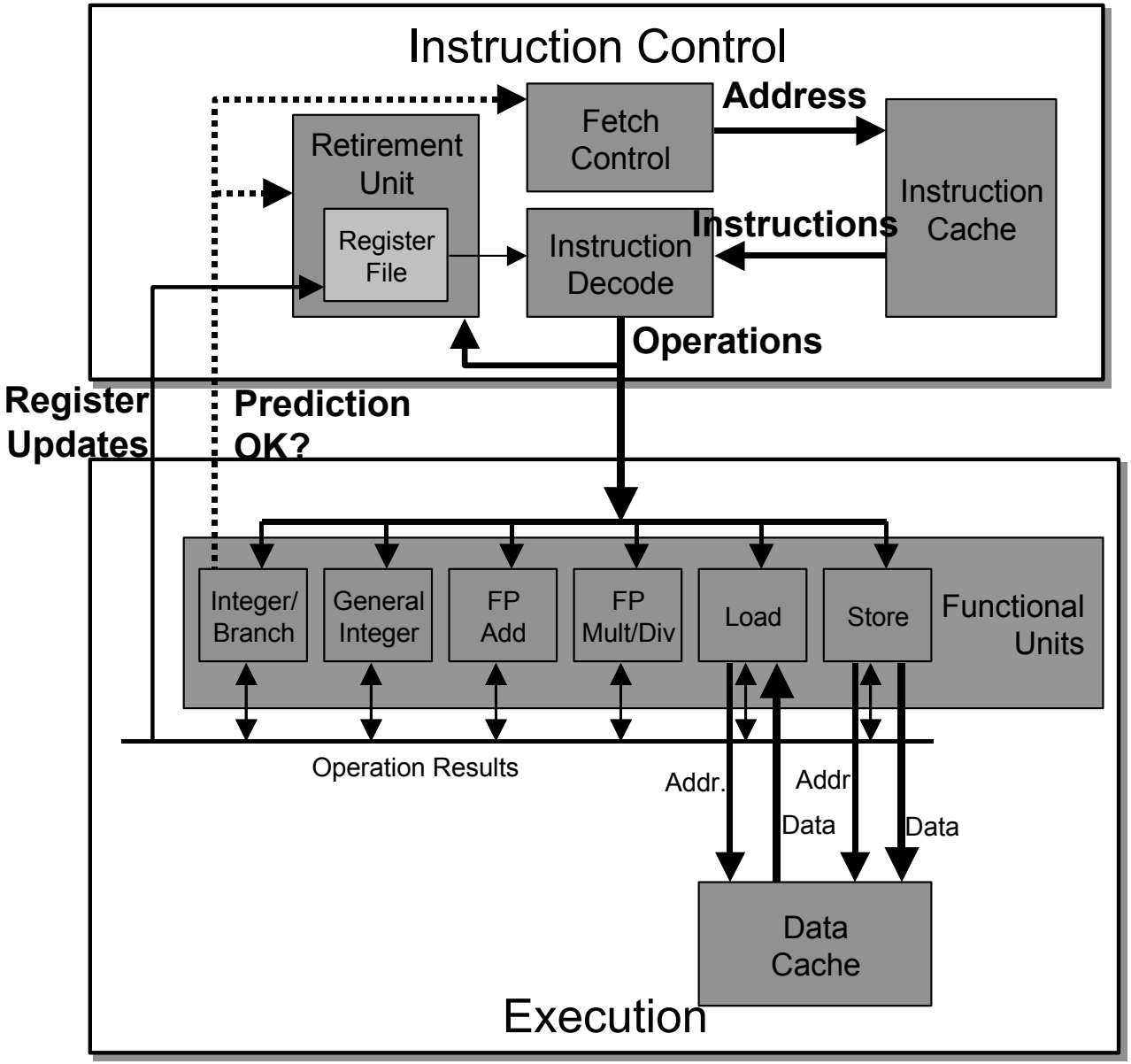
- **60-Bit Incrementer**
  - Acts as soon as PC selected
  - Output not needed until final MUX
  - Works in parallel with memory read

More Realistic Fetch Logic

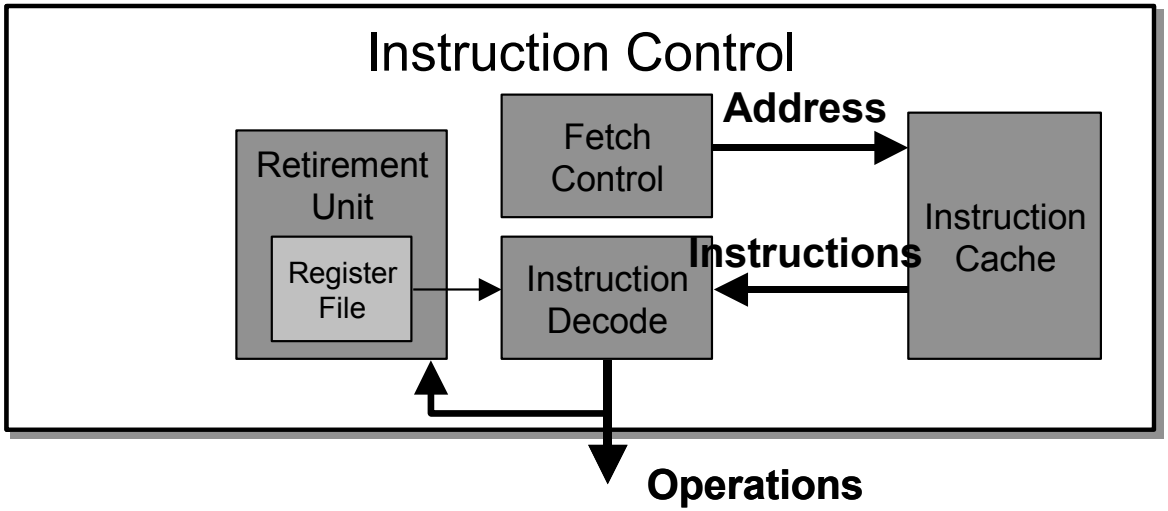


- **Fetch Box**
  - Integrated into instruction cache
  - Fetches entire cache block (16 or 32 bytes)
  - Selects current instruction from current block
  - Works ahead to fetch next block
    - As reaches end of current block
    - At branch target

Modern CPU Design

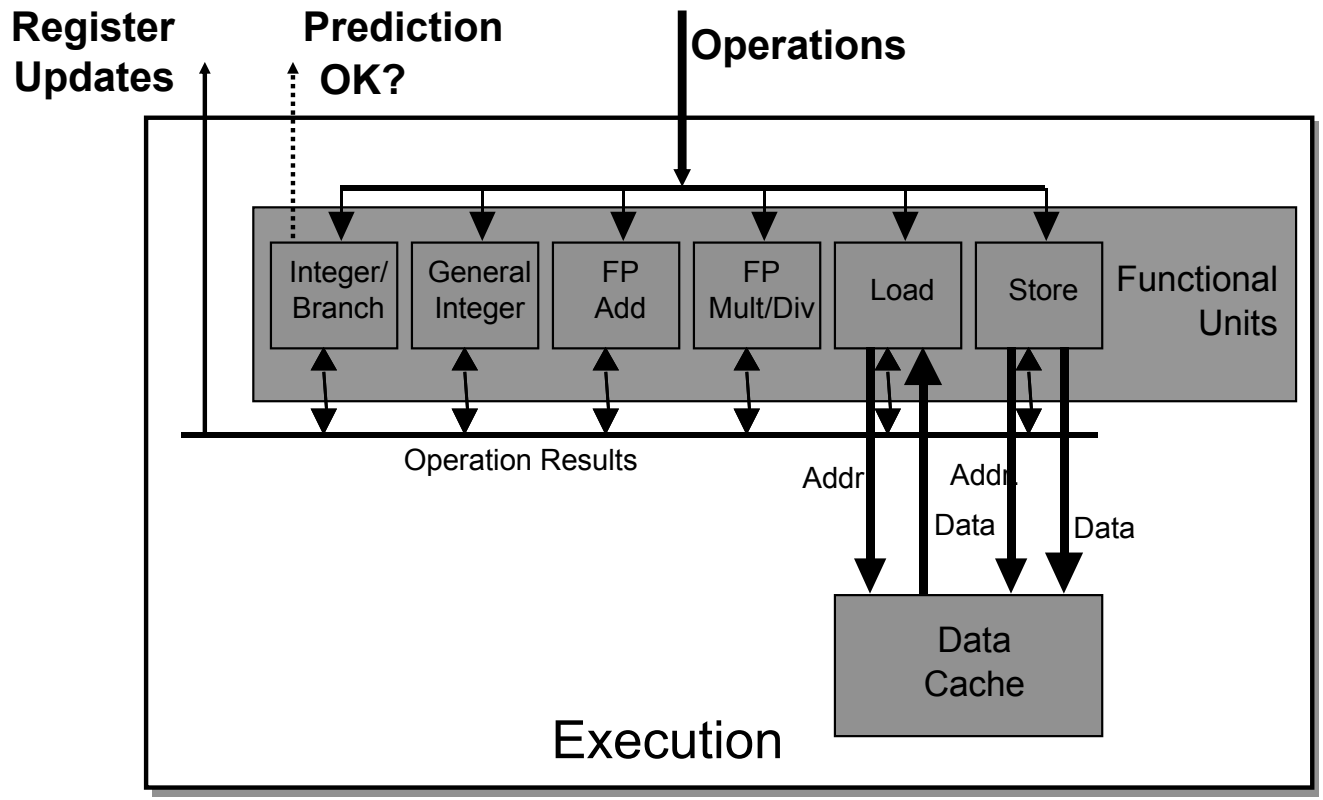


Instruction Control



- **Grabs Instruction Bytes From Memory**
  - Based on Current PC + Predicted Targets for Predicted Branches
  - Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target
- **Translates Instructions Into *Operations***
  - Primitive steps required to perform instruction
  - Typical instruction requires 1–3 operations
- **Converts Register References Into *Tags***
  - Abstract identifier linking destination of one operation with sources of later operations

Execution Unit

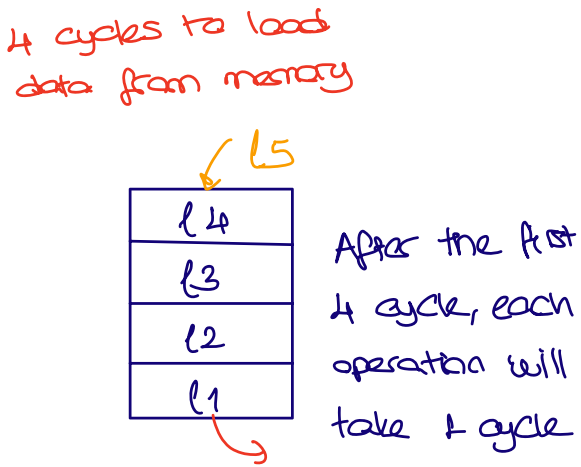


- Multiple functional units
  - Each can operate in independently
- Operations performed as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

CPU Capabilities of Intel Haswell

- **Multiple Instructions Can Execute in Parallel**
  - 2 load
  - 1 store
  - 4 integer
  - 2 FP multiply
  - 1 FP add / divide
- **Some Instructions Take > 1 Cycle, but Can be Pipelined**

Instruction	Latency	Cycles/Issue
Load / Store	4	1
Integer Multiply	3	1
Integer Divide	3—30	3—30
Double/Single FP Multiply	5	1
Double/Single FP Add	3	1
Double/Single FP Divide	10—15	6—11



Haswell Operation

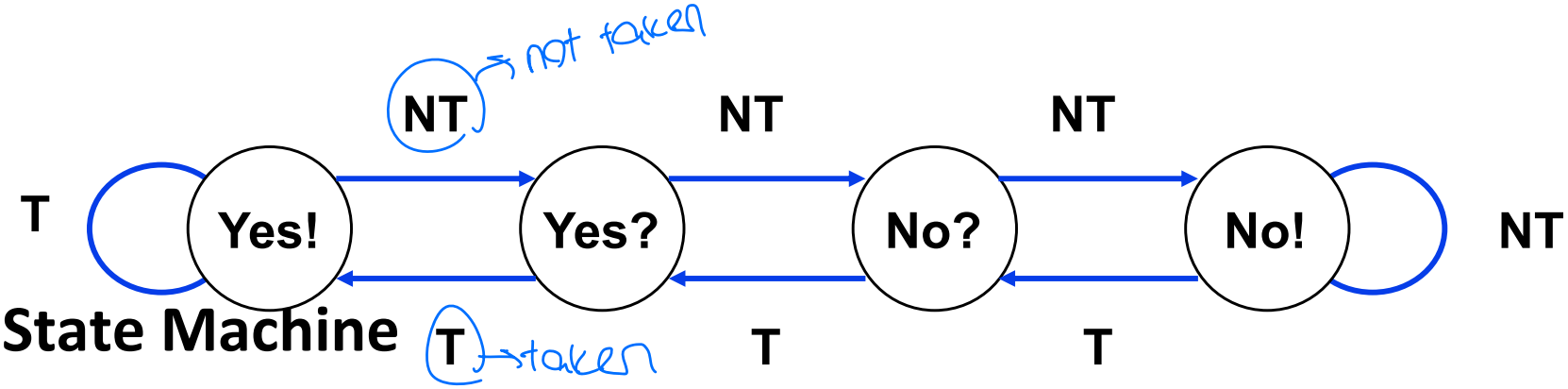
- **Translates instructions dynamically into "Uops"**
  - ~118 bits wide
  - Holds operation, two sources, and destination
- **Executes Uops with "Out of Order" engine**
  - Uop executed when
    - Operands available
    - Functional unit available
  - Execution controlled by "Reservation Stations"
    - Keeps track of data dependencies between uops
    - Allocates resources

High-Performance Branch Prediction

- **Critical to Performance**
  - Typically 11–15 cycle penalty for misprediction
- **Branch Target Buffer**
  - 512 entries
  - 4 bits of history
  - Adaptive algorithm
    - Can recognize repeated patterns, e.g., alternating taken–not taken
- **Handling BTB misses**
  - Detect in ~cycle 6
  - Predict taken for negative offset, not taken for positive
    - Loops vs. conditionals

Example Branch Prediction

- **Branch History**
  - Encode information about prior history of branch instructions
  - Predict whether or not branch will be taken



- **State Machine**
  - Each time branch taken, transition to right
  - When not taken, transition to left
  - Predict branch taken when in state Yes! or Yes?

Processor Summary

- **Design Technique**
  - Create uniform framework for all instructions
    - Want to share hardware among instructions
  - Connect standard logic blocks with bits of control logic
- **Operation**
  - State held in memories and clocked registers
  - Computation done by combinational logic
  - Clocking of registers/memories sufficient to control overall behavior
- **Enhancing Performance**
  - Pipelining increases throughput and improves resource utilization
  - Must make sure to maintain ISA behavior