Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

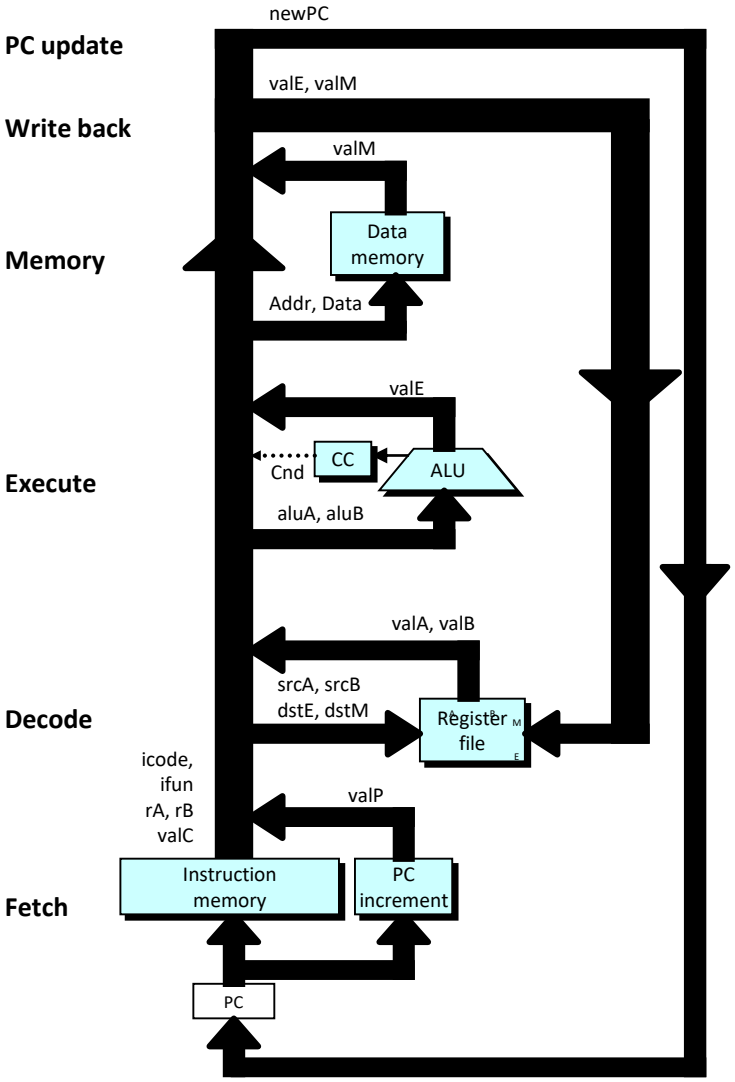
Systems Software &
Architecture Lab.

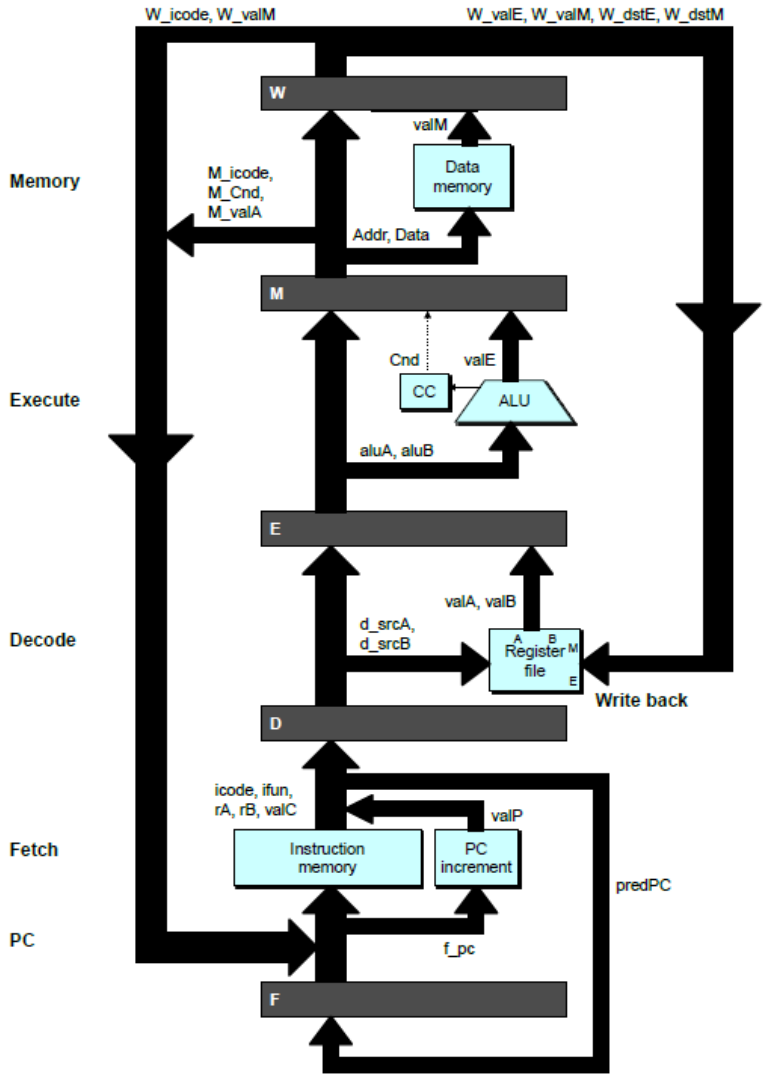
Seoul National University

Spring 2019

PIPE: Pipelined Y86-64 Implementation

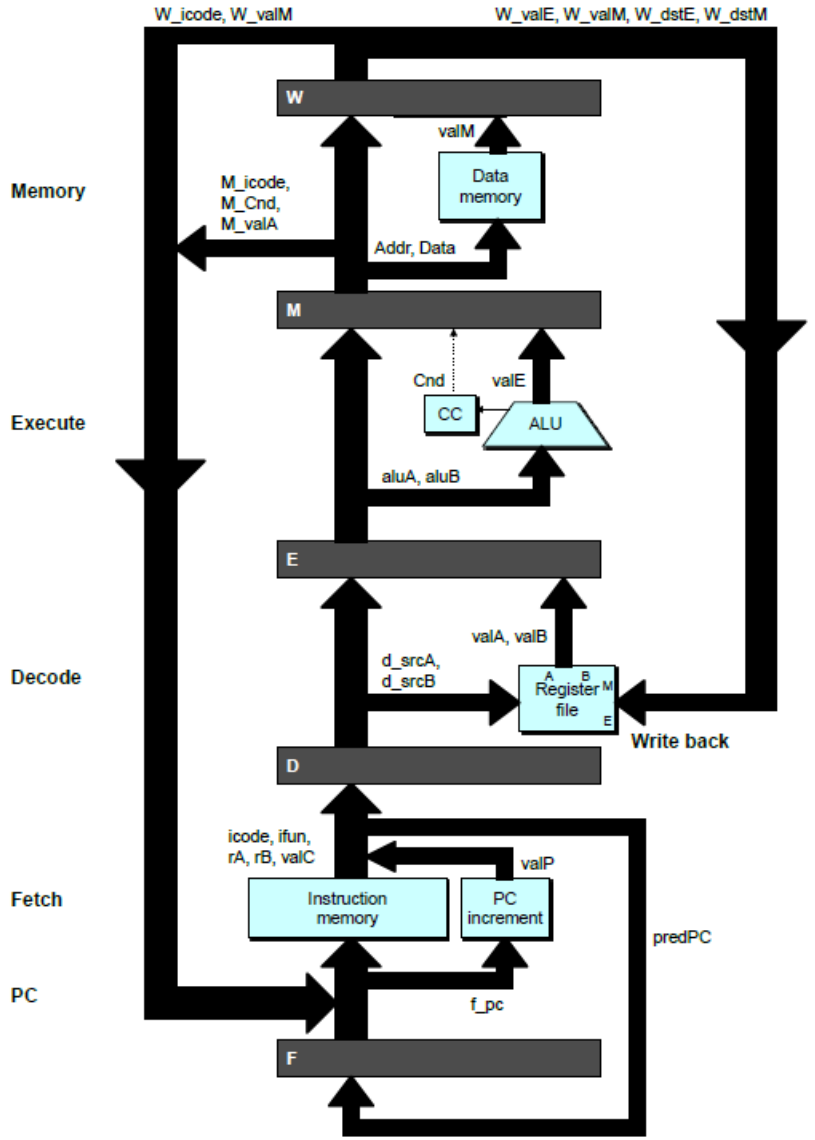
Adding Pipeline Registers





Pipeline Stages

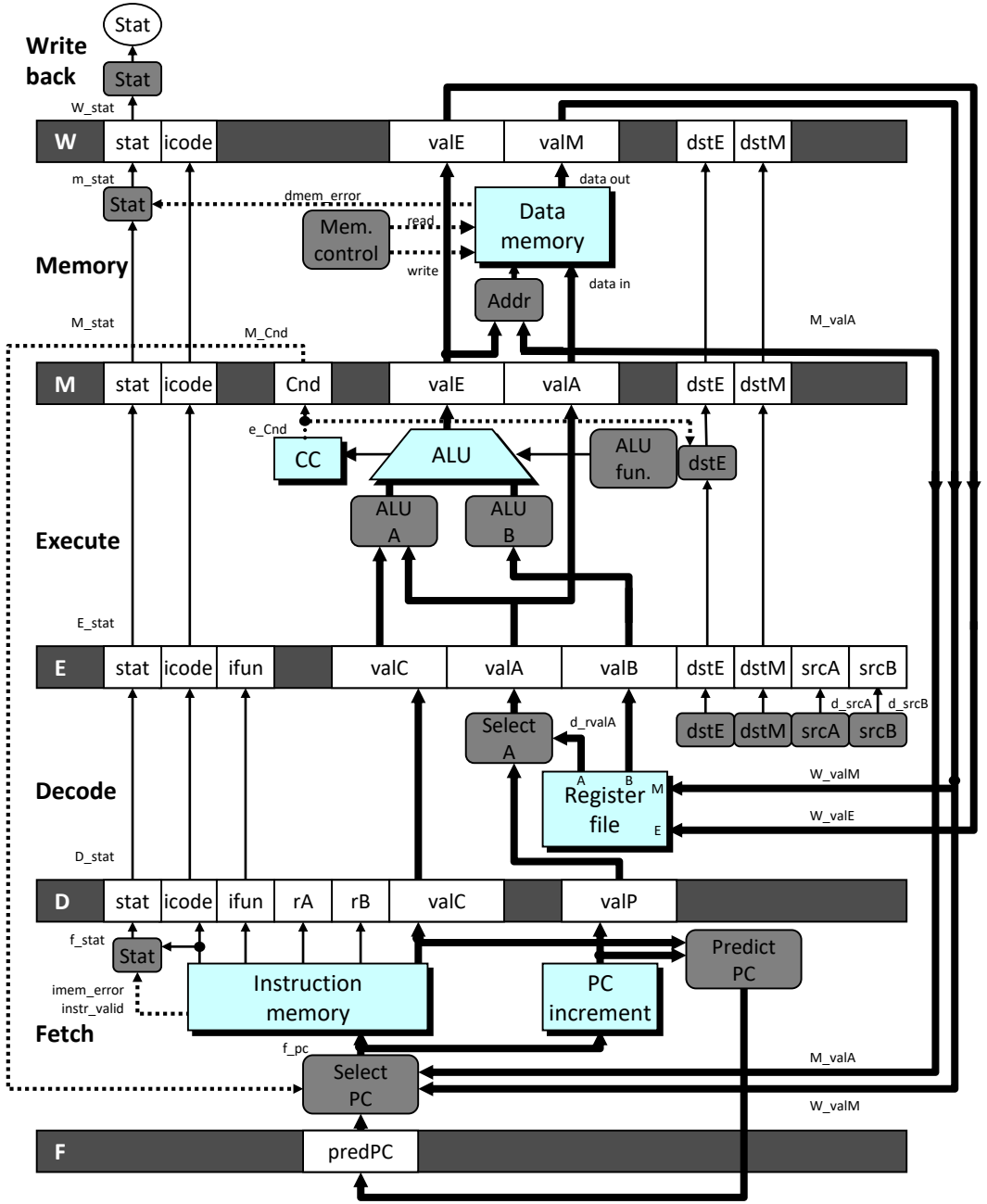
- **Fetch** — Select current PC
Read instruction
Compute incremented PC
- **Decode** — Read program registers
- **Execute** — Operate ALU
- **Memory** — Read or write data memory
- **Write Back** — Update register file



PIPE- Hardware

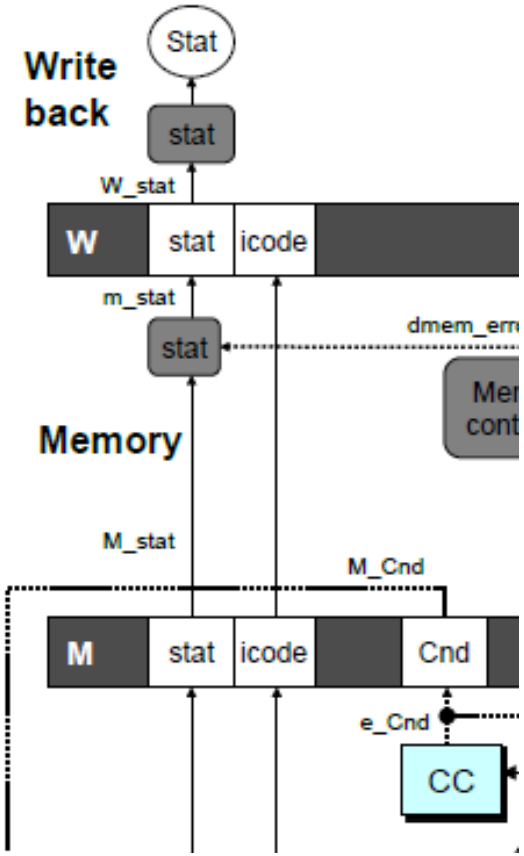
- Pipelined registers hold intermediate values from instruction execution

- Forward (Upward) paths
  - Values passed from one stage to next
  - Cannot jump past stages
    - e.g., `valC` passes through decode



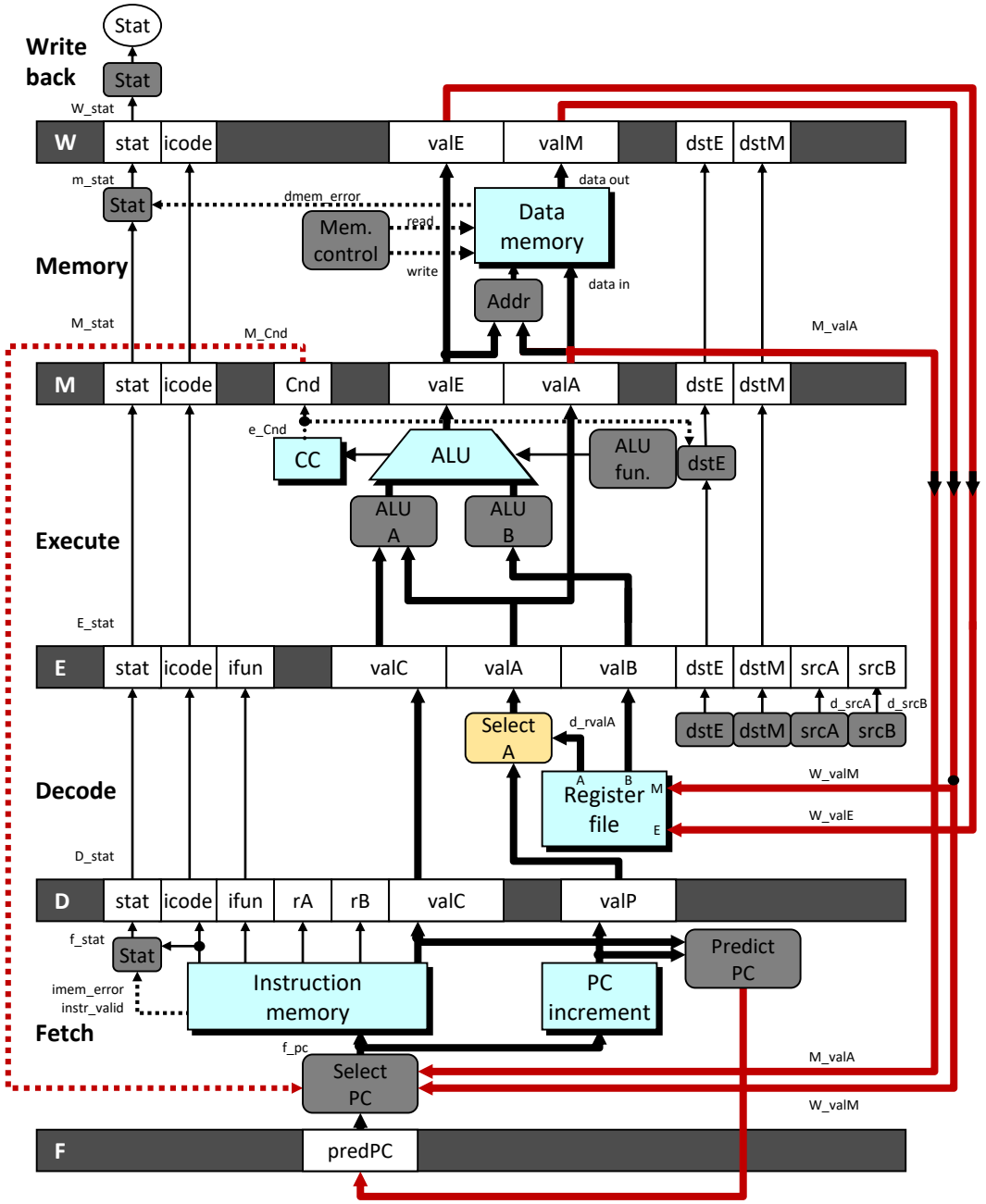
Signal Naming Conventions

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



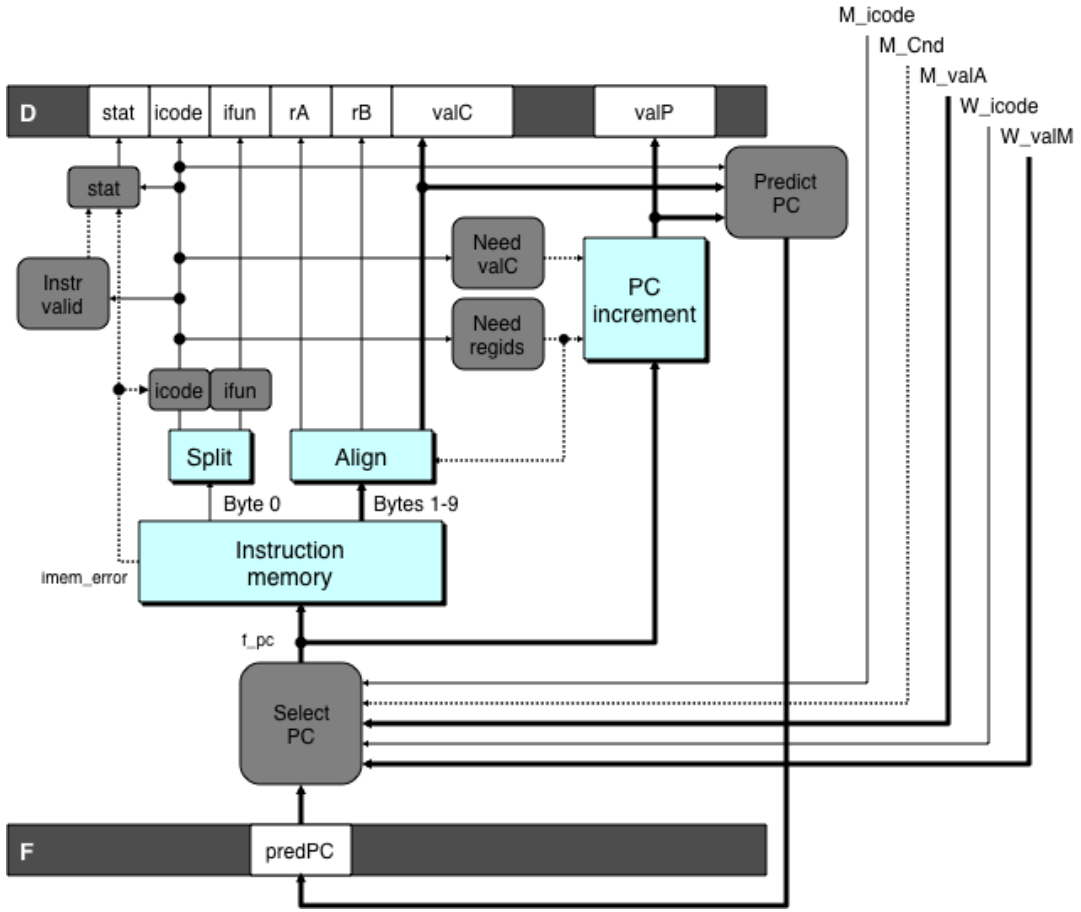
Feedback Paths

- 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

- Goal: issue a new instruction on every clock cycle
- 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



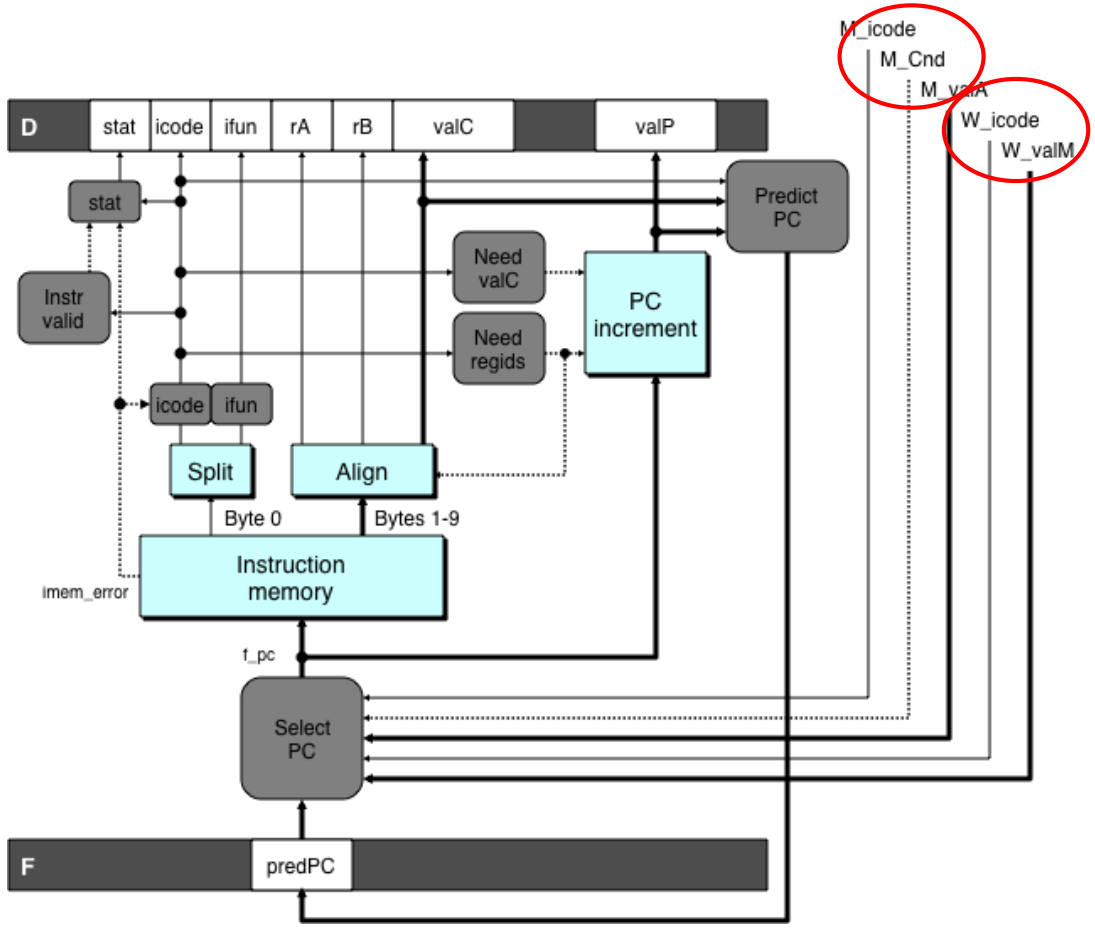
Our Prediction Strategy

- 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 (stall the pipeline)

Recovering from PC Misprediction

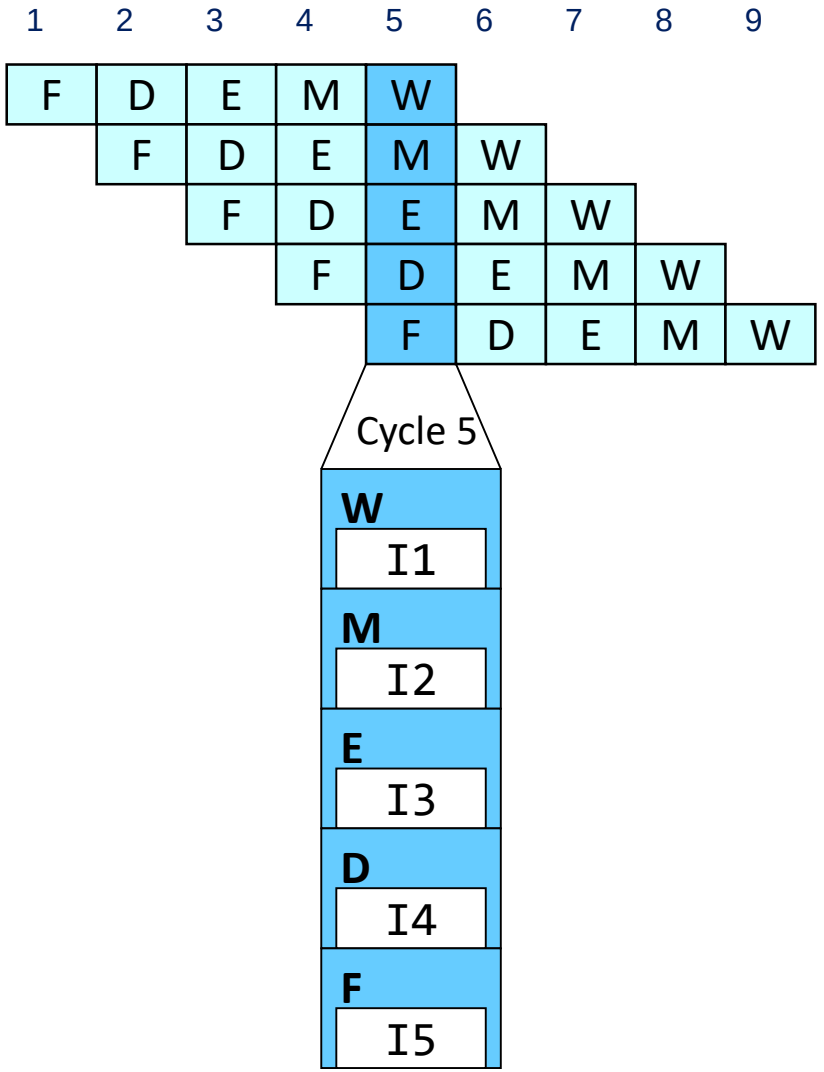
- Mispredicted jump
  - Will see branch condition flag once instruction reaches memory stage
  - Can get fall-through PC from **`valA`** (value **`M_valA == D_valP`**)

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



Pipeline Demonstration

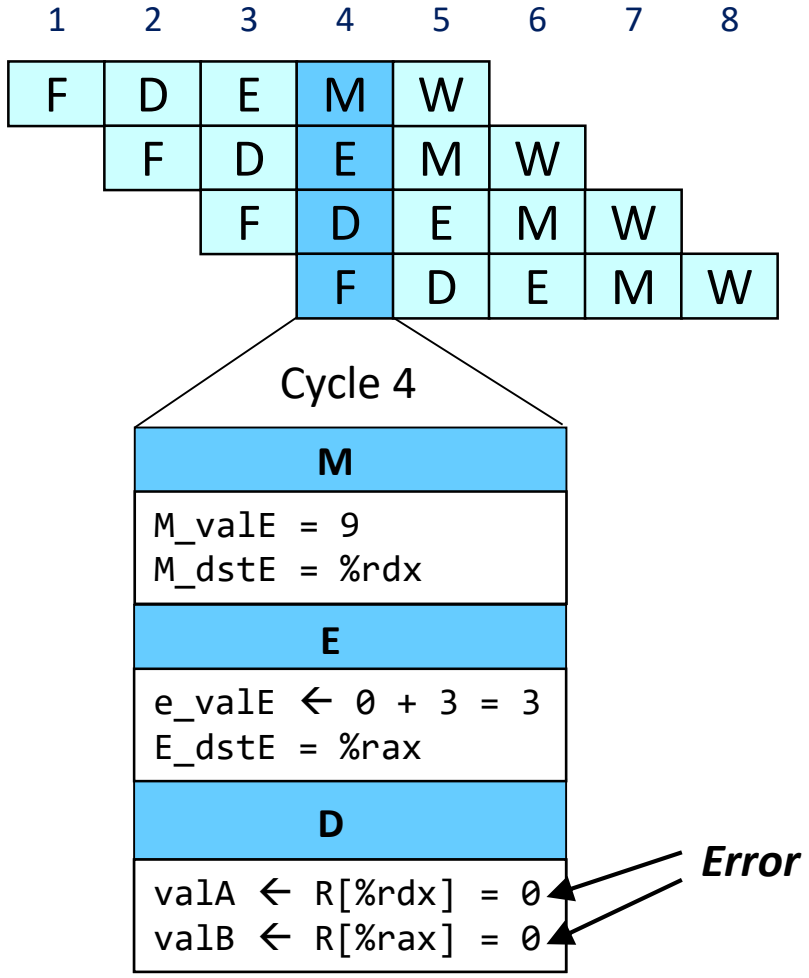
```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



Example: Data Hazard (1)

- No nop

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   addq     %rdx,%rax
0x016:   halt
```

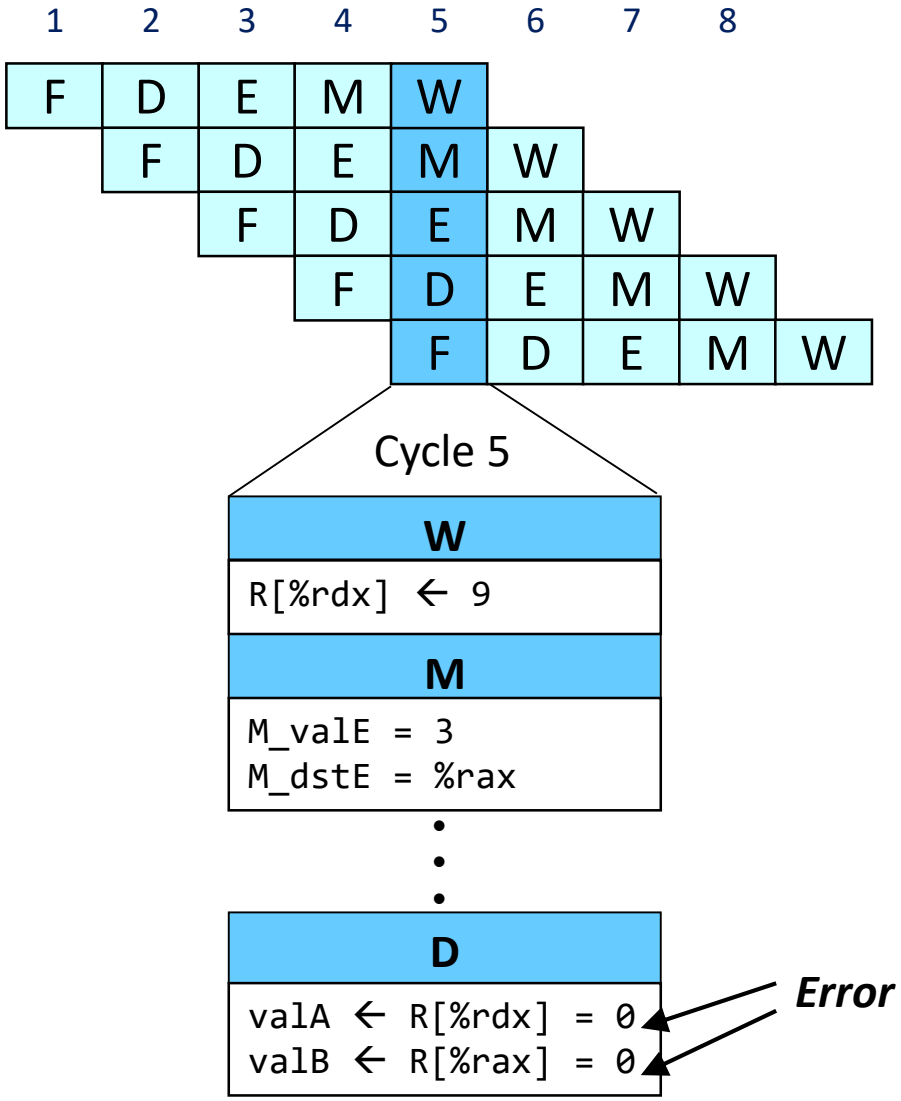


(Both %rax and %rdx are initialized to 0)

Example: Data Hazard (2)

- I nop

```
0x000:   irmovq    $9,%rdx
0x00a:   irmovq    $3,%rax
0x014:   nop
0x015:   addq      %rdx,%rax
0x017:   halt
```

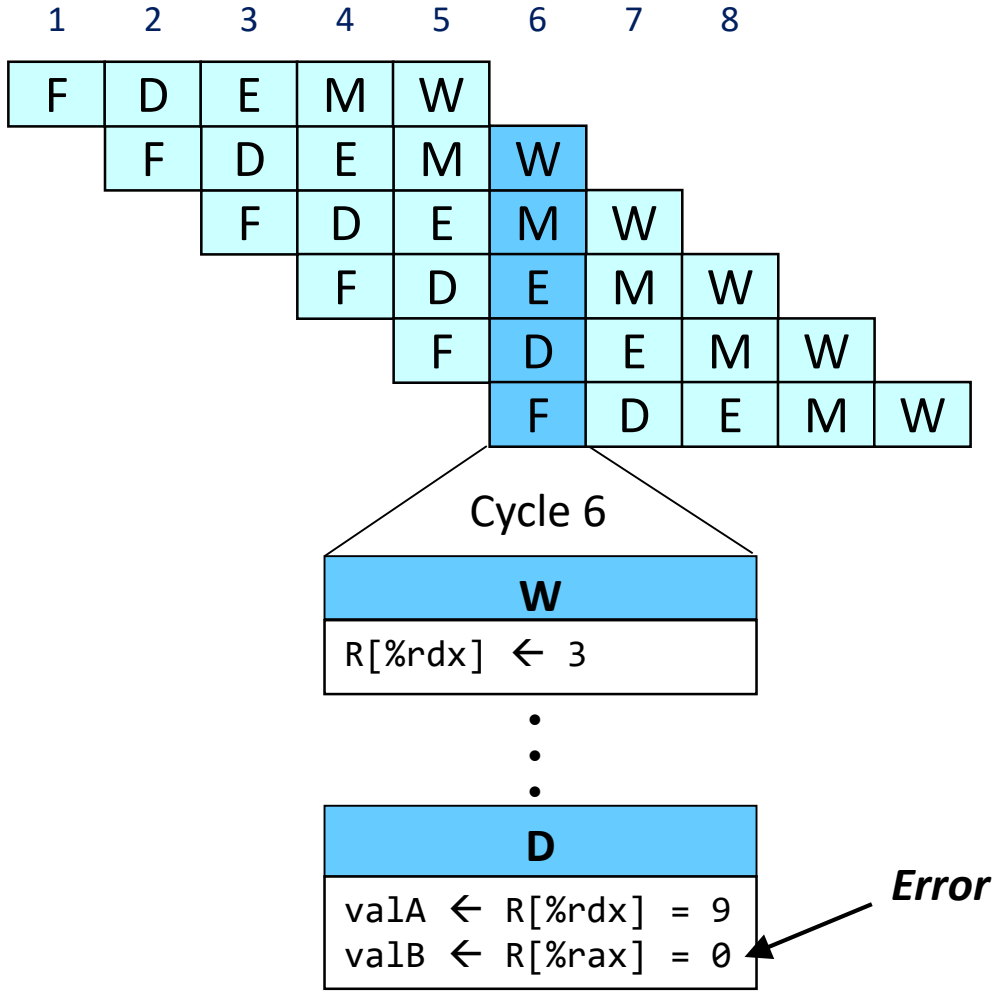


(Both %rax and %rdx are initialized to 0)

Example: Data Hazard (3)

- 2 nop's

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   nop
0x015:   nop
0x016:   addq     %rdx,%rax
0x018:   halt
```

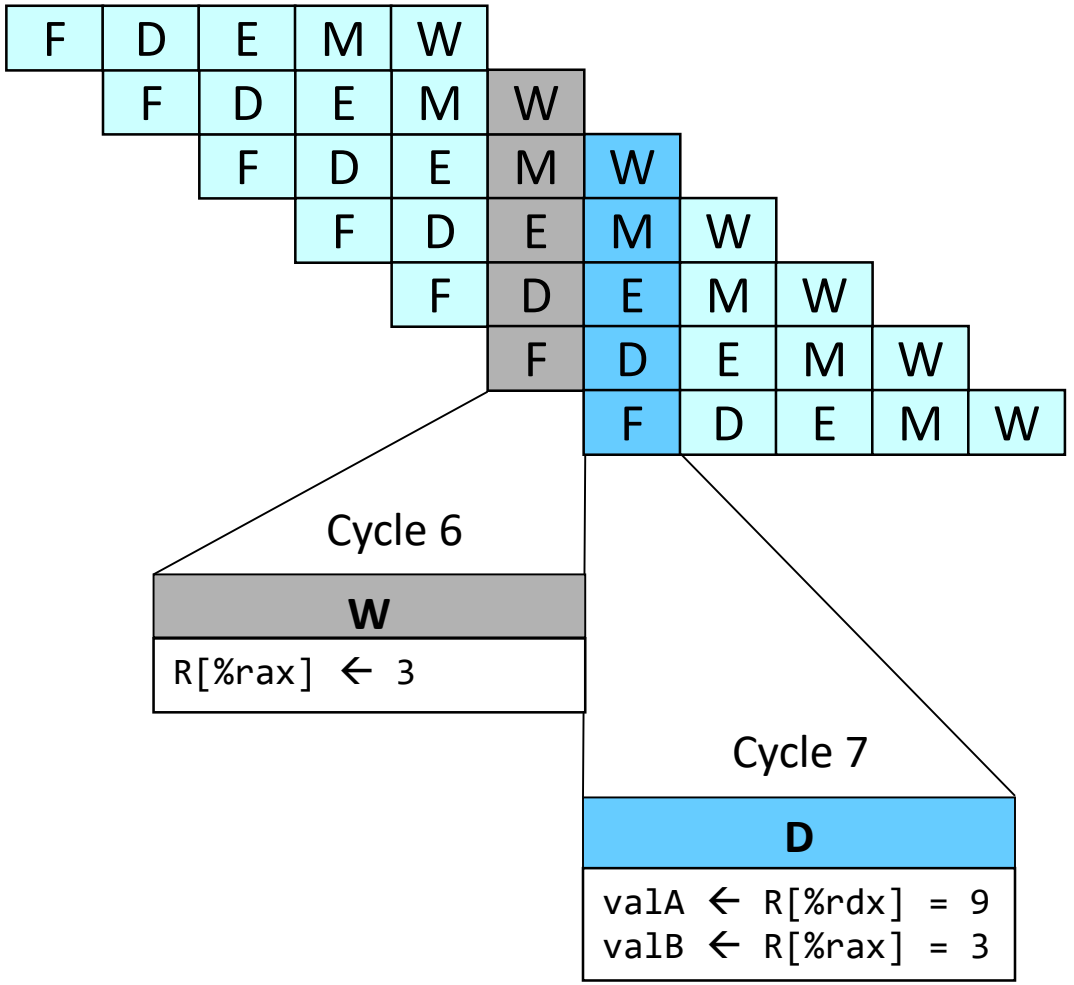


(Both %rax and %rdx are initialized to 0)

Example: Data Hazard (4)

- 3 nop's

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   nop
0x015:   nop
0x016:   nop
0x017:   addq     %rdx,%rax
0x019:   halt
```



PIPE- 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 dependency
    - 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
- How to fix them?

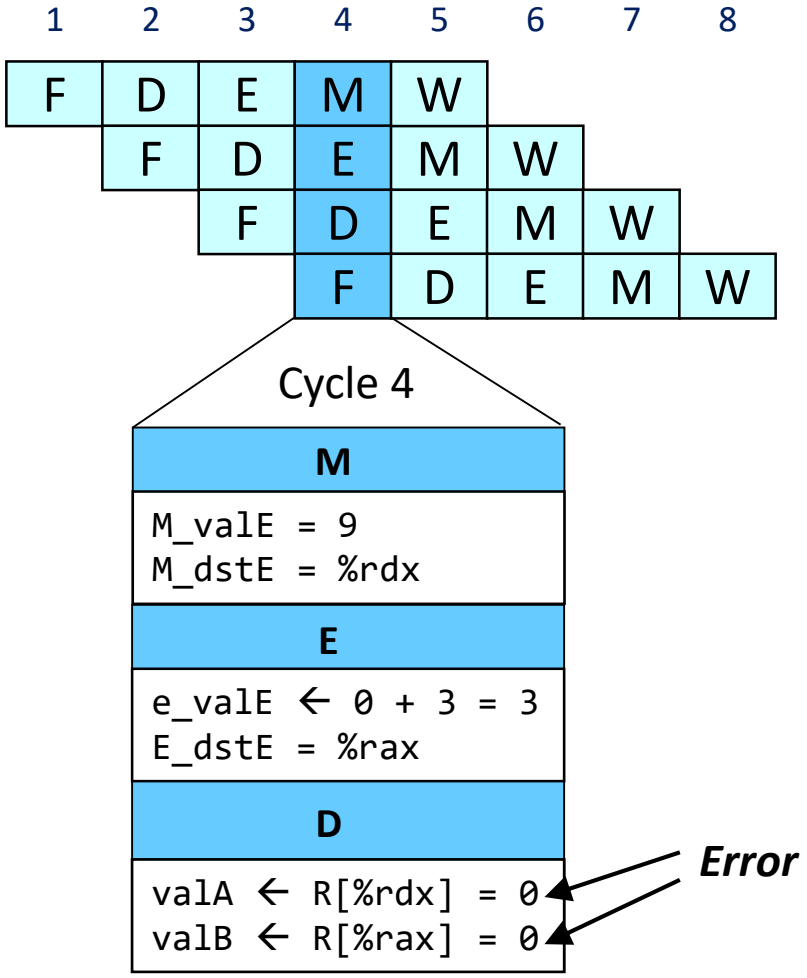
Make the Pipelined Processor Work!

- 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
  - Mispredicted 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?

Data Dependencies (Revisited)

- No nop

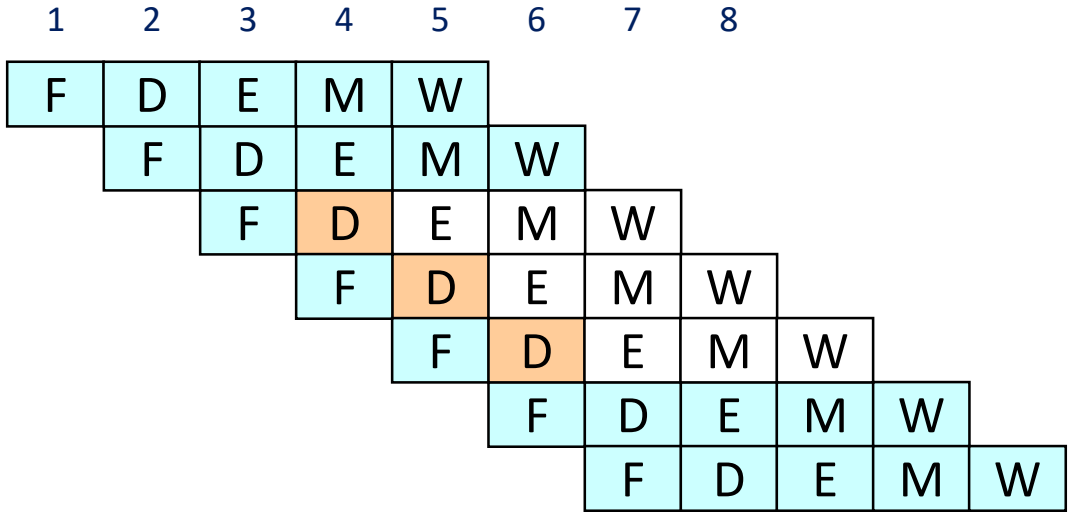
```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   addq     %rdx,%rax
0x016:   halt
```



(Both %rax and %rdx are initialized to 0)

Stalling for Data Dependencies

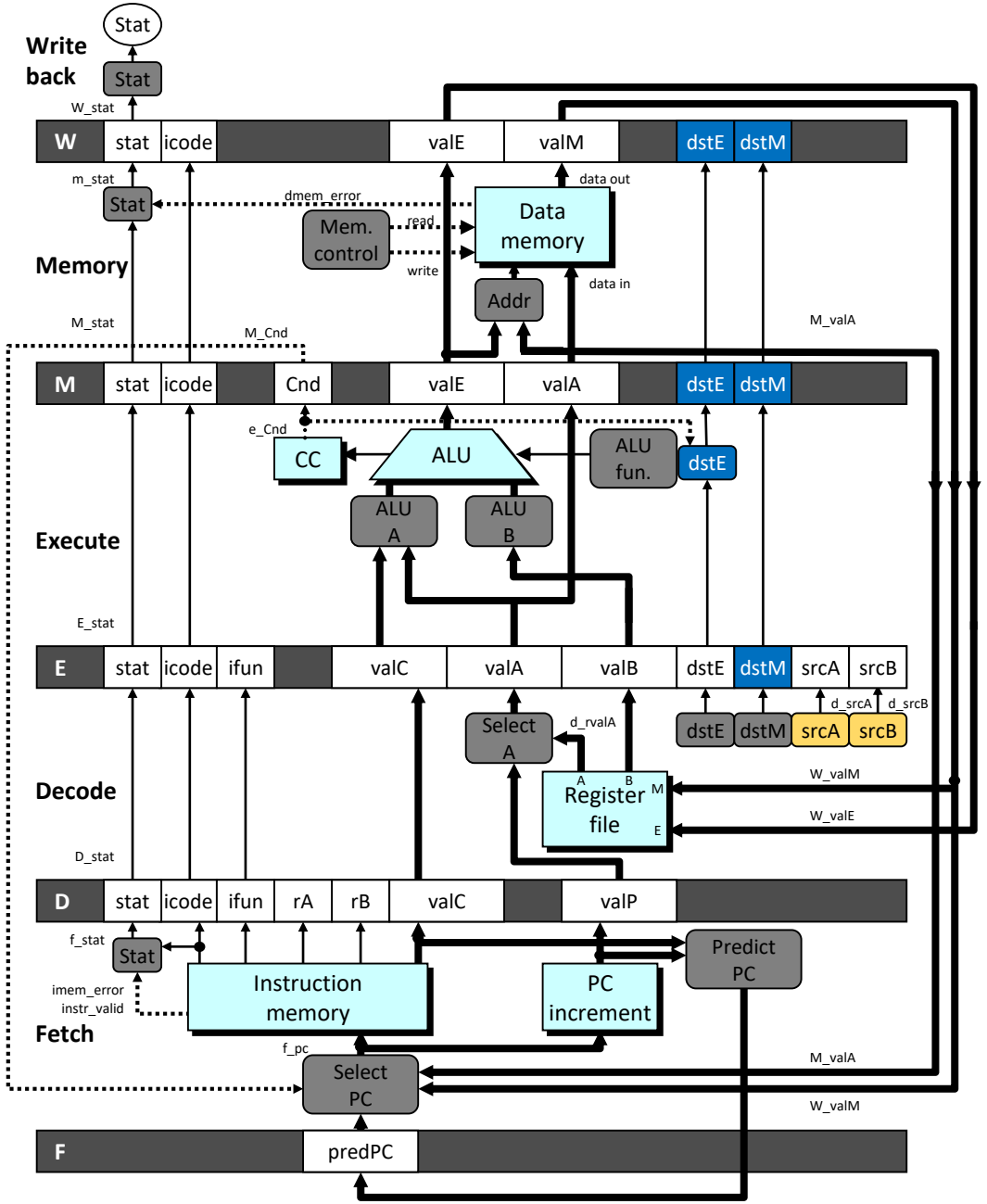
```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
0x014:   bubble
         bubble
         bubble
0x014:   addq     %rdx,%rax
0x016:   halt
```



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

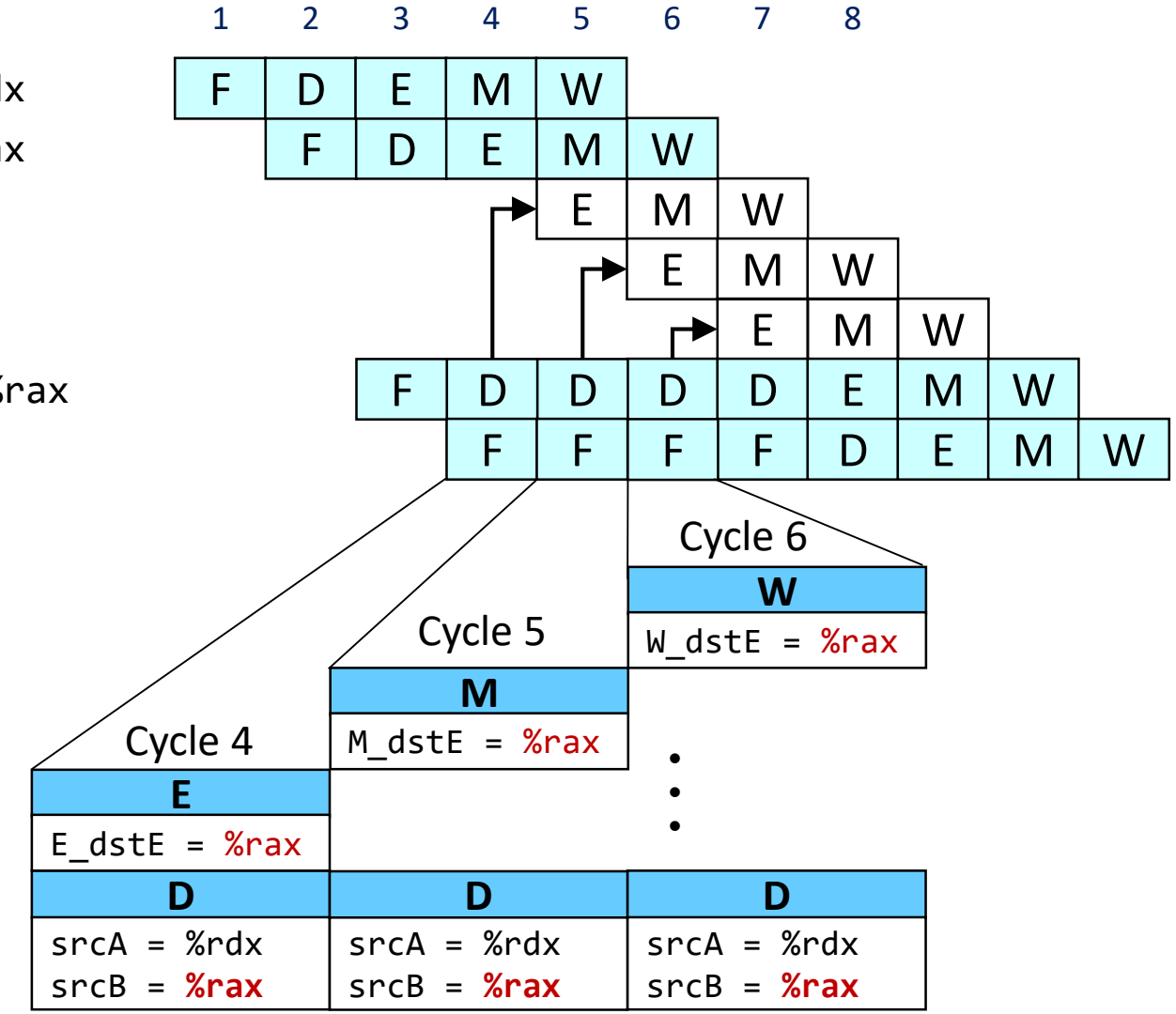
Stall Condition

- 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 conditional move



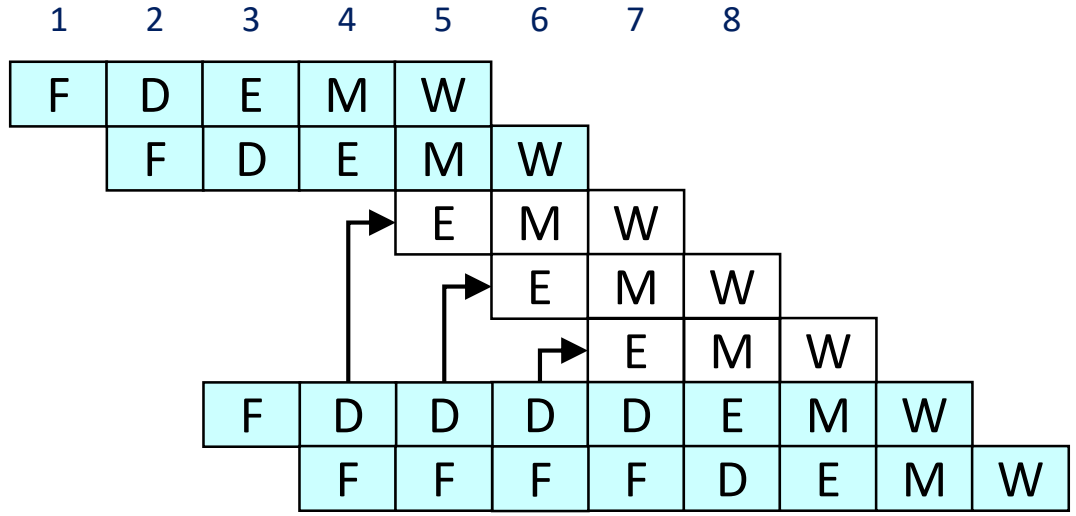
Stalling X3

```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
         bubble
         bubble
         bubble
0x014:   addq     %rdx,%rax
0x016:   halt
```



What Happens When Stalling?

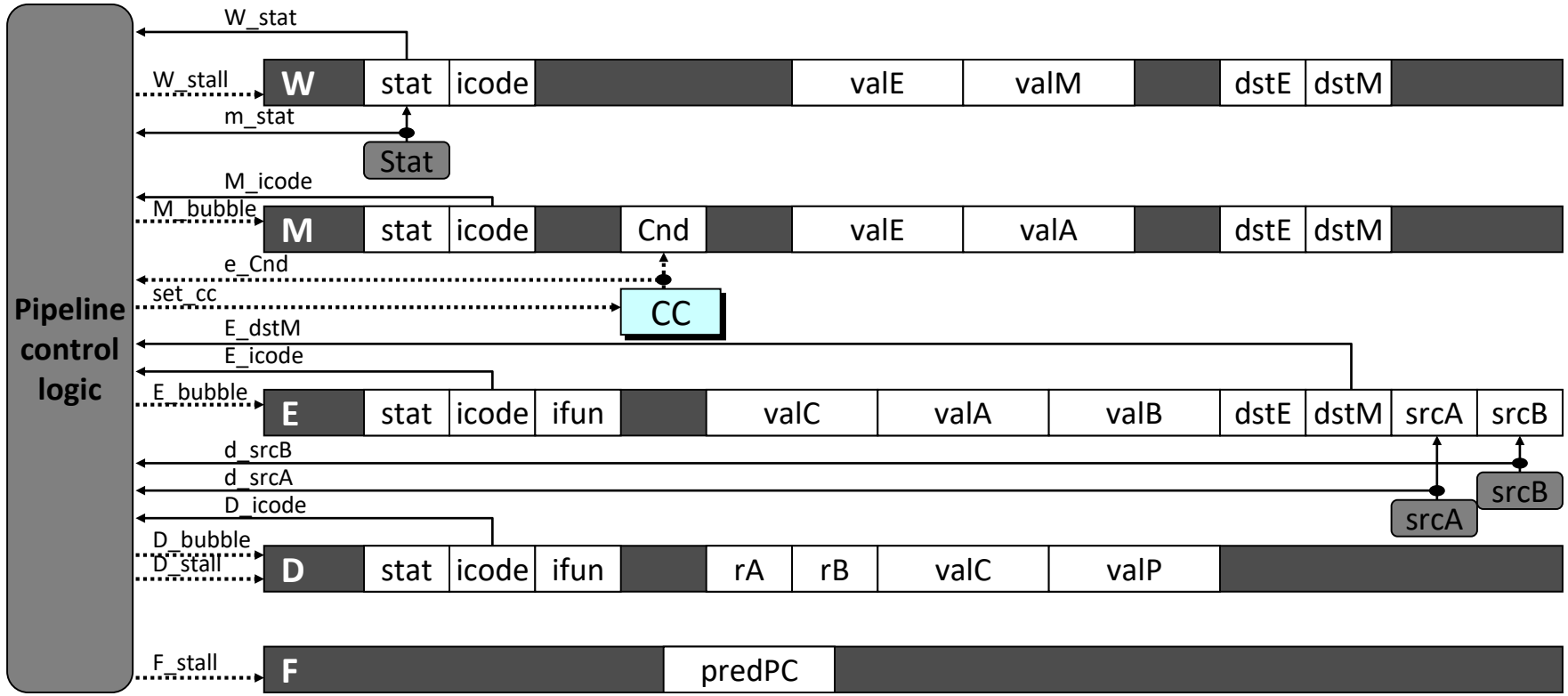
```
0x000:   irmovq   $9,%rdx
0x00a:   irmovq   $3,%rax
         bubble
         bubble
         bubble
0x014:   addq     %rdx,%rax
0x016:   halt
```



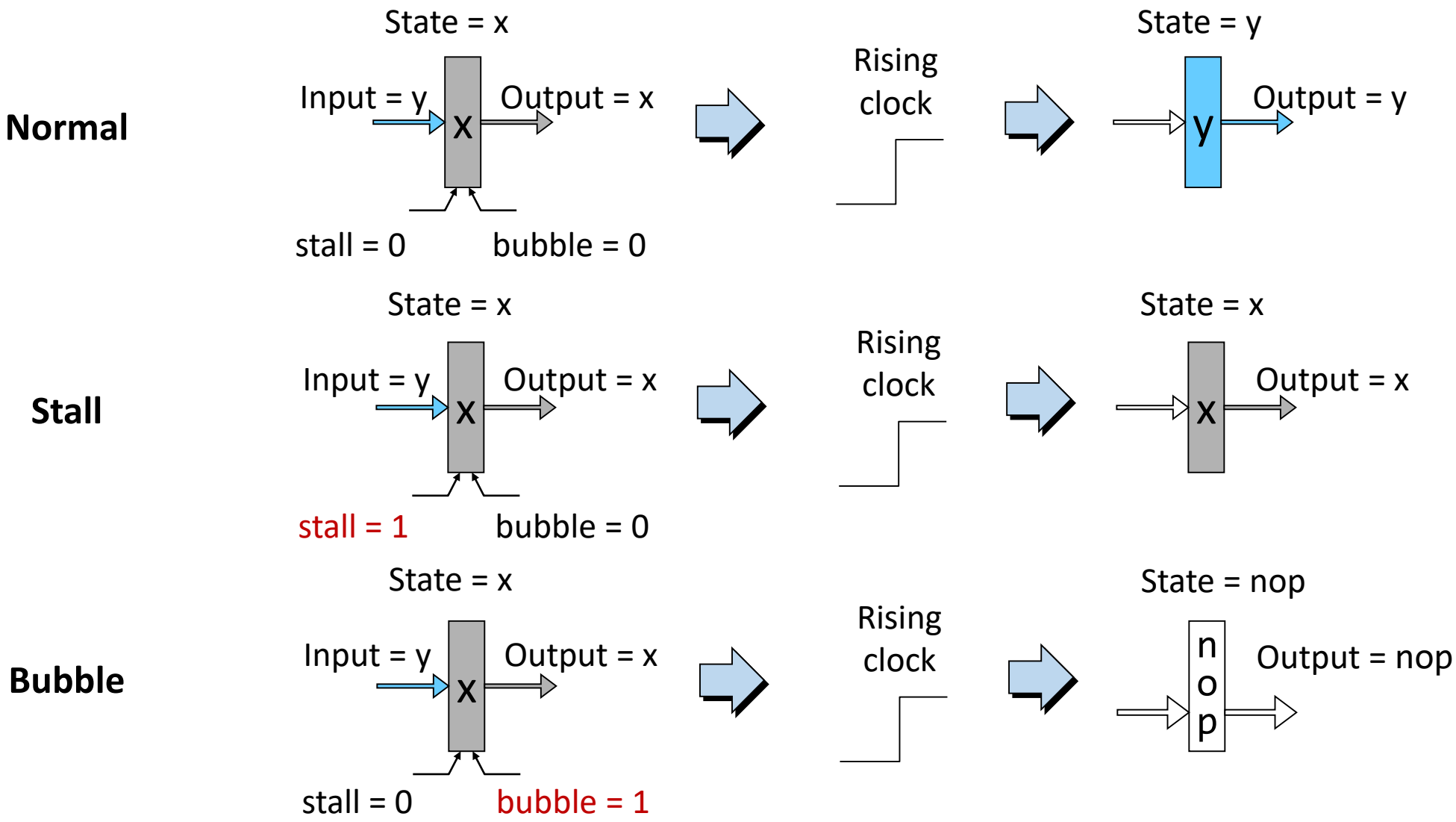
- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

Implementing Stalling

- Pipeline control
  - Combinational logic detects stall condition
  - Sets mode signals for how pipeline registers should be updated



Pipeline Register Modes

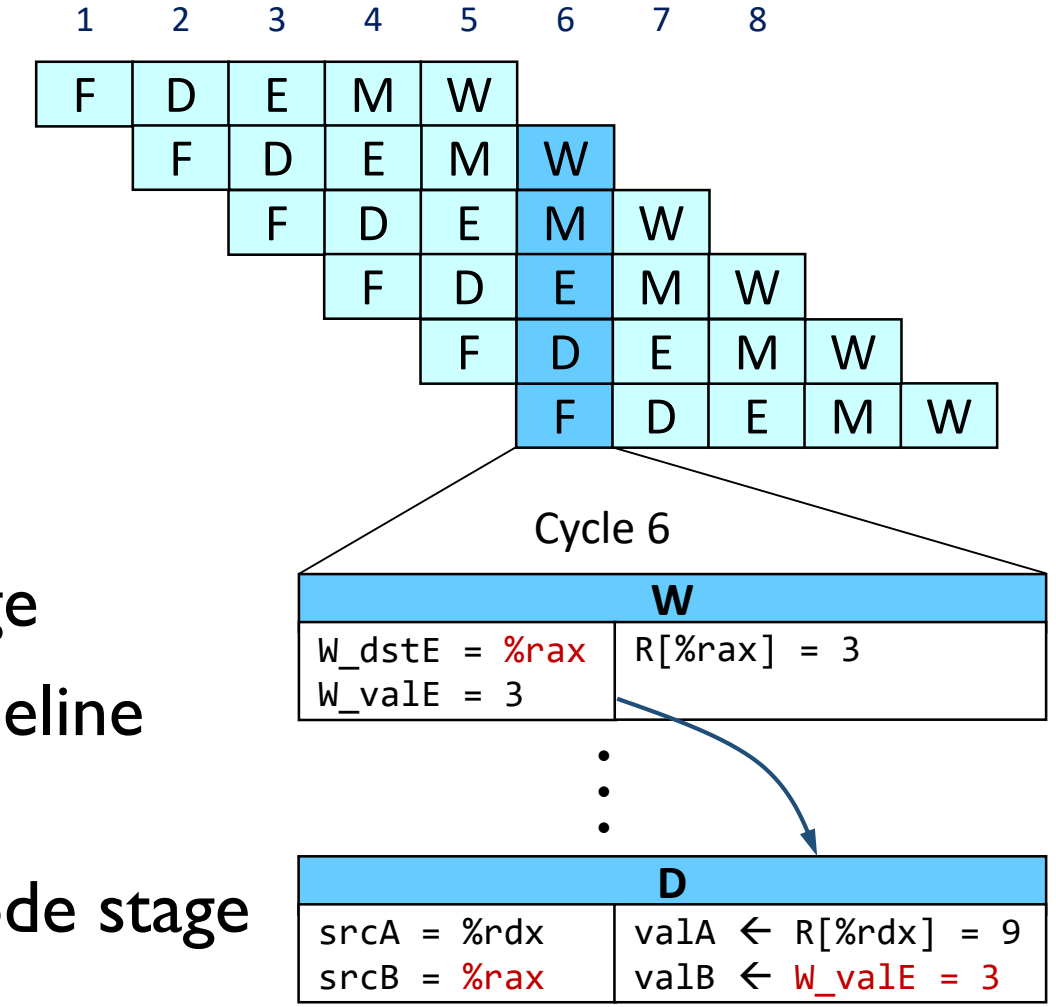


Data Forwarding

- 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 to be written to register file generated much earlier (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:   irmovq    $9,%rdx
0x00a:   irmovq    $3,%rax
0x014:   nop
0x015:   nop
0x016:   addq      %rdx,%rax
0x018:   halt
```

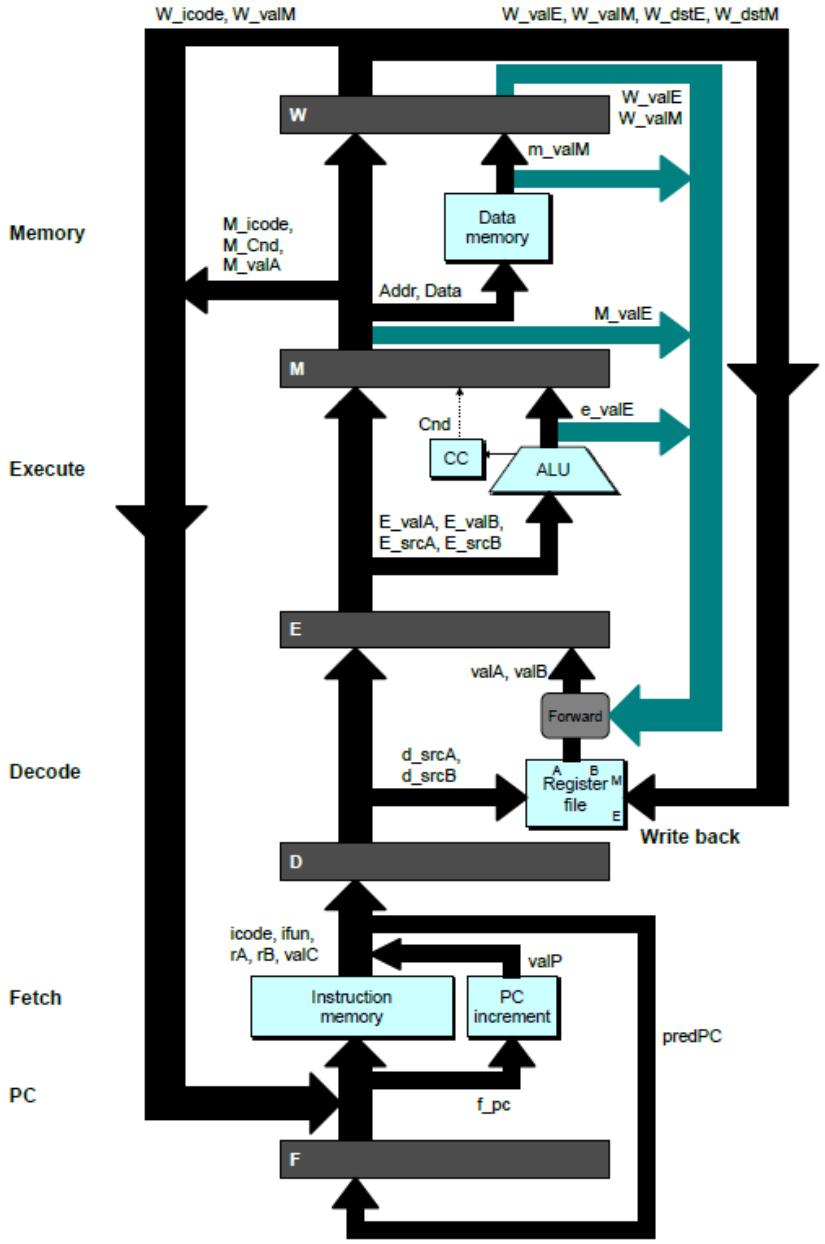


- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as `valB` for decode stage

Bypass Paths

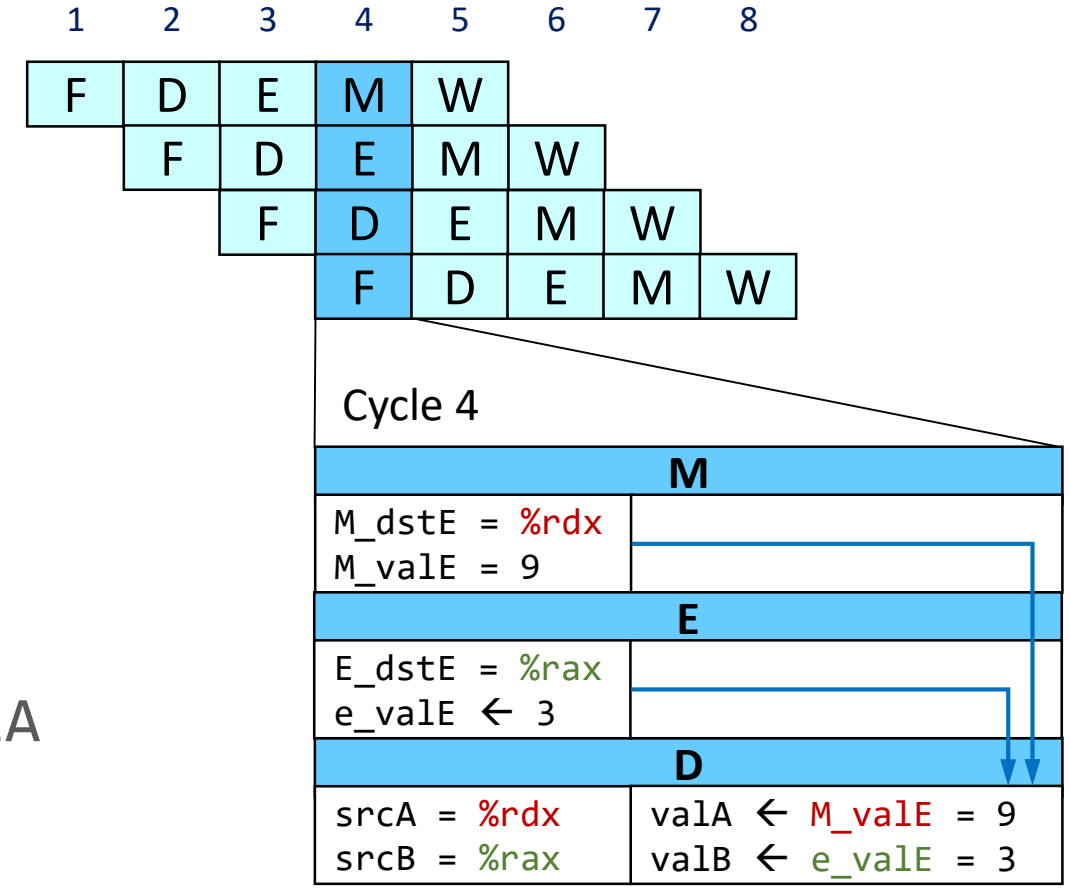
- Decode stage
  - Forwarding logic selects `valA` and `valB`
  - Normally from register file
  - Forwarding: get `valA` or `valB` from later pipeline stage

- Forwarding sources
  - Execute: `valE`
  - Memory: `valE`, `valM`
  - Write back: `valE`, `valM`



Data Forwarding Example #2

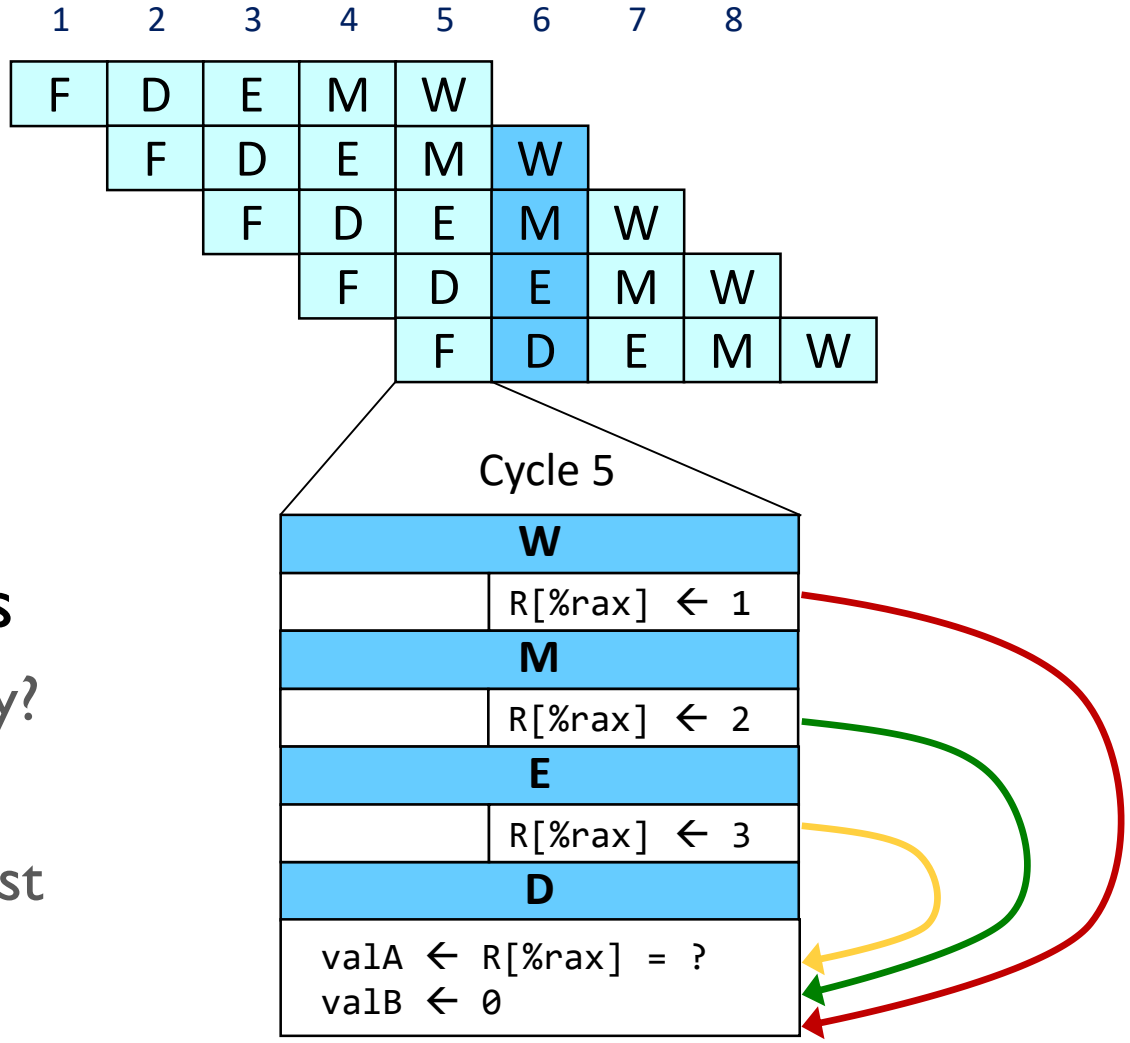
```
0x000:   irmovq    $9,%rdx
0x00a:   irmovq    $3,%rax
0x014:   addq      %rdx,%rax
0x016:   halt
```



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

Forwarding Priority

```
0x000:   irmovq   $1,%rax
0x00a:   irmovq   $2,%rax
0x014:   irmovq   $3,%rax
0x01e:   rrmovq   %rax,%rdx
0x020:   halt
```

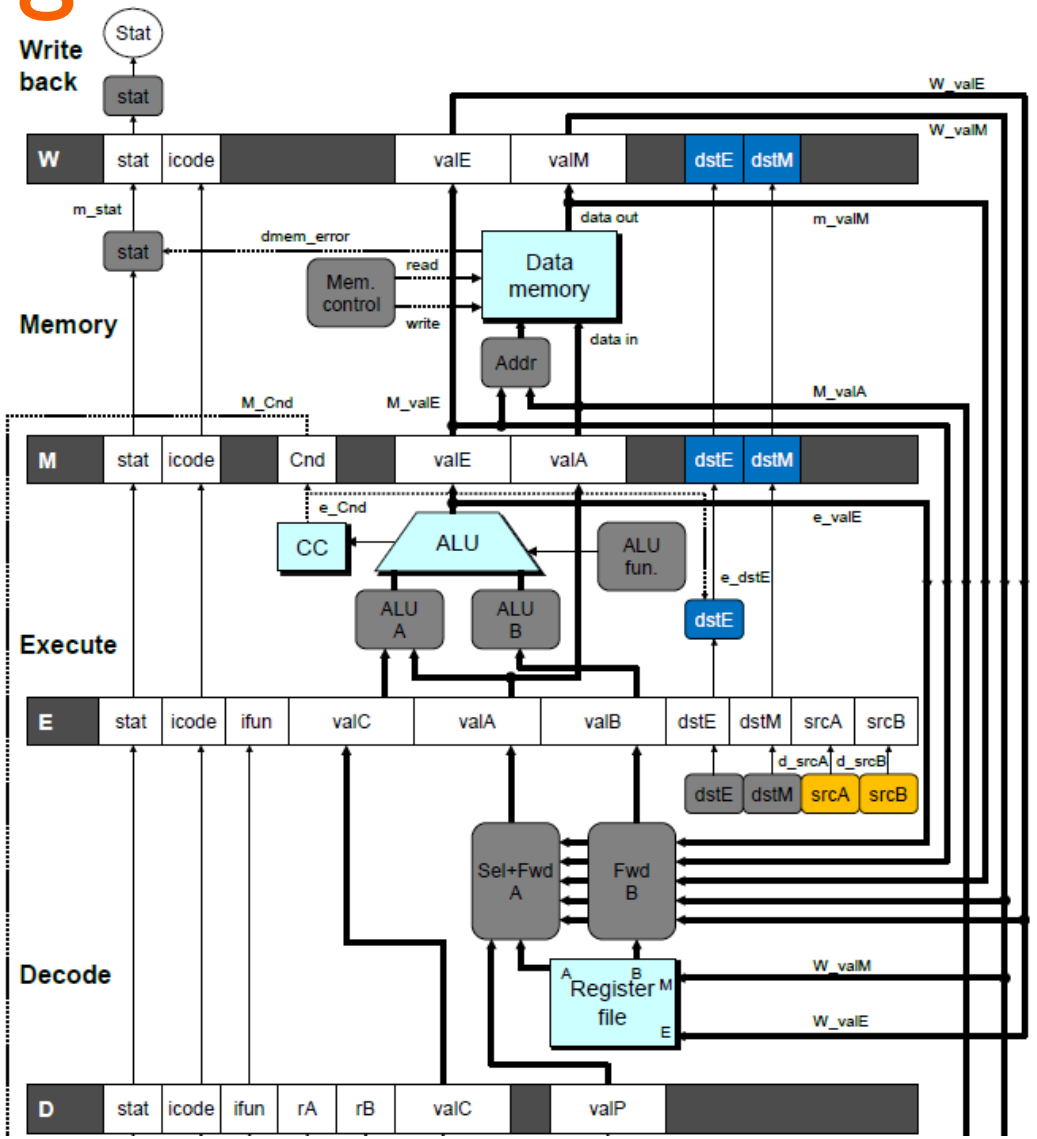


- Multiple forwarding choices
  - Which one should have priority?
  - Match serial semantics
  - Use matching value from earliest pipeline stage

Implementing Forwarding

- Add additional feedback paths from E, M, and W pipeline registers into decode stage

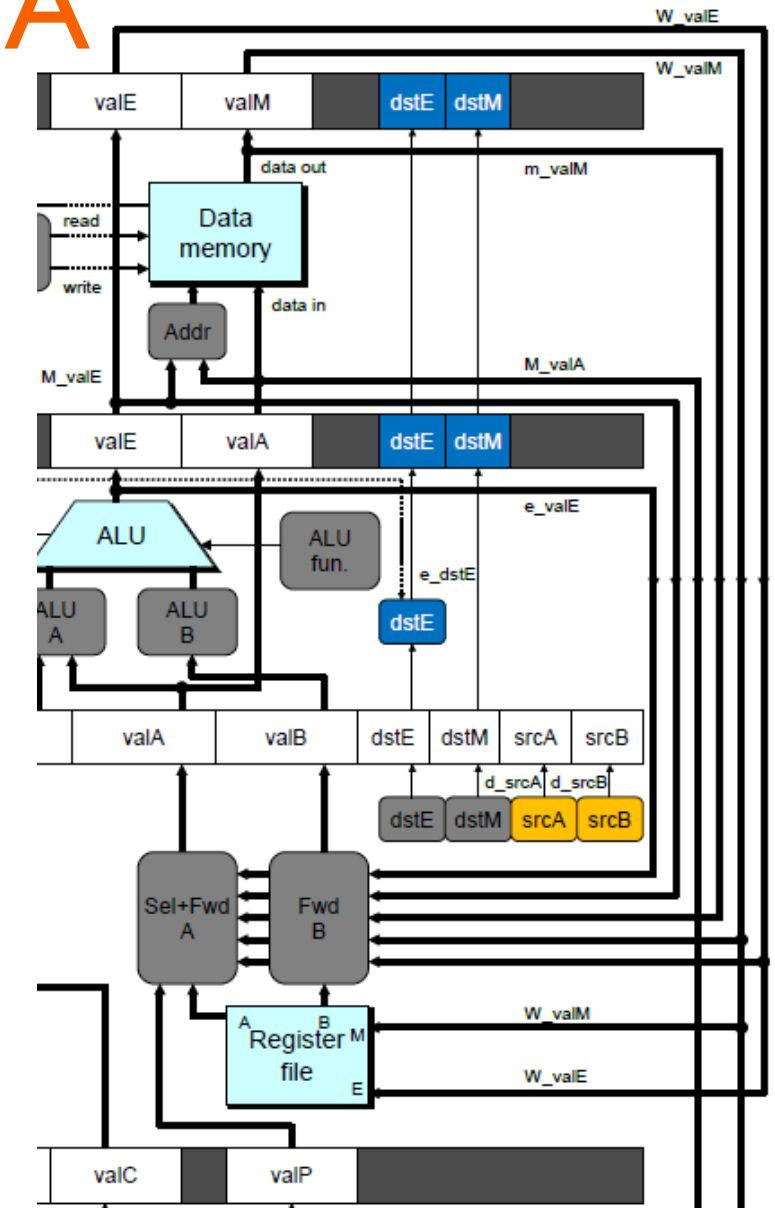
- Create logic blocks to select from multiple sources for `valA` and `valB` in decode stage



Implementing Forwarding: valA

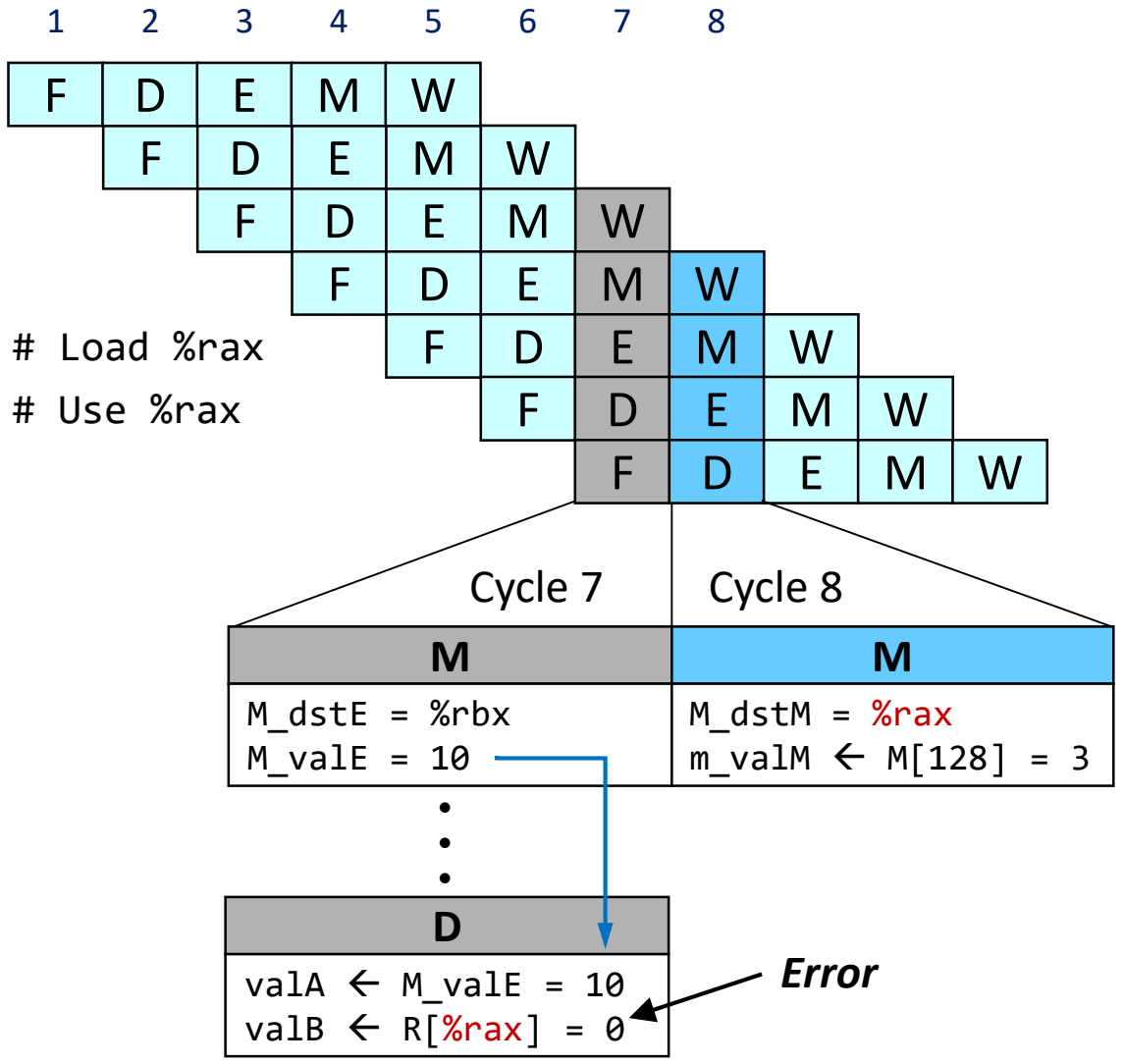
- What should be the A value?

```
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;
];
```



Load/Use Hazard

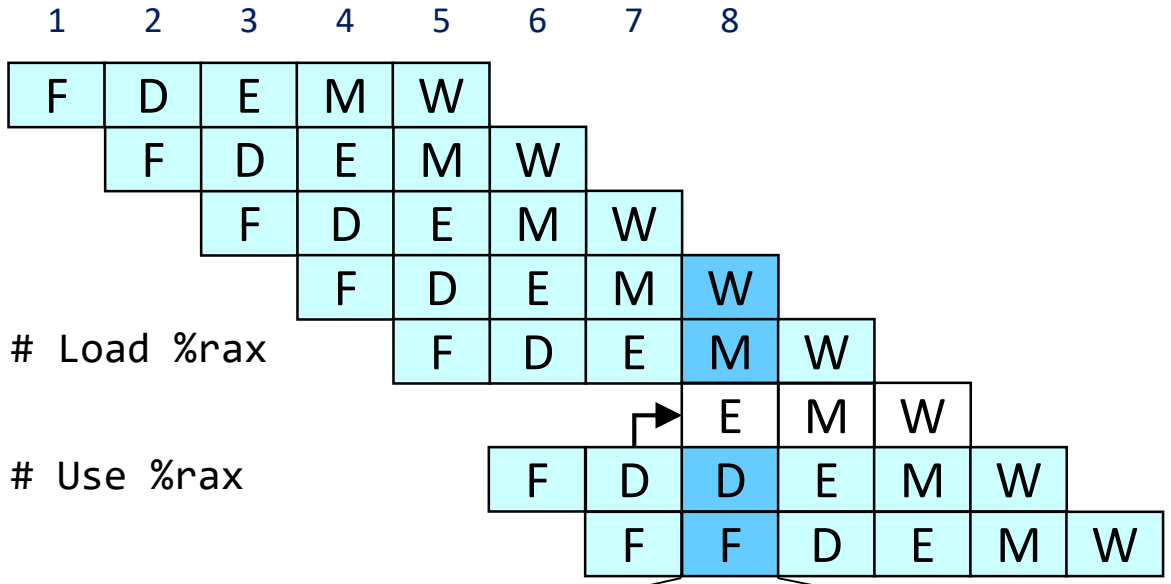
```
0x000:  irmovq   $128,%rdx
0x00a:  irmovq   $3,%rcx
0x014:  rmmovq   %rcx, 0(%rdx)
0x01e:  irmovq   $10,%rbx
0x028:  mrmovq   0(%rdx),%rax  # Load %rax
0x032:  addq     %rbx,%rax     # Use %rax
0x034:  halt
```

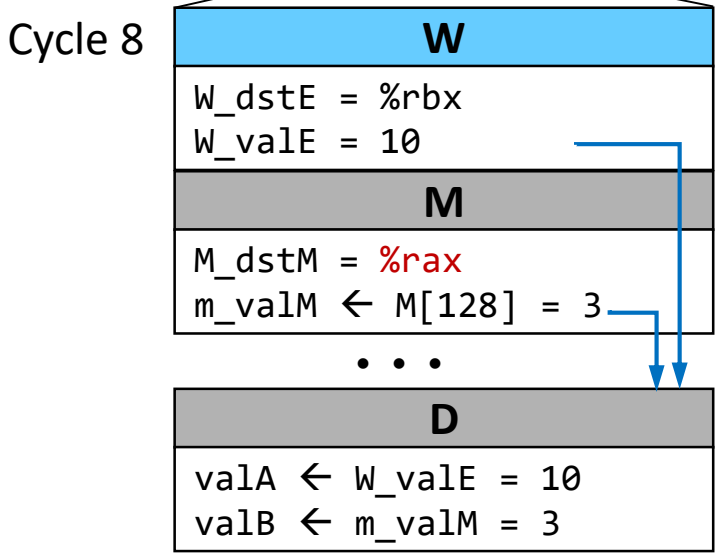


- Load-Use dependency
  - Value needed by end of decode stage in cycle 7
  - Value read from memory in memory stage of cycle 8

Avoiding Load/Use Hazard

```
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
```

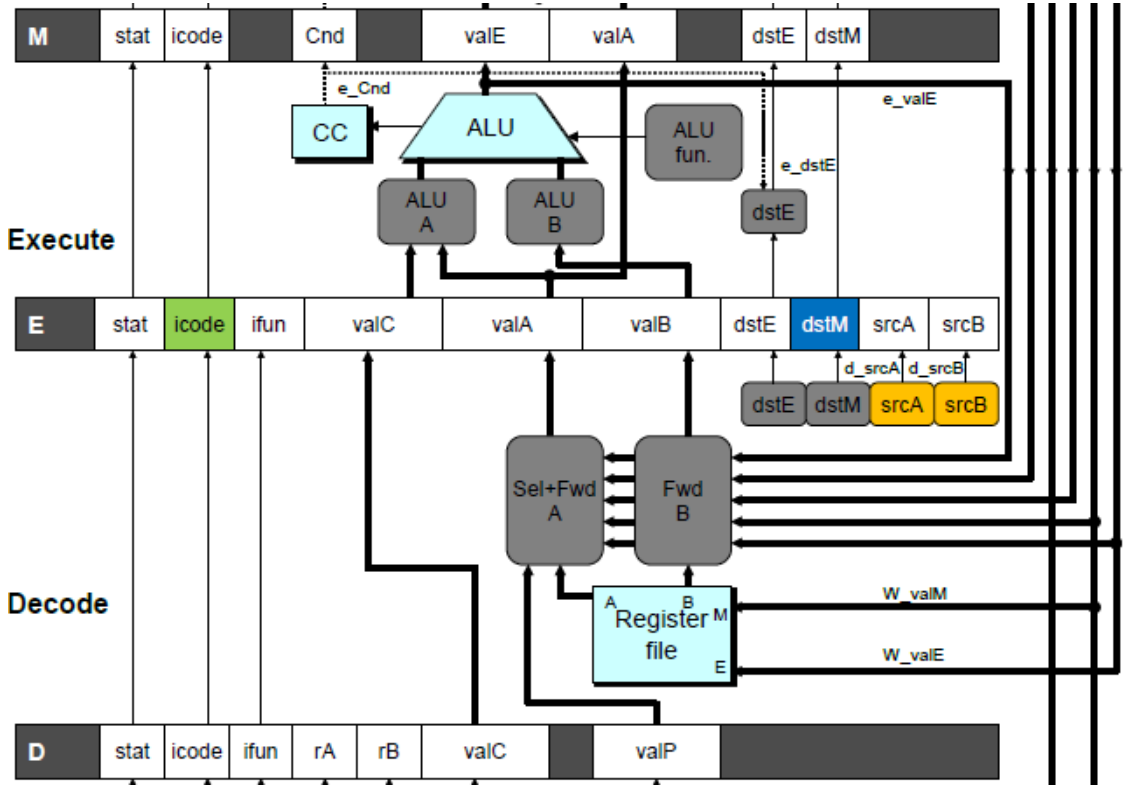




- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

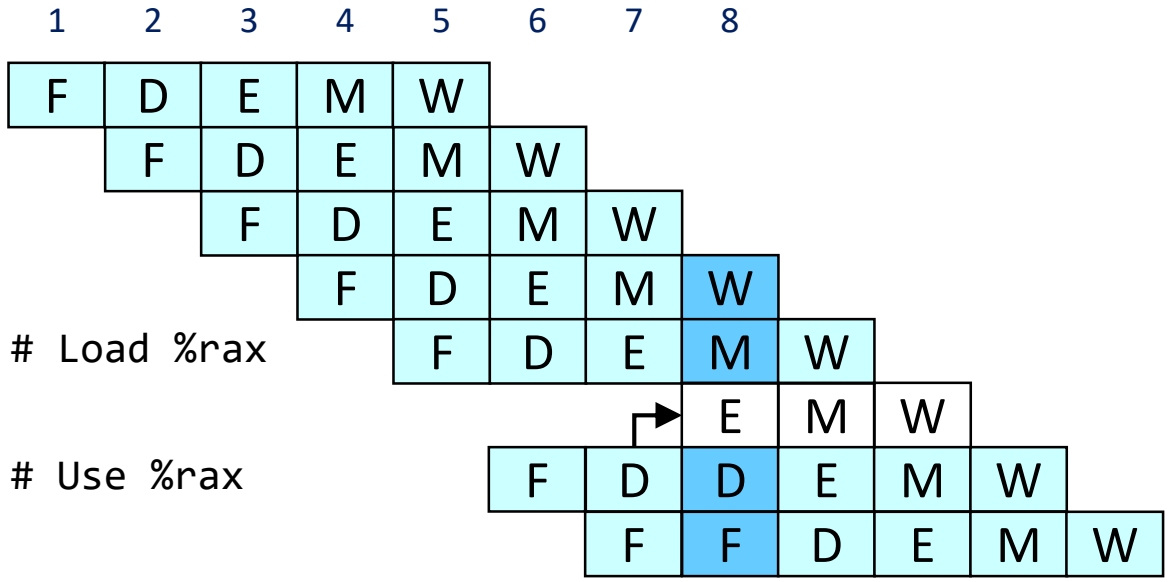
Detecting Load/Use Hazard

```
 # Conditions for a load/use hazard

bool F_stall =
      E_icode in { IMRMOVQ, IPOPQ } &&
      E_dstM in { d_srcA, d_srcB } || ...
```



Control for Load/Use Hazard

```
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 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

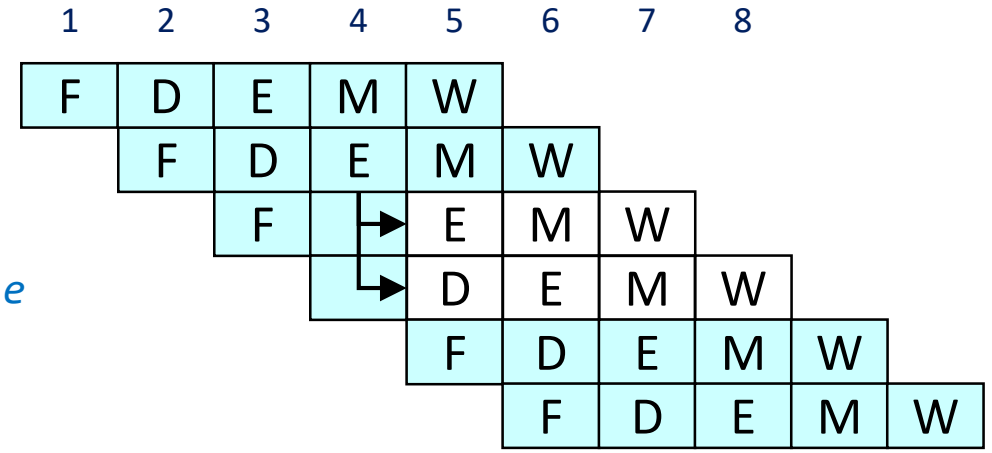
Branch Misprediction Example

- Should only execute first 7 instructions

```
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
```

Handling Misprediction

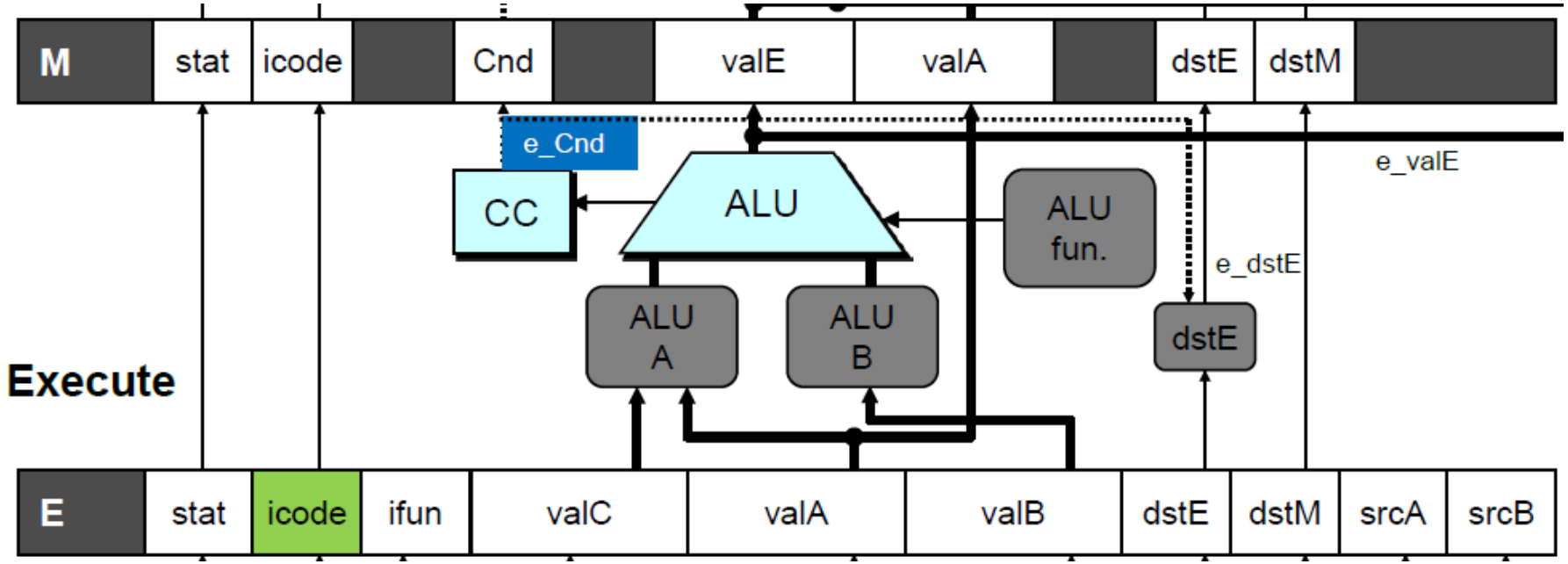
```
0x000:  xorq    %rax,%rax
0x002:  jne     target      # not taken
0x016:  irmovq  $3,%rdx     # target → bubble
0x020:  irmovq  $4,%rbx     # target+1 → bubble
0x00b:  irmovq  $1,%rax     # fall through
0x015:  halt
```



- 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

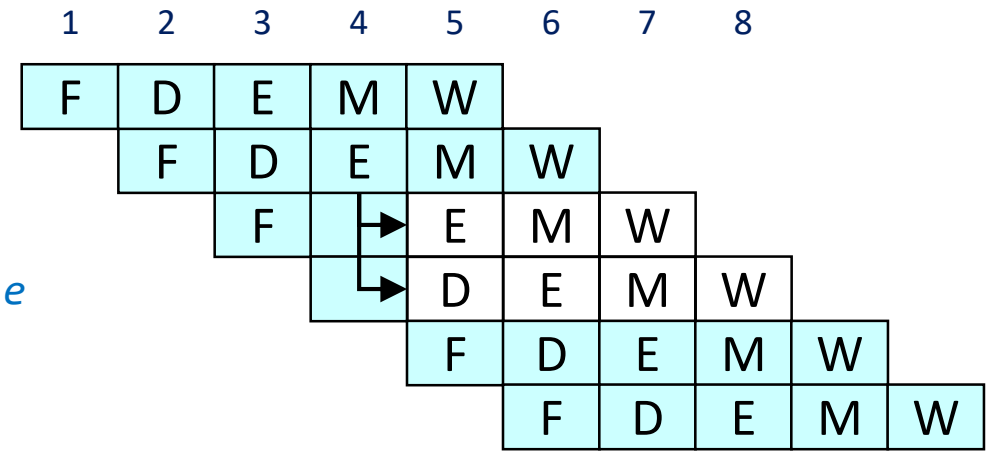
Detecting Mispredicted Branch

```
 # Mispredicted branch

bool D_bubble =
      (E_icode == IJXX && !e_Cnd) || ...
```



Control for Misprediction

```
0x000:   xorq     %rax,%rax
0x002:   jne      target      # not taken
0x016:   irmovq   $3,%rdx     # target → bubble
0x020:   irmovq   $4,%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

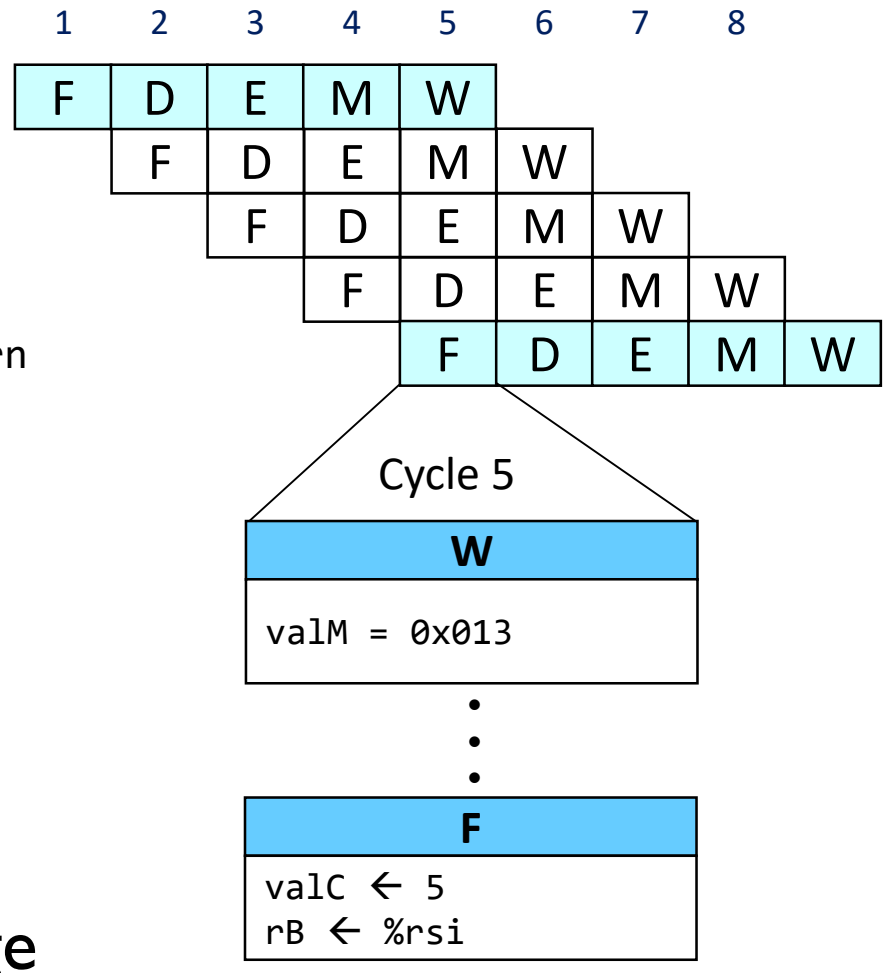
Return Example

- Previously executed three additional instructions

```
0x000:     irmovq  Stack, %rsp   # Initialize 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:                   # Initial stack pointer
```

Correct Return Example

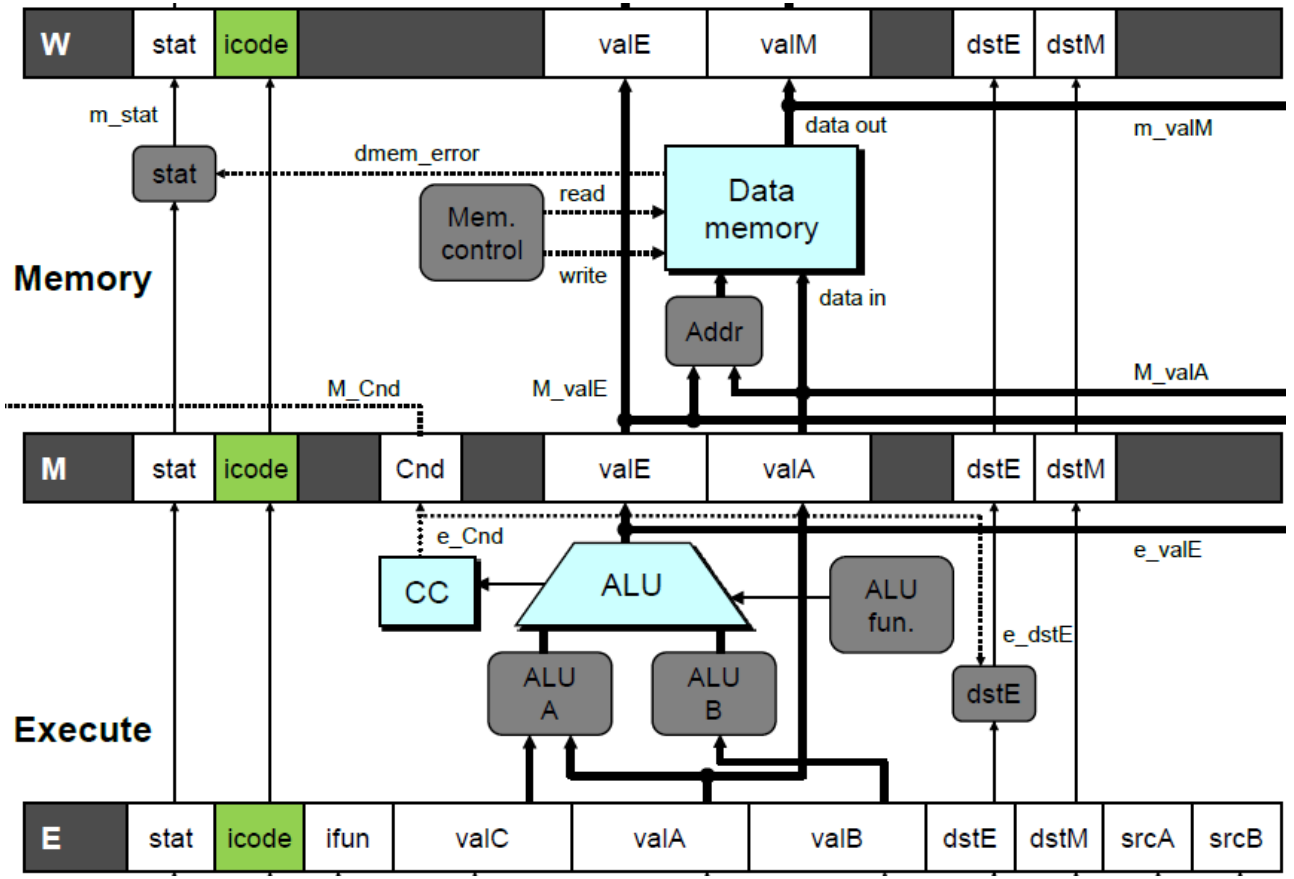
```
0x02a:   ret
         bubble
         bubble
         bubble
0x013:   irmovq   $5,%rsi     # Return
```



- 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

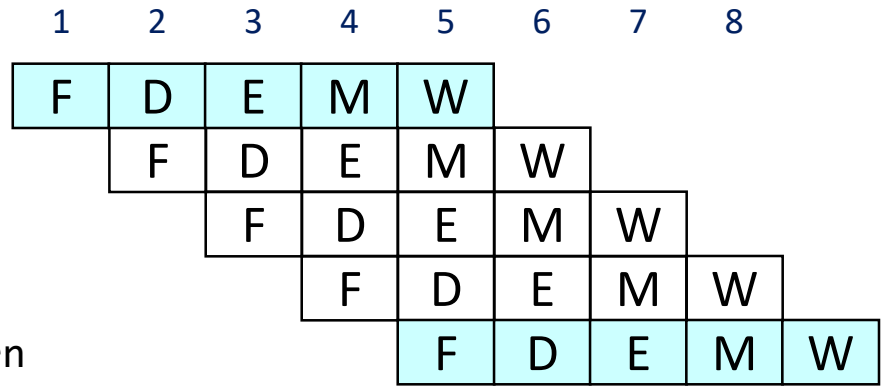
Detecting Return

```
# Processing ret

bool F_stall = … ||

    IRET in { D_icode,
              E_icode,
              M_icode};
```



Control for Return

```
0x02a:   ret
         bubble
         bubble
         bubble
0x013:   irmovq   $5,%rsi     # Return
```



Condition	F	D	E	M	W
Processing ret	Stall	Bubble	Normal	Normal	Normal

Special Control Cases

- Detection

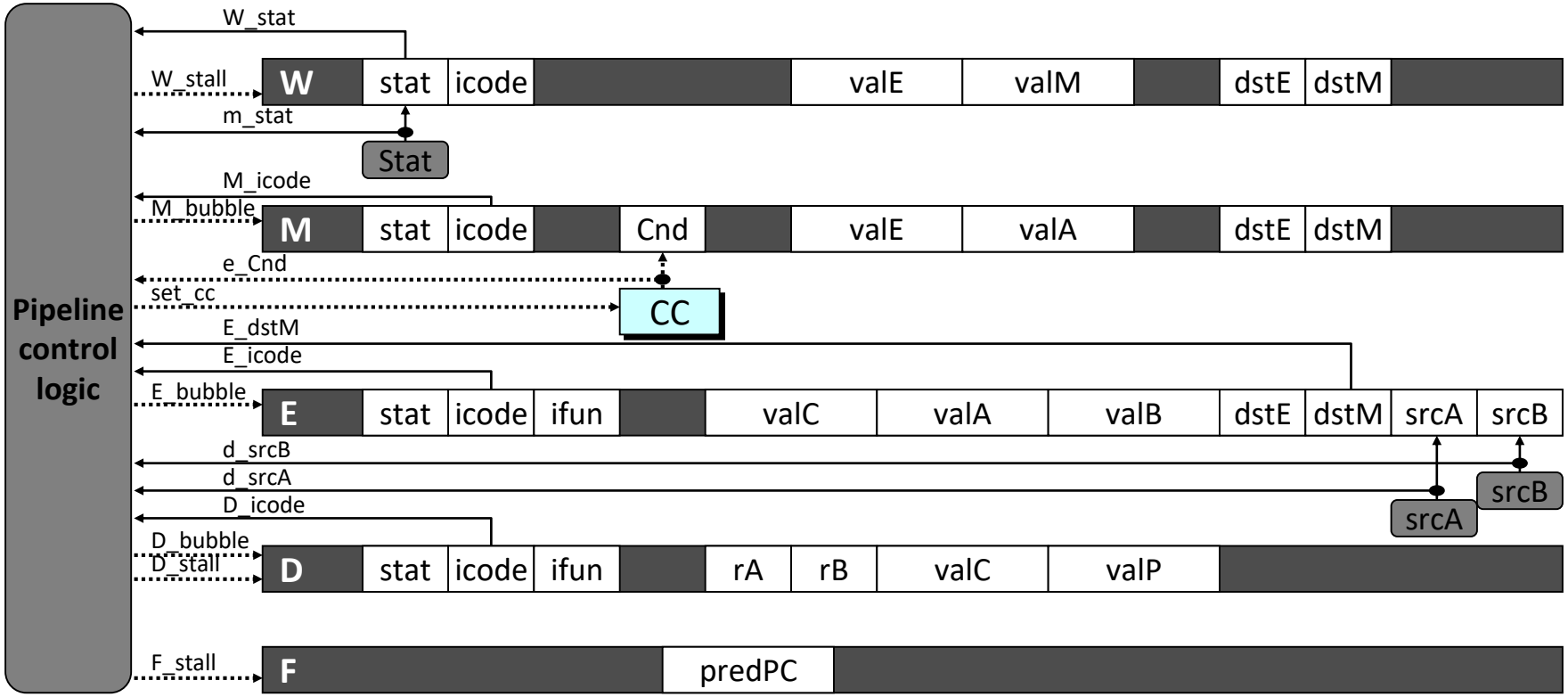
Condition	Trigger
Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`
Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in {d_srcA, d_srcB }`
Mispredicted Branch	`E_icode == IJXX && !e_Cnd`

- 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

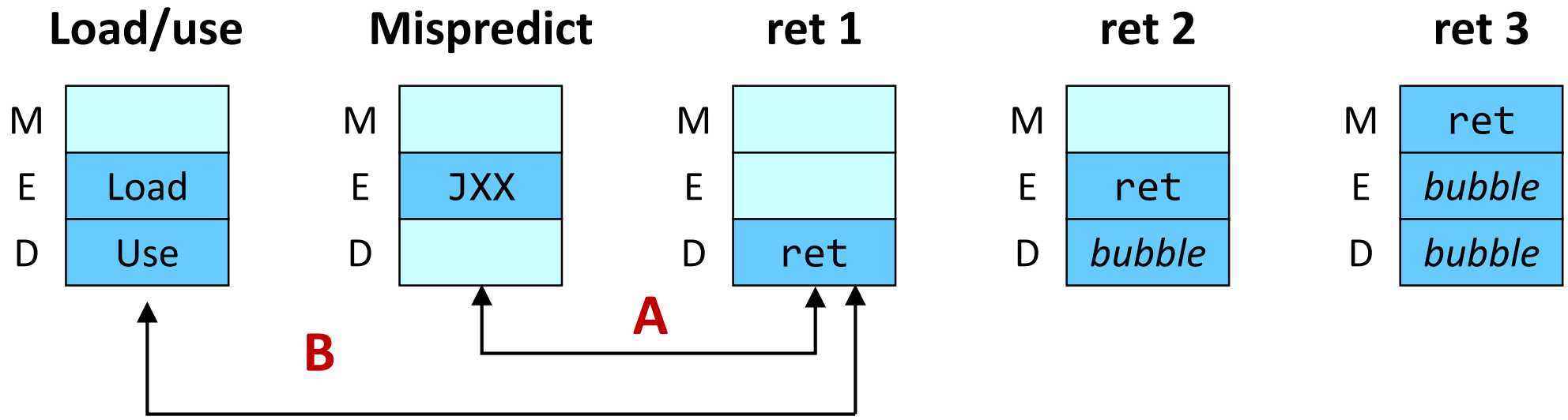


Initial Version of Pipeline Control

```
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

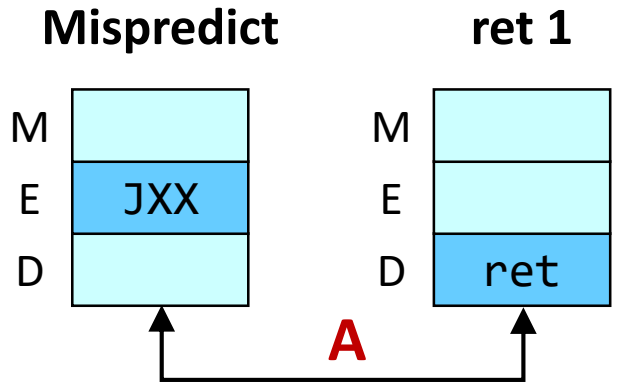
- Special cases that can arise on same clock cycle

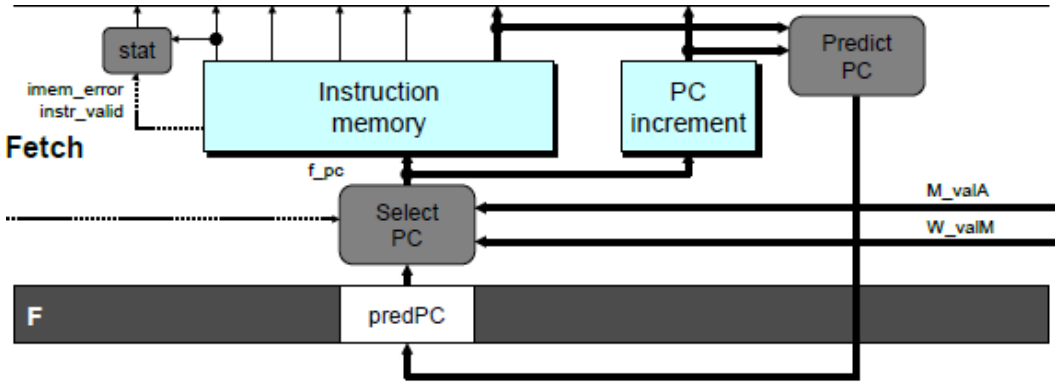


- Combination A
  - Not-taken branch
  - `ret` instruction at branch target
- Combination B
  - Instruction that reads from memory to %rsp
  - Followed by `ret` instruction

Control Combination A

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

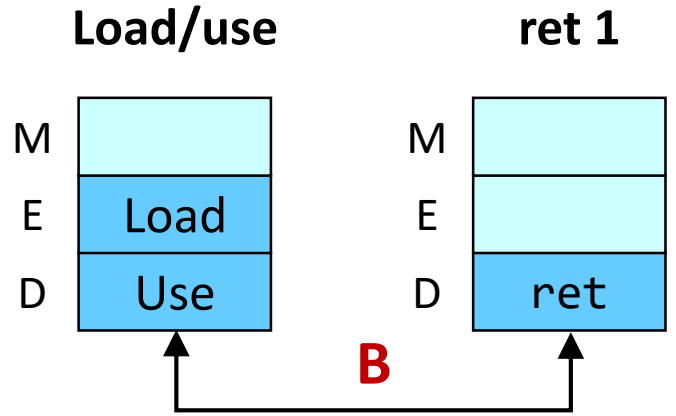




Condition	F	D	E	M	W
Processing `ret`	Stall	Bubble	Normal	Normal	Normal
Mispredicted Branch	Normal	Bubble	Bubble	Normal	Normal
Combination	*Stall*	*Bubble*	*Bubble*	*Normal*	*Normal*

Control Combination B

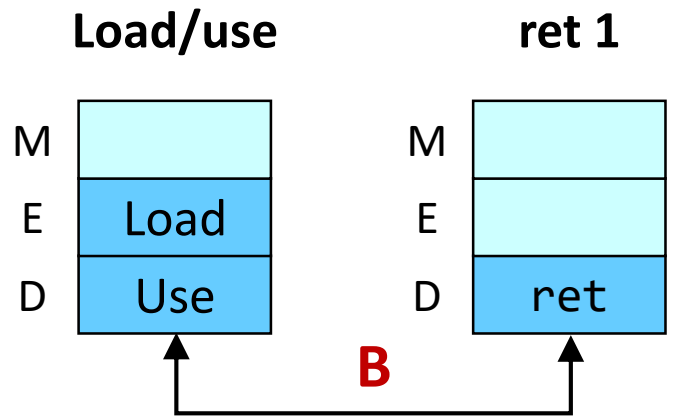
- Would attempt to bubble *and* stall pipeline register D
  - Signaled by processor as pipeline error



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*

Handling Control Combination B

- 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*

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*

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