CS429: Computer Organization and Architecture

Pipeline III

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: July 11, 2019 at 15:02

Data Hazard vs. Control Hazard

There are two types of *hazards* that interfere with flow through a pipeline.

Data hazard: values produced from one instruction are not available when needed by a subsequent instruction.

Control hazard: a branch in the control flow makes ambiguous what is the next instruction to fetch.

How Do We Fix the Pipeline? Possibilities:

1. Pad the program with NOPs. That could mean two things:
    - Change the program itself. That violates our Pipeline Correctness Axiom. Why?
    - Make the implementation *behave* as if there were NOPs inserted.
2. That's called *stalling the pipeline*
    - Data hazards:
        - Wait for producing instruction to complete
        - Then proceed with consuming instruction
    - Control hazards:
        - Wait until new PC has been determined, then fetch
        - Make a guess and patch later, if wrong
    - How is this better than inserting NOPs into the program?

How Do We Fix the Pipeline?

3. Forward data within the pipeline
    - Grab the result from somewhere in the pipe
        - After it has been computed
        - But before it has been written back
    - This gives an opportunity to avoid performance degradation due to stalling for hazards.
4. Do some clever combination of these.

The implemented solution (4) is a combination of 2 and 3: forward data when possible and stall the pipeline only when necessary.

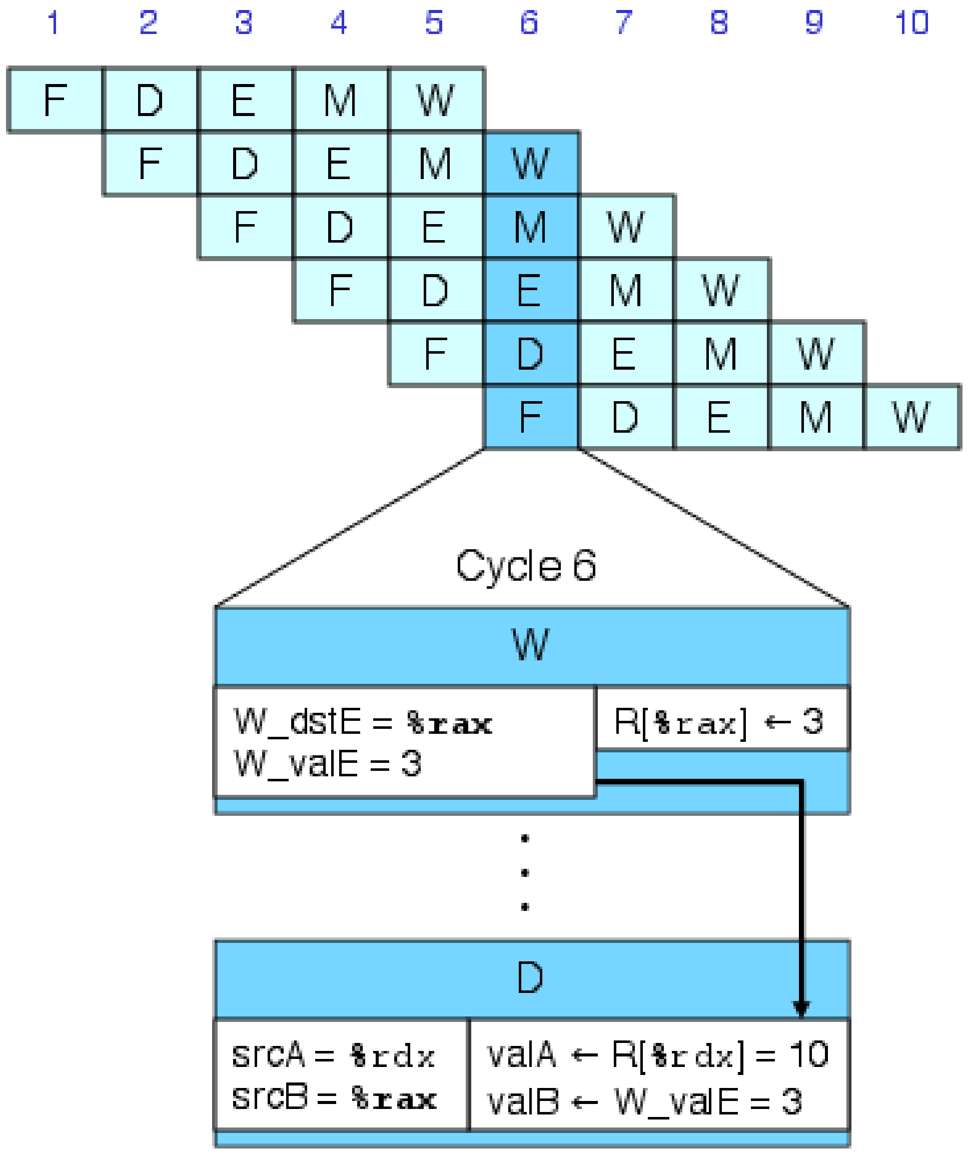
Data Forwarding

```
irmovq   $10, %rdx
irmovq   $3, %rax
addq     %rdx, %rax
```

- **Naive 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 was available 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

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



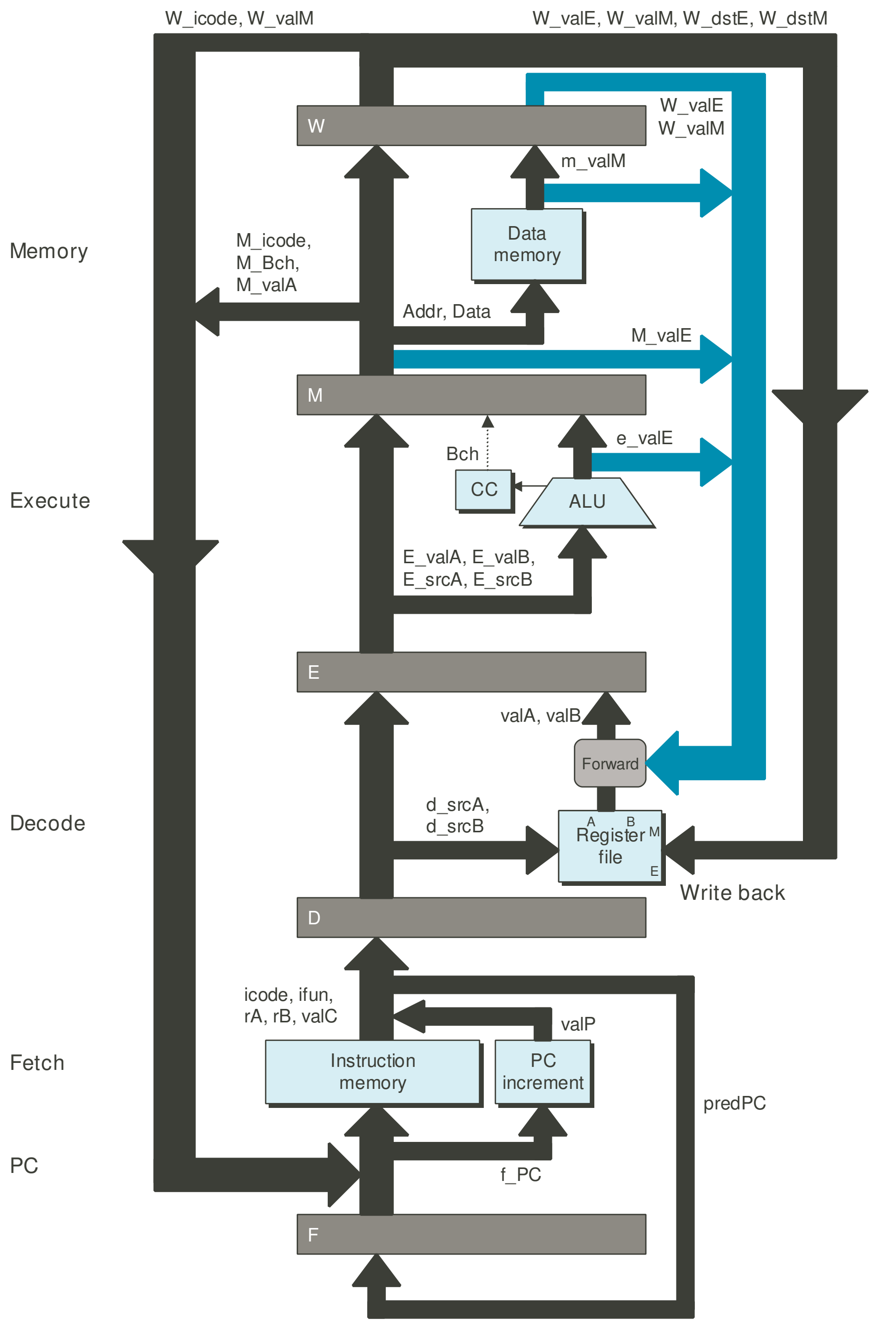
Bypass Paths

Decode Stage:

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

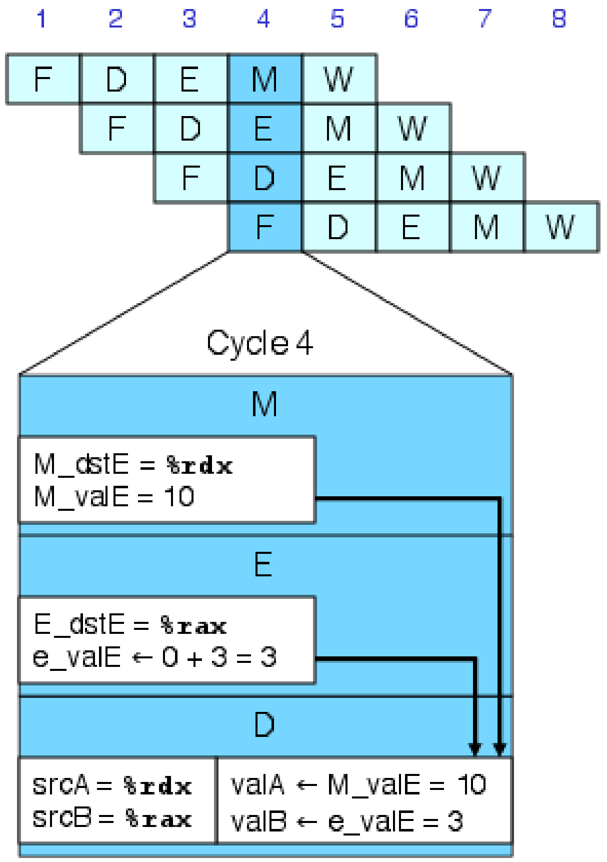
Forwarding Sources:

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



Data Forwarding Example 2

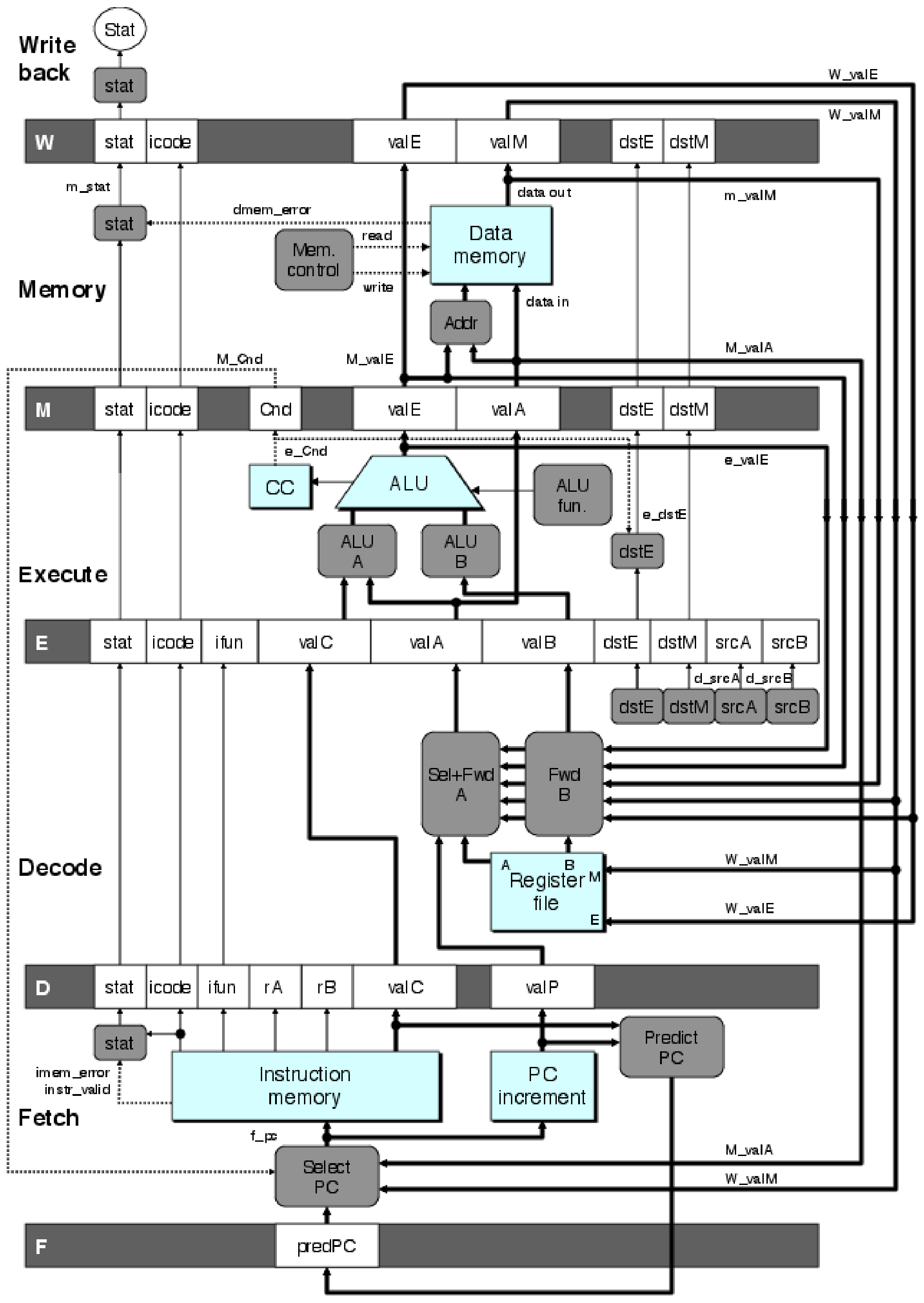
```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



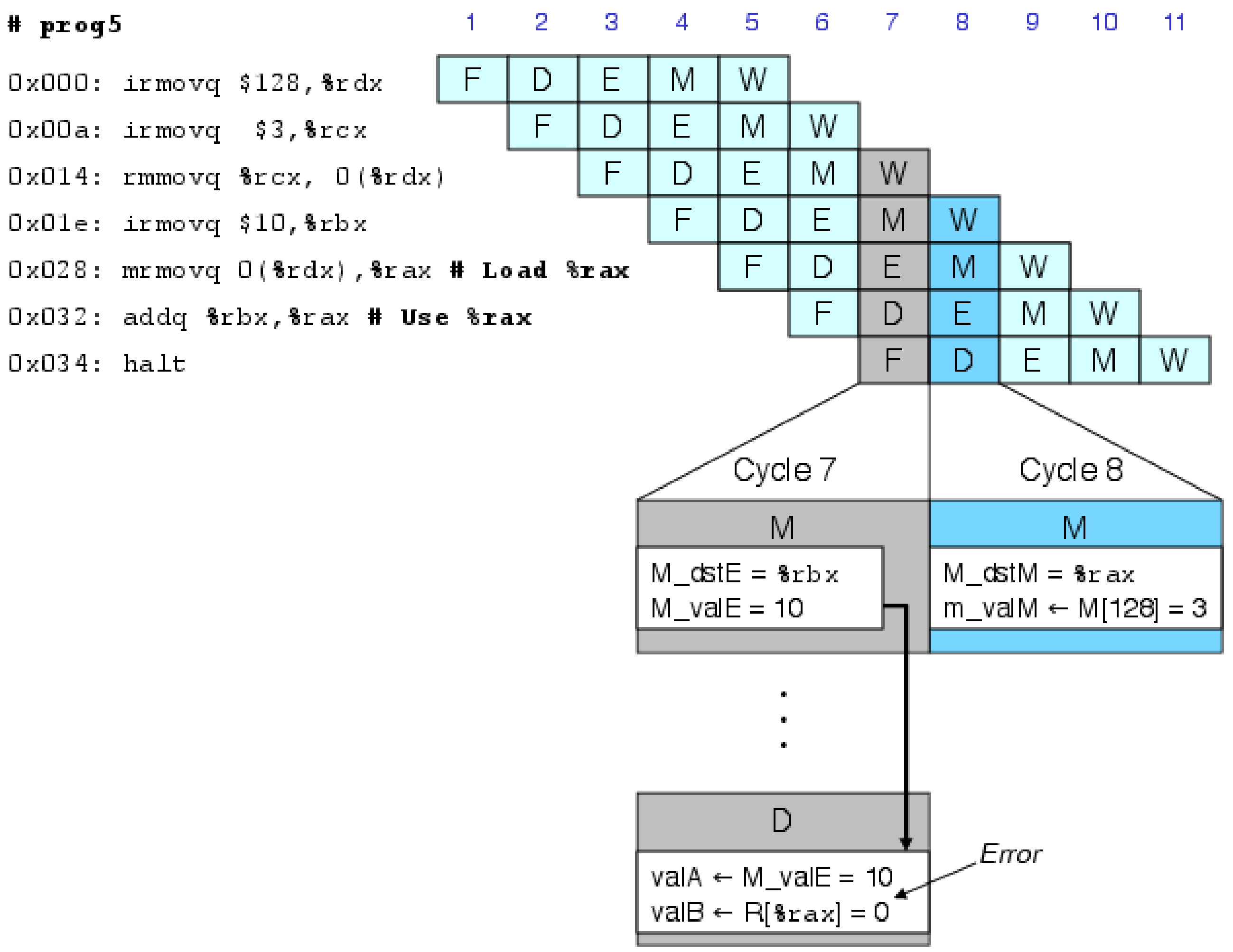
- Register %rdx: generated by ALU during previous cycle; forwarded from memory as valA.
- Register %rax: value just generated by ALU; forward from execute as valB.

Implementing Forwarding

- Add new 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.


Stat
Write back
stat
W
stat
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_stat
data out
m_valM
dmem_error
read
write
Mem. control
Data memory
Memory
Addr
data in
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
e_valE
CC
ALU
ALU fun.
e_dstE
dstE
ALU A
ALU B
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
Decode
A
B
Register file
M
E
W_valM
W_valE
D
stat
icode
ifun
rA
rB
valC
valP
stat
imem_error
instr_valid
Instruction memory
PC increment
Predict PC
Fetch
f_pc
Select PC
M_valA
W_valM
F
predPC

Limitation of Forwarding

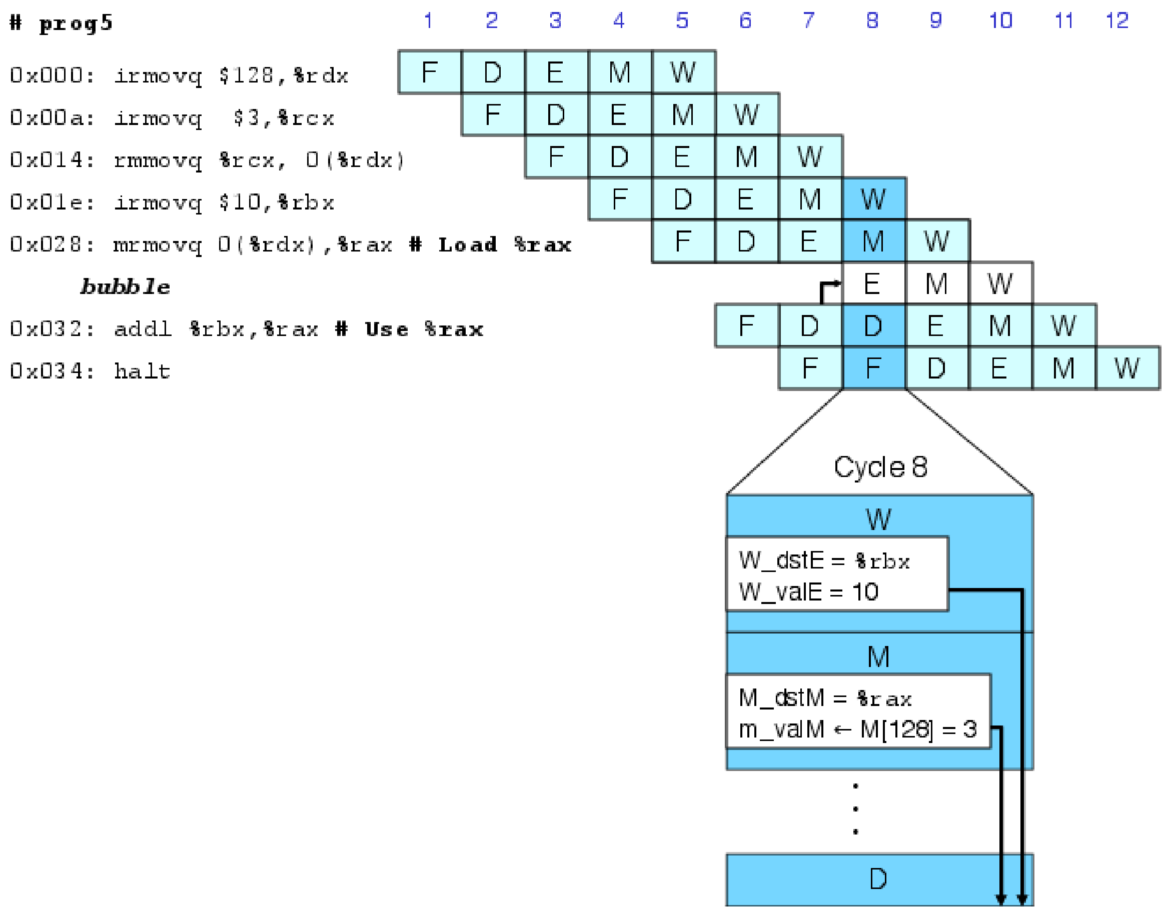


```
# prog5
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```

Load-use (data) dependency:

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

Dealing with Load/Use Hazard



```
# prog5
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: addl %rbx,%rax # Use %rax
0x034: halt
```

- Notice that value needed *is not* in any pipeline register
- Stall using instruction for one cycle; requires one bubble.
- Can pick up loaded value by forwarding from memory stage.

What's a Bubble

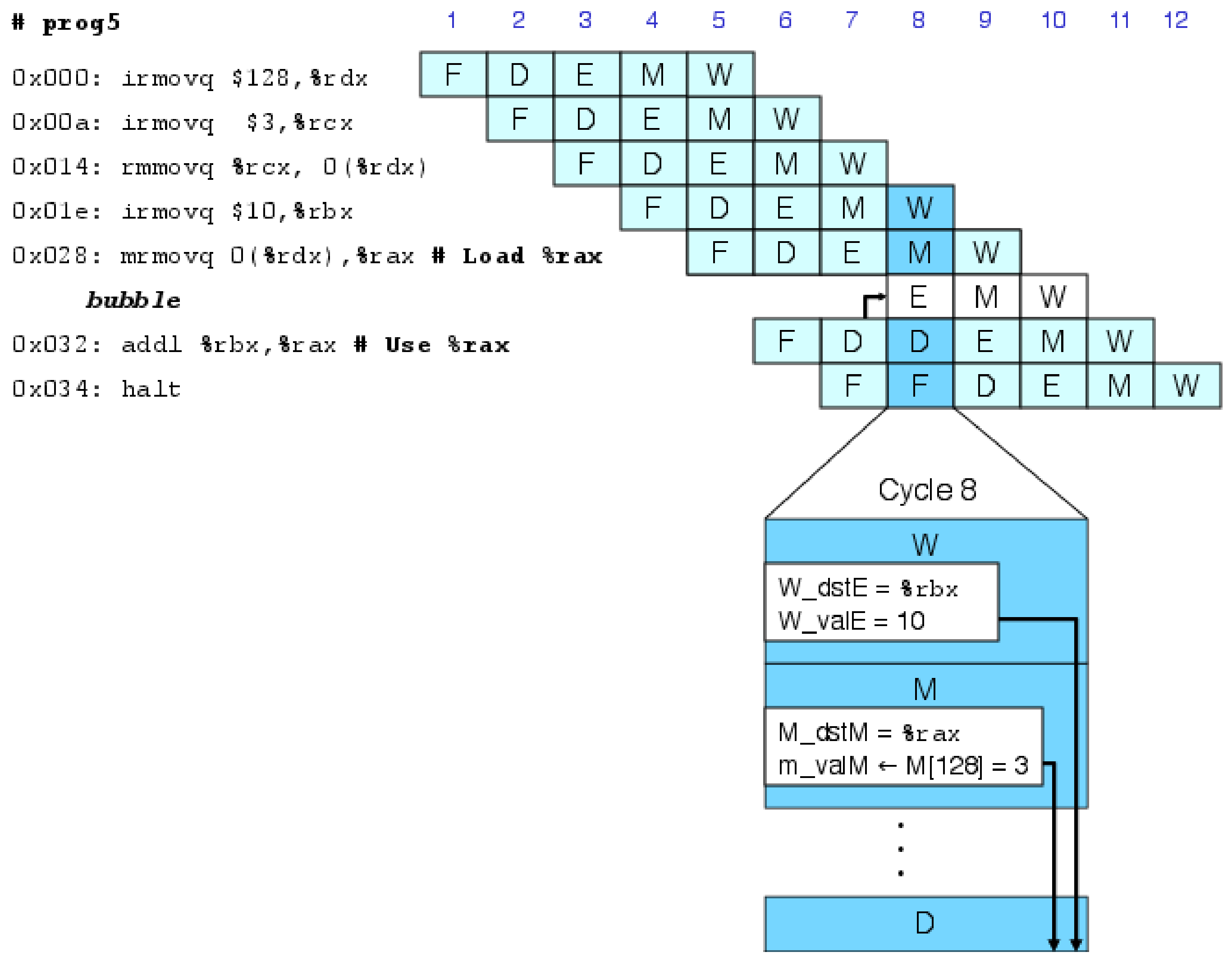
If we stall the pipeline at one stage and let the instructions ahead proceed, that creates a gap that has to be filled.

A *bubble* is a "virtual nop" created by populating the pipeline registers at that stage with values *as if had there been a nop at that point in the program*. The bubble can flow through the pipeline just like any other instruction.

A bubble is used for two purposes:

1. fill the gap created when the pipeline is stalled;
2. replace a real instruction that was fetched erroneously.

Control for Load/Use Hazard

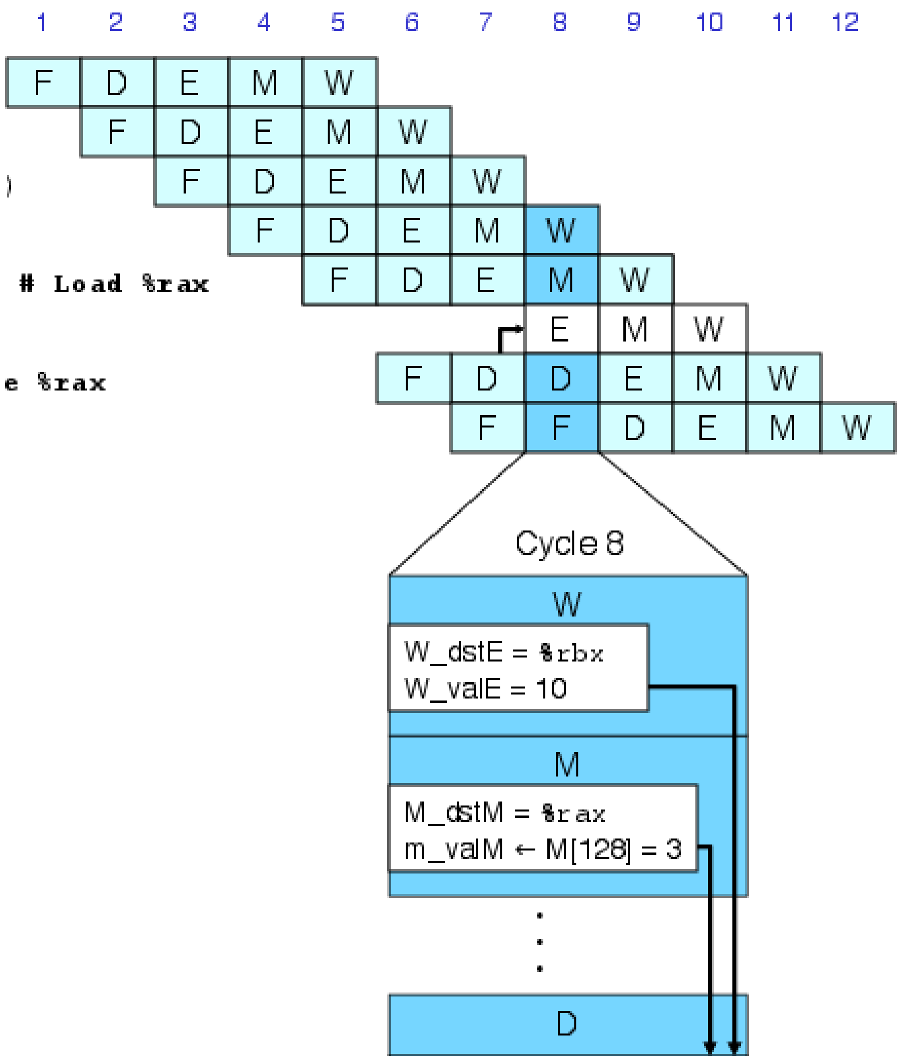


- Stall instructions in fetch and decode stages
- Inject bubble into execute stage.

Control for Load/Use Hazard

```
# prog5

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: addl %rbx,%rax # Use %rax
0x034: halt
```



Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

Control Hazards: Recall Our Prediction Strategy

- **Instructions that don't transfer control:**
  - Predict next PC to be valP; this is always reliable.
- **Call and Unconditional Jumps:**
  - Predict next PC to be valC (destination); this is always reliable.
- **Conditional Jumps:**
  - Predict next PC to be valC (destination).
  - Only correct if the branch is taken; right about 60% of the time.
- **Return Instruction:**
  - Don't try to predict.

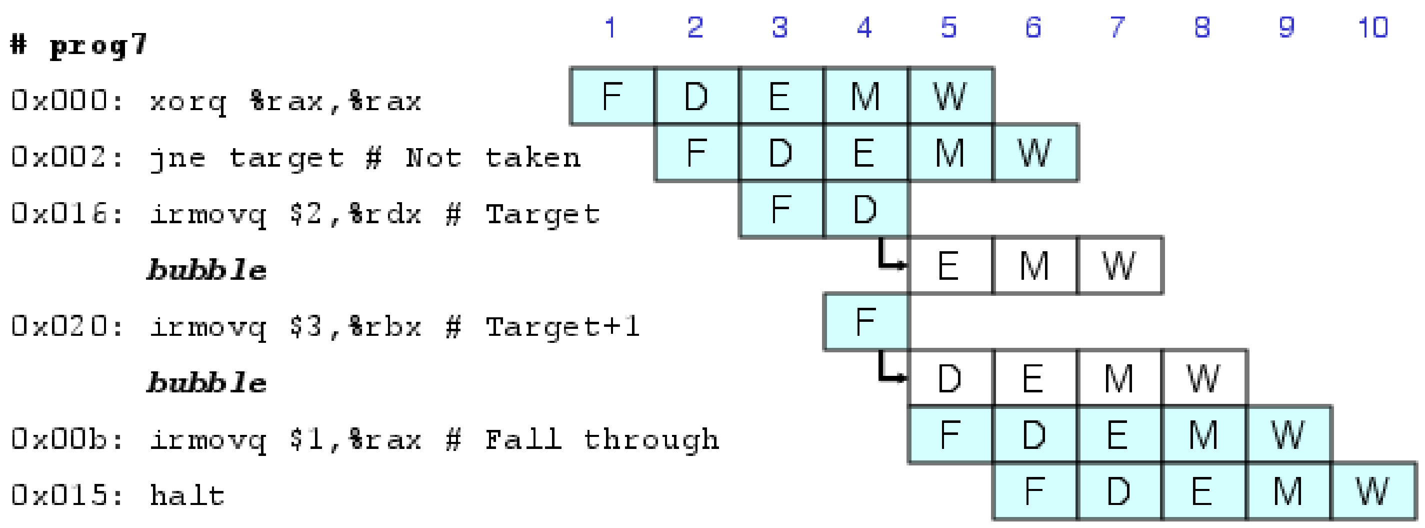
Note that we could have used a different prediction strategy

Branch Misprediction Example

```
0x000:     xorq   %rax, %rax
0x002:     jne    target          # Not taken
0x00b:     irmovq $1, %rax        # Fall through
0x015:     halt
0x016: target:
0x016:     irmovq $2, %rdx        # Target
0x020:     irmovq $3, %rcx        # Target + 1
0x02a:     halt
```

Should only execute the first 4 instructions.

Handling Misprediction

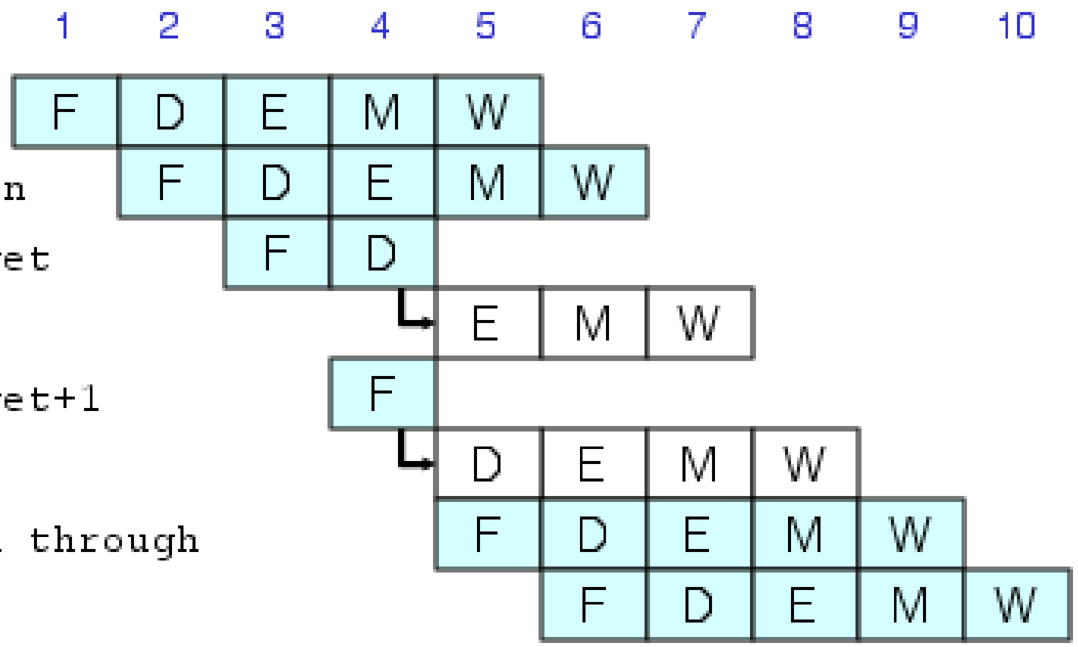


```
# prog7
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

- **Predict branch as taken**
  - Fetch 2 instructions at target
- **Cancel when mispredicted**
  - Detect branch not taken in execute stage
  - On following cycle, replace instruction in execute and decode stage by bubbles.
  - No side effects have occurred yet.

Control for Misprediction

```
# prog7
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

Return Example

```
    irmovq  Stack, %rsp    # Initialize stack pointer
    call    p              # Procedure call
    irmovq  $5, %rsi       # Return point
    halt
 .pos 0x20
p: irmovq  $-1, %rdi      # procedure
    ret
    irmovq  $1, %rax       # should not be executed
    irmovq  $2, %rcx       # should not be executed
    irmovq  $3, %rdx       # should not be executed
    irmovq  $4, %rbx       # should not be executed
 .pos 0x100
Stack:                     # Stack pointer
```

Without stalling, could execute three additional instructions.

Correct Return Example

```
ret
bubble
bubble
bubble
irmovq  $5, %rsi    # Return
```

- As `ret` passes through pipeline, stall at fetch stage—while in decode, execute, and memory stages.
- Inject bubble into decode stage.
- Release stall when `ret` reaches write-back stage.

Control for Return

This is a bit confusing, because there are actually three bubbles inserted. Stall until the ret reaches write back.

```
ret
bubble
bubble
bubble
irmovq  $5, %rsi    # Return
```

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

Pipeline Summary

Data Hazards

- Most handled by forwarding with no performance penalty
- Load / use hazard requires one cycle stall

Control Hazards

- Cancel instructions when detect mispredicted branch; two cycles wasted
- Stall fetch stage while `ret` pass through pipeline; three cycles wasted.

Control Combinations

- Must analyze carefully
- First version had a subtle bug
- Only arises with unusual instruction combination

Performance Analysis with Pipelining

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Cycle}}$$

- Ideal pipelined machine: Cycles per Instruction (CPI) $= 1$
  - One instruction completed per cycle.
  - But much faster cycle time than unpipelined machine.
- However, hazards work against the ideal
  - Hazards resolved using forwarding are fine with no penalty.
  - Stalling degrades performance and instruction completion rate is interrupted.
- CPI is a measure of the “architectural efficiency” of the design.

Computing CPI

CPI is a function of useful instructions and bubbles:

$$CPI = \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i}$$

You can reformulate this to account for:

- load/use penalties (lp): 1 bubble
- branch misprediction penalties (mp): 2 bubbles
- return penalties (rp): 3 bubbles

$$CPI = 1.0 + \frac{lp + mp + rp}{C_i}$$

Computing CPI (2)

- So, how do we determine the penalties?
  - Depends on how often each situation occurs on average.
  - How often does a load occur and how often does that load cause a stall?
  - How often does a branch occur and how often is it mispredicted?
  - How often does a return occur?
- We can measure these using:
  - a simulator, or
  - hardware performance counters.
- We can also estimate them through historical averages.
  - Then use estimates to make early design tradeoffs for the architecture.

Computing CPI (3)

Assume some hypothetical counts:

Cause	Name	Instruction Frequency	Condition Frequency	Stalls	Product
Load/use	lp	0.30	0.3	1	0.09
Mispredict	mp	0.20	0.4	2	0.16
Return	rp	0.02	1.0	3	0.06
Total penalty					0.31

$$\text{CPI} = 1 + 0.31 = 1.31 == 31\%$$

This is not ideal.

This gets worse when:

- you also account for non-ideal memory access latency;
- deeper pipeline (where stalls per hazard increase).