

CS429: Computer Organization and Architecture

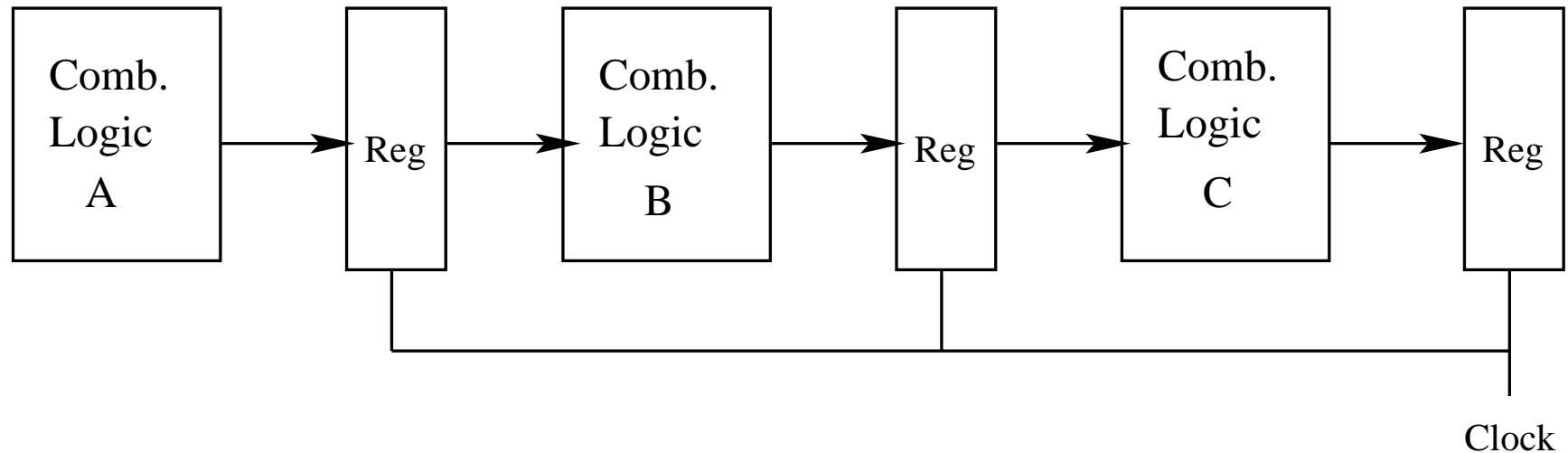
Pipeline II

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

Last updated: July 11, 2019 at 14:45

Pipeline Registers

Recall that one requirement of pipelining is inserting sequential logic between pipeline stages to hold the intermediate values.

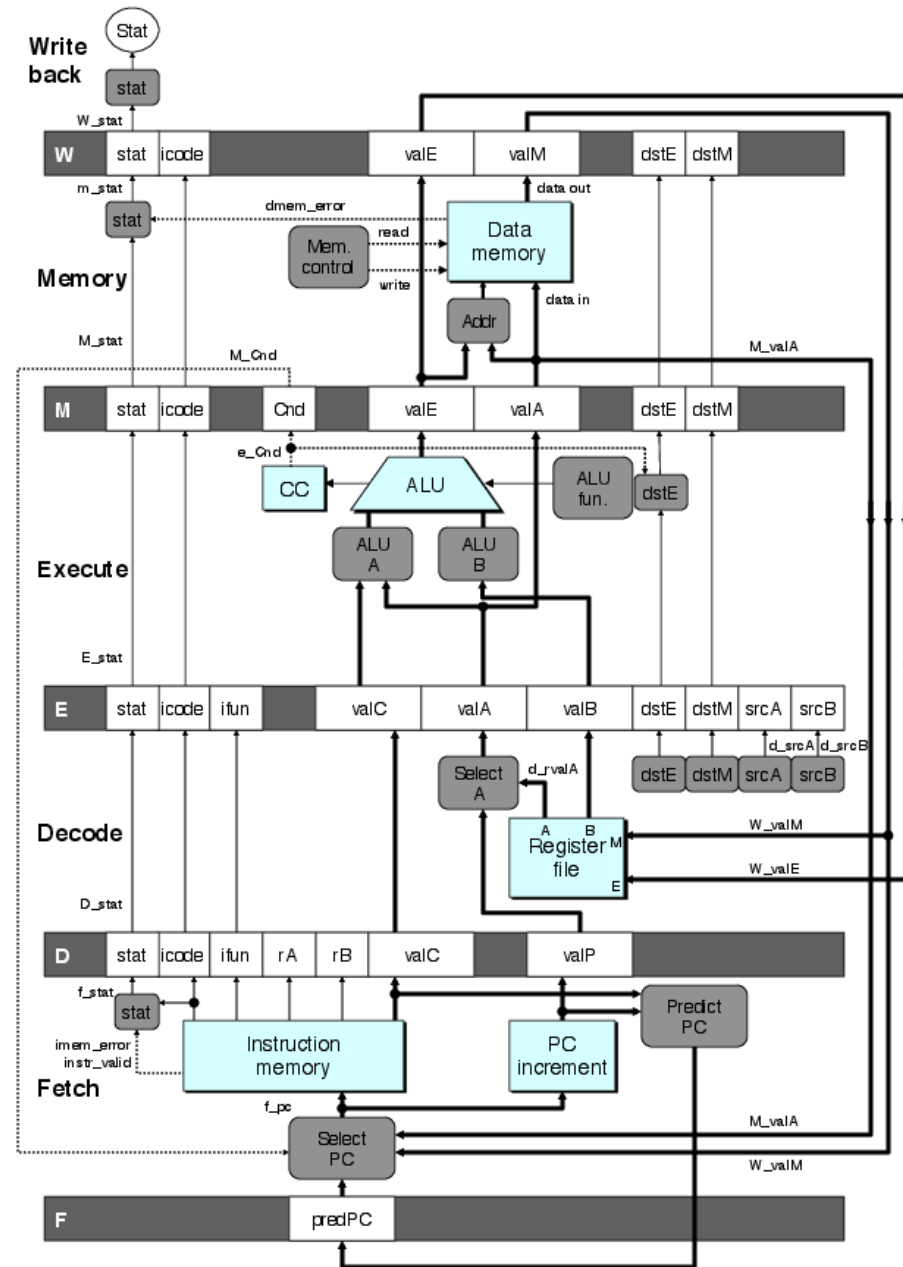


In general, these are called *pipeline registers*.

Idea: Insert “pipeline registers” to hold intermediate values after each pipeline stage.

Forward (Upward) Paths

- Values passed from one stage to the next.
- Cannot jump past stages.
- E.g., valC must pass through decode



Pipeline Registers

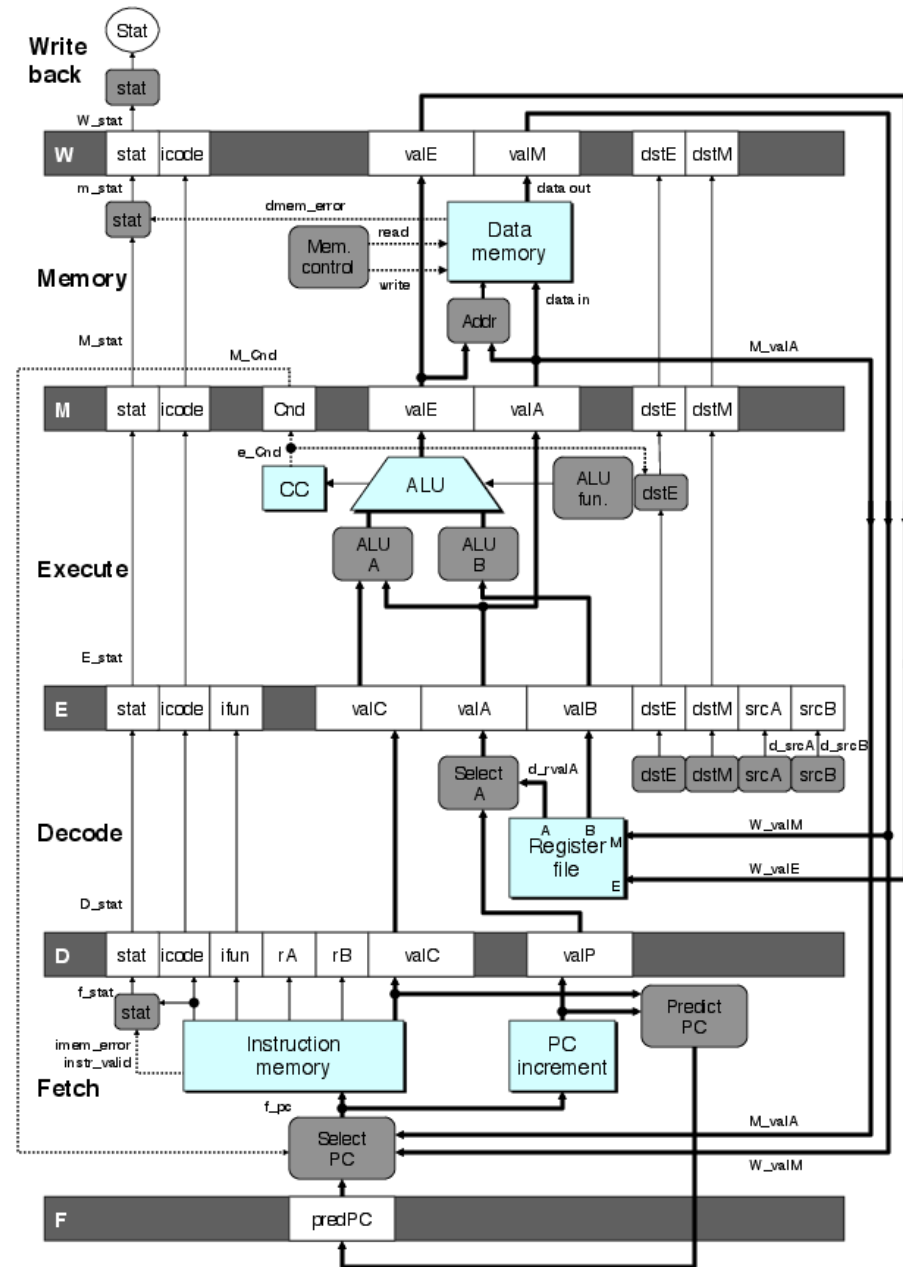
The term “register” is overloaded. Don’t confuse the two uses.

- ① It means the 16 named registers in the register file.
- ② It also means data storage items within the implementation.

Pipeline “registers” are not user-visible processor registers.

Feedback Paths

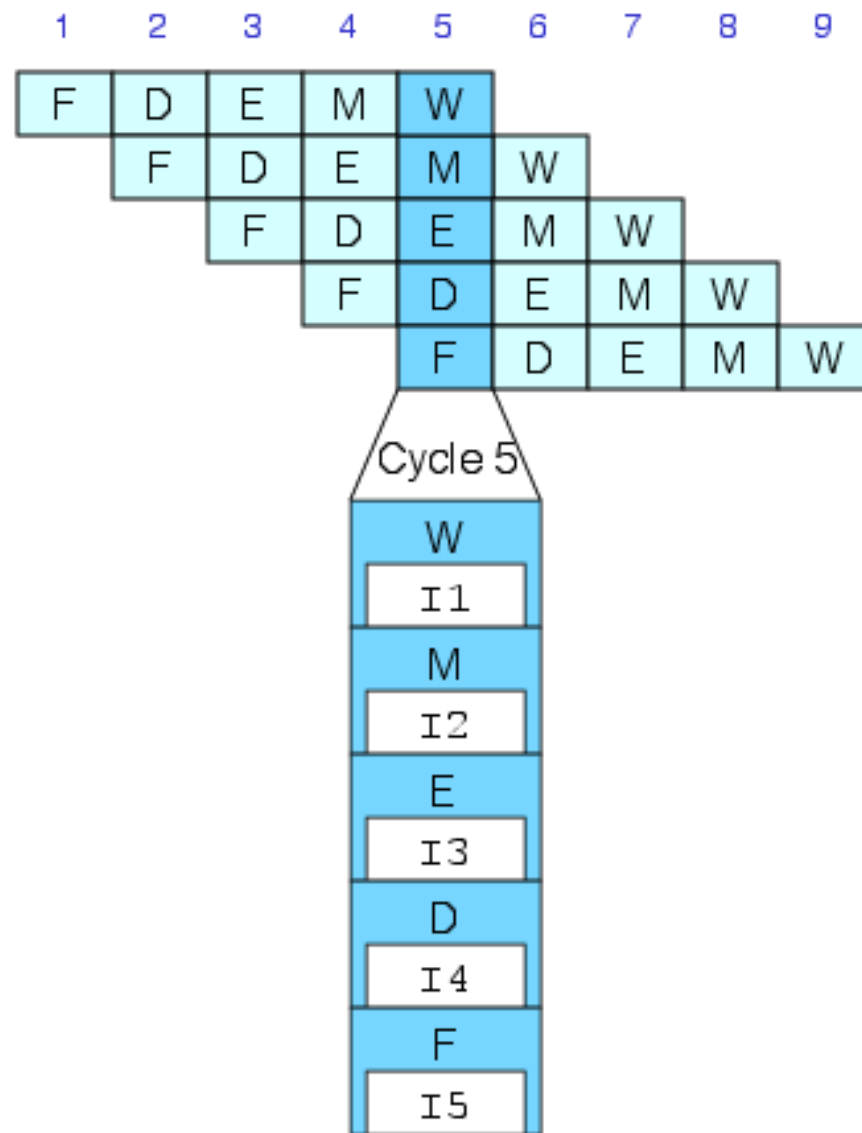
- Predicted PC: guess value of next PC
- Branch information:
 - Jump taken/not taken
 - Fall-through or target address
- Return address: read from memory (stack)
- Register updates: To register file write ports



The Pipeline Ideal

Suppose all registers are initialized to zero.

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



Data Dependencies: 3 Nop's

prog1

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

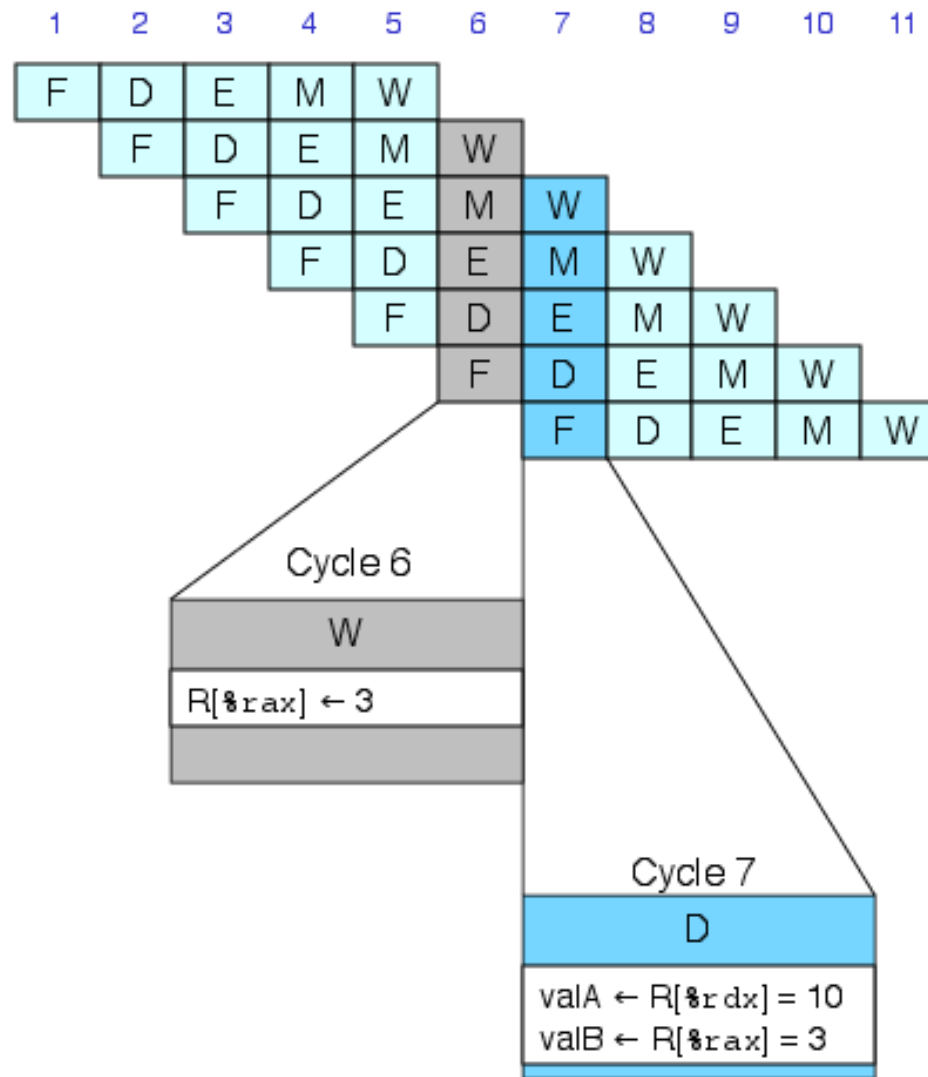
0x014: nop

0x015: nop

0x016: nop

0x017: addq %rdx,%rax

0x019: halt



Data Dependencies: 2 Nop's

prog2

0x000: irmovq \$10,%rdx

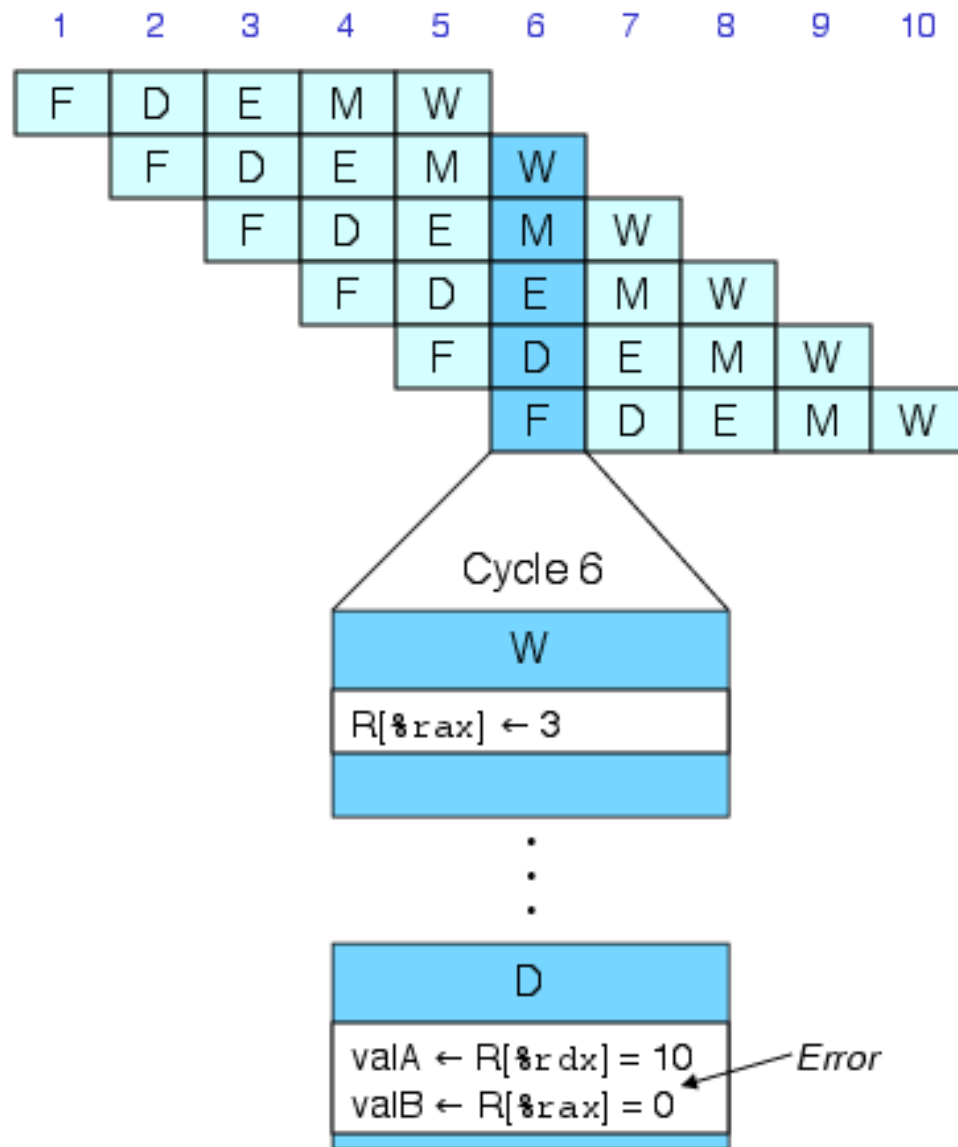
0x00a: irmovq \$3,%rax

0x014: nop

0x015: nop

0x016: addq %rdx,%rax

0x018: halt



Data Dependencies: 1 Nop

prog3

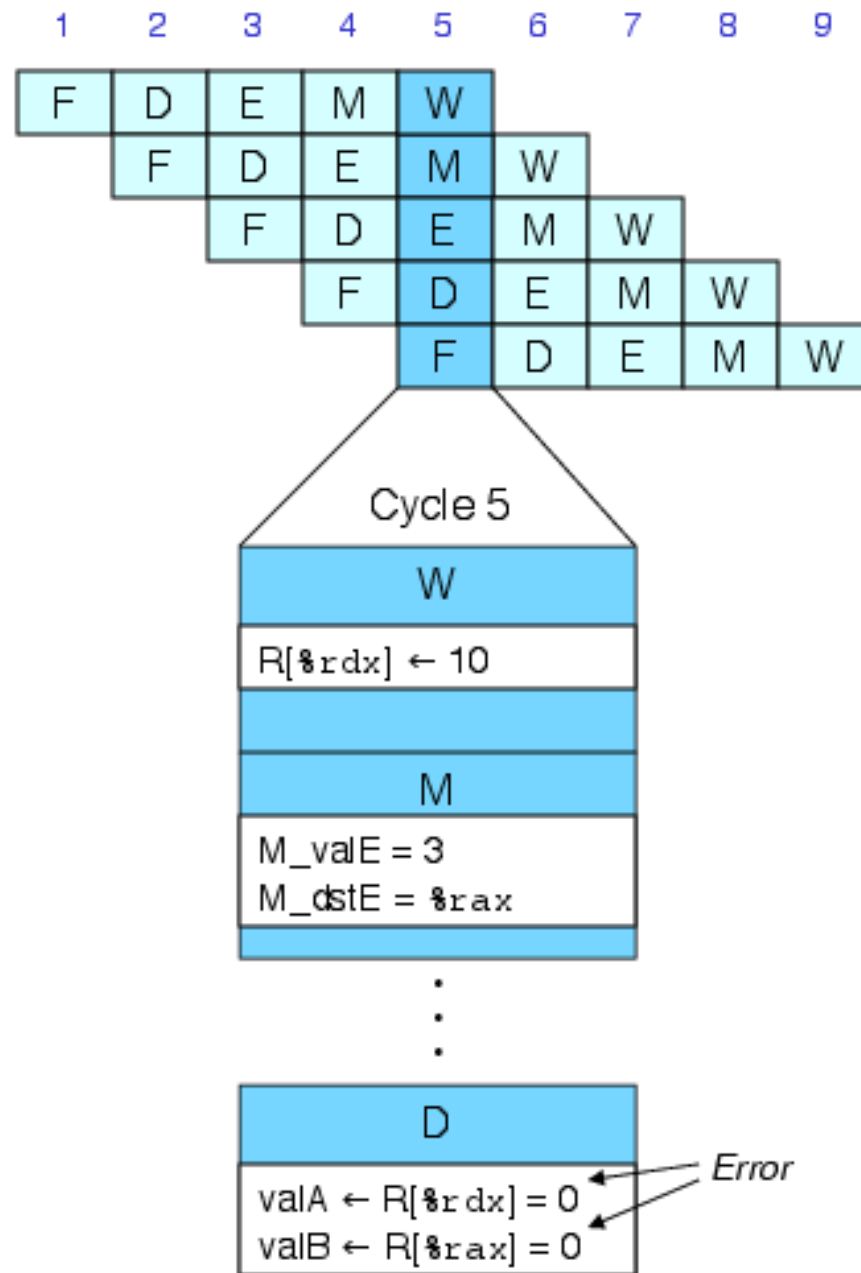
0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

0x014: nop

0x015: addq %rdx,%rax

0x017: halt



Data Dependencies: No Nop's

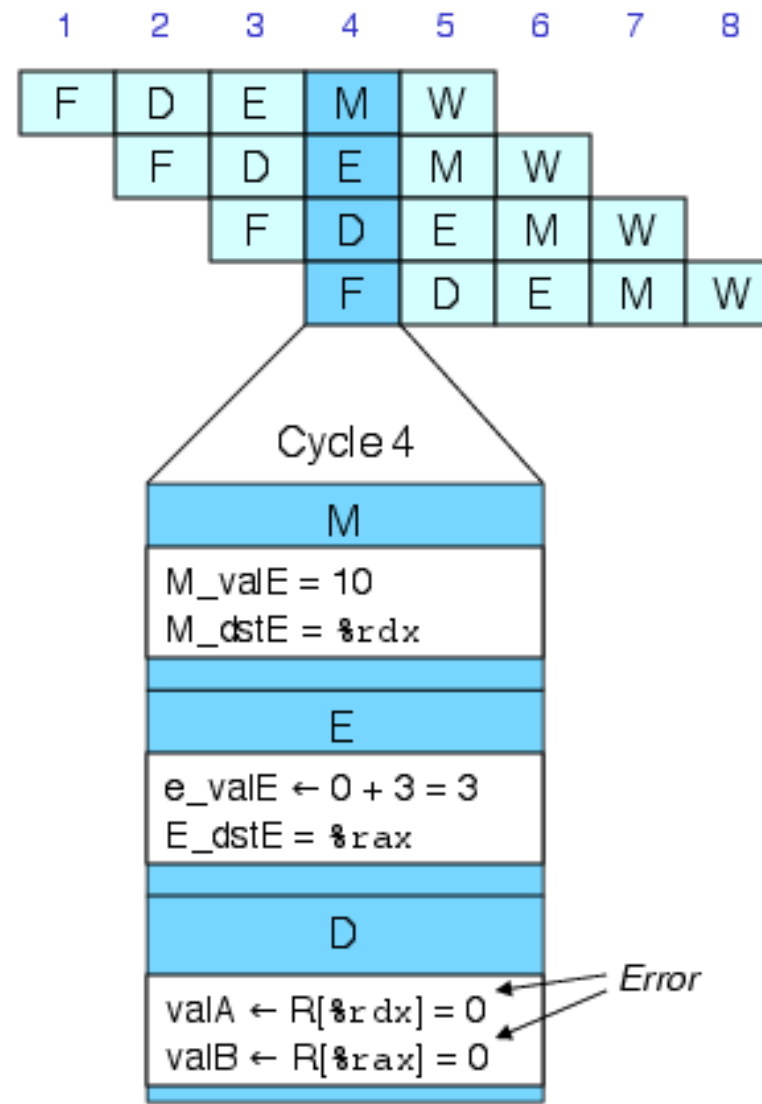
prog4

0x000: irmovq \$10,%rdx

0x00a: irmovq \$3,%rax

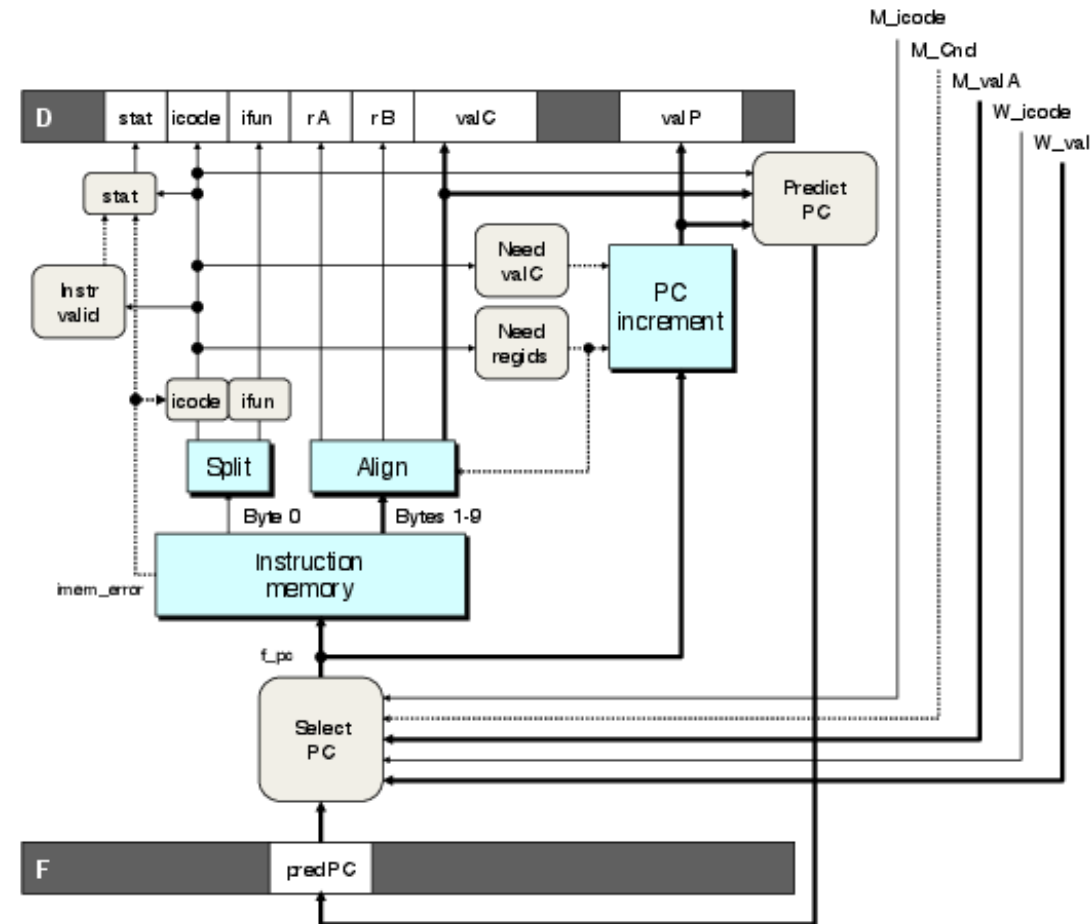
0x014: addq %rdx,%rax

0x016: halt



Control Hazards: Predicting the PC

- Start fetch of a new instruction after the current one has completed the fetch stage.
- There's not enough time to reliably determine the next instruction.
- Guess which instruction will follow.
- Then, recover if the prediction was incorrect.



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. *Why do you suppose it's better than 50%*
- **Return Instruction:**
 - Don't try to predict.

Mispredicted Jump:

- Will see branch flag once instruction reaches memory stage.
- Can get fall-through PC from `valP`.
- Must throw away instructions fetched between prediction and resolution. **How many instructions?**

Return Instruction:

- Will get return PC when `ret` reaches write-back stage.
- Since we can't predict, we don't fetch anything; no clean-up is needed, but 3 cycles are lost.