Pipelining

’20H2

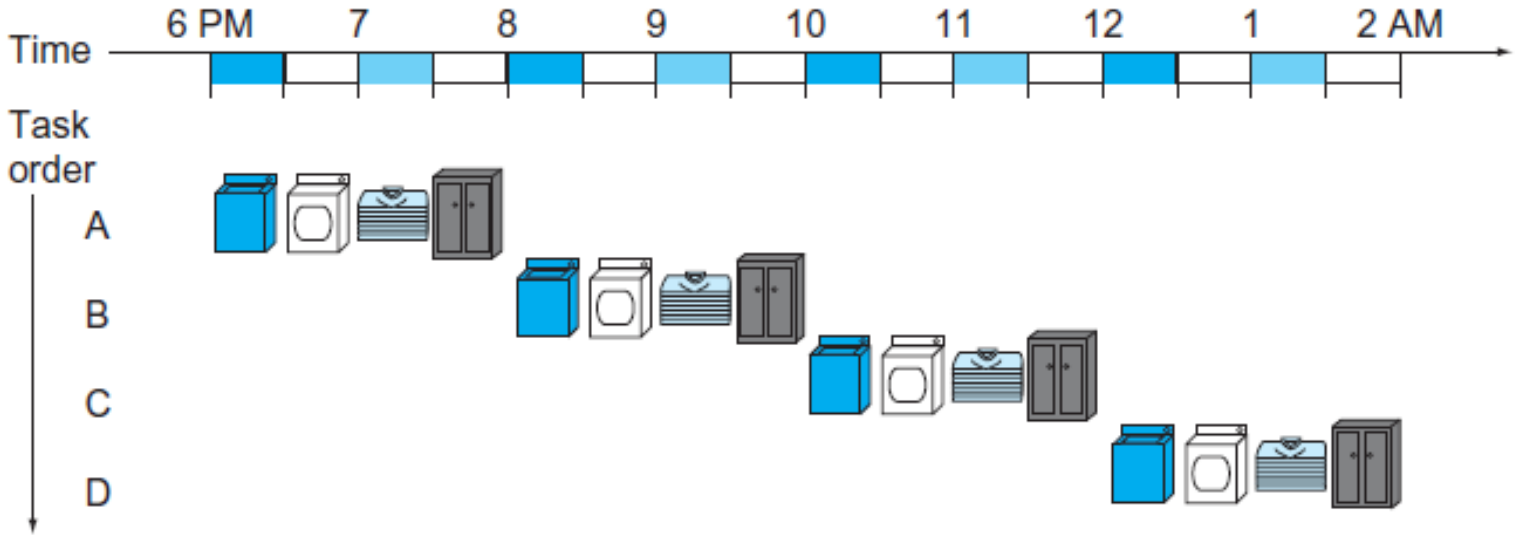
송 인 식

Outline

- Pipelining Concepts
- Pipeline Hazards

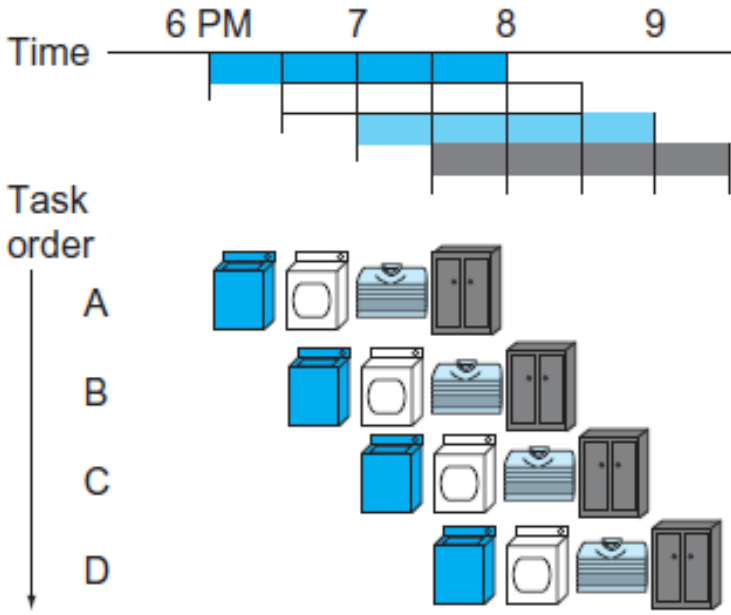
Sequential Processing

- Sequential processing: Wash-Dry-Fold-Store



Pipelined Processing

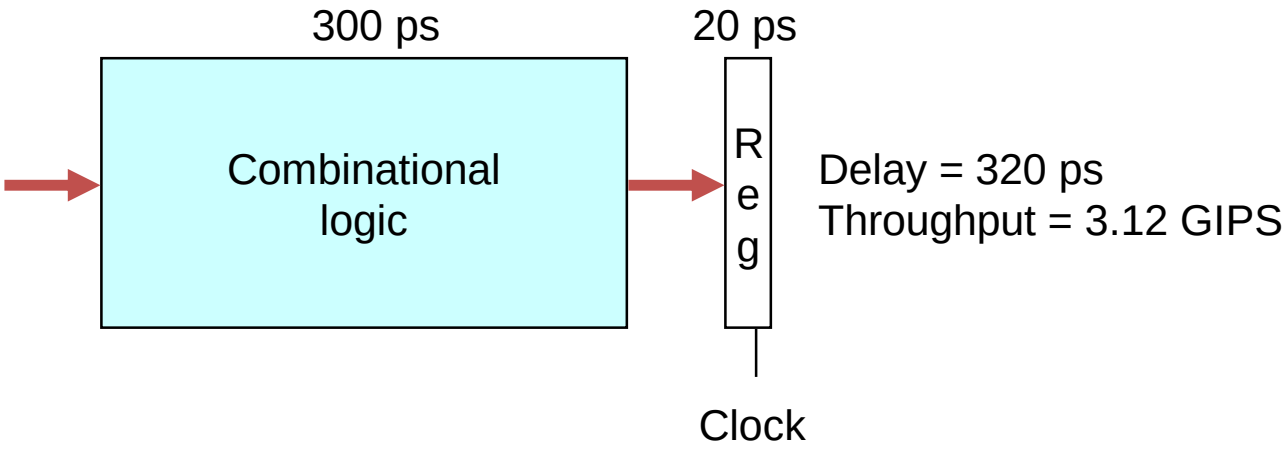
- Overlapping execution
- Parallelism improves performance



- Four loads:
  Speedup = 8 / 3.5 = 2.3
- Non-stop:
  Speedup = $2n$ / (0.5n + 1.5) ≈ 4
  = number of stages

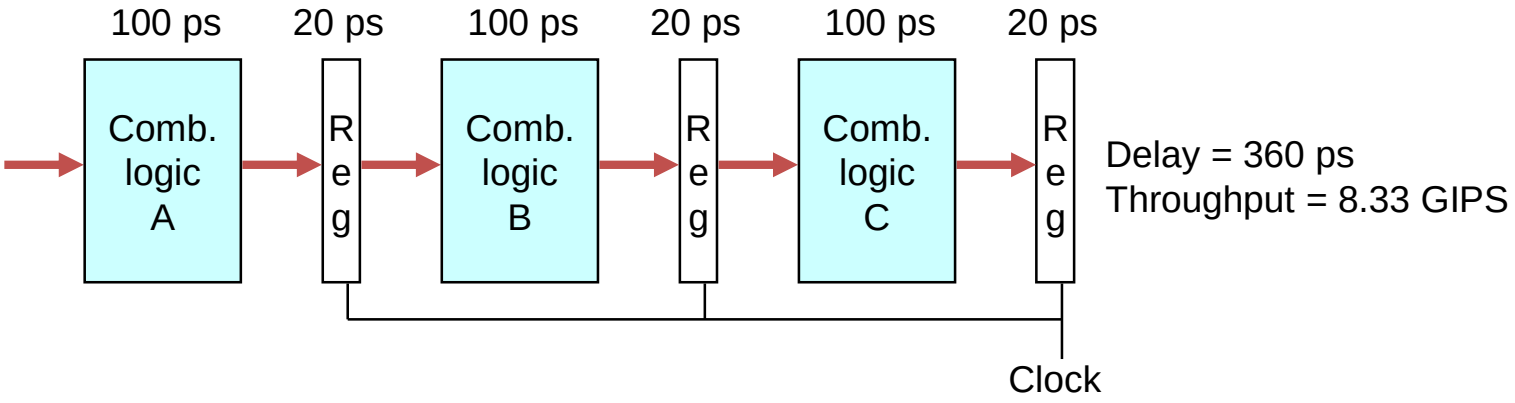
Sequential Execution

- An example CPU
  - Instruction execution requires total of 300 ps
  - Additional 20 ps to save results in registers
  - Must have clock cycle of at least 320 ps



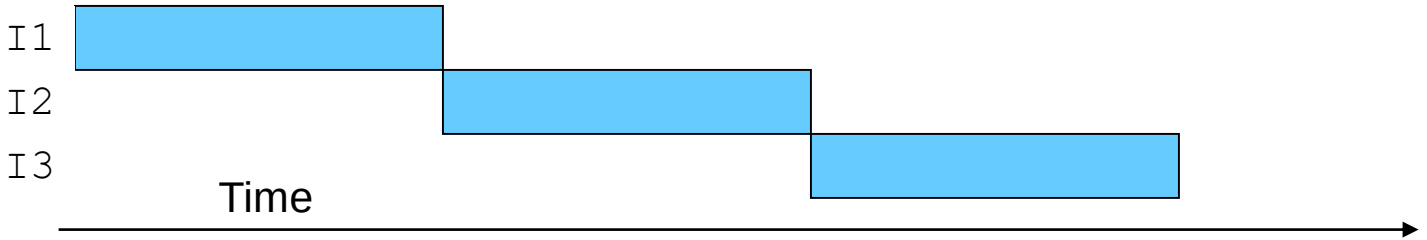
Pipelined Execution

- Example: 3-way pipelined version
  - 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
  - But, throughput improved by 2.67x

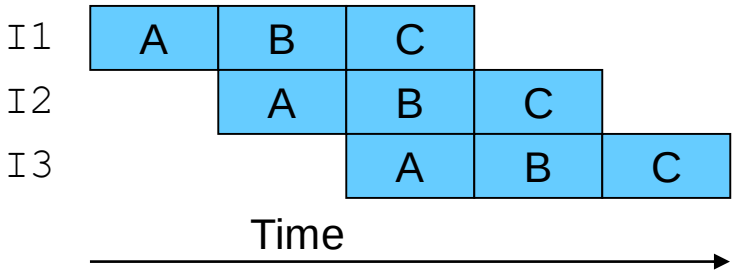


Pipeline Diagrams

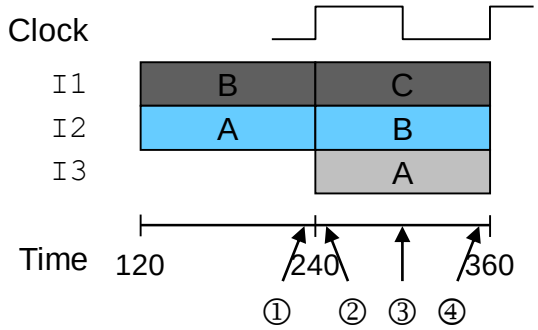
- Unpipelined (= sequential)
  - Cannot start new operation until previous one completes



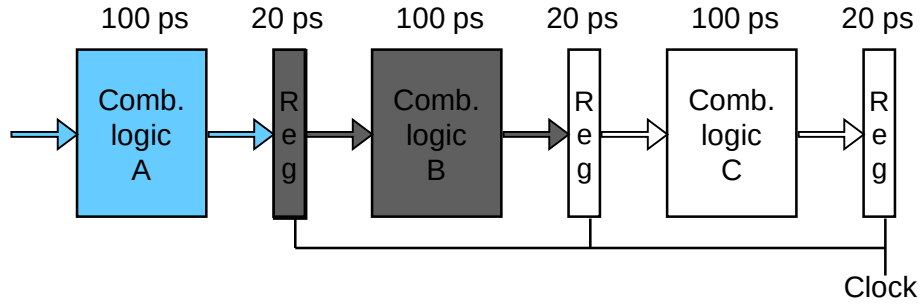
- 3-way pipelined
  - Up to 3 operations in process simultaneously



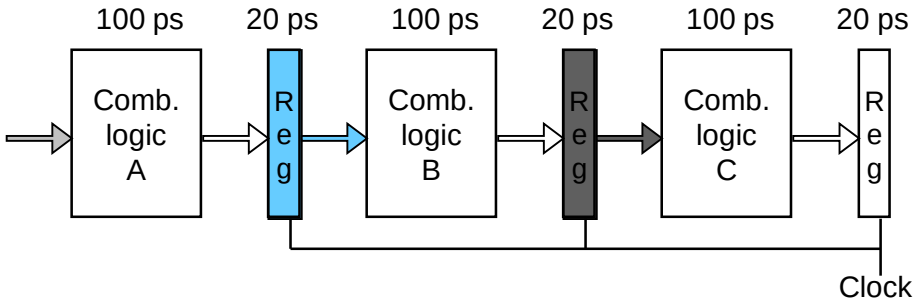
Operating a Pipeline



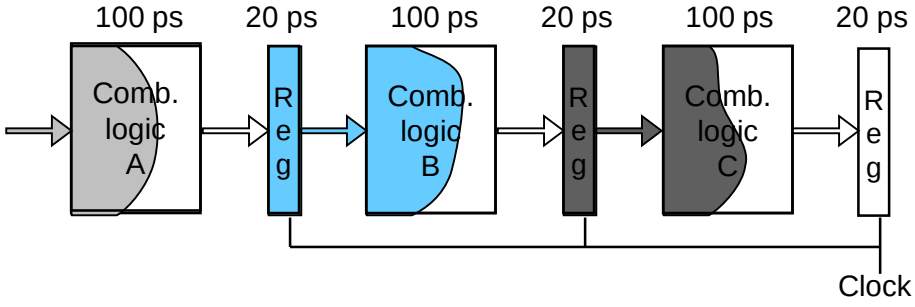
① Time = 239



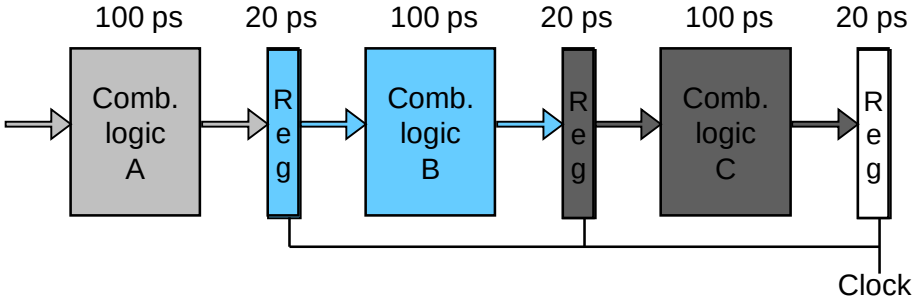
② Time = 241



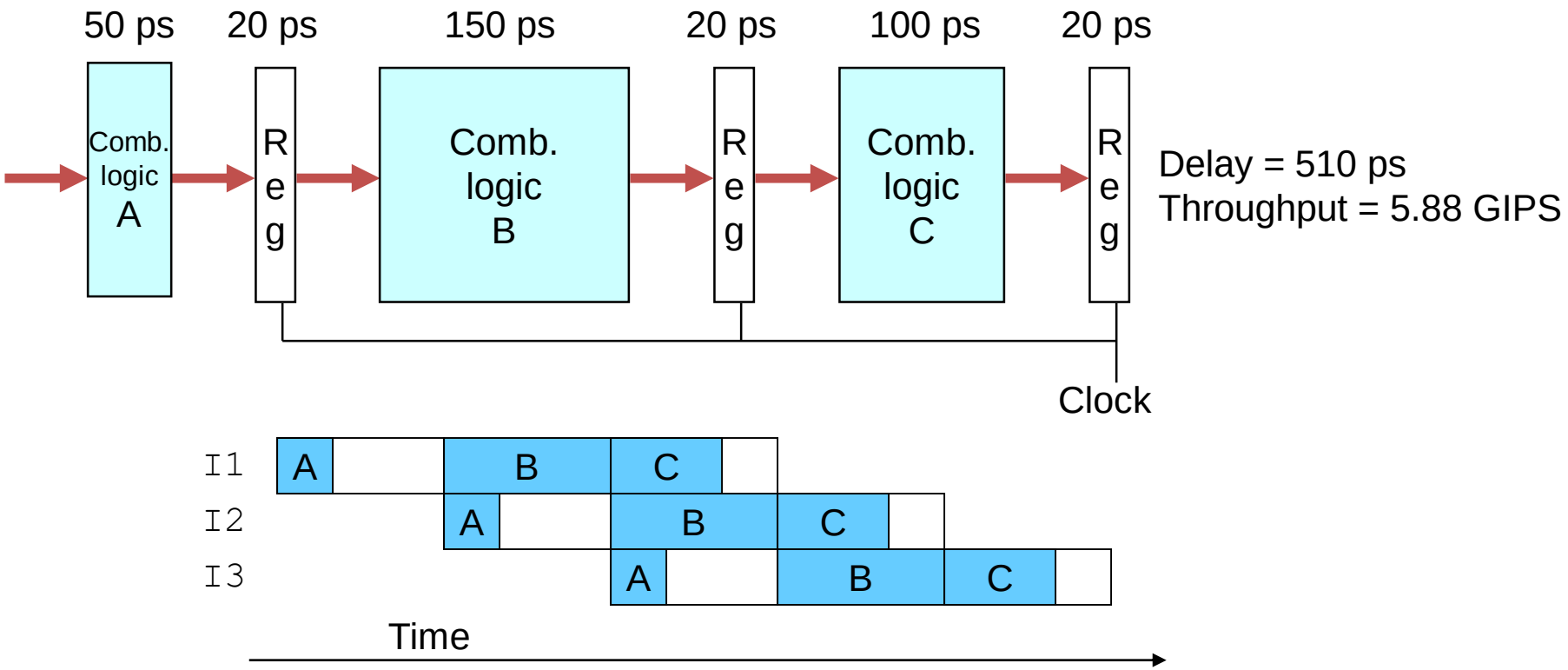
③ Time = 300



④ Time = 359

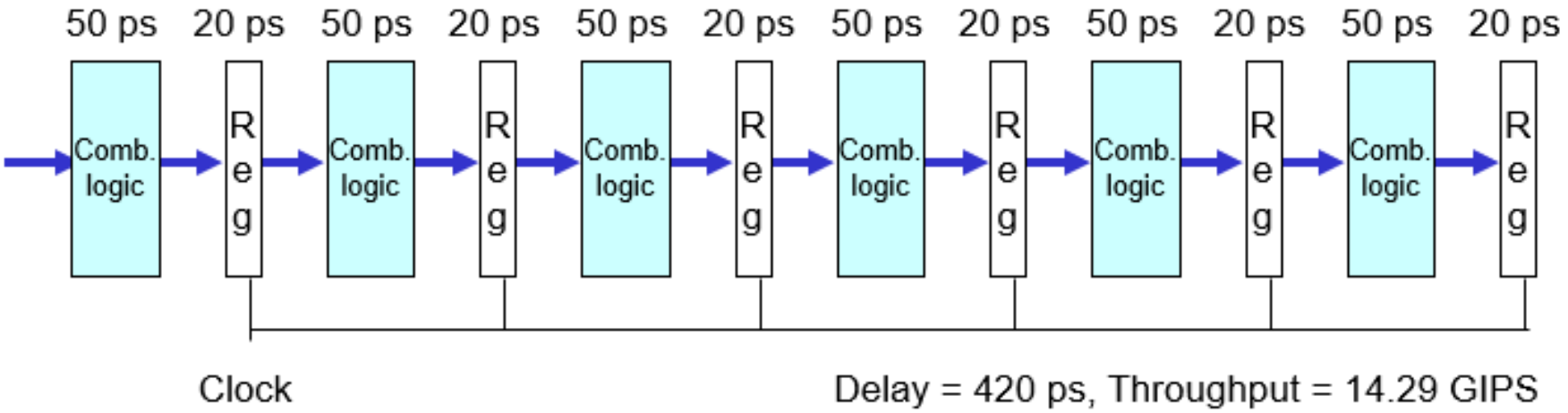


Limitations: Nonuniform Delays



- 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 pipeline deepens, 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

Outline

- Pipelining Concepts
- Pipeline Hazards

Basic Steps of Execution

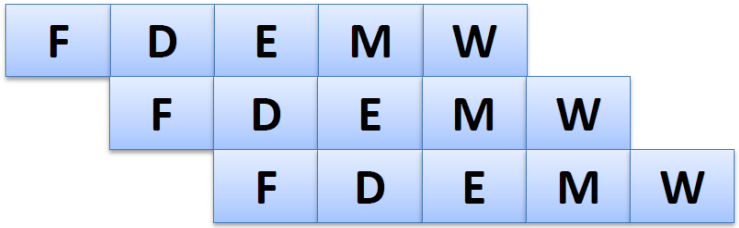
1. Instruction fetch step (F)
2. Instruction decode/register fetch step (D)
3. Execution/effective address step (E)
4. Memory access (M)
5. Register write-back step (W)

- Used in pipelined Y86-64 implementation later

Pipelined Instruction Execution

- Sequential Execution

- Pipelined Execution



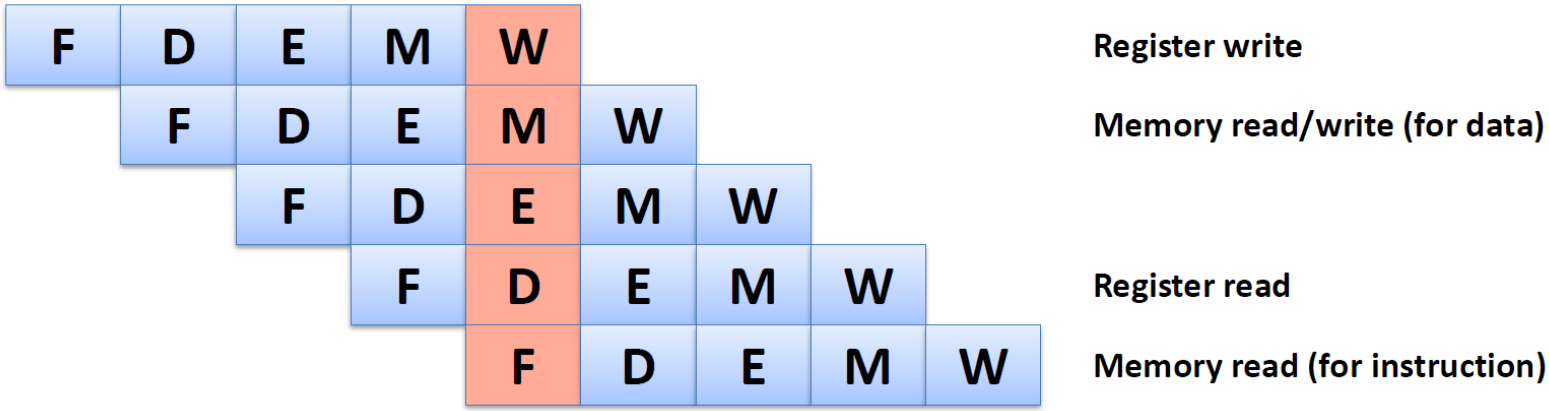

```
addq %rcx, %rax
    subq %rdx, %rbx
       andq %rdx, %rcx
```

Major Hurdles (Hazards) of Pipelining

- Situations that prevent starting the next instruction in the next cycle

- Structural hazard
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structural Hazard

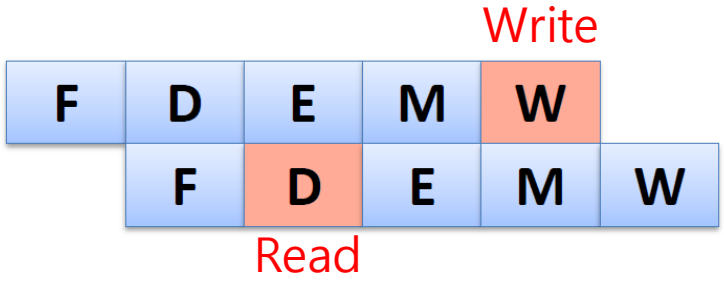


- Solution: resource duplication
  - Separate I and D caches for memory access conflict
  - Time-multiplexed or multi-port register file for register file access conflict

Data Hazard

- Read After Write (RAW) Hazard
  - An instruction tries to read operand before the previous instruction writes it
  - Called a "(true) dependence"
  - This hazard results from an actual need for communication

```
addq    %rdx, %rax
subq    %rax, %rcx
```



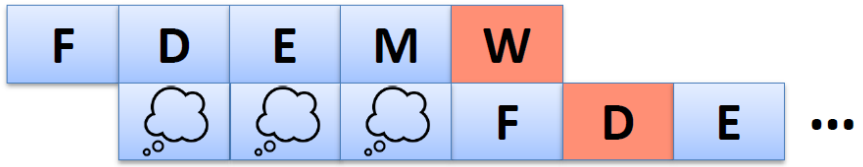
Solutions to Data Hazard

- Freezing the pipeline
- (Internal) Forwarding
- Compiler scheduling

Freezing The Pipeline

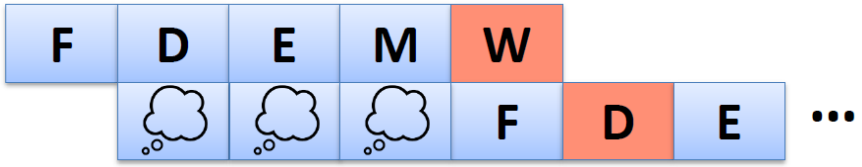
- Stall the pipeline until dependences are resolved
- ALU result to next instruction (3 stalls)

```
addq    %rdx, %rax
subq    %rax, %rcx
```



- Load result to next instruction (3 stalls)

```
mrmovq 0(%rdx), %rax
addq    %rax, %rcx
```

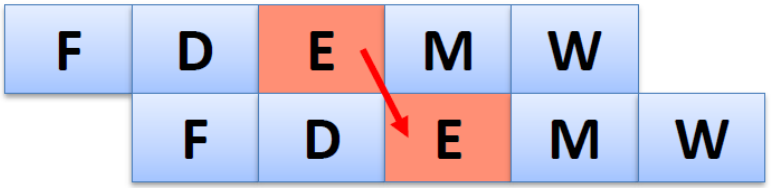


(Internal) Forwarding

- Don't wait for the result to be stored in a register
- Requires extra connections in the datapath

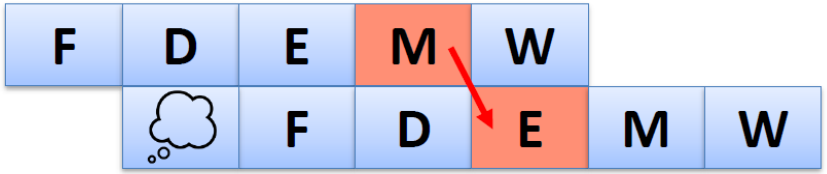
- ALU result to next instruction (no stall)

```
addq    %rdx, %rax
subq    %rax, %rcx
```



- Load result to next instruction (1 stall)

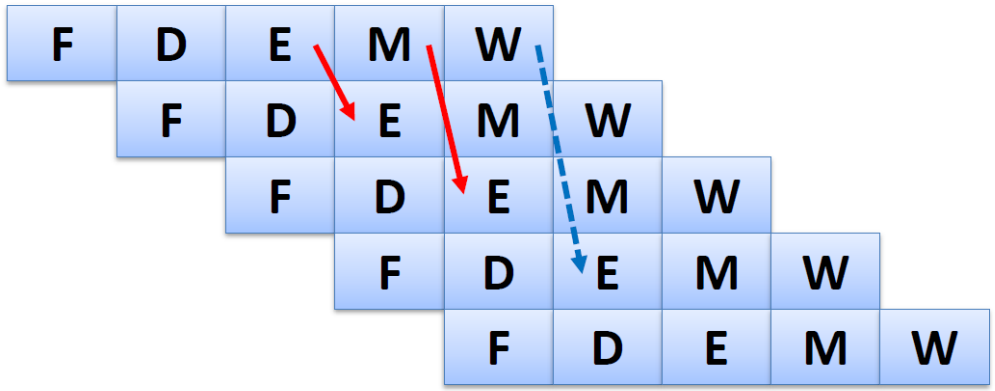
```
mrmovq 0(%rdx), %rax
addq    %rax, %rcx
```



Load interlock

Forwarding: More Example

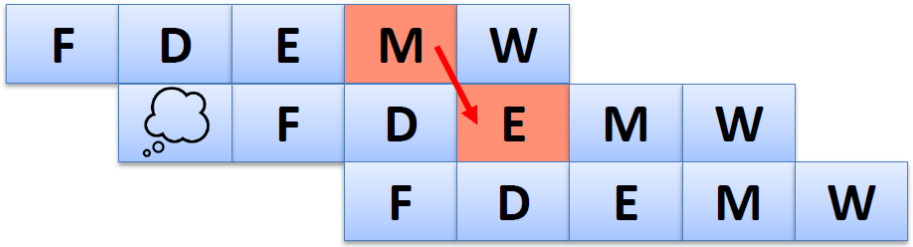
```
addq    %rdx, %rax
subq    %rax, %rcx
andq    %rax, %rbx
addq    %rax, %rsi
xorq    %rax, %rdi
```



Compiler Scheduling

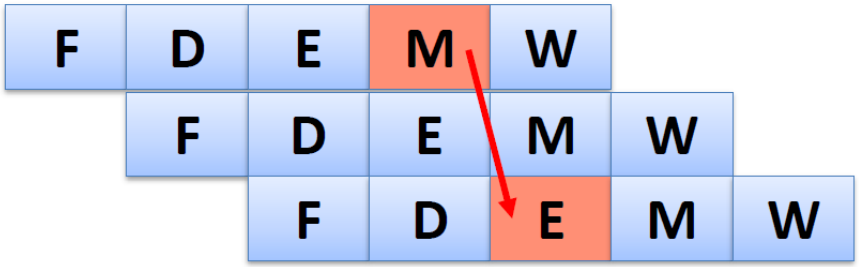
- Without compiler scheduling (1 stall)

```
mrmovq 0(%rdx), %rax
addq   %rax, %rcx
subq   %rdi, %rsi
```



- With compiler scheduling (no stall)

```
mrmovq 0(%rdx), %rax
subq   %rdi, %rsi
addq   %rax, %rcx
```



Control Hazard

- Caused by PC-changing instructions
  - (conditional/unconditional) Jump, Call / Return
- Stall-on-branch example: branch outcome determined in E stage
  - 2-cycle penalty for every branch
  - CPI (Cycles Per Instruction) = 1.3 for 15% branch frequency

Branch Instruction	F	D	E	M	W				
Branch successor		F	*stall*	F	D	E	M	W	
Branch successor + 1					F	D	E	M	W
Branch successor + 2						F	D	E	M
Branch successor + 3							F	D	E
Branch successor + 4								F	D
Branch successor + 5									F

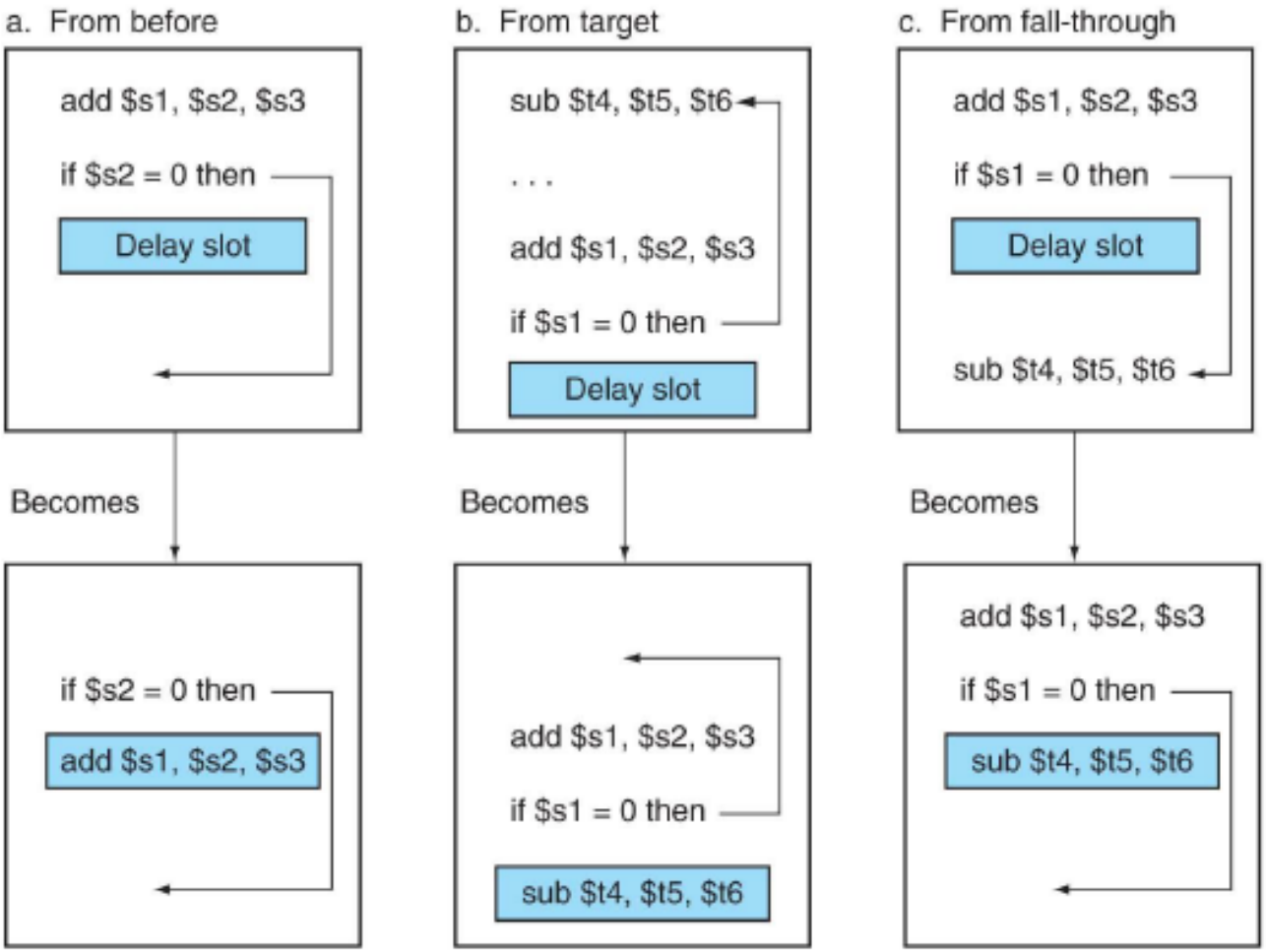
Solutions to Control Hazard

- Delayed branch
- Branch prediction

Delayed Branch

- Compiler approach
  - Branch delay slot filled by useful instructions
  - Branch delay slot becomes longer in modern processors
  - Microarchitecture-dependent

Delayed Branch



Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - (e.g., record recent history of each branch)
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
  - Modern processors favor dynamic branch prediction

Branch Prediction Example

- Predict the branch outcome early in the pipeline
- Example: (static) predict-taken

Taken branch instruction	F	D	E	M	W					
Branch target		F	D	E	M	W				
Branch target + 1			F	D	E	M	W			
Branch target + 2				F	D	E	M	W		
Branch target + 3					F	D	E	M	W	

Untaken branch instruction	F	D	E	M	W					
Branch target		F	D	*idle*	*idle*	*idle*				
Branch target + 1			F	*idle*	*idle*	*idle*	*idle*			
Untaken branch instruction + 1				F	D	E	M	W		
Untaken branch instruction + 2					F	D	E	M	W	

Questions?