Computer Architecture

Pipeline

Lynn Choi
Korea University

高麗大學校

1905

Computer System Laboratory

Motivation

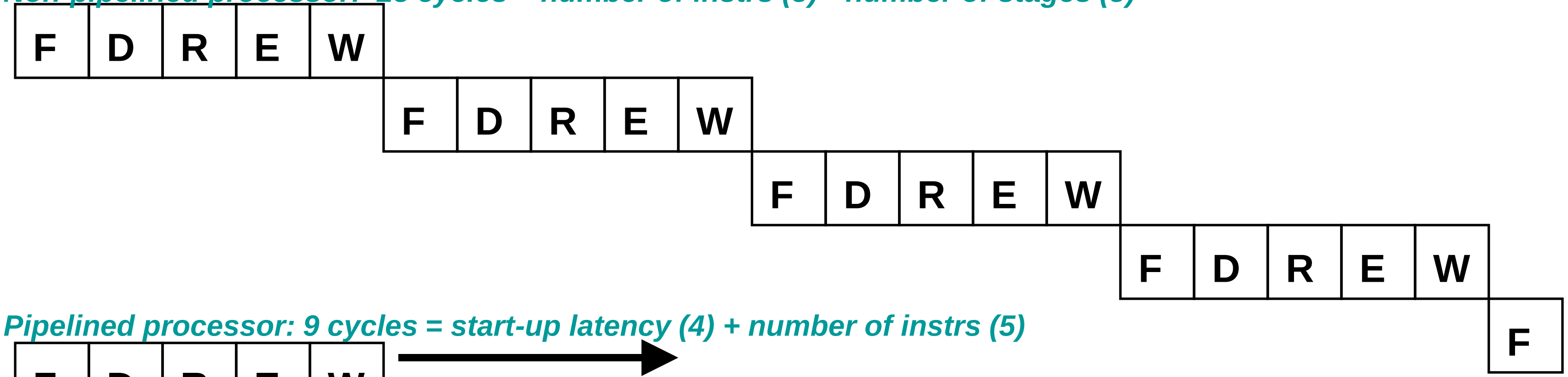
- **Non-pipelined design**
  - Single-cycle implementation
    - The cycle time depends on the slowest instruction
    - Every instruction takes the same amount of time
  - Multi-cycle implementation
    - Divide the execution of an instruction into multiple steps
    - Each instruction may take variable number of steps (clock cycles)
- **Pipelined design**
  - Divide the execution of an instruction into multiple steps (*stages*)
  - Overlap the execution of different instructions in different stages
    - Each cycle different instructions are executed in different stages
    - For example, 5-stage pipeline (Fetch-Decode-Read-Execute-Write),
      - 5 instructions are executed concurrently in 5 different pipeline stages
      - Complete the execution of one instruction every cycle (instead of every 5 cycle)
      - Can increase the throughput of the machine by 5 times

Pipeline Example

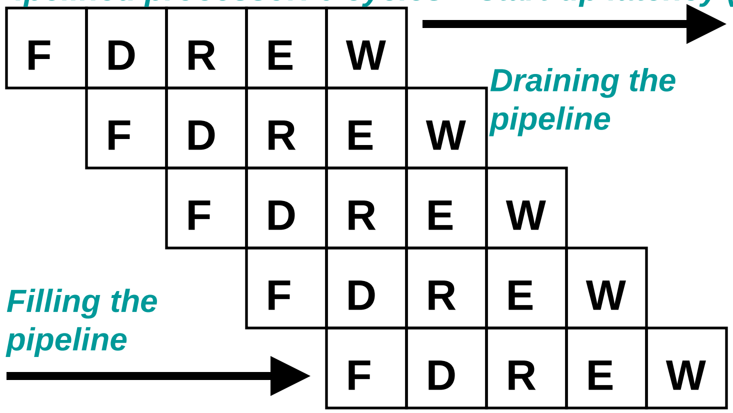
LD R1 <- A
ADD R5, R3, R4
LD R2 <- B
SUB R8, R6, R7
ST C <- R5

5 stage pipeline:
Fetch – Decode – Read – Execute - Write

*Non-pipelined processor: 25 cycles = number of instrs (5) * number of stages (5)*



Pipelined processor: 9 cycles = start-up latency (4) + number of instrs (5)



Data Dependence & Hazards

Data Dependence

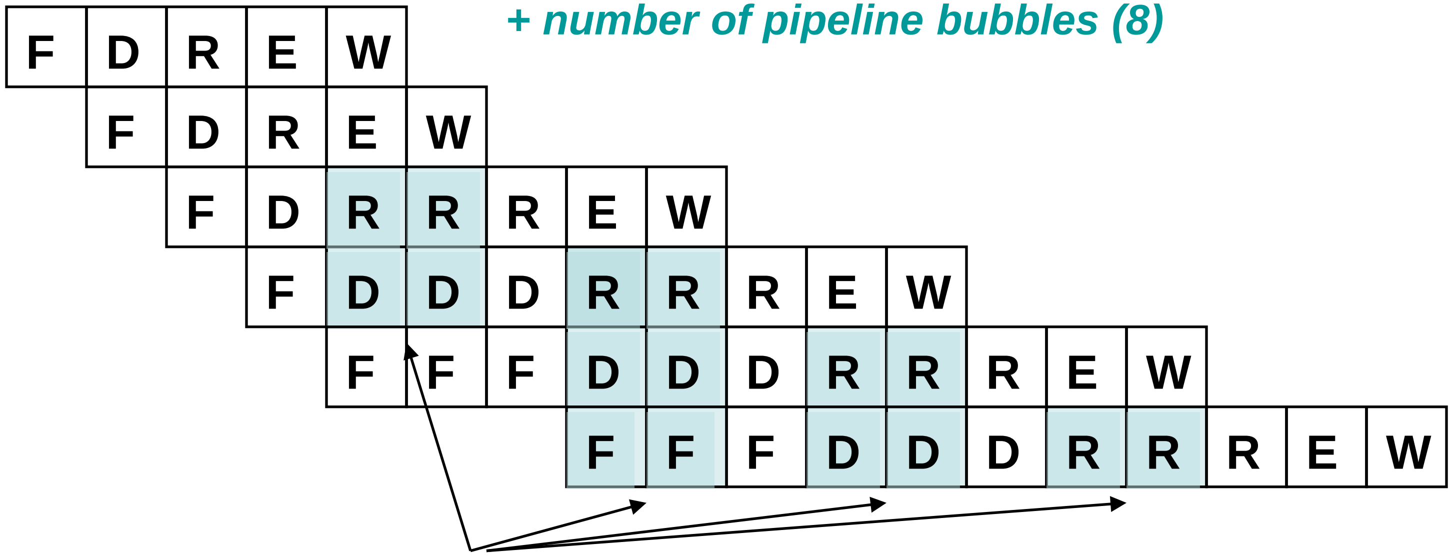
- Read-After-Write (RAW) dependence
  - True dependence
  - Must consume data after the producer produces the data
- Write-After-Write (WAW) dependence
  - Output dependence
  - The result of a later instruction can be overwritten by an earlier instruction
- Write-After-Read (WAR) dependence
  - Anti dependence
  - Must not overwrite the value before its consumer
- Notes
  - WAW & WAR are called ***false dependences***, which happen due to storage conflicts
  - All three types of dependences can happen for ***both registers and memory locations***
  - Characteristics of ***programs*** (not machines)
  - ***Must be preserved*** during execution to produce the correct output

Example 1

1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 ADD R4, R3, R2
5 SUB R3, R3, R4
6 ST A <- R3

RAW dependence:
1->3, 2-> 3, 2->4, 3 -> 4, 3 -> 5, 4-> 5, 5-> 6
WAW dependence:
3-> 5
WAR dependence:
4 -> 5, 1 -> 6 (memory location A)

Execution Time: 18 cycles = start-up latency (4) + number of instrs (6)
+ number of pipeline bubbles (8)



F	D	R	E	W													
	F	D	R	E	W												
		F	D	R	R	R	E	W									
			F	D	D	D	R	R	R	E	W						
				F	F	F	D	D	D	R	R	R	E	W			
							F	F	F	D	D	D	R	R	R	E	W

Pipeline bubbles due to RAW dependences (Data Hazards)

Example 2

1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 ADD R4, R5, R6
5 SUB R3, R1, R4
6 ST A <- R3

Changes:
1. Assume that MULT execution takes 6 cycles Instead of 1 cycle
2. Assume that we have separate ALUs for MULT and ADD/SUB

Dead Code

Execution Time: 18 cycles = start-up latency (4) + number of instrs (6) + number of pipeline bubbles (8)

F	D	R	E	W													
	F	D	R	E	W												
		F	D	R	R	R	E	E	E	E	E	E	W				
			F	D	D	D	R	E	W								
				F	F	F	D	R	R	R	R	R	E	W			
							F	D	D	D	D	D	R	R	R	E	W

due to RAW

due to WAW

Out-of-order (OOO) Completion

Multi-cycle execution like MULT can cause out-of-order completion

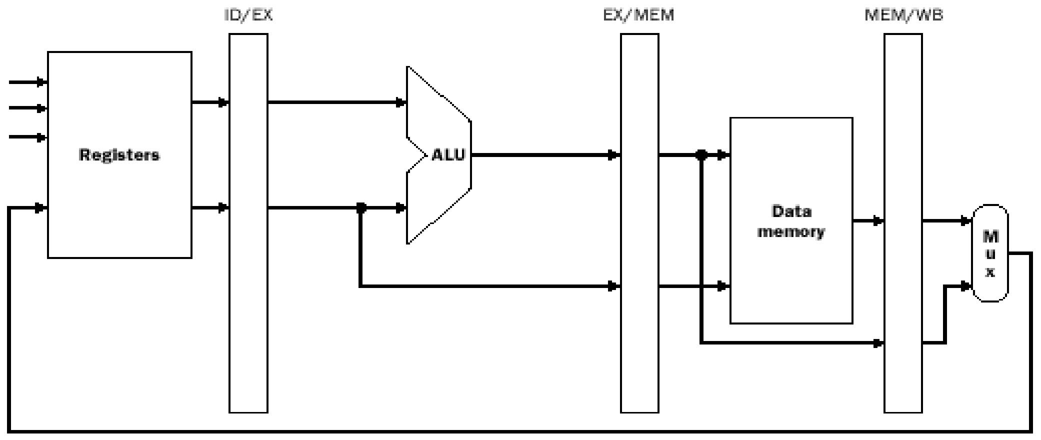
Pipeline stalls

- **Need reg-id comparators for**
  - RAW dependences
    - Reg-id comparators between the sources of a consumer instruction in REG stage and the destinations of producer instructions in EXE, WRB stages
  - WAW dependences
    - Reg-id comparators between the destination of an instruction in REG stage and the destinations of instructions in EXE stage (if the instruction in EXE stage takes more execution cycles than the instruction in REG)
  - WAR dependences
    - Can never cause the pipeline to stall since register read of an instruction always happens earlier than the write of a following instruction
- **If there is a match, recycle dependent instructions**
    - The current instruction in REG stage need to be recycled and all the instructions in FET and DEC stage need to be recycled as well
- **Also, called** pipeline interlock

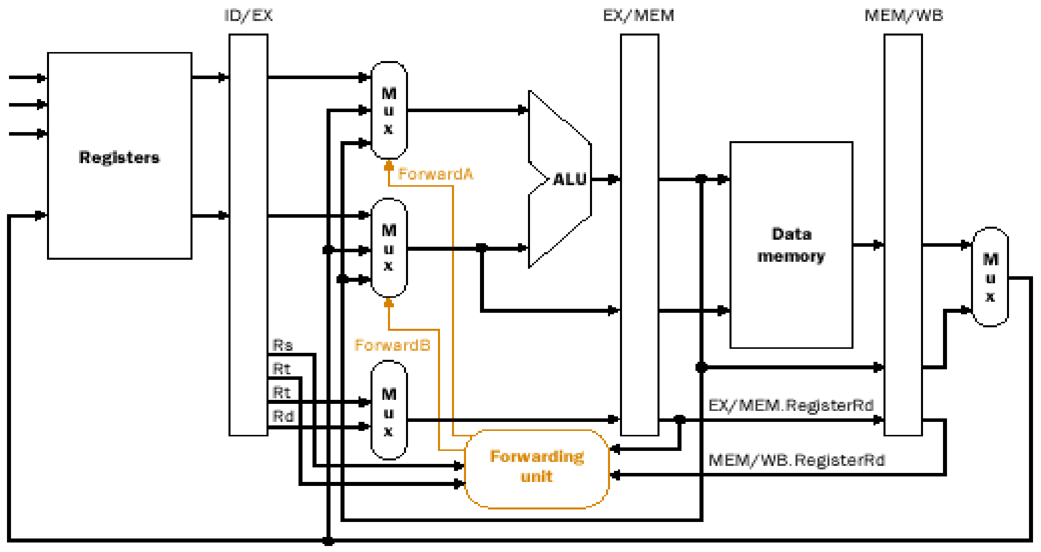
Data Bypass (Forwarding)

- **Motivation**
  - Minimize the pipeline stalls due to data dependence (RAW) hazards
- **Idea**
  - Let's propagate the result as soon as the result is available from ALU or from memory (in parallel with register write)
  - Requires
    - Data path from ALU output to the input of execution units (input of integer ALU, address or data input of memory pipeline, etc.)
    - Register Read stage can read data from register file or from the output of the previous execution stage
      - Require MUX in front of the input of execution stage

Datapath w/ Forwarding



a. No forwarding



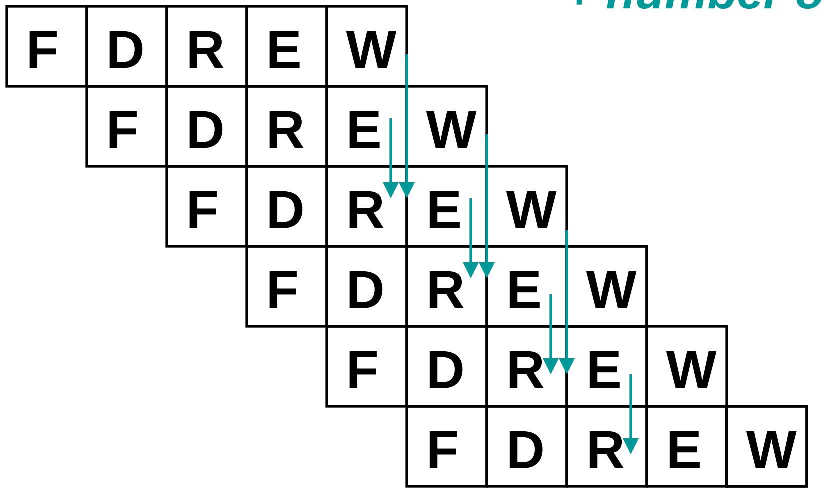
b. With forwarding

Elsevier Inc. All rights reserved

Example 1 with Bypass

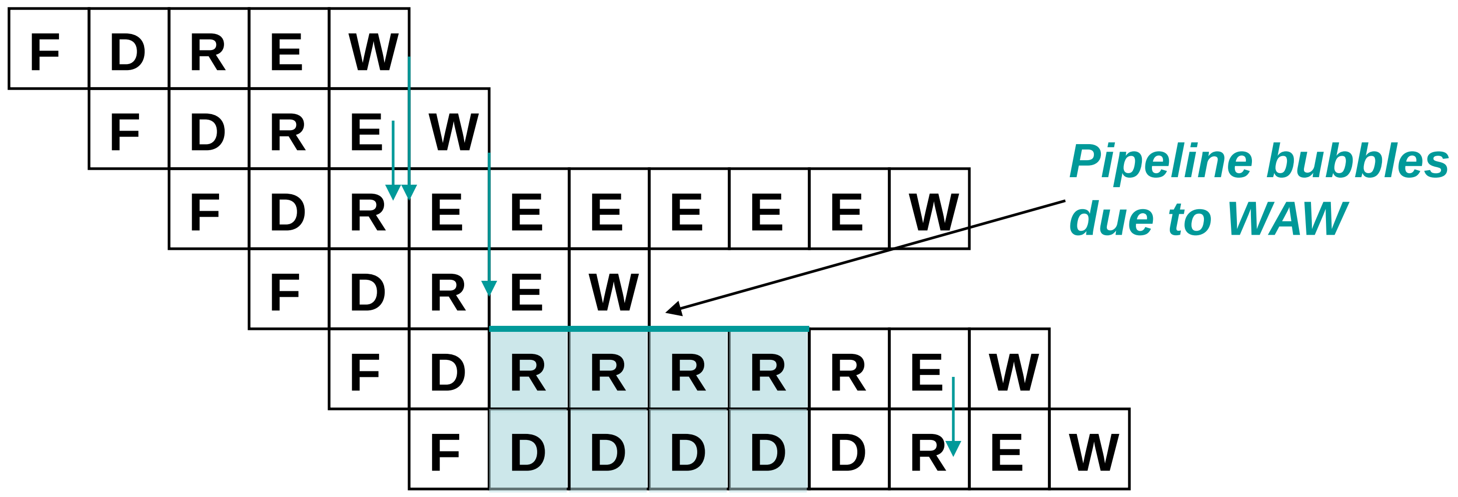
1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 ADD R4, R3, R2
5 SUB R3, R3, R4
6 ST A <- R3

Execution Time: 10 cycles = start-up latency (4) + number of instrs (6)
+ number of pipeline bubbles (0)



Example 2 with Bypass

1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 ADD R4, R5, R6
5 SUB R3, R1, R4
6 ST A <- R3



F	D	R	E	W									
	F	D	R	E	W								
		F	D	R	E	E	E	E	E	E	W		
			F	D	R	E	W						
				F	D	R	R	R	R	R	E	W	
					F	D	D	D	D	D	R	E	W

Pipeline Hazards

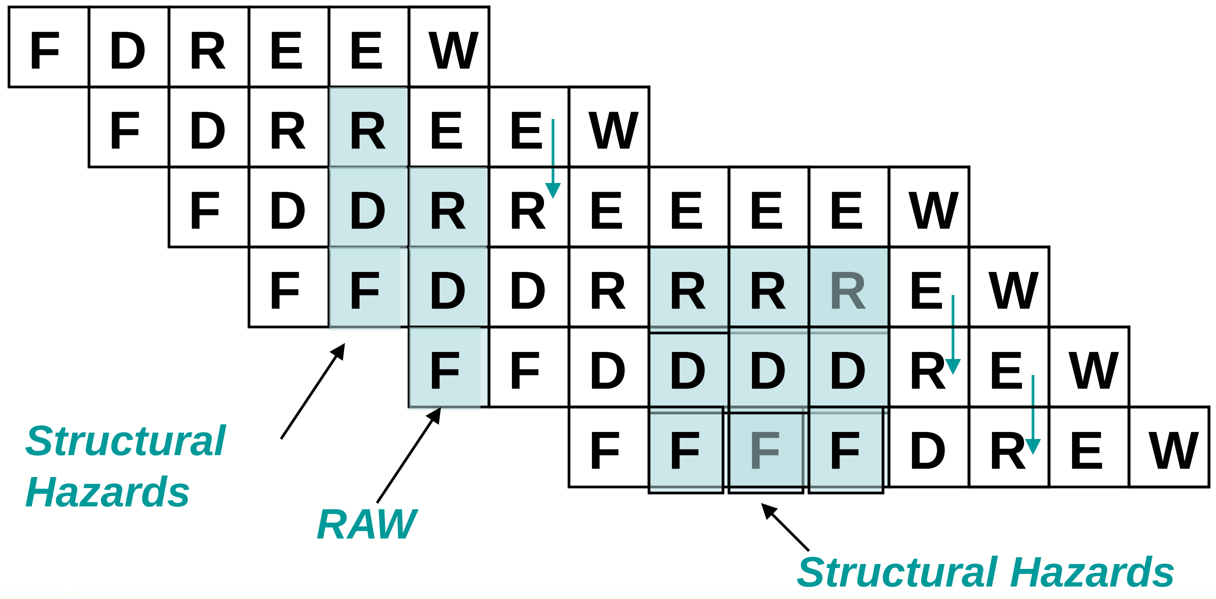
- **Data Hazards**
  - Caused by data (RAW, WAW, WAR) dependences
  - Require
    - Pipeline interlock (stall) mechanism to detect dependences and generate machine stall cycles
      - Reg-id comparators between instructions in REG stage and instructions in EXE/WRB stages
  - Stalls due to RAW hazards can be reduced by bypass network
    - Reg-id comparators + data bypass paths + mux
- **Structural Hazards**
  - Caused by resource constraints
  - Require pipeline stall mechanism to detect structural constraints
- **Control (Branch) Hazards**
  - Caused by branches
  - Instruction fetch of a next instruction has to wait until the target (including the branch condition) of the current branch instruction is resolved
  - Use
    - Predict the next target address (branch prediction) and if wrong, flush all the speculatively fetched instructions from the pipeline

Structural Hazard Example

1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 ADD R4, R5, R6
5 SUB R3, R1, R4
6 ST A <- R3

Assume that

1. *We have 1 memory unit and 1 integer ALU unit*
2. *LD takes 2 cycles and MULT takes 4 cycles*



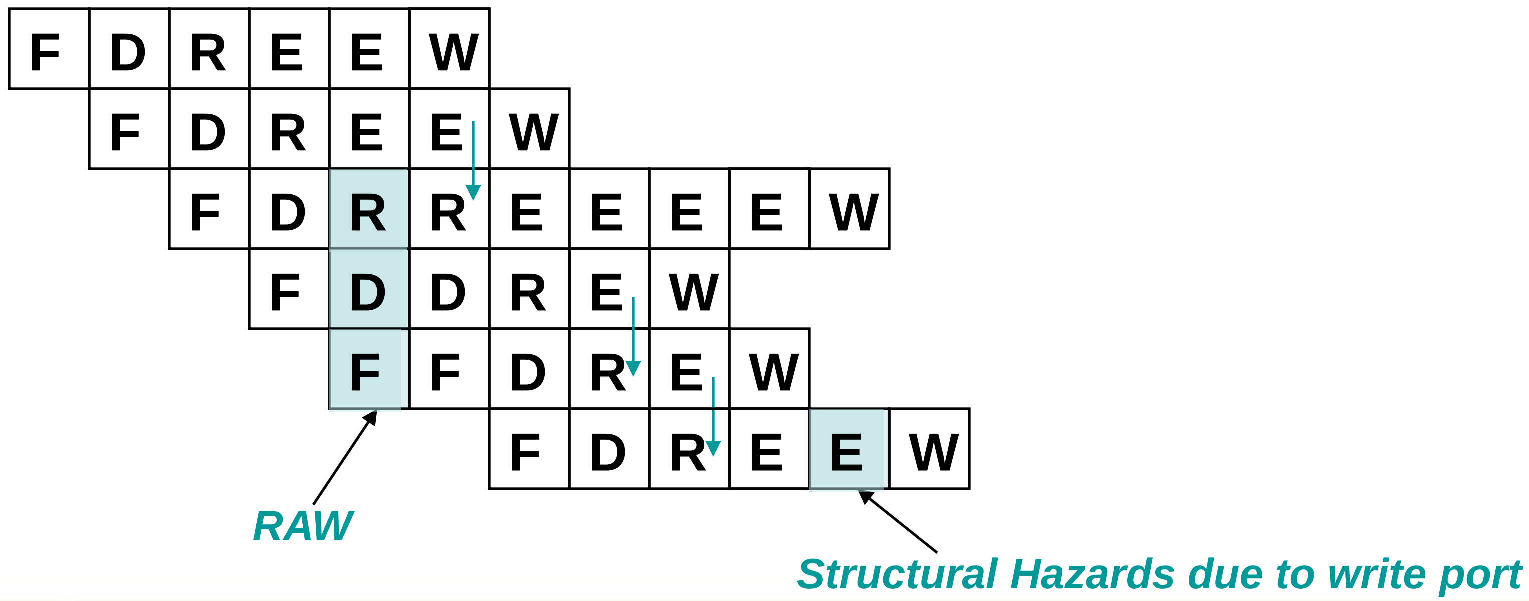
F	D	R	E	E	W									
	F	D	R	R	E	E	W							
		F	D	D	R	R	E	E	E	E	W			
			F	F	D	D	R	R	R	R	E	W		
					F	F	D	D	D	D	R	E	W	
							F	F	F	F	D	R	E	W

Structural Hazard Example

1 LD R1 <- A
2 LD R2 <- B
3 MULT R3, R1, R2
4 ADD R4, R5, R6
5 SUB R3, R1, R4
6 OR R10 <- R3, R1

Assume that

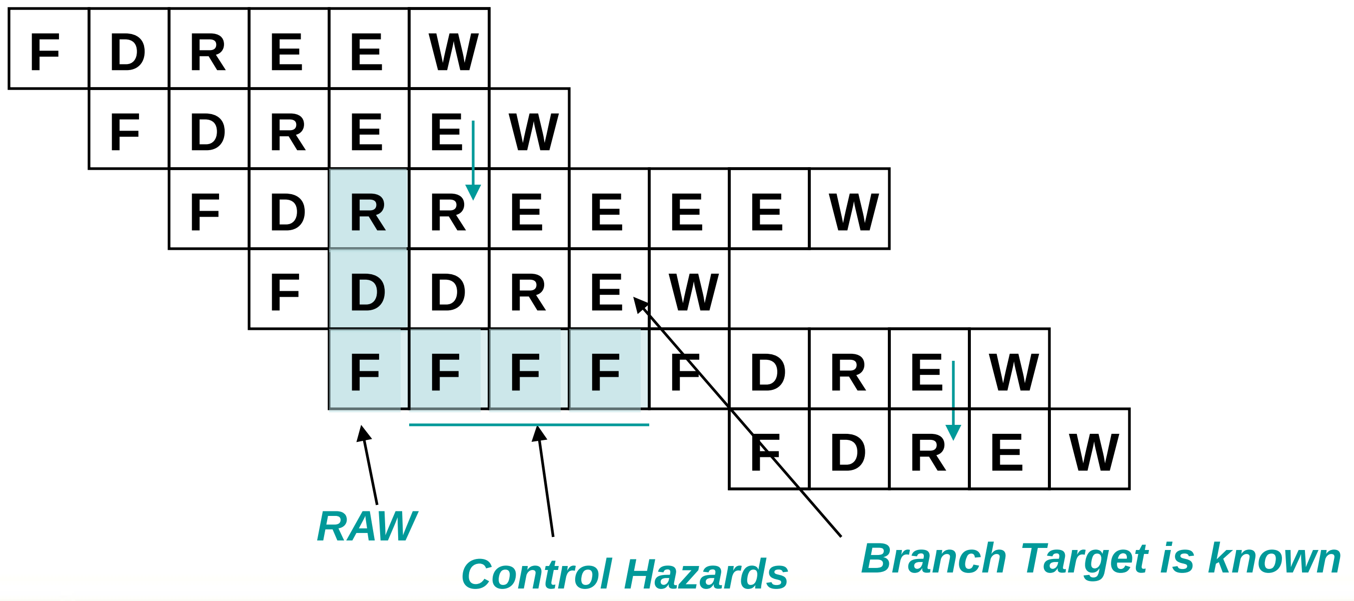
1. *We have 1 memory pipelined unit and and 1 integer add unit and 1 integer multiply unit*
2. *LD takes 2 cycles and MULT takes 4 cycles*



F	D	R	E	E	W						
	F	D	R	E	E	W					
		F	D	R	R	E	E	E	E	W	
			F	D	D	R	E	W			
				F	F	D	R	E	W		
						F	D	R	E	E	W

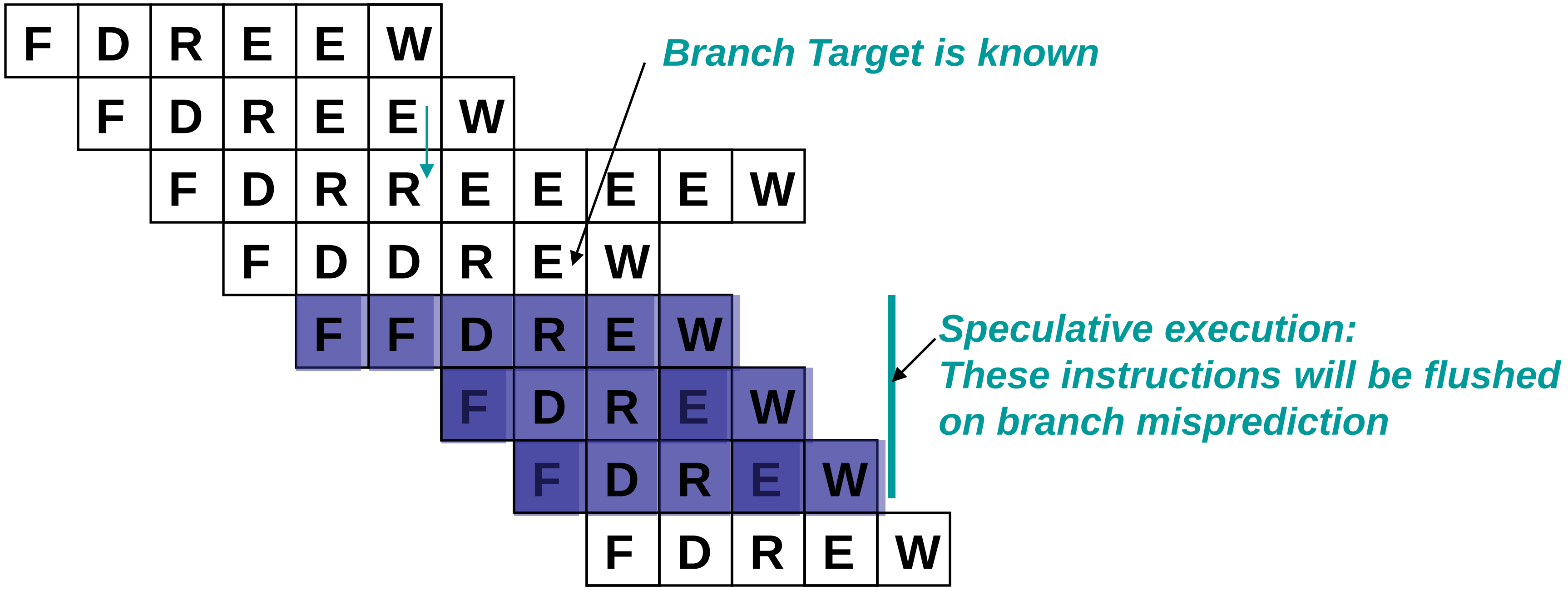
Control Hazard Example (Stall)

```
1              LD R1 <- A
2              LD R2 <- B
3              MULT R3, R1, R2
4              BEQ R1, R2, TARGET
5              SUB R3, R1, R4
6              ST A <- R3
7    TARGET:
```



Control Hazard Example (Flush)

```
1             LD R1 <- A
2             LD R2 <- B
3             MULT R3, R1, R2
4             BEQ R1, R2, TARGET
5             SUB R3, R1, R4
6             ST A <- R3
7    TARGET:  ADD R4, R1, R2
```



Branch Prediction

- **Branch Prediction**
  - Predict branch condition & branch target
  - Predictions are made even before the branch is fetched and decoded
  - Prefetch from the branch target before the branch is resolved (***Speculative Execution***)
  - A simple solution: PC <- PC + 4, prefetch the next sequential instruction
- **Branch condition (Path) prediction**
  - Only for conditional branches
  - Branch Predictor
    - Static prediction – at compile time
    - Dynamic prediction – at runtime using execution history
- **Branch target prediction**
  - Branch Target Buffer (BTB) or Target Address Cache (TAC)
    - Store target address for each branch and accessed with current PC
    - Do not store fall-through address since it is PC +4 for most branches
    - Can be combined with branch condition prediction, but separate branch prediction table is more accurate and common in recent processors
  - Return stack buffer (RSB)
    - Store return address (fall-through address) for procedure calls
    - Push return address on a call and pop the stack on a return

Branch Target Buffer

- ***For BTB to make a correct prediction, we need***
  - BTB hit: the branch instruction should be in the BTB
  - Prediction hit: the prediction should be correct
  - Target match: the target address must not be changed from last time
    - For *direct branches*, the target address is never changed
- ***Example:*** BTB hit ratio of 96%, 97% prediction hit, 1.2% of target change,

  The overall prediction accuracy = 0.96 * 0.97 *0.988 = 92%
- ***Implementation:*** Accessed with VA and need to be flushed on context switch

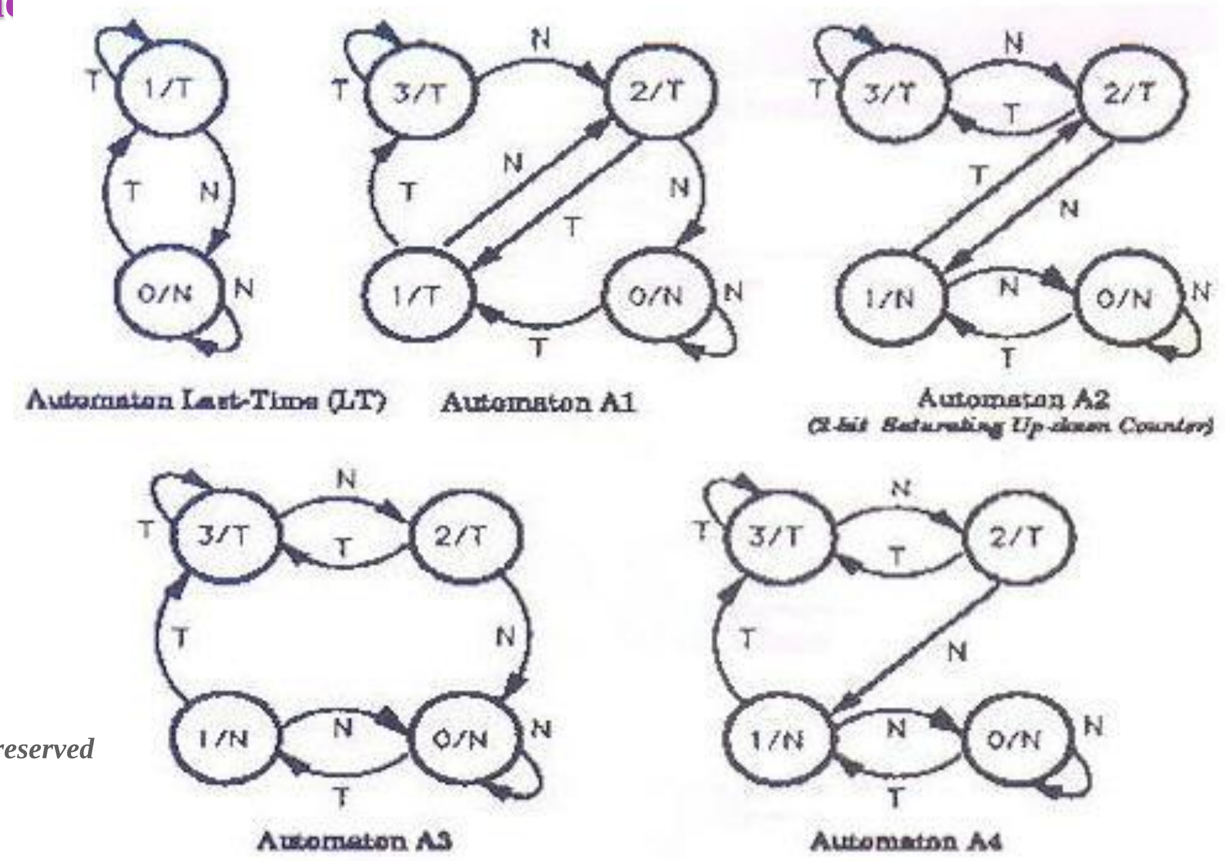
Branch Instruction Address	**Branch Target Address**	**Branch Condition Prediction (bimodal)**
⋮	⋮	⋮

Branch Prediction

- **Static prediction**
  - Assume all branches are taken : 60% of conditional branches are taken
  - Backward Taken and Forward Not-taken scheme: 69% hit rate
  - Profiling
    - Measure the tendencies of the branches and preset a prediction bit in the opcode
    - Sample data sets may have different branch tendencies than the actual data sets
    - 92.5% hit rate
  - Used as safety nets when the dynamic prediction structures need to be warmed up
- **Dynamic schemes- use runtime execution history**
  - LT (last-time) prediction - 1 bit, 89%
  - Bimodal predictors - 2 bit
    - 2-bit saturating up-down counters (Jim Smith), 93%
  - Two-level adaptive training (Yeh & Patt), 97%
    - First level, *branch history register* (BHR)
    - Second level, *pattern history table* (PHT)

Bimodal Predictor

S(I): State at time I
G(S(I)) -> T/F: Prediction decision function
E(S(I), T/N) -> S(I+1): State transition function
Performan



IEEE All rights reserved

Superscalar Processors

- **Exploit instruction level parallelism (ILP)**
  - Fetch, decode, and execute multiple instructions per cycle
  - Today's microprocessors issue 2 ~ 6 instructions per cycle
- **In-order pipeline versus Out-of-order pipeline**
  - In-order pipeline
    - When there is a data hazard stall, all the instructions following the stalled instruction must be stalled as well
  - Out-of-order pipeline (dynamic scheduling)
    - After the instruction fetch and decode phases, instructions are put into buffers called *instruction windows*. Instructions in the windows can be executed out-of-order when their operands are available
- **Examples**
  - Pentium IV: 3-way OOO
  - MIPS R10000: 4-way OOO
  - Ultrasparc II V9: 4-way in-order
  - Alpha 21264: 4-way OOO

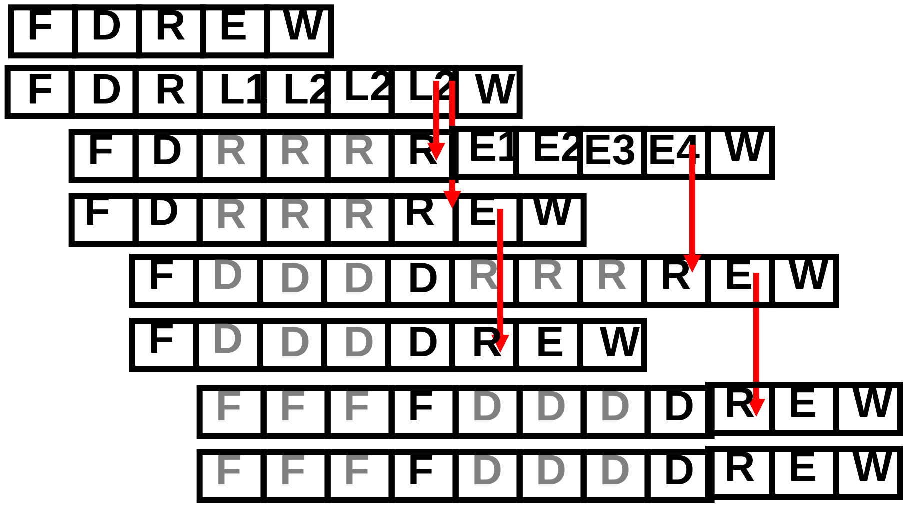
Superscalar Example

Assume 2-way superscalar processor with the following pipeline:

1 ADD/SUB ALU pipeline (1-Cycle INT-OP)
1 MULT/DIV ALU pipelines (4-Cycle INT-OP such as MULT)
2 MEM pipelines (1-Cycle (L1 hit) and 4-Cycle (L1 miss) MEM OP)

Show the pipeline diagram for the following codes assuming the bypass network:

LD R1 <- A (L1 hit); LD R2 <- B (L1 miss)
MULT R3, R1, R2; ADD R4, R1, R2
SUB R5, R3, R4; ADD R4, R4, 1
ST C <- R5; ST D <- R4



Exercises and Discussion

- WAR dependence violation cannot happen in in-order pipeline. Prove why?
- What is pipeline interlock? Explain the difference between pipeline interlock HW and data bypass HW.
- How do execution pipelines such as FPU pipeline affect the processor performance?

Homework 5

- Read Chapter 5
- Exercise
  - 4.2
  - 4.6
  - 4.11
  - 4.14
  - 4.16
  - 4.21
  - 4.24