

Lecture 4  
Pipeline (Appendix A in 3<sup>rd</sup> ed.)

Instructor: Jun Yang

Review: 5-stage Execution

• 5 canonical stage "RISC" load-store architecture

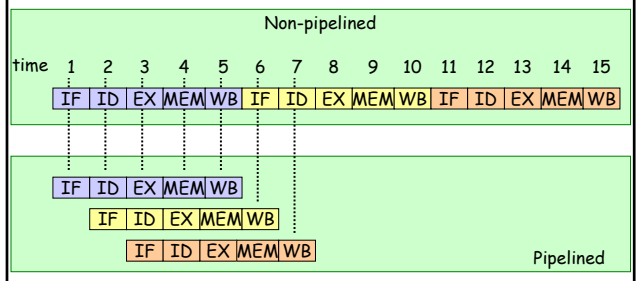
1. Instruction fetch (IF):
  - get instruction from memory/cache
2. Instruction decode, Register read (ID):
  - translate opcode into control signals and read regs
3. Execute (EX):
  - perform ALU operation, load/store address, branch outcomes
4. Memory (MEM):
  - access memory if load/store, everyone else idle
5. Writeback/retire (WB):
  - write results to register file

Performance of Serial Execution

- Performance:
  - Suppose each stage takes 1 cycle; branches take 2 cycles, branches are 12% of workload; stores take 4 cycles, 10% of workload; all other instructions take 5 cycles: CPI = 4.54.
  - Not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance level.
- Problem:
  - A new instruction can not start until the previous one finishes.

Solution

- Overlap execution of instructions
  - Start instruction on **every** cycle, e.g. the new instruction can be fetched while the previous one is decoded - **pipeline**. Each cycle performing a specific task; number of stages is called pipeline depth (5 here)

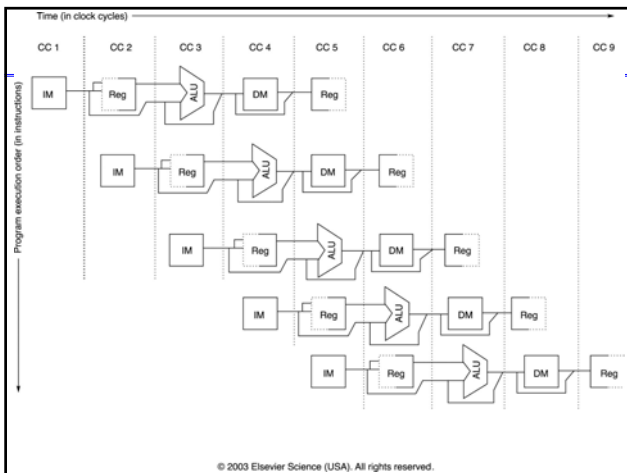
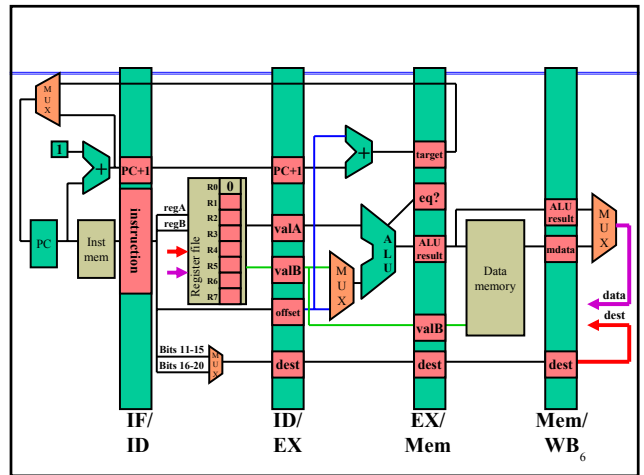


Pipelined

## Hardware Issues

- Simply put, we at least need:
  - IF - instruction memory
  - ID - instruction decoder, register file
  - EX - ALU
  - MEM - data memory
  - WB - register file
- With pipelining
  - hardware resources should be separated for each stage.
  - separate instruction and data memory access.
  - Memory delivers 5x the bandwidth.
  - 2 reads and one write of registers per cycle.
  - Latches between the 5 stages: holds the context of the current instruction in the stage.

5



© 2003 Elsevier Science (USA). All rights reserved.

## Easier Representation

Inst. No.	1	2	3	4	5	6	7	8	9
Inst. I	IF	ID	EX	ME	WB				
Inst. I+1		IF	ID	EX	ME	WB			
Inst. I+2			IF	ID	EX	ME	WB		
Inst. I+3				IF	ID	EX	ME	WB	
Inst. I+4					IF	ID	EX	ME	WB

Is that all?

8

## Pipeline Hazards

• Hazards are caused by conflicts between instructions. Will lead to incorrect behavior if not fixed.

- Three types:

- **Structural:** two instructions use same h/w in the same cycle - resource conflicts (e.g. one memory port, unpipelined divider etc).
- **Data:** two instructions use same data storage (register/memory) - dependent instructions.
- **Control:** one instruction affects which instruction is next - PC modifying instruction, changes control flow of program.

9

## Handling Hazards

- Force stalls or bubbles in the pipeline.
  - Stop some younger instructions in the stage when hazard happen
  - Make younger instr. Wait for older ones to complete
  - Implementation: de-assert write-enable signals to pipeline registers
- Flush pipeline
  - Blow instructions out of the pipeline
  - Refetch new instructions later - solving control hazards
  - Implementation: assert clear signals on pipeline registers

10

## Structural Hazards

• Example

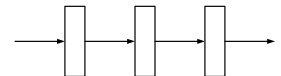
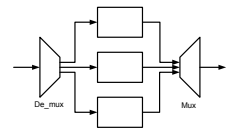
- Assume unified cache memory, i.e., instruction and data are stored in a single cache, and each cycle only one request can be processed (either instruction or data) - this cache has only one port

	1	2	3	4	5	6	7	8	9
Load	f	d	x	m	w				
inst1		f	d	x	m	w			
inst2			f	d	x	m	w		
inst3				f	d	x	m	w	

11

## Dealing with Structural Hazards

- Stall
  - + simple, low cost in h/w
  - Decrease IPC
- Replicate the resource
  - + good for performance
  - Increase h/w and area
  - Used for cheap resources
- Pipeline the resource
  - + good for performance
  - Complexity, e.g. RAM
  - Useful for multicycle resources



12

## Fixing Structural Hazards Using Stalls

- Which one to stall?
  - Always safe to stall younger instructions. Why?

	1	2	3	4	5	6	7	8	9	10
Load	f	d	x	m	w					
inst1		f	d	x	m	w				
inst2			f	d	x	m	w			
inst3				-	f	d	x	m	w	
inst4				-	-	f	d	x	m	w

13

## Data Hazards

- Two different instructions use the same storage location
  - It must appear as if they executed in sequential order

add R1, R2, R3	add R1, R2, R3	add R1, R2, R3
sub R2, R4, R1	sub R2, R4, R1	sub R2, R4, R1
or R1, R6, R3	or R1, R6, R3	or R1, R6, R3
read-after-write (RAW)	write-after-read (WAR)	write-after-write (WAW)
True dependence (real)	anti dependence (artificial)	output dependence (artificial)

What about read-after-read dependence ?

14

## Two Stall Timings

- Depending on how ID and WB share the register file (RF)
  - Each gets RF for half a cycle

1<sup>st</sup> half ID reads, 2<sup>nd</sup> half WB writes - 3 cycle bubble

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	f	d	x	m	w				
sub R2, R4, R1		f	-	-	-	d	x	m	w

1<sup>st</sup> half WB writes, 2<sup>nd</sup> half ID reads - 2 cycle bubble

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	f	d	x	m	w				
sub R2, R4, R1		f	-	-	d	x	m	w	

15

## Reducing RAW Hazards: Bypassing

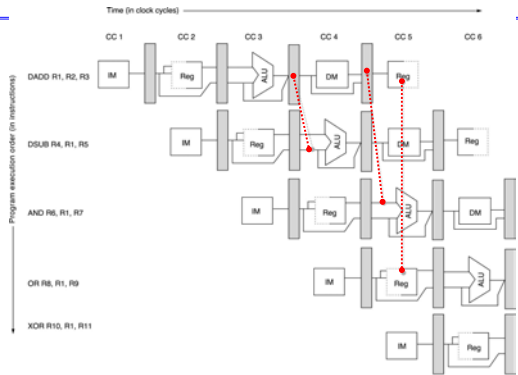
- Data available at the end of EX stage, why wait until WB stage?
  - Bypass (forward) data directly to input of EX
    - Reduces/avoids stalls in a big way
      - Large fraction of input operands are bypassed
    - Complex
  - Important: does not relieve you from having to perform WB

	1	2	3	4	5	6	7	8	9
add R1, R2, R3	f	d	x	m	w				
sub R2, R4, R1		f	d	x	m	w			

- Can bypass from MEM also

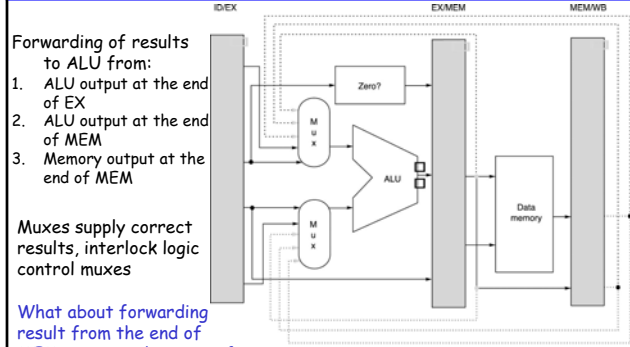
16

## Minimizing Data Hazard Stalls by Forwarding



© 2003 Elsevier Science B.V. All rights reserved.

## Forwarding Implementation Example



- Forwarding of results to ALU from:
1. ALU output at the end of EX
  2. ALU output at the end of MEM
  3. Memory output at the end of MEM

Muxes supply correct results, interlock logic control muxes

What about forwarding result from the end of MEM stage to the input of memory?

© 2003 Elsevier Science B.V. All rights reserved.

## But ...

- Even with bypassing, not all RAWs stalls can be avoided
  - Load to an ALU immediately after
  - Can be eliminated with compiler scheduling

	1	2	3	4	5	6	7	8	9
lw R1, 16(R3)	f	d	x	m	w				
sub R2, R4, R1		f	-	d	x	m	w		

You can also stall before EX stage, but it is better to separate stall logic from bypassing logic

## Compiler Scheduling

- Compiler moves instructions around to reduce stalls
  - E.g. code sequence:  $a = b + c$ ,  $d = e - f$

before scheduling

```
lw Rb, b
lw Rc, c
add Ra, Rb, Rc //stall
sw Ra, a
stall
lw Re, e
lw Rf, f
sub Rd, Re, Rf //stall
Rf //no stall
sw Rd, d
```

after scheduling

```
lw Rb, b
lw Rc, c
lw Re, e
add Ra, Rb, Rc //no
stall
lw Rf, f
sw Ra, a
sub Rd, Re,
Rf
```

## WAR: Why do they exist?

- Recall WAR

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```
- Problem: swap means introducing false RAW hazards
- Artificial: can be removed if sub used a different destination register
- Can't happen in in-order pipeline since reads happen in ID but writes happen in WB
- Can happen in out-of-order reads, e.g. out-of-order execution

21

## WAW

```
add R1, R2, R3
sub R2, R4, R1
or R1, R6, R3
```

- Problem: scheduling would leave wrong value in R1 for the sub
- Artificial: using different destination register would solve
- Can't happen in in-order pipeline in which every instruction takes same cycles since writes are in-order
- Can happen in the presence of multi-cycle operations, i.e., out-of-order writes

22