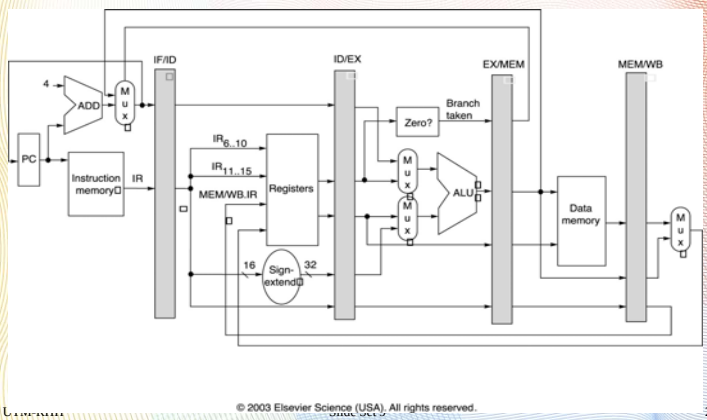
Microprocessor Design & Organisation HCA2102

Pipeling and Hazards

Pipelined DLX Datapath

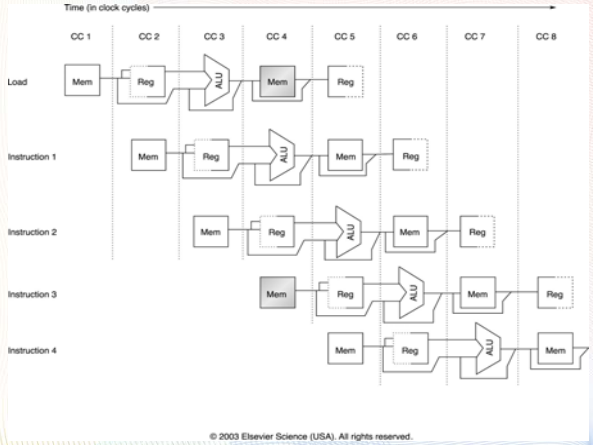


Hazards

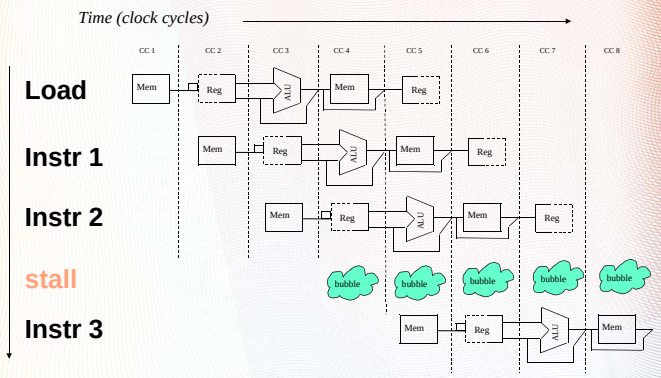
Hazards are situations that hamper execution flow

- Structural Hazards:
  - Resource Conflict, hardware cannot support all possible combinations of instructions simultaneously.
- Data Hazards:
  - Source operands are not available: instruction depends on results of previous instructions still in the pipeline
- Control Hazards:
  - Changes in program counter

Structural Hazards



One Memory Port/Structural Hazards



Structural Hazard: Single Memory

Clock cycle number

Instruction	1	2	3	4	5	6	7	8	9	10
Load	IF	ID	EX	MEM	WB					
Instr. 1		IF	ID	EX	MEM	WB				
Instr. 2			IF	ID	EX	MEM	WB			
Instr. 3				Stall	IF	ID	EX	MEM	WB	
Instr. 4						IF	ID	EX	MEM	WB
Instr. 5							IF	ID	EX	MEM
Instr. 6								IF	ID	EX

Speed Up Equation for Pipelining

$$\text{Speedup from pipelining} = \frac{\text{Avg. Instr. Time Unpipelined}}{\text{Avg. Instr. Time Pipelined}}$$

$$= \frac{\text{CPI}_{\text{unpipelined}} \times \text{Clock Cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock Cycle}_{\text{pipelined}}}$$

$$= \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

Ideal CPI = $\text{CPI}_{\text{unpipelined}}$ /Pipeline depth

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

Speed Up Equation for Pipelining

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}$$

$$\text{Speedup} = \frac{\text{Ideal CPI x Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a clock rate that is 1.2 times faster
- Ideal CPI=1 for both
- Loads and stores are 40% of instructions executed

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1+0) \times (\text{clock}_{\text{unpipe}}/\text{clock}_{\text{pipe}})$$
$$= \text{Pipeline Depth}$$
$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1+0.4 \times 1) \times (\text{clock}_{\text{unpipe}}/(\text{clock}_{\text{unpipe}}/1.2))$$
$$= (\text{Pipeline Depth}/1.4) \times 1.2$$
$$= 0.86 \times \text{Pipeline Depth}$$
$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth}/(0.86 \times \text{Pipeline Depth}) = 1.17$$

Machine A is 1.17 times faster

Data Hazards

```
sub   R2, R1, R3      ; R2 written by sub
and   R12, R2, R5     ; first operand (R2) depends on sub
or    R13, R6, R2     ; second operand (R2) depends on sub
add   R14, R2, R2     ; both operands depend on sub
sw    100 (R2), R15   ; index (R2) depends on sub
```

Notice that the value written into R2 by the subtract instruction is needed in **all** of the following instructions

Classification of Data Hazards

Consider instructions *i* and *j*, where *i* occurs before *j*.

- RAW (read after write) — *j* tries to read a source before *i* writes it, so *j* gets the old value
- WAW (write after write) — *j* tries to write an operand before it is written by *i* (only possible in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled)
- WAR (write after read) — *j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value (only possible when some instructions can write results early in the pipeline and other instructions can read sources late in the pipeline)

Software Solution

Compiler recognizes data hazard and adds nops to eliminate it

```
sub R2, R1, R3     ; register R2 written by sub
nop                ; no operation
nop                ; no operation
nop                ; no operation
and R12, R2, R5    ; now, result from sub available
or  R13, R6, R2
add R14, R2, R2
sw  100 (R2), R15
```

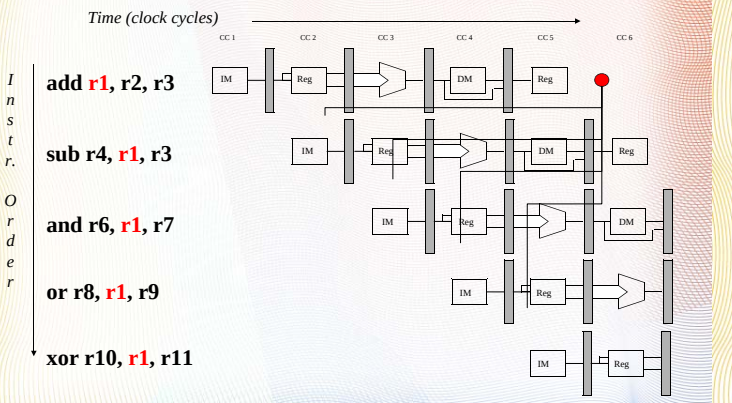
Data Hazard Control: Stalls

- Hazard occurs when instruction reads (in ID stage) register that will be written by an earlier instruction (in WB stage)
- Idea: Detect hazard and stall instructions in pipeline until hazard is resolved
- Detect hazard by comparing read fields in IF/ID pipeline register with write fields in later pipeline registers (ID/EX, EX/MEM, MEM/WB)
- To add bubble in pipeline
  - Preserve PC register and IF/ID pipeline register
  - Change EX, MEM, and WB control fields of ID/EX pipeline register to do nothing

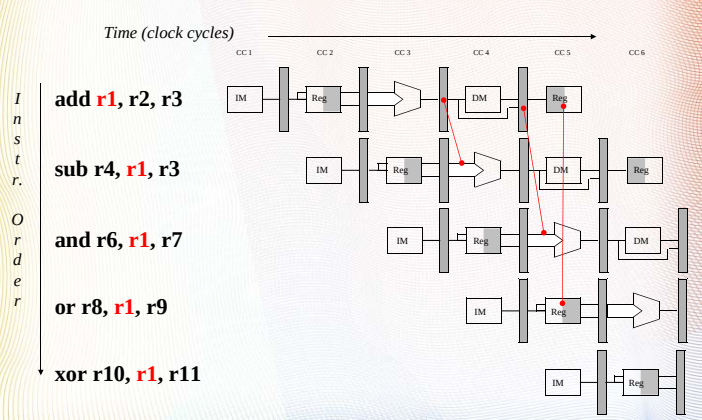
Data Hazard Reduction: Forwarding

- Needed result is available before it is written into register file in WB stage
- Idea: Use temporary results instead of waiting for registers to be written
- Cannot solve problem of write (load) followed by read
- Almost all pipelined machines today use some form of forwarding

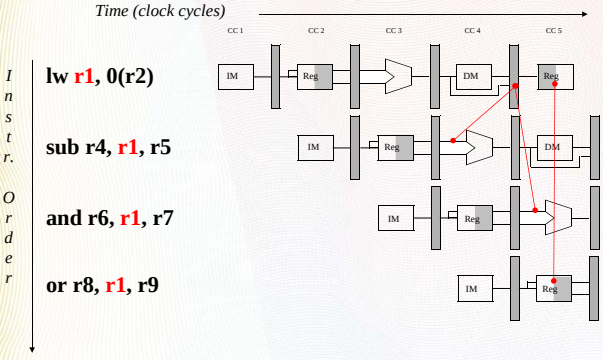
Data Hazard on R1



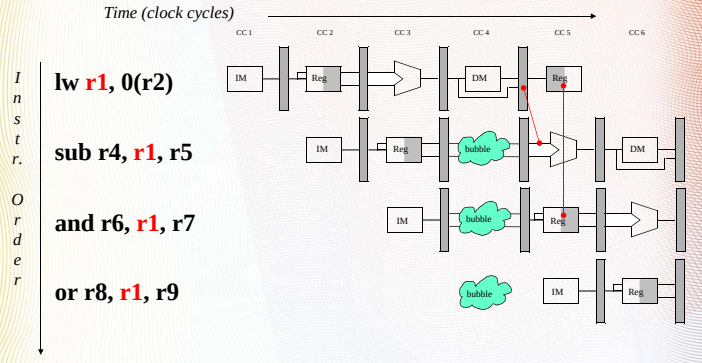
Forwarding to Avoid Data Hazard



Data Hazard Even with Forwarding



Data Hazard Even with Forwarding

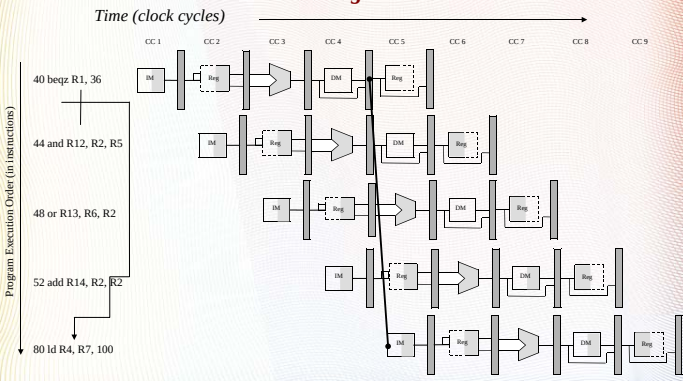


LW R1, 0 (R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0 (R2)	IF	ID	EX	MEM	WB
SUB R4, R1, R5					
AND R6, R1, R7					
OR R8, R1, R9					

Control Hazard on Branches

Three Stage Stall



Branch instruction	IF	ID	EX	MEM	WB				
Branch successor		IF	*stall*	*stall*	IF	ID	EX	MEM	WB
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

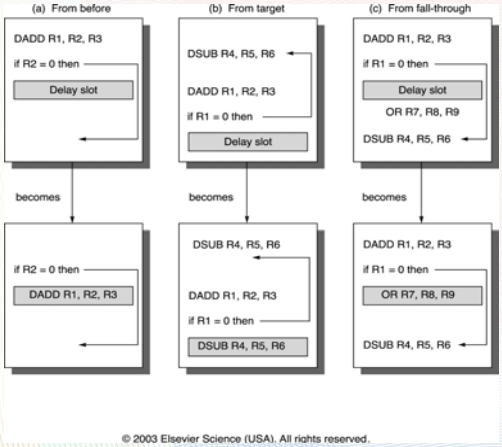
Branch Stall Impact

- If CPI = 1, 30% branches, 3-cycle stall ⇒ new CPI = 1.9!

Two simple solutions:

- Predict not taken
  - Continue with decoding code that is already in Instruction Cache
  - Usually < 50% correct, however, no stalls when correct

Delayed Branch



Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Canceling branches allow more slots to be filled
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

$$\text{Pipeline Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stalls}}$$

$$= \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Evaluating Branch Alternatives