Advanced Computer Architecture
(0630561)

Lecture 6

Pipeline Hazards

Prof. Kasim M. Al-Aubidy
Computer Eng. Dept.

Pipeline Conflicts : 3 major difficulties

- 1) Resource conflicts
  memory access by two segments at the same time
- 2) Data dependency
  when an instruction depend on the result of a previous instruction, but this result is not yet available
- 3) Branch difficulties
  branch and other instruction (interrupt, ret, ..) that change the value of PC

Data Dependency

- Hardware » Hardware Interlock »Operand Forwarding
- Software » Delayed Load

Handling of Branch Instructions

- Prefetch target instruction
- Branch Target Buffer : **BTB**
- Loop Buffer
- Branch Prediction

Delayed Branch

Clock cycles :	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

(a) Using no-operation instructions

Clock cycles :	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

(b) Rearranging the instructions

Example of delayed branch

RISC Pipeline

RISC CPU

- Instruction Pipeline
- Single-cycle instruction execution
- Compiler support

Example : Three-segment Instruction Pipeline

- 3 Suboperations Instruction Cycle
  - » 1) I : Instruction fetch
  - » 2) A : Instruction decoded and ALU operation
  - » 3) E : Transfer the output of ALU to a register, memory, or PC(Program control Inst.=JMP/CALL)
- Delayed Load :
- Delayed Branch :

Conflict

Clock cycles :	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

(a) Pipeline timing with data conflict

Clock cycles :	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

(b) Pipeline timing with delayed load

Pipeline Hazards

- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. The instruction is said to be *stalled*. When an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled. Instructions earlier than the stalled one can continue. No new instructions are fetched during the stall.

- Types of hazards:
  - *structural*: two instructions use same h/w in same cycle
  - *data*: two instructions use same data (register/memory)
  - *control*: one instruction affects which instruction is next

Structural Hazards

- Structural hazards occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.

Instruction ADD R4,X fetches in the FO stage operand X from memory. The memory doesn't accept another access during that cycle.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R4,X	FI	DI	CO	FO	EI	WO						
Instr. i+1		FI	DI	CO	FO	EI	WO					
Instr. i+2			FI	DI	CO	FO	EI	WO				
Instr. i+3				stall	FI	DI	CO	FO	EI	WO		
Instr. i+4						FI	DI	CO	FO	EI	WO	

Penalty: 1 cycle

- Certain resources are duplicated in order to avoid structural hazards. Functional units (ALU, FP unit) can be pipelined themselves in order to support several instructions at a time. A classical way to avoid hazards at memory access is by providing separate data and instruction caches.

Data Hazards

- We have two instructions, I1 and I2. In a pipeline the execution of I2 can start before I1 has terminated. If in a certain stage of the pipeline, I2 needs the result produced by I1, but this result has not yet been generated, we have a data hazard.

I1:	MUL	R2,R3	R2 ← R2 * R3
I2:	ADD	R1,R2	R1 ← R1 + R2

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	stall	FO	EI	WO			
Instr. i+2			FI	DI			CO	FO	EI	WO		

Before executing its FO stage, the ADD instruction is stalled until the MUL instruction has written the result into R2.

Penalty: 2 cycles

Data Hazards

two different instructions use the same storage location

- we must preserve the illusion of sequential execution

read-after-write (RAW)

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

read-after-write (RAW) = true dependence (dataflow)

- problem: `sub` reads `R1` before `add` has written it
  - Pipelining enables this overlapping to occur
  - But this violates sequential execution semantics!
  - Recall: user just sees ISA and expects sequential execution

detect RAW and stall instruction at ID before it reads registers

- mechanics? disable PC, F/D write
- RAW detection? *compare register names*

WAR: Write After Read

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

- problem: **add** could use wrong value for **R2**
- can't happen in vanilla pipeline (reads in ID, writes in WB)
  - can happen if: early writes (e.g., auto-increment) + late reads (??)
  - can happen if: out-of-order reads (e.g., out-of-order execution)
- artificial: using different output register for **sub** would solve
  - The dependence is on the name **R2**, but not on actual dataflow

WAW: Write After Write

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

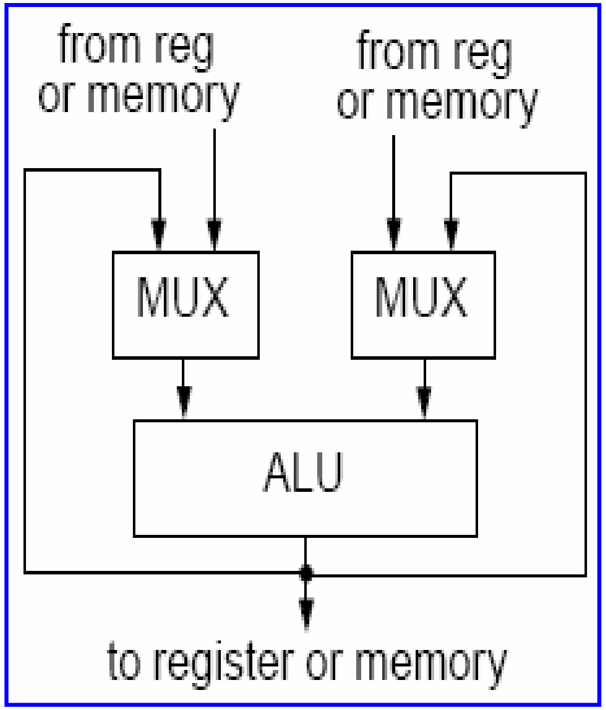
- problem: reordering could leave wrong value in **R1**
  - later instruction that reads **R1** would get wrong value
- can't happen in vanilla pipeline (register writes are in order)
  - another reason for making ALU ops go through MEM stage
  - can happen: multi-cycle operations (e.g., FP, cache misses)
- artificial: using different output register for **or** would solve
  - Also a dependence on a name: **R1**

RAR: Read After Read

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

- no problem: R3 is correct even with reordering

- Some of the penalty produced by data hazards can be avoided using a technique called *forwarding* (bypassing).
- The ALU result is always fed back to the ALU input. If the hardware detects that the value needed for the current operation is the one produced by the previous operation (but which has not yet been written back) it selects the forwarded result as the ALU input, instead of the value read from register or memory.



Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
MUL R2,R3	FI	DI	CO	FO	EI	WO						
ADD R1,R2		FI	DI	CO	stall	FO	EI	WO				

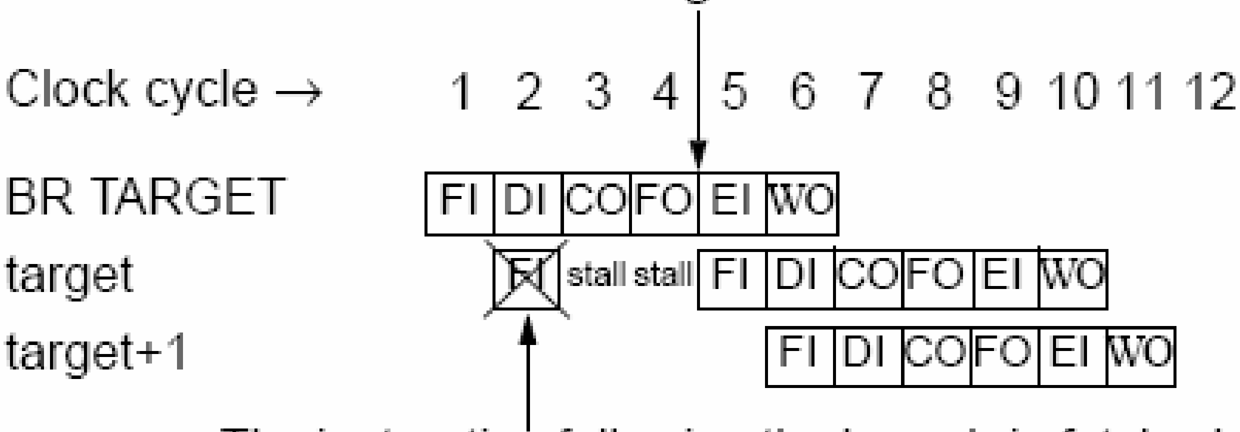
After the EI stage of the MUL instruction the result is available by forwarding. The penalty is reduced to one cycle.

Control Hazards are produced by branch instructions.

Unconditional branch

```
              ---------------
              BR   TARGET
              ---------------
TARGET        ---------------
```

After the FO stage of the branch instruction the address of the target is known and it can be fetched



The instruction following the branch is fetched; before the DI is finished it is not known that a branch is executed. Later the fetched instruction is discarded

Penalty: 3 cycles

Conditional branch

```
                ADD       R1,R2          R1 ← R1 + R2
                BEZ       TARGET         branch if zero
                instruction i+1
                - - - - - - - - - - - - - -
      TARGET    - - - - - - - - - - - - - -
```

<u>Branch is taken</u>

At this moment, both the condition (set by ADD) and the target address are known.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TARGET		FI	DI	CO	FO	EI	WO					
target			~~FI~~	stall	stall	FI	DI	CO	FO	EI	WO	

Penalty: 3 cycles

<u>Branch **not** taken</u>

At this moment the condition is known and instr+1 can go on.

Clock cycle →	1	2	3	4	5	6	7	8	9	10	11	12
ADD R1,R2	FI	DI	CO	FO	EI	WO						
BEZ TARGET		FI	DI	CO	FO	EI	WO					
instr i+1			FI	stall	stall	DI	CO	FO	EI	WO		

Penalty: 2 cycles