Advanced Computer Architecture

Pipeline Hazard – Control Hazard

Prof. Roger Luis Uy
De La Salle University
College of Computer Studies

Pipeline Hazards

- Hazards – situation that prevent the next instruction in the instruction stream from executing during its designated clock cycle

Pipeline Hazards

- To eliminate a hazard, it often requires that some instructions in the pipeline be allowed to proceed while others are delayed
- All instructions issued earlier than the stalled instruction must continue, and all instructions issued later than the stalled instruction must be stalled
- No new instruction shall be fetched during the stall

Pipeline Hazards

Three type of pipeline hazards:

1. Structural Hazards
   - Arises from resource contention
2. Data Hazards
   - Arise when an instruction depend on the results of a previous instruction
3. Control Hazards
   - Arise from the pipelining of branches and other instructions that change the PC

Control Hazard

- In the worst case, the branch instruction could cause stall up to 3 clock cycles long (i.e., 2 stall cycles + 1 repeated IF cycle)
- Solution #1: Simple compile time static branch solution (i.e., fixed for each branch during the entire execution)

4 simple static branch solutions

- Pipeline Freeze
- Pipeline Flush
- Predict-not-taken
- Predict-taken

Pipeline Freeze

- Holding (Stalling) any fetched instructions after the branch until the branch destination is known

Illustration of Pipeline Freeze

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R1, R2, R3		IF	*	*	ID	EX	MEM	WB	
DADDU R4, R5, R6									
DADDU R7, R8, R9									
L1:DSUBU R1, R2, R3									

Branch not taken (above) ; Branch Taken (below)

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R1, R2, R3		IF	*	*					
DADDU R4, R5, R6									
DADDU R7, R8, R9									
L1:DSUBU R1, R2, R3					IF	ID	EX	MEM	B

Pipeline Flush

- Deleting (flushing) any instructions after the branch until the branch destination is known
- Since the instruction after the branch is flush and if branch is not taken, the instruction after the branch will need to be re-fetch

Illustration of Pipeline Flush

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R1, R2, R3		IF			IF	ID	EX	MEM	WB
DADDU R4, R5, R6			IF						
DADDU R7, R8, R9				IF					
L1:DSUBU R1, R2, R3									

Branch not taken (above) ; Branch Taken (below)

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R1, R2, R3		IF							
DADDU R4, R5, R6			IF						
DADDU R7, R8, R9				IF					
L1:DSUBU R1, R2, R3					IF	ID	EX	MEM	B

Predict-not-taken

- Continuous fetching of instructions as if branch instruction is a normal instruction
- If the branch is taken: the fetched instruction has to be changed into a NOP instruction and restart the fetch at the branch address

Illustration of Predict-not-taken

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R1, R2, R3		IF	ID	EX	MEM	WB			
DADDU R4, R5, R6			IF	ID	EX	MEM	WB		
DADDU R7, R8, R9				IF	ID	EX	MEM	WB	
L1:DSUBU R1, R2, R3									

Branch not taken (above) ; Branch Taken (below)

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R1, R2, R3		IF	ID	EX	*				
DADDU R4, R5, R6			IF	ID	*				
DADDU R7, R8, R9				IF	*				
L1:DSUBU R1, R2, R3					IF	ID	EX	MEM	B

Predict-taken

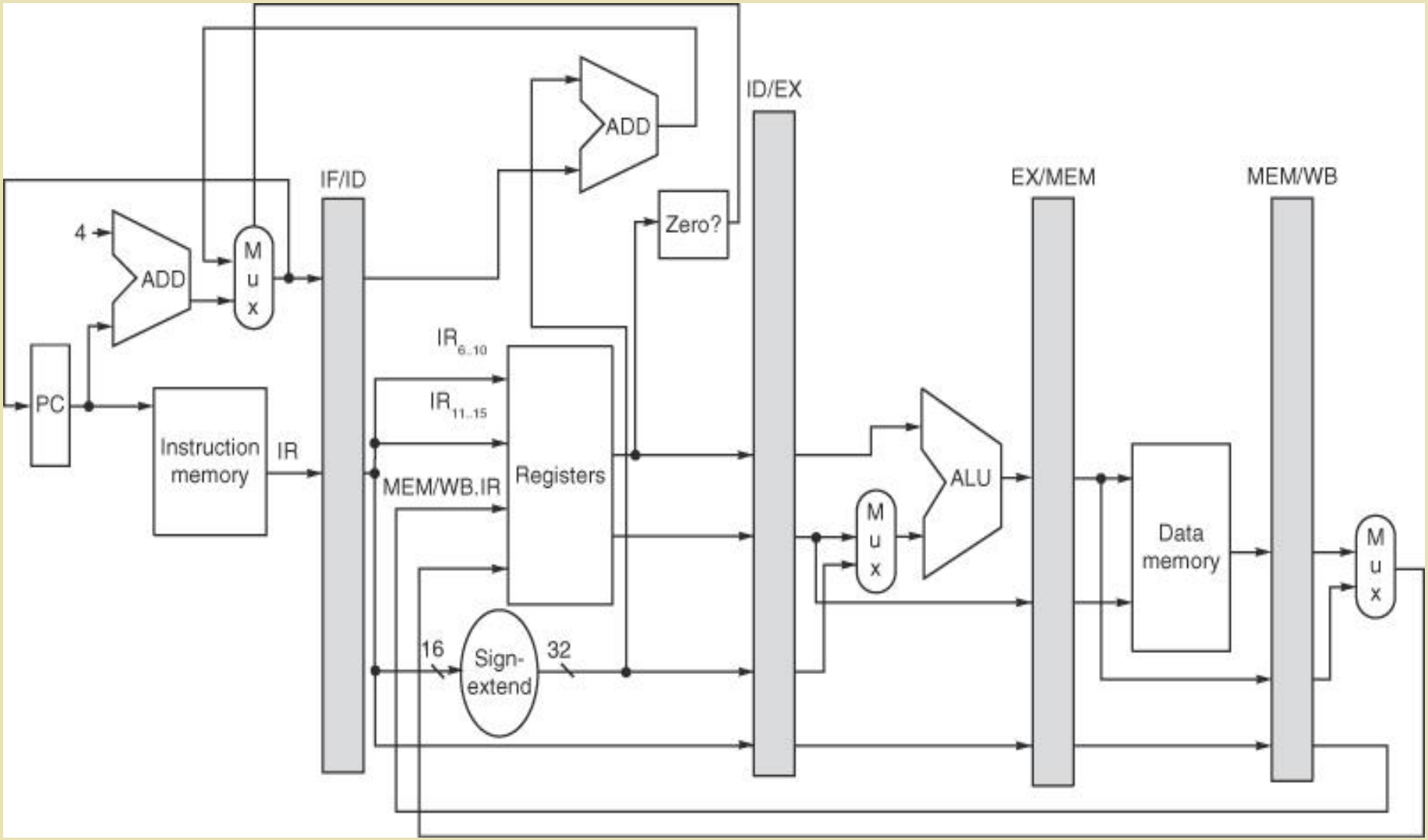
- Treat every branch instruction as taken
- Not useful in MIPS architecture since the branch address is not known ahead of decoder

Control Hazard

- In the worst case, the branch instruction could cause stall up to 3 clock cycles long (i.e., 2 stall cycles + 1 repeated IF cycle)
- The number of clock cycles in a branch stall can be reduced by two steps:
  - Find out whether the branch is taken or not taken earlier in the pipeline
  - Compute the branch address earlier

Control Hazard

- Solution #2: Thus, the pipeline structure can be redesign as follows:
  - move the zero equality test from pipe-stage 3 to pipe-stage 2
  - move the computation of target PC from pipe-stage 3 to pipe-stage 2



Modified Pipeline Structure to Minimize Branch Penalty

Modified Pipeline Structure (algorithm)

Pipe stage	Branch instruction
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) & (Regs[IF/ID.IR$_{6..10}$] op 0)) {IF/ID.NPC + sign-extended (IF/ID.IR[immediate field] << 2) else {PC + 4});**
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$
EX	
MEM	
WB	

MIPS Cycles (Pipeline)

1. Instruction fetch cycle (IF)

IF/ID.IR ← Mem [PC];
IF/ID.NPC, PC ← if ((EX/MEM.opcode == branch) & EX/MEM.cond) {EX/MEM.ALUOutput} else {PC + 4});

MIPS Cycle (Modified Pipeline)

1. Instruction fetch cycle (IF)

IF/ID.IR ← Mem [PC];
IF/ID.NPC, PC ← if ((IF/ID.opcode == branch) & Regs[$IF/ID.IR_{6..10}$] op 0)) {IF/ID.NPC + $(IF/ID.IR_{16})^{48}$ ## $(IF/ID.IR_{16..31}$ <<2)} else {PC + 4});

MIPS Cycles (Pipeline)

2. Instruction decode/register fetch cycle (ID)

$$ID/EX.A \leftarrow Regs\,[IF/ID.IR_{6..10}];$$
$$ID/EX.B \leftarrow Regs\,[IF/ID.IR_{11..15}];$$
$$ID/EX.Imm \leftarrow ((IF/ID.IR_{16})^{48} \,\#\#\, IF/ID.IR_{16..31});$$
$$ID/EX.NPC \leftarrow IF/ID.NPC;$$
$$ID/EX.IR \leftarrow IF/ID.IR;$$

MIPS Cycle (Modified Pipeline)

2. Instruction decode/register fetch cycle (ID)

$$ID/EX.A \leftarrow Regs\,[IF/ID.IR_{6..10}];$$
$$ID/EX.B \leftarrow Regs\,[IF/ID.IR_{11..15}];$$
$$ID/EX.Imm \leftarrow ((IF/ID.IR_{16})^{48} \,\#\#\, IF/ID.IR_{16..31});$$
$$ID/EX.IR \leftarrow IF/ID.IR;$$

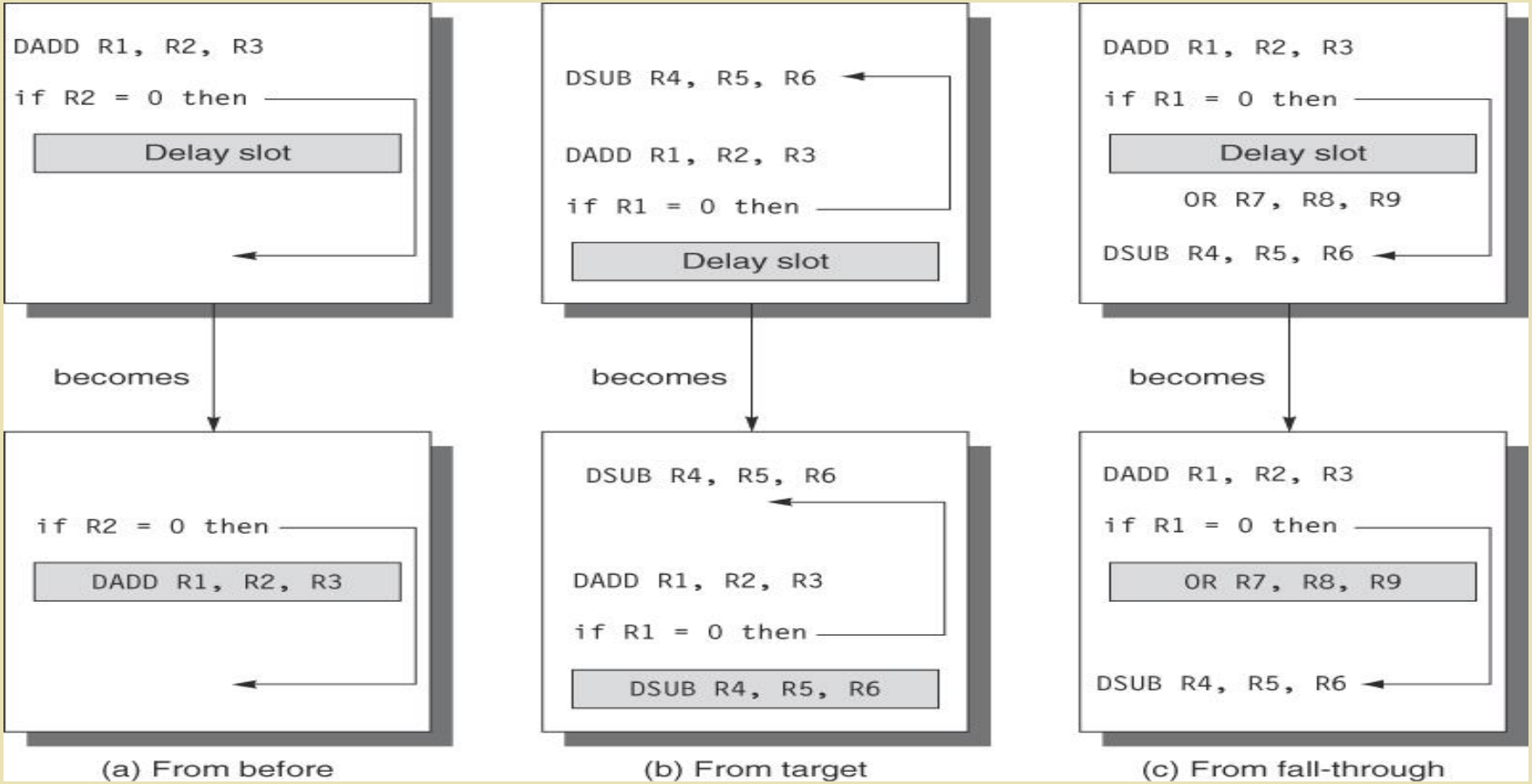
Control Hazard & Delay Slot Branch

- There is still 1-clock cycle stall on branches (known as branch delay)
- The number of branch-delay instruction is depended on the branch delay length
- Solution #3: Insert branch-delay instruction immediately after branch instruction
- Compiler can be used to improve performance by organizing the code so that and independent instruction can be place in branch delay slot

Illustration of Delay Slot branching

Instruction	1	2	3	4	5	6	7	8	9
BEQZ R1, L1	IF	ID	EX	MEM	WB				
DADDU R0, R0, R0		IF	ID	EX	MEM	WB			
DADDU R1, R2, R3									
DADDU R4, R5, R6									
DADDU R7, R8, R9									
L1:DSUBU R1, R2, R3									

DADDU R0, R0, R0 is an independent instruction that is placed in the delayed branch slot. This instruction is executed regardless if the branch instruction will cause a branch or not



Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b), the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.