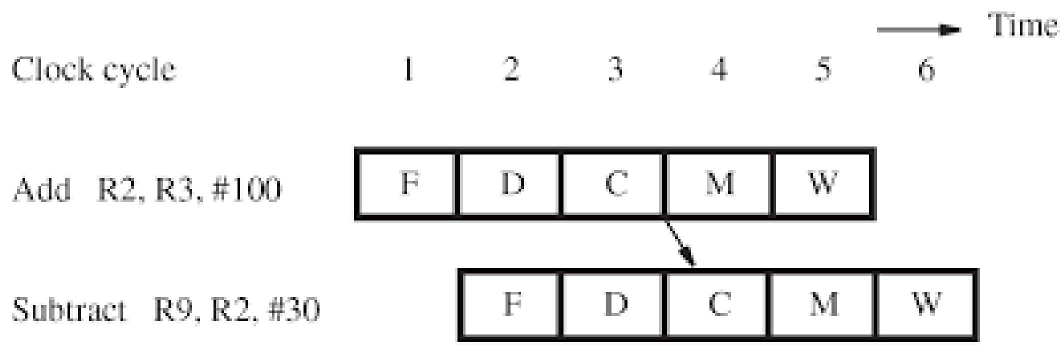
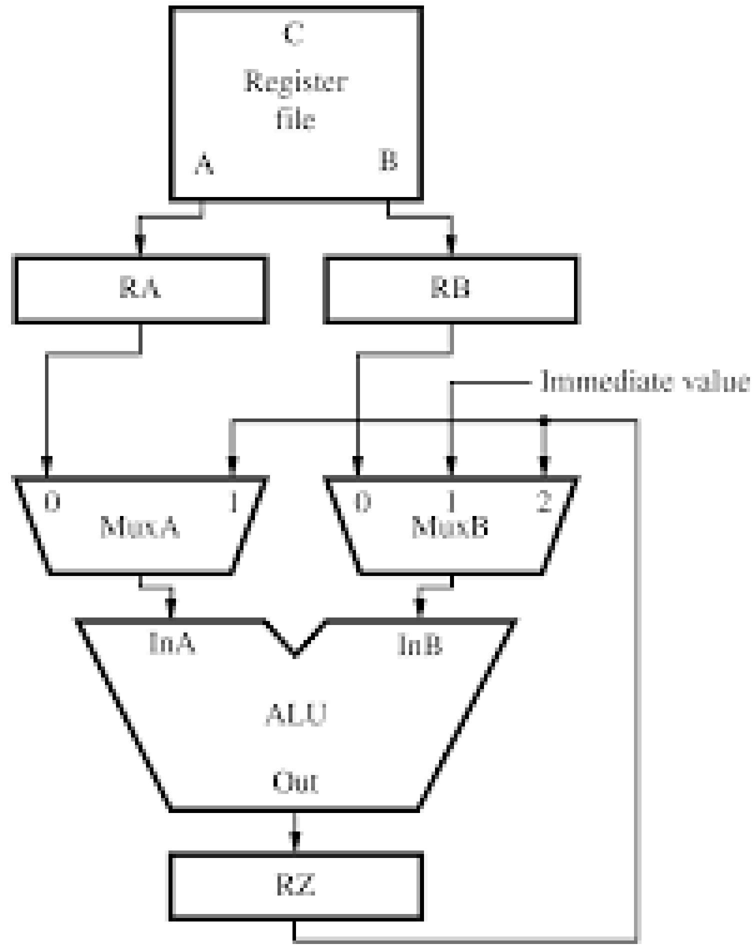
Computer Organization & Architecture

Operand forwarding



- A new multiplexer, MuxA, is inserted before input InA of the ALU, and the existing multiplexer MuxB is expanded with another input.
- The multiplexers select either a value read from the register file in the normal manner, or the value available in register RZ.


C
Register
file
A
B
RA
RB
Immediate value
0
1
MuxA
0
1
2
MuxB
InA
InB
ALU
Out
RZ

Operand forwarding

- Forwarding can also be extended to a result in register RY

  Add R2, R3, #100

  Or R4, R5, R6

  Subtract R9, R2, #30

- When the Subtract instruction is in the Compute stage:
  - Or instruction is in the Memory stage (where no operation is performed)
  - Add instruction is in the Write stage.

Operand forwarding

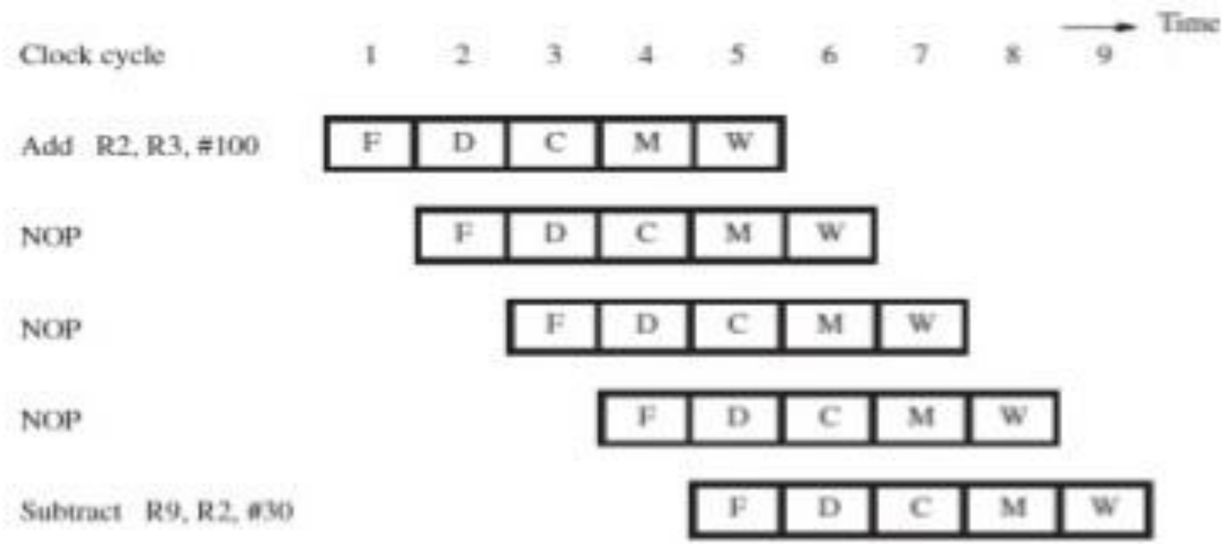
- New value of register R2 generated by Add instruction is in register RY.
- Forwarding this value from register RY to ALU input InA makes it possible to avoid stalling the pipeline.
  - MuxA requires another input for the value of RY.
  - Similarly, MuxB is extended with another input.

Data dependencies

- Handling data dependencies in software:
  - Task of detecting data dependencies and dealing with them to the compiler.
  - When compiler identifies a data dependency between two successive instructions I_j and I_{j+1}
    - Can insert three explicit NOP (No-operation) instructions between them.
    - NOPs introduce necessary delay to enable instruction I_{j+1} to read new value from register file after it is written.

Add R2, R3, #100
Subtract R9, R2, #30

Add R2, R3, #100
NOP
NOP
NOP
Subtract R9, R2, #30

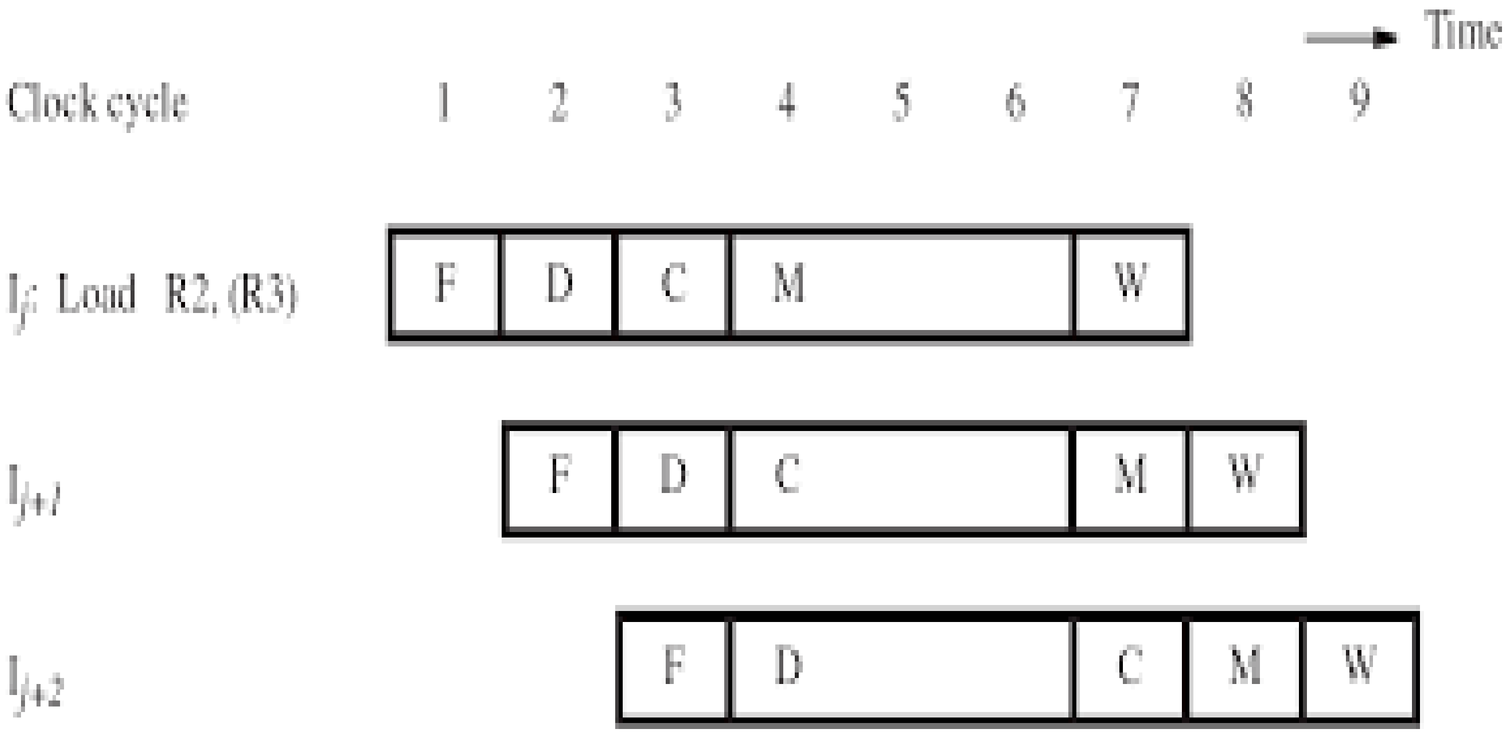


Data dependencies

- Compiler identifies dependencies and inserts NOP instructions
  - Simplifies hardware implementation of the pipeline.
  - However, code size increases and execution time is not reduced as it would be with operand forwarding.
- Compiler can attempt to optimize code to improve performance and reduce code size
  - Reordering instructions to move useful instructions into NOP slots.
  - Must consider data dependencies between instructions, which constrain the extent to which the NOP slots can be usefully filled.

Memory delays

- Delays arising from memory accesses are another cause of pipeline stalls.
- Ex., Load instruction may require more than one clock cycle to obtain its operand from memory.
  - May occur because the requested instruction or data are not found in the cache, resulting in a cache miss.
  - Cache miss causes all subsequent instructions to be delayed.
- A similar delay can be caused by a cache miss when fetching an instruction.


Time
Clock cycle
1
2
3
4
5
6
7
8
9
I_j: Load R2, (R3)
F
D
C
M
W
I_{j+1}
F
D
C
M
W
I_{j+2}
F
D
C
M
W

Memory delays

- Consider the instructions:

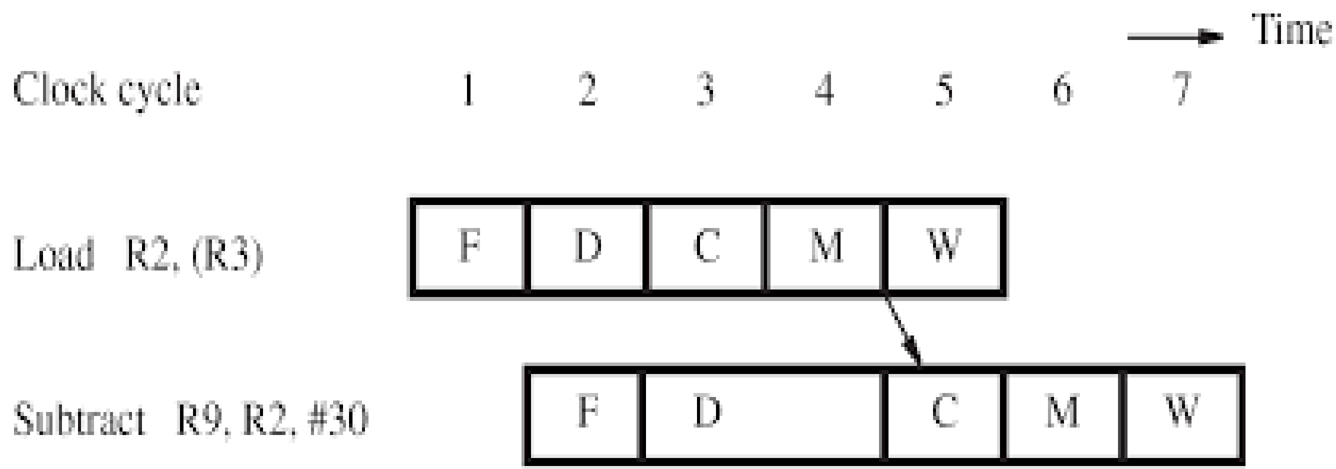
  Load R2, (R3)

  Subtract R9, R2, #30

  - Assume data for Load instruction is found in cache, requiring only one cycle to access the operand.
  - Destination register R2 for Load instruction is a source register for the Subtract instruction.

Memory delays

- Operand forwarding cannot be done
  - Data read from cache are not available until they are loaded into register RY at beginning of cycle 5.
- Subtract instruction must be stalled for one cycle, to delay the ALU operation.
- Memory operand, which is now in register RY, can be forwarded to the ALU input in cycle 5.


Time
Clock cycle
1
2
3
4
5
6
7
Load R2, (R3)
F
D
C
M
W
Subtract R9, R2, #30
F
D
C
M
W

Memory delays

- Compiler can eliminate one-cycle stall for this type of data dependency:
  - Reordering instructions to insert a useful instruction between the Load instruction and the instruction that depends on the data read from the memory.
  - Inserted instruction fills bubble that would otherwise be created.
  - If a useful instruction cannot be found by the compiler, then the hardware introduces the one-cycle stall automatically.
  - If the processor hardware does not deal with dependencies, then the compiler must insert an explicit NOP instruction.

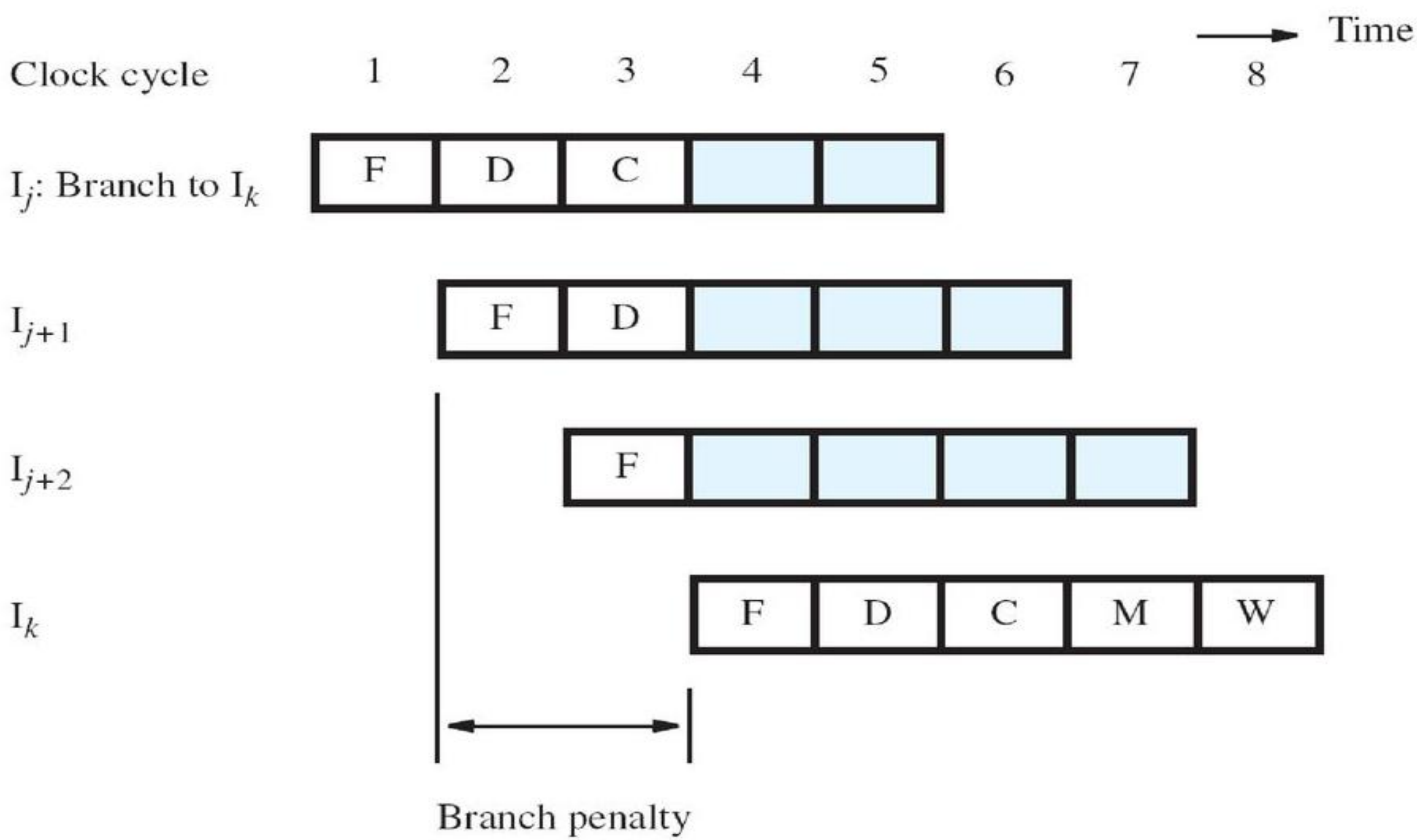
Control hazards: Branch delays

- Branch instructions can alter the sequence of execution
  - But they must first be executed to determine whether and where to branch.
- Pipelines delays due to branch instructions are referred to as ‘control hazards.’

Unconditional branch

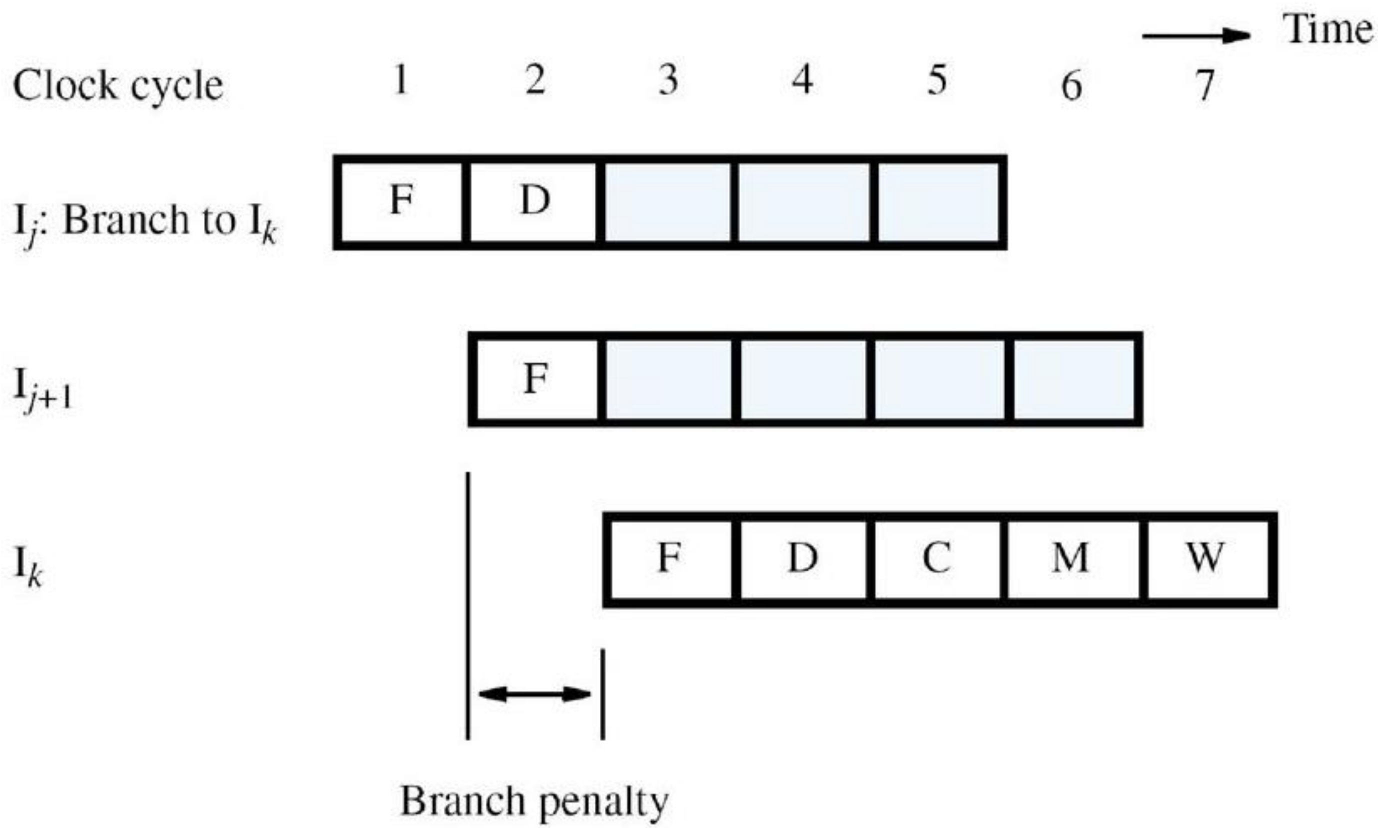
- In pipelined execution, instructions I_{j+1} and I_{j+2} are fetched in cycles 2 and 3
  - Before branch instruction is decoded and its target address is known.
  - Must be discarded.
  - Resulting two-cycle delay constitutes a **branch penalty**.
- Branch instructions occur frequently.
  - Represent about 20 percent of dynamic instruction count of most programs.
  - Dynamic count is number of instruction executions - some instructions in a program are executed many times, because of loops.

Unconditional branch



Unconditional branch

- With a two-cycle branch penalty:
  - Relatively high frequency of branch instructions could increase execution time for a program by as much as 40%
- Reducing branch penalty:
  - Requires branch target address to be computed earlier in the pipeline.
  - Rather than Compute stage, possible to determine target address and update program counter in Decode stage.
  - Instruction I_k can be fetched one clock cycle earlier, reducing branch penalty to one cycle
  - Only one instruction, I_{j+1}, is fetched incorrectly, because the target address is determined in the Decode stage.


Time
Clock cycle
1
2
3
4
5
6
7
I_j: Branch to I_k
F
D
I_j+1
F
I_k
F
D
C
M
W
Branch penalty

Unconditional branch

- Hardware must be modified to implement change.
  - Adder is needed to increment PC in every cycle.
  - Second adder is needed in Decode stage to compute a branch target address for every instruction.
- When instruction decoder determines that instruction is indeed a branch instruction
  - Computed target address will be available before end of cycle.
  - Can then be used to fetch target instruction in next cycle.

Conditional branches

- Consider conditional branch instruction such as

    Branch_if_[R5]=[R6] LOOP

  - Result of comparison in third step determines whether the branch is taken.
- For pipelining, branch condition must be tested as early as possible to limit branch penalty.
  - Target address for unconditional branch instruction can be determined in Decode stage.

Conditional branches

- Similarly, comparator that tests branch condition can also be moved to Decode stage
  - Enabling conditional branch decision to be made at same time that target address is determined.
  - Comparator uses values from outputs A and B of register file directly.
  - Moving branch decision to Decode stage ensures a common branch penalty of only one cycle for all branch instructions.

Branch penalty

Add R7, R8, R9

Branch_if_[R3]=0 TARGET

I_{j+1}

. . .

TARGET: I_k

- Assume branch target address and branch decision are determined in Decode stage
  - At same time that instruction I_{j+1} is fetched.

Branch delay slot

- Branch instruction may cause instruction I_{j+1} to be discarded after branch condition is evaluated.
  – If condition is true, then there is a branch penalty of one cycle before the correct target instruction I_k is fetched.
  – If the condition is false, then instruction I_{j+1} is executed, and there is no penalty.
  – In both of these cases, the instruction immediately following the branch instruction is always fetched.
- Location that follows a branch instruction is called the **branch delay slot**.

Branch delay slot

- Rather than conditionally discard instruction in delay slot:
  - Arrange to have pipeline always execute this instruction, whether or not the branch is taken.
- Instruction in the delay slot cannot be I_{j+1}, one that may be discarded depending on branch condition.
  - Compiler attempts to find a suitable instruction to occupy the delay slot
  - One that needs to be executed even when branch is taken.

Branch delay slot

- Compiler can do so by moving one of the instructions preceding the branch instruction to the delay slot
  - This can only be done if any data dependencies involving instruction being moved are preserved
- If a useful instruction is found, there will be no branch penalty
- If no useful instruction can be placed in delay slot because of constraints arising from data dependencies:
  - A NOP must be placed there instead
  - In this case, there will be a penalty of one cycle whether or not the branch is taken

Delayed branching

Add R7, R8, R9

Branch_if_[R3]=0 TARGET

I_{j+1}

. . .

TARGET: I_k

- Add instruction can safely be moved into the branch delay slot
  - Add instruction is always fetched and executed, even if the branch is taken.

Delayed branching

- Instruction I_{j+1} is fetched only if branch is not taken.
- Logically, execution proceeds as though branch instruction were placed after the Add instruction.
  - Branching takes place one instruction later than where the branch instruction appears in the instruction sequence.
  - This technique is called **delayed branching**.
  - Effectiveness of delayed branching depends on how often the compiler can reorder instructions to usefully fill the delay slot.
  - Experimental data indicate that compiler can fill a branch delay slot in 70 % or more of cases

END