ECE 341

Lecture # 13

Instructor: Zeshan Chishti
zeshan@ece.pdx.edu

November 12, 2014

Portland State University

Lecture Topics

- Pipelining
  - Pipeline Hazards
  - Data Dependencies
    - Operand Forwarding
    - Dealing with Memory Delays
  - Instruction Hazards
    - Branch penalty
    - Branch delay slot optimization

- Reference:
  - Chapter 6: Sections 6.3, 6.4, 6.5 and 6.6 (Pages 197 - 205 of textbook)

Pipeline Hazards

- Any condition that causes a pipeline to stall is called a *hazard*
- Pipeline stalls cause degradation in pipeline performance

We need to identify all pipeline hazards and find ways to minimize their impact

Types of Pipeline Hazards

There are three types of pipeline hazards:

- **Data hazard** – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
  - Instruction processing needs to be delayed until the operands become available
- **Instruction (control) hazard** – a delay in the availability of an instruction or the memory address needed to fetch the instruction
- **Structural hazard** – a situation where two (or more) instructions require the use of a given hardware resource at the same time.

Data Hazards

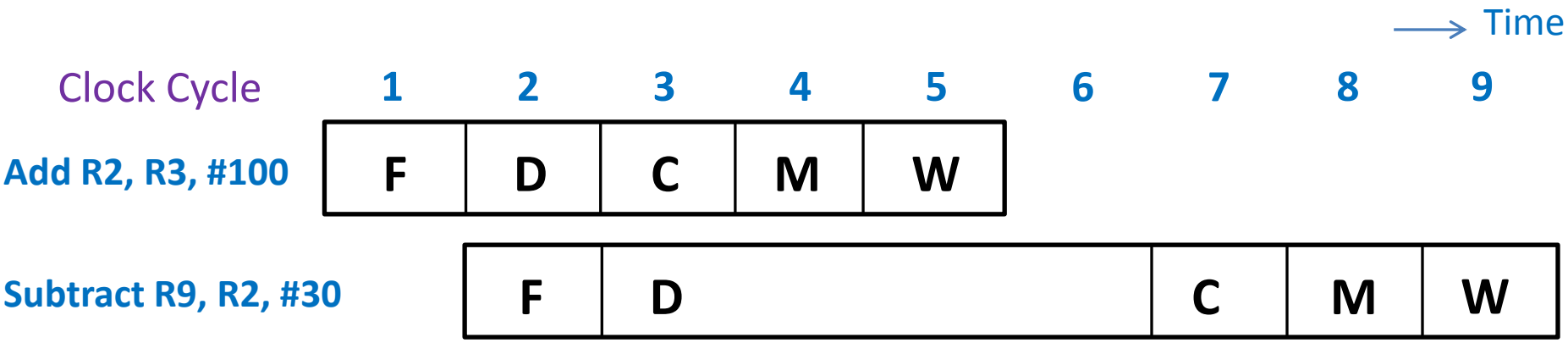
- Consider the following two instructions executed in a program sequence:

  Add R2, R3, #100

  Subtract R9, R2, #30

- The destination register (R2) for the *first* instruction is a source register for the *second* instruction
- Register R2 carries data from the *first* instruction to the *second* instruction => There is a **data dependency** between these two instructions
- *First* instruction writes to register R2 in stage-5 of the pipeline (Writeback stage)
- *Second* instruction reads register R2 in stage-2 of the pipeline (decode stage)
- If *second* instruction reads R2 *before* the *first* instruction writes R2, the result of *second* instruction would be incorrect, as it would be based on R2's old value
- To obtain the correct result, second instruction needs to wait until the first instruction has written to R2

Stalls Caused by Data Hazards



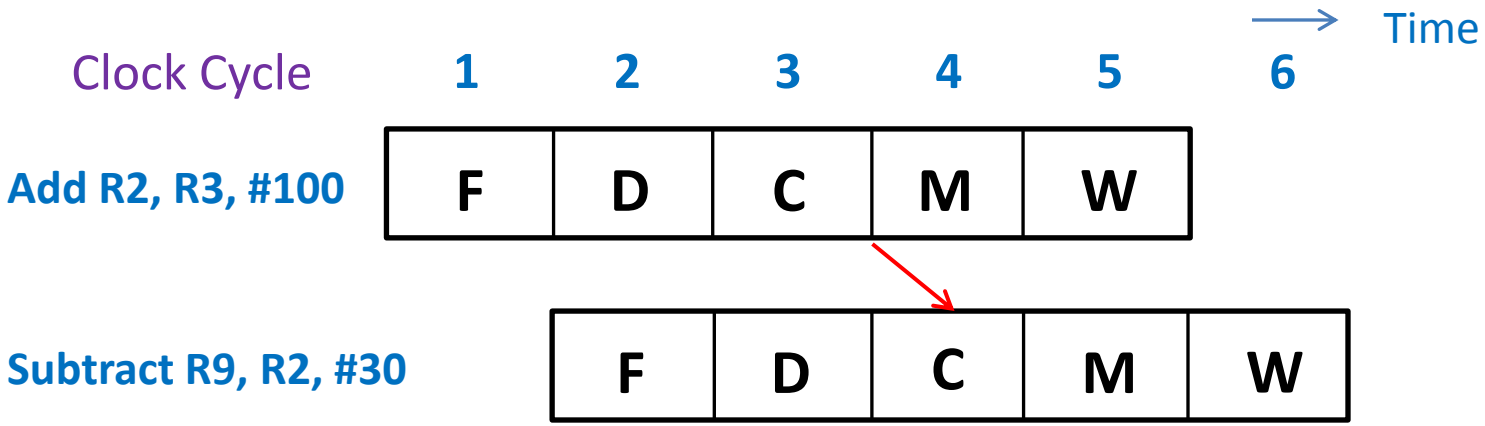
- *Second* instruction is stalled in decode stage for three additional cycles to delay reading R2 until the new value of R2 has been written by *first* instruction

How is this pipeline stall implemented in hardware?

- Control circuitry recognizes the data dependency when it decodes the second instruction (comparing source register ids in inter-stage buffer B1 with destination register id in inter-stage buffer B2)
- During cycles 3 to 5:
  - first instruction proceeds through the pipe
  - second instruction is held in inter-stage buffer B1
  - control circuitry inserts NOP (no-operation) instructions in buffer B2. These instructions pass through the pipeline stages C, M and W with no impact

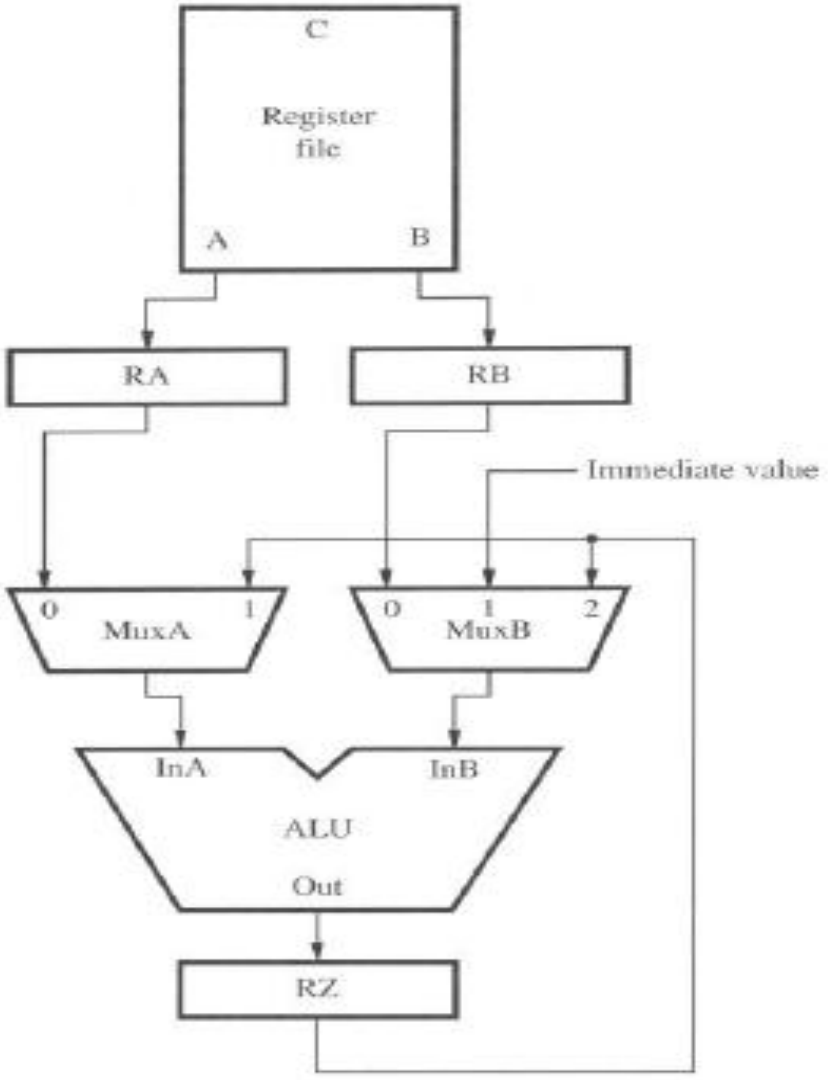
Operand Forwarding - Alleviating Data Hazards

- Stalls due to data dependencies can be avoided by operand forwarding
- Consider the two instructions discussed in the previous example
  - The result of *first* instruction is actually available after the completion of *C* stage (cycle 3), when the ALU completes its computation
  - This result is stored in inter-stage buffer B3 at the end of cycle 3
  - Instead of stalling the *second* instruction, the hardware can ***forward*** the result from B3 to the ALU input, where it can be used by the *second* instruction
  - The arrow in below figure shows data being forwarded from *C* stage of first instruction to *C* stage of the second instruction



Operand Forwarding (cont.)

Modification of the datapath to support operand forwarding



- Added a new multiplexer MuxA to select between the forwarded value and the value read from source register

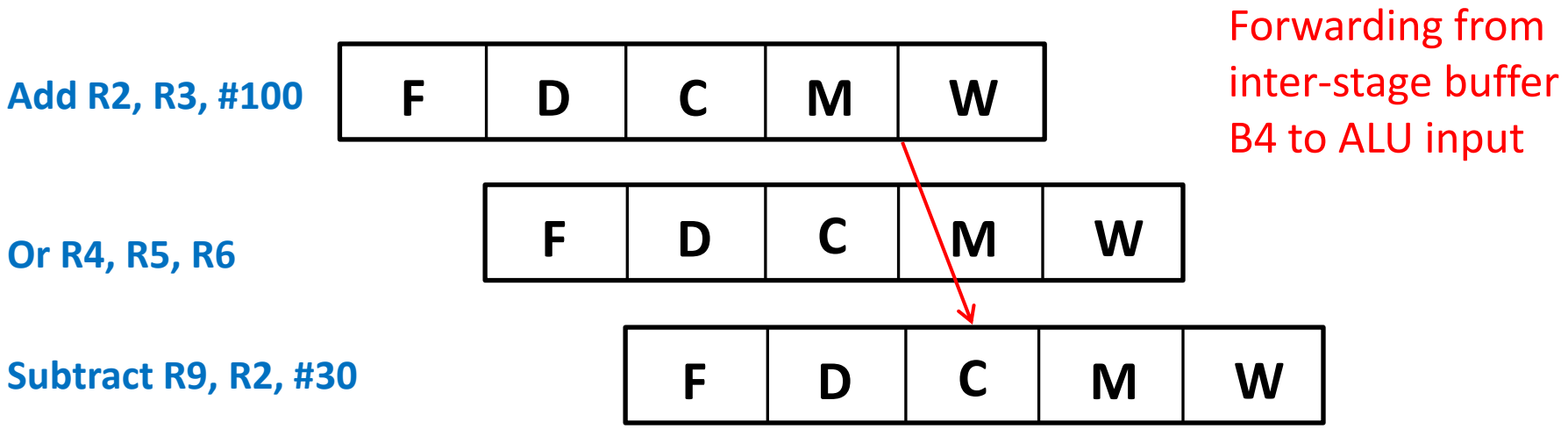
- Added a new input to MuxB for forwarded data

Operand Forwarding (cont.)

- So far, we have looked at forwarding from *C* stage to *C* stage
- Such forwarding mitigates data hazards between successive instructions

- How can we avoid stalls when instruction I_{j+2} depends on instruction I_j?
- Solution: Forward operands from *M* stage to *C* stage

- **Example:**



Stalls Caused by Loads

- Load instructions can also cause pipeline stalls

- **Example 1 (No data dependency):** A Load instruction *misses* in the cache and needs to stall for multiple cycles until the data is read from memory. All the subsequent instructions stall (Figure 6.7), even if they have *no* data dependency with the Load (no data hazard)
  - Operand forwarding cannot mitigate these stalls

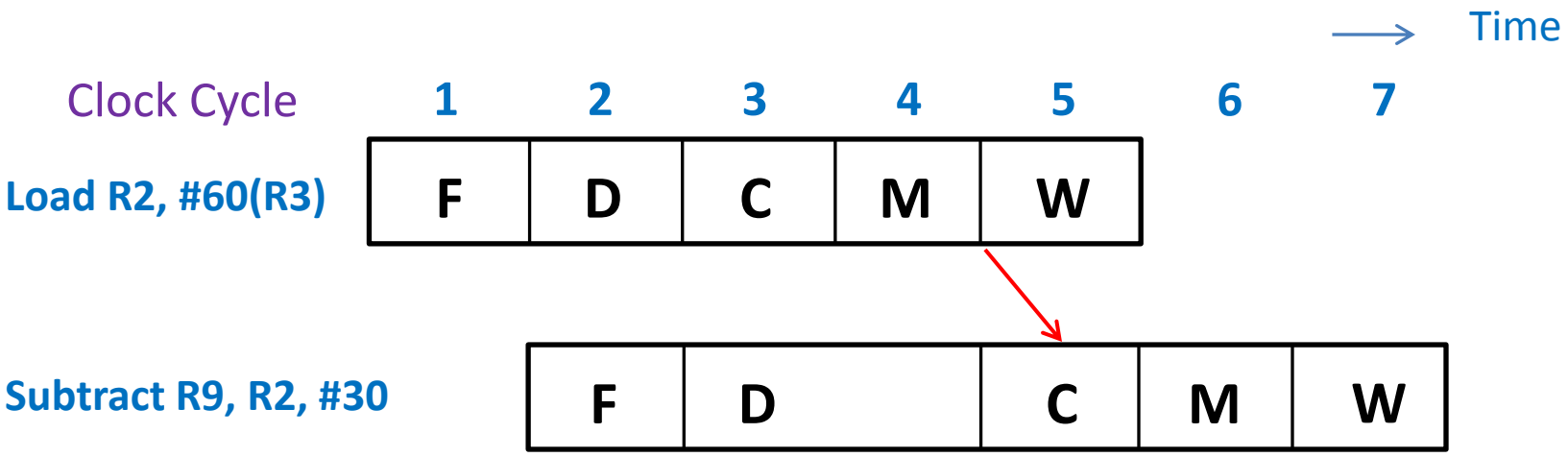
- **Example 2 (Data dependency):** Consider the following instruction sequence:

  Load R2, #60(R3)

  Subtract R9, R2, #30

  - Register R2 causes a data dependency between the two instructions
  - Assume that the Load instruction hits in the cache (no stall). Data read from memory is available after the *M* stage and written to R2 in the *W* stage
  - Subtract instruction will need to stall in the *D* stage, because it needs the data loaded into register R2

Operand Forwarding for Load Instructions



- Operand forwarding from Load instructions cannot happen before the data has been read from memory (end of cycle 4)
- Subtract instruction will need to stall for one cycle

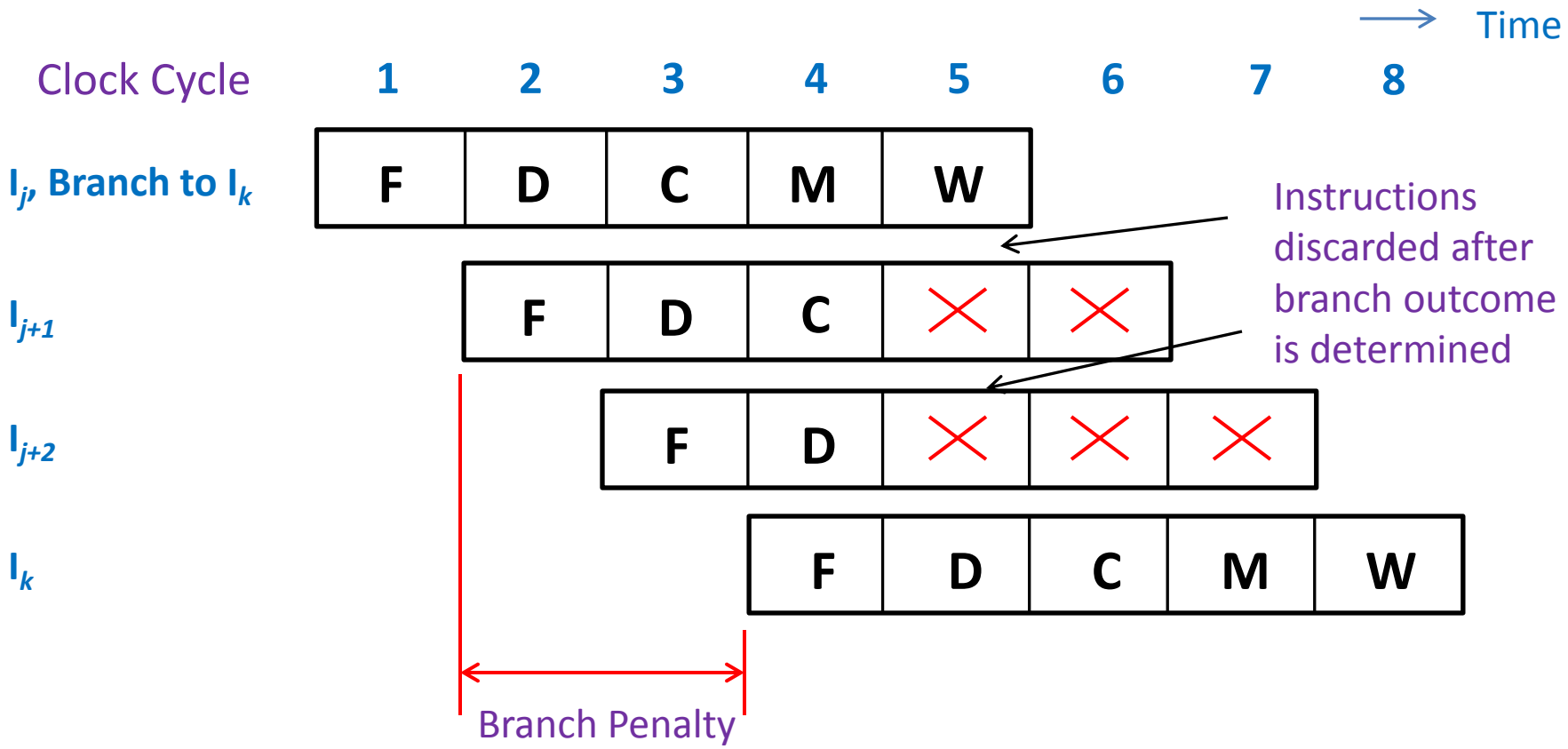
Operand forwarding from Load instruction reduces stalls but does not completely eliminate stalls

Instruction Hazards

Instruction Hazards

- Instruction hazards (also called control hazards) are caused by a delay in the availability of an instruction or the memory address needed to fetch the instruction
- In ideal pipelined execution, a new instruction is fetched every cycle, while the previous instruction is being decoded
- This will work fine as long as the instruction addresses follow a pre-determined sequence (e.g., PC ← [PC]+4)
- However, branch instructions can alter this sequence
- Branch instructions first need to be executed to determine *whether* and *where* to branch
- Pipeline needs to be stalled before the branch outcome is decided

Unconditional Branches

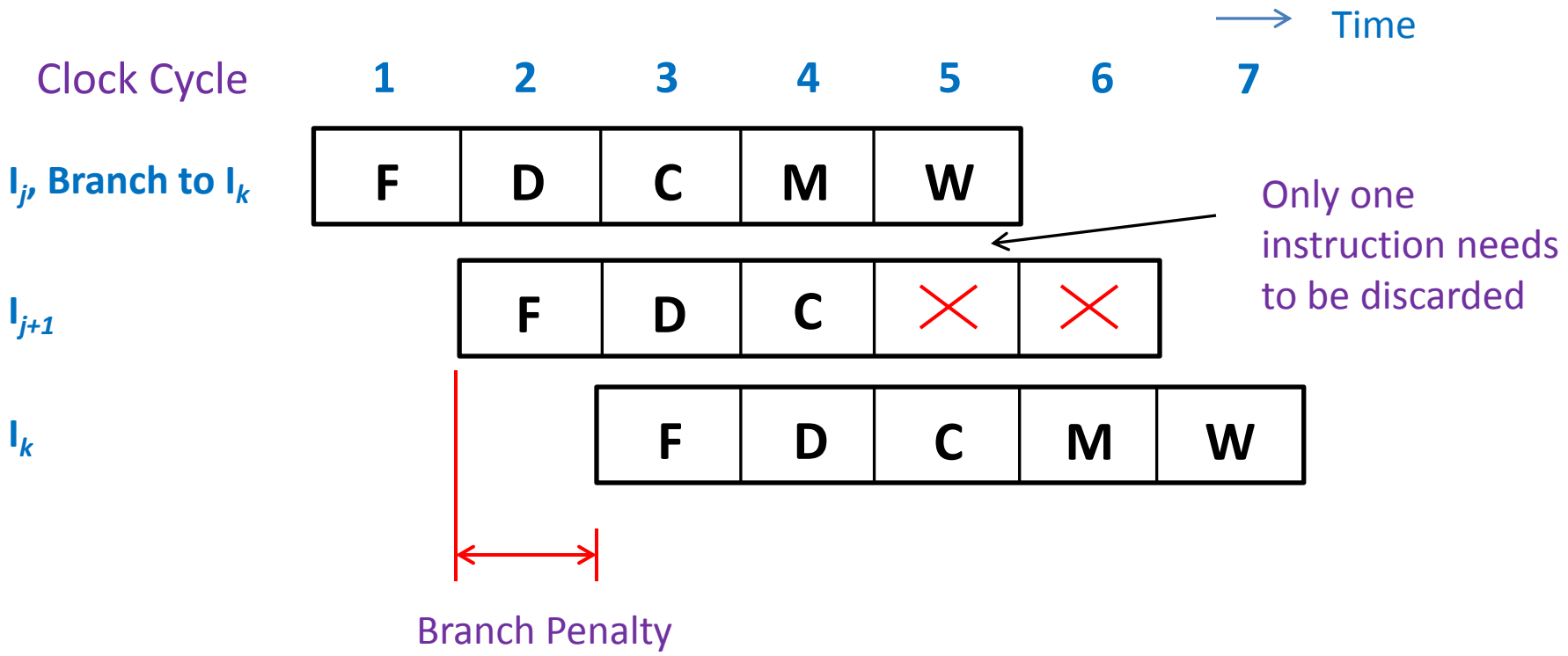


- Branch target computed in cycle-3. Two instructions on the wrong path (I_{j+1} and I_{j+2}) have already been fetched in cycles 2 and 3 and need to be discarded
- Branch Penalty in this case is 2 cycles

Reducing Branch Penalty

- Branch instructions occur frequently (~ 20% of dynamic instruction count in a typical program), 2-cycle branch penalty can increase execution time by 40%
- How to reduce branch penalty for unconditional branches?
- Compute branch target address earlier in the pipeline
  - Include an adder in the “decode” stage to compute target address
  - When the decoder determines that the instruction is an unconditional branch, the computed target address is available before the end of the decode stage

Reducing Branch Penalty (cont.)



- Branch target address computed in cycle # 2
- Branch penalty reduced to one cycle
- Only one instruction (I_{j+1}) is fetched incorrectly and needs to be discarded

Conditional Branches

- Consider a conditional branch instruction such as

  Branch_if_[R5]=[R6] LOOP

- Testing the branch condition (comparison between [R5] and [R6]) determines whether the branch is *taken* or not *taken*
- Recall from the datapath description that this comparison is done in compute stage (cycle # 3 of instruction processing)
- If the branch is taken, the penalty will be 2 cycles

- How to reduce the branch penalty for conditional branches?
- Test the branch condition in the "Decode" stage
  - Move the comparator from "Compute" stage to "Decode" stage
  - Branch condition tested in parallel with target address computation
  - Branch penalty reduced to one cycle

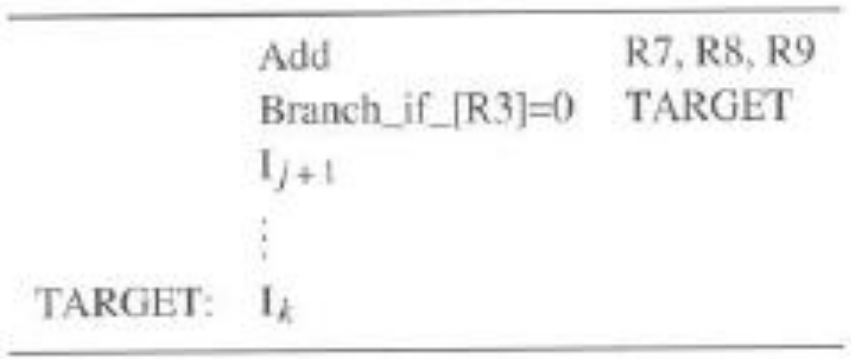
Branch Delay Slot

- Moving the branch decision and effective address calculation to the "Decode" stage reduces the branch penalty from two to one cycle
- Are there ways to reduce the branch penalty to zero?
- Yes. The basic idea behind Branch delay slot is to find an instruction that appears before the branch in program order but the branch outcome is independent of that instruction
- **If** the compiler finds such an instruction, it places this instruction in the branch delay slot (the location immediately after the branch instruction)
- This instruction is always executed irrespective of whether the branch is taken or not taken => no discarding and no branch penalty

Branch Delay Slot Example

Without the delay slot optimization:

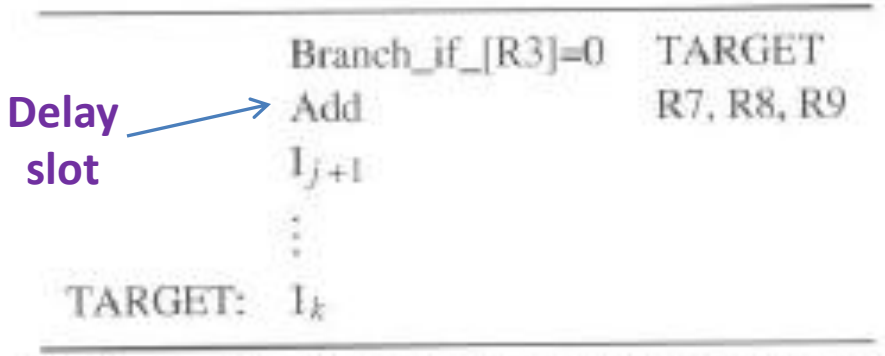
- If the branch condition is evaluated to be true ([R3] == 0), the branch is *taken* and instruction I_{j+1} needs to be discarded => penalty = 1 cycle
- If the branch condition is false ([R3] ! = 0), there is no penalty



(a) Original sequence of instructions containing a conditional branch instruction

After the delay slot optimization:

- The "Add" instruction is always executed, even if the branch is taken
- Instruction I_{j+1} is fetched only if the branch is *not taken*
- Branch penalty if zero irrespective of the branch outcome



(b) Placing the Add instruction in the branch delay slot where it is always executed

Branch Delay Slot Limitations

- The delay slot optimization must preserve all the data dependences
- In the previous example, it was safe to move the “Add” instruction to the branch delay slot, because the branch outcome was independent of that particular “Add” instruction
- But because of data dependences, it is not always possible to find a suitable instruction to fill the delay slot
- If no useful instruction can be placed in the delay slot, the compiler places a NOP (no operation) instruction in that slot
  - In this case, branch penalty = 1, irrespective of the branch outcome
- Effectiveness of delay slot optimization depends on the ability of compiler to find a useful delay slot candidate (~ 70% of the cases)
- Other ways to reduce branch penalty to zero?