Computer Organization & Architecture

6-2 Pipeline Issues

Wang Guohua

School of Software Engineering

Contents of this lecture

- Pipeline Issues
- Data Dependency
  - Operand Forwarding
  - Extension of Operand Forwarding
  - Software Handling of Dependencies
- Memory Delays
- Resource Limitations
- Branch Delays

Pipeline Issues (1)

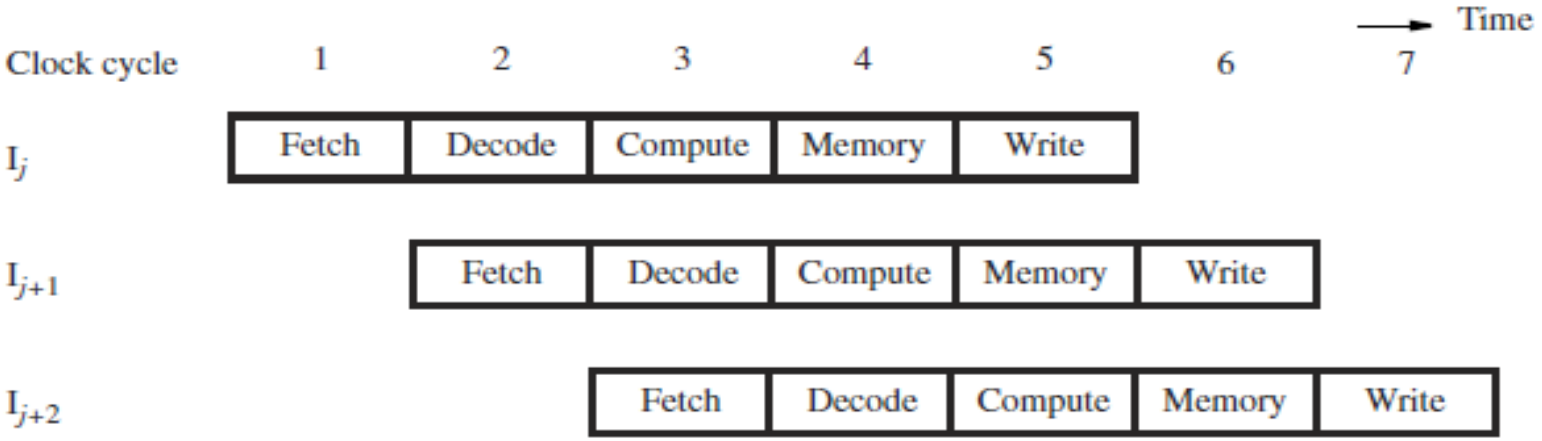


Figure 6.1 Pipelined execution—the ideal case.

- Consider instructions I_j and I_{j+1}, where the destination register for instruction I_j is a source register for instruction I_{j+1}.

Pipeline Issues (2)

- Any condition that causes a pipeline to stall is called a hazard.
- Three Types of Hazard
  - Data Hazard
    - Any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
  - Instruction (control) Hazard
    - A delay in the availability of an instruction causes the pipeline to stall.
  - Structural Hazard
    - The situation when two instructions require the use of a given hardware resource at the same time.

Data Dependency (1)

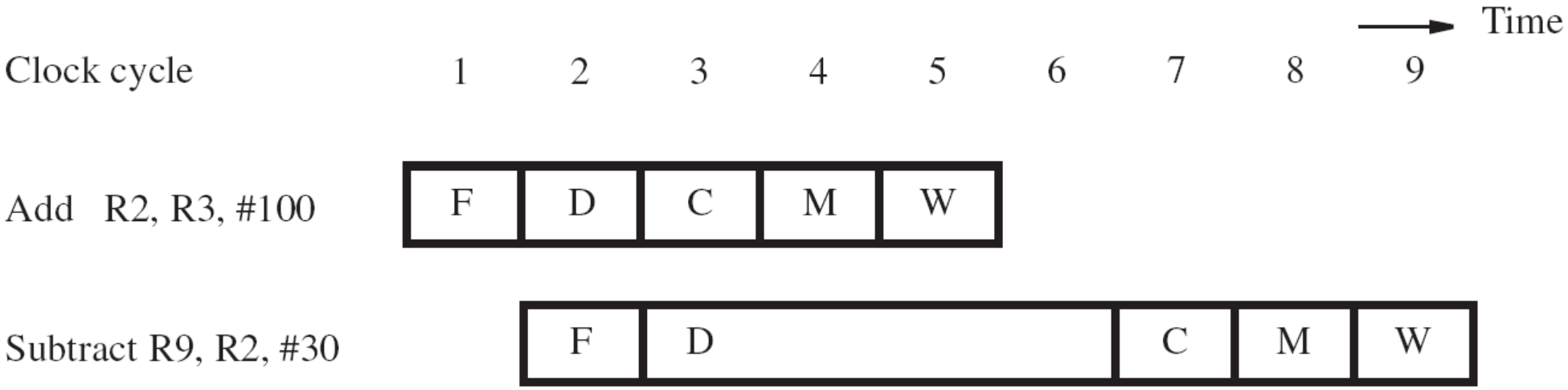
- Example

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

  – Destination R2 of Add is a source for Subtract
  – There is a *data dependency* between them because R2 carries data from Add to Subtract
  – On *non*-pipelined datapath, result is available in R2 because Add completes before Subtract

Data Dependency (2)

- Stalling the Pipeline
  - With pipelined execution, old value is still in register R2 when Subtract is in Decode stage
  - So stall Subtract for 3 cycles in Decode stage
  - New value of R2 is then available in cycle 6

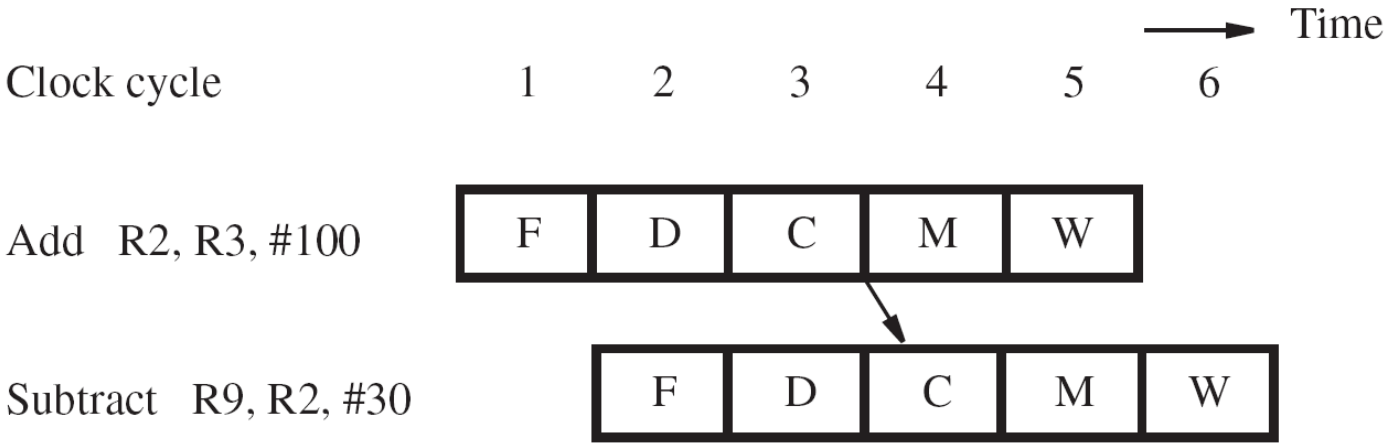


Data Dependency (3)

- Details for Stalling the Pipeline
  - Control circuitry must recognize dependency while Subtract is being decoded in cycle 3
  - Interstage buffers carry register identifiers for source(s) and destination of instructions
  - In cycle 3, compare destination identifier in Compute stage against source(s) in Decode
  - R2 matches, so Subtract kept in Decode while Add allowed to continue normally

Operand Forwarding (1)

- Operand forwarding handles dependencies without the penalty of stalling the pipeline
- For the preceding sequence of instructions, new value for R2 is available at end of cycle 3
- *Forward* value to where it is needed in cycle 4



Operand Forwarding (2)

- Hardware Details
  - Introduce multiplexers before ALU inputs to use contents of register RZ as forwarded value
  - Control circuitry now recognizes dependency in cycle 4 when Subtract is in Compute stage
    - Interstage buffers still carry register identifiers
    - Compare destination of Add in Memory stage with source(s) of Subtract in Compute stage
  - Set multiplexer control based on comparison

Operand Forwarding (3)

- Hardware Details (ctd.)

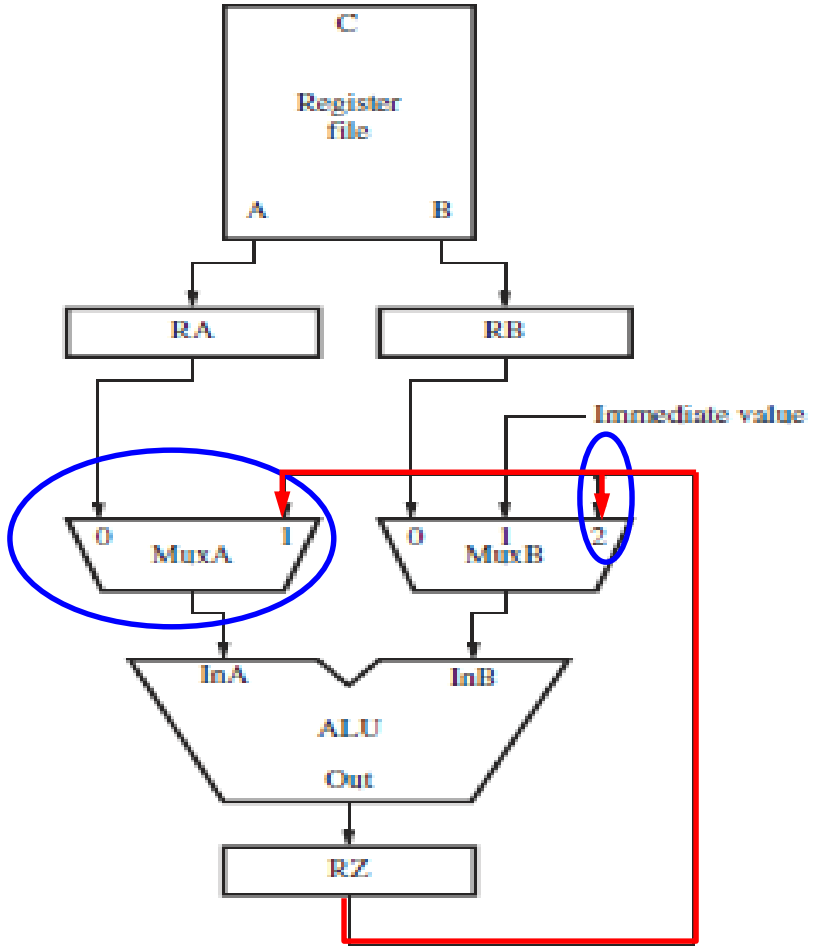


Figure 6.5 Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

Extension of Operand Forwarding (1)

- Forwarding can also be extended to a result in register RY in Figure 5.8.
- This would handle a data dependency such as the one involving register R2 in the following sequence of instructions:

```
Add       R2, R3, #100
Or        R4, R5, R6
Subtract  R9, R2, #30
```

Extension of Operand Forwarding (2)

- Hardware Detail

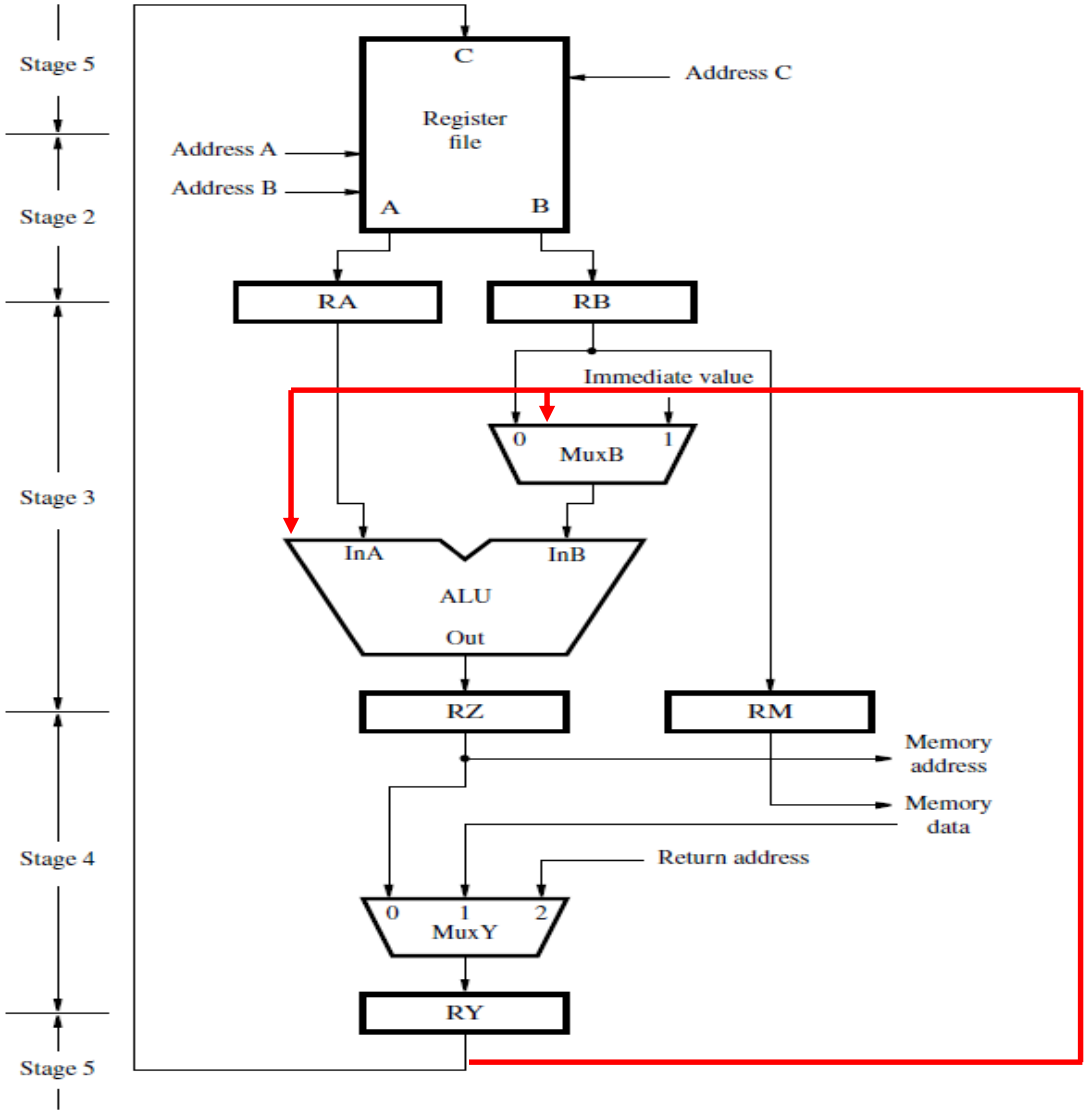
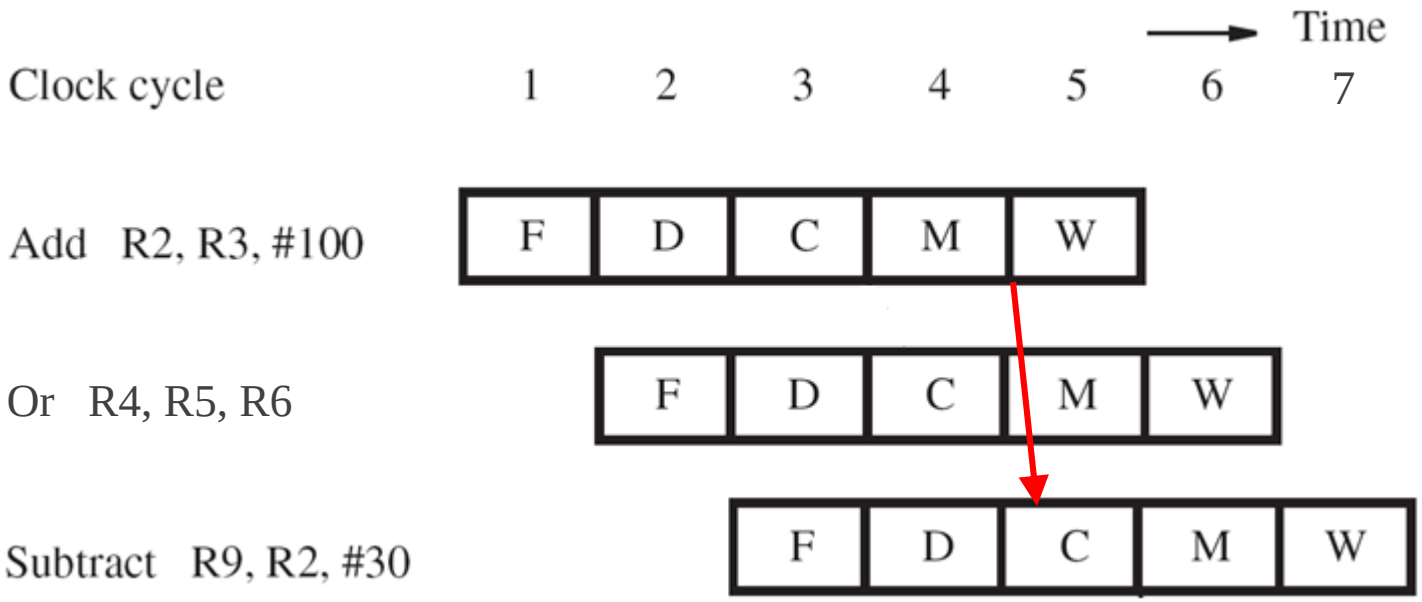


Figure 5.8 Datapath in a processor.

Extension of Operand Forwarding (3)

- Forward value to where it is needed in cycle 5



Software Handling of Dependencies (1)

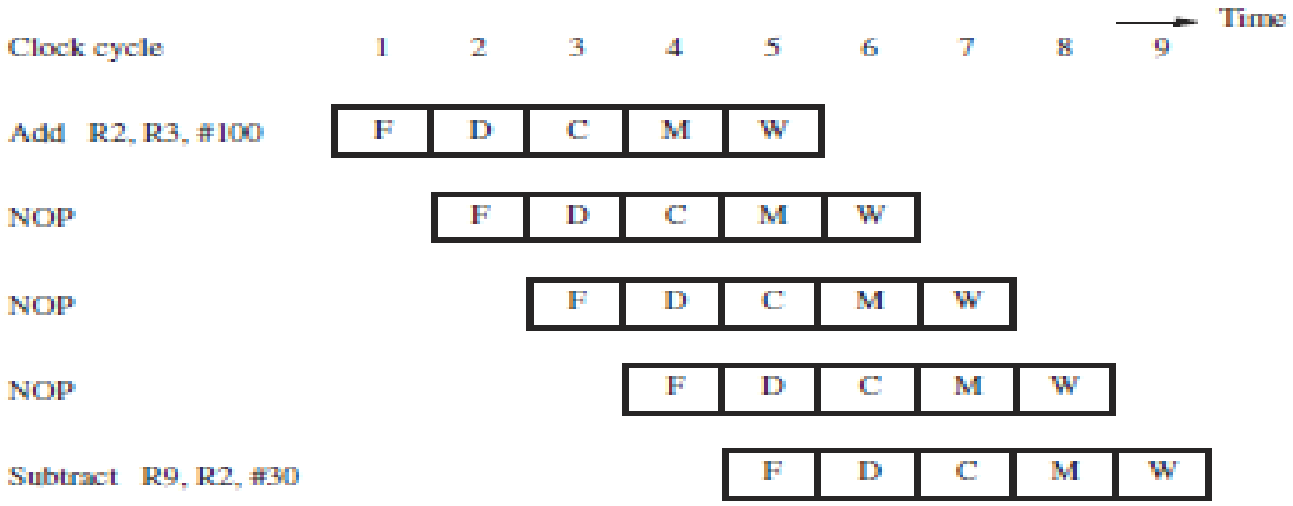
- Compiler can generate & analyze instructions
- Data dependencies are evident from registers
- Compiler puts three *explicit* NOP instructions between instructions having a dependency
- Delay ensures new value available in register but causes total execution time to increase
- Compiler can *optimize* by moving instructions into NOP slots (if data dependencies permit)

Software Handling of Dependencies (2)

- Example (ctd.)

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

(a) Insertion of NOP instructions for a data dependency



(b) Pipelined execution of instructions

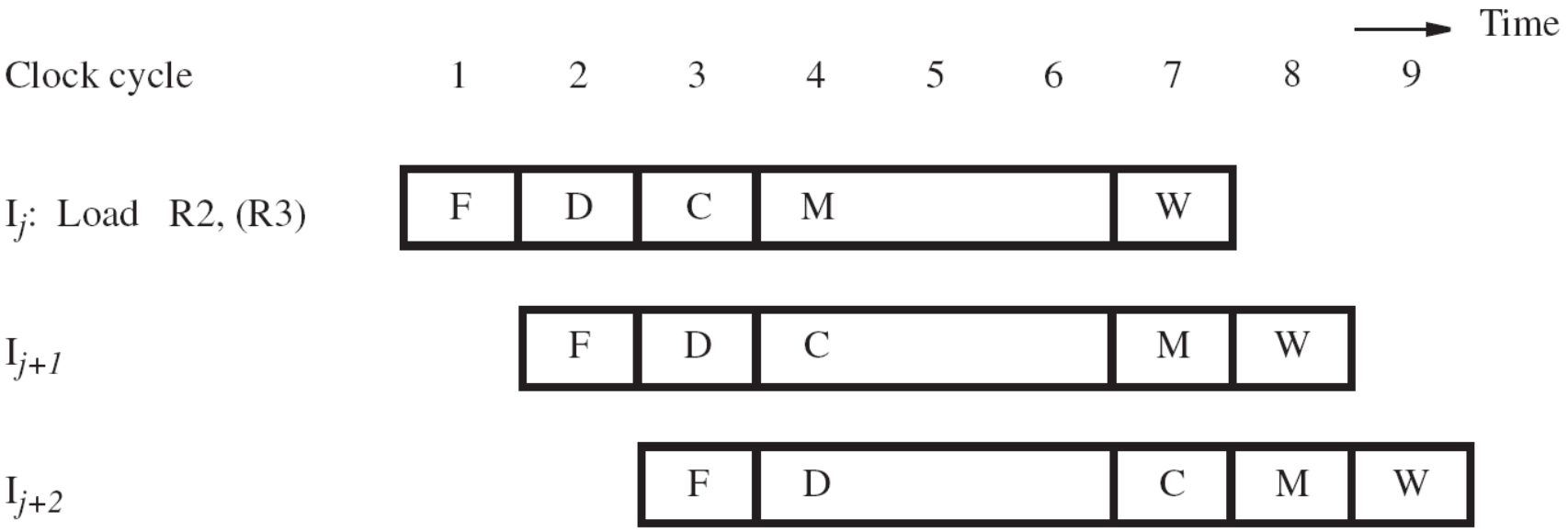
Figure 6.6 Using NOP instructions to handle a data dependency in software.

Memory Delays (1)

- Memory delays can also cause pipeline stalls.
- A cache memory can hold instructions and data from the main memory, and it is faster to access.
- With a cache, typical access time is one cycle.
- But a cache *miss* requires accessing slower main memory with a much longer delay.
- In pipeline, memory delay for one instruction causes subsequent instructions to be delayed.

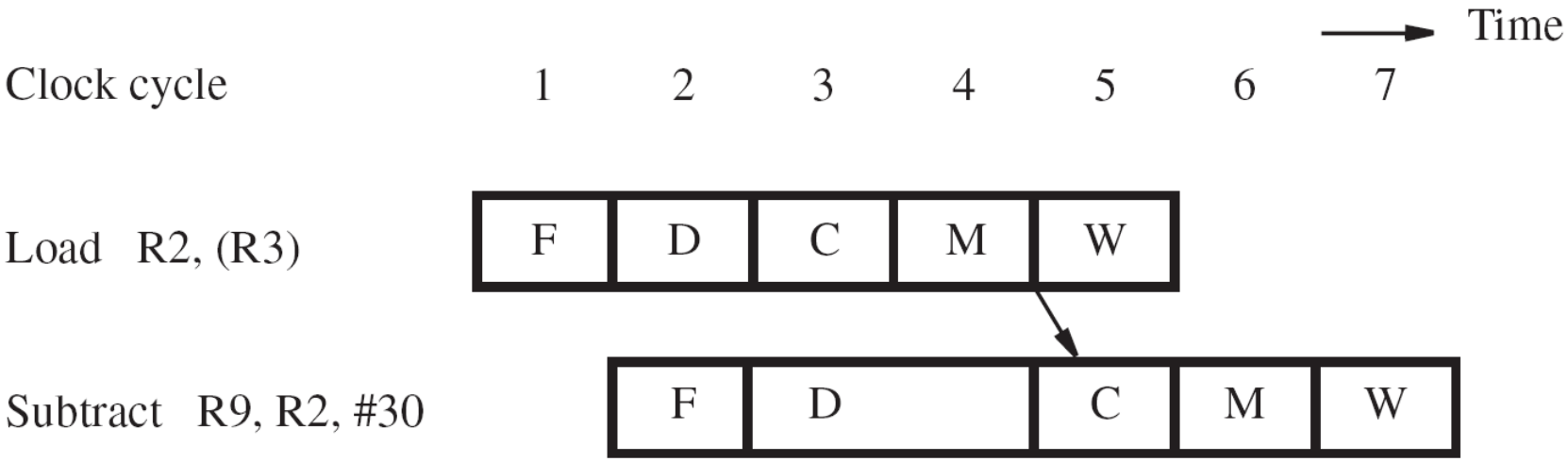
Memory Delays (2)

- Example



Memory Delays (3)

- Even with a cache *hit*, a Load instruction may cause a short delay due to a data dependency.
- One-cycle stall required for correct value to be forwarded to instruction needing that value.
- Optimize with useful instruction to fill delay.



Resource Limitations (1)

- Two instructions may need to access the same resource in the same clock cycle.
- One instruction must be stalled to allow the other instruction to use the resource.
- This can be prevented by providing additional hardware.

Resource Limitations (2)

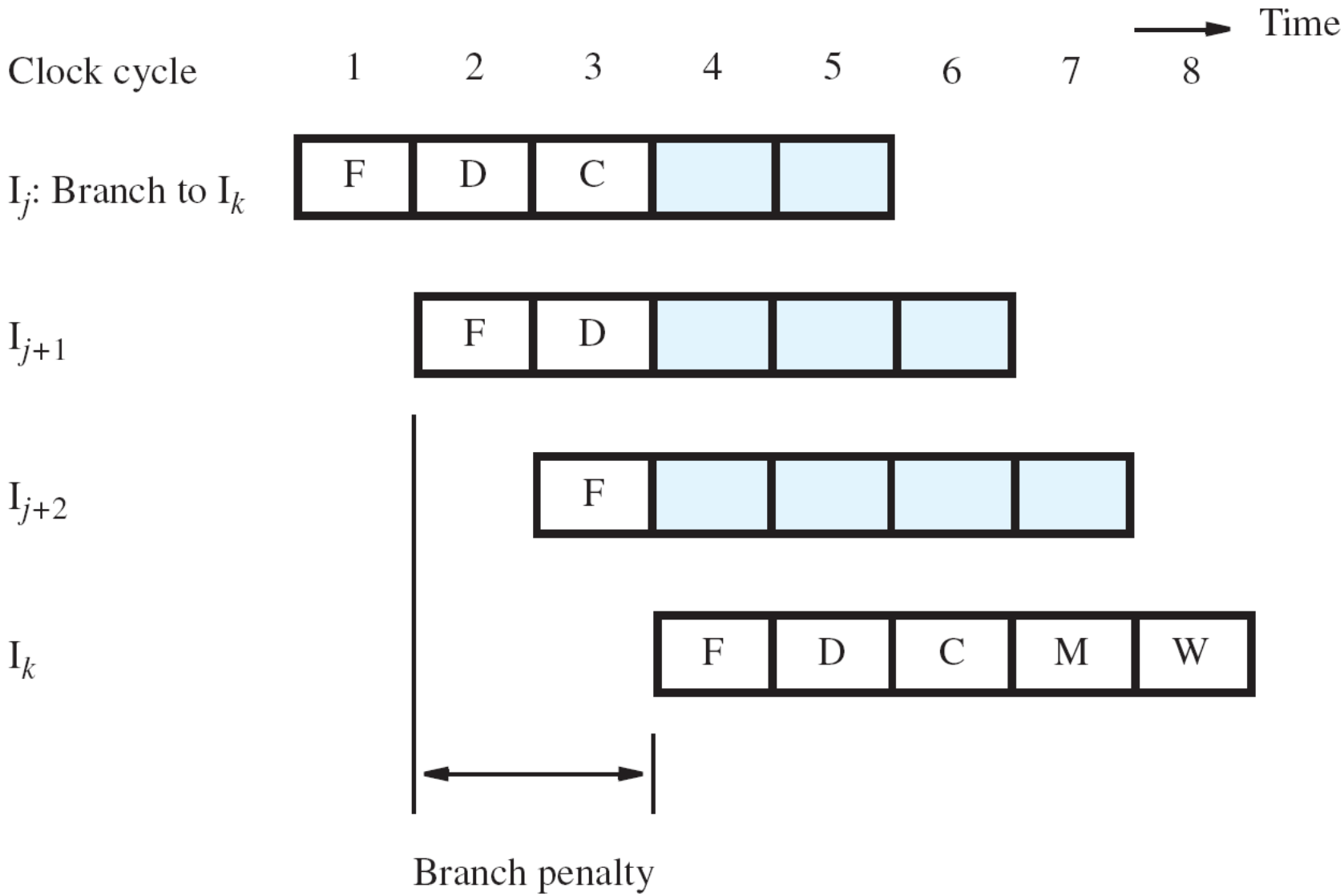
- E.g., If both the Fetch and Memory stages are connected to the cache
  - Normally, the Fetch stage accesses the cache in every cycle.
  - When there is a Load or Store instruction in the Memory stage also needing to access the cache, the activity must be stalled for one cycle.
  - We can use separate caches for instructions and data to allow the Fetch and Memory stages to proceed simultaneously without stalling.

Branch Delays

- Ideal pipelining: fetch each new instruction while previous instruction is being decoded.
- Branch instructions alter execution sequence, but they must be processed to determine:
  - Whether and where to branch
- Any delay for determining branch outcome leads to an increase in total execution time.

Unconditional Branches

- Consider instructions I_j , I_{j+1} , I_{j+2} in sequence, I_j is an unconditional branch with target I_k.
- In Chapter 5, the Compute stage determined the target address using offset and the updated PC value.
- In pipeline, target I_k is known for I_j in cycle 4, but instructions I_{j+1} , I_{j+2} fetched in cycles 2 & 3.
- Target I_k should have followed I_j immediately, so discard I_{j+1} , I_{j+2} and incur two-cycle *branch penalty.*

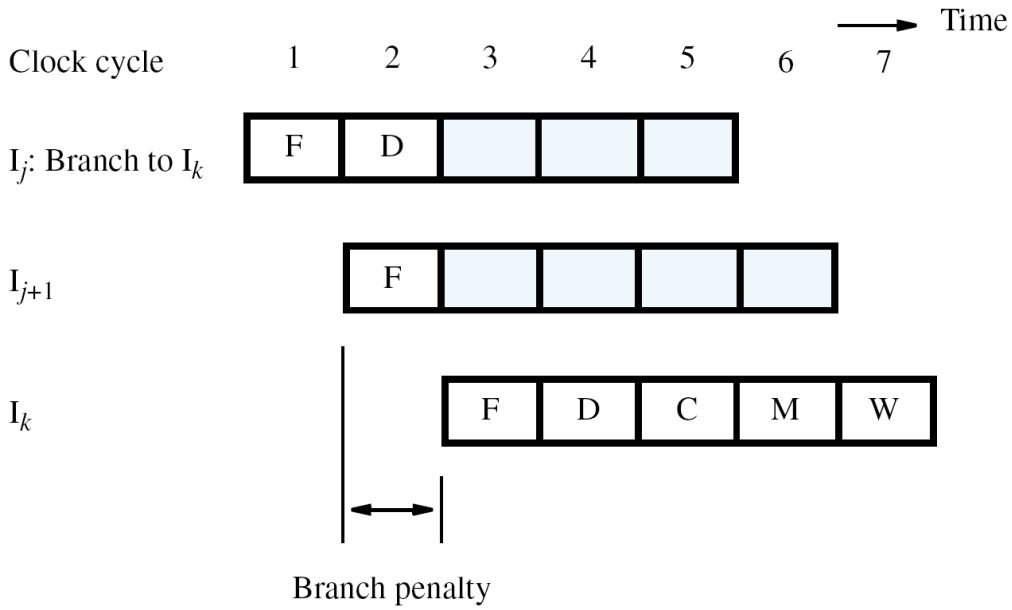

Time
Clock cycle
1
2
3
4
5
6
7
8
I_j: Branch to I_k
F
D
C
I_{j+1}
F
D
I_{j+2}
F
I_k
F
D
C
M
W
Branch penalty

Reducing the Branch Penalty

- Require the branch target address to be computed earlier in the pipeline.
- Determine the target address and update the PC in the Decode stage, rather than the Compute stage.
- So introduce a second adder in the Decode stage just for branches.

Reducing the Branch Penalty

- For previous example, now only I_{j+1} is fetched.
- Only one instruction needs to be discarded.
- The branch penalty is reduced to one cycle.



Conditional Branches

- Consider a conditional branch instruction:
  Branch_if_[R5]=[R6] LOOP
- Requires not only target address calculation, but also requires comparison for condition.
- In Chapter 5, ALU performed the comparison.
- Target address now calculated in Decode stage.
- To maintain one-cycle penalty, we introduce a comparator just for branches in Decode stage.

The Branch Delay Slot

- Let both branch decision and target address be determined in Decode stage of pipeline.
- Instruction immediately following a branch is always fetched, regardless of branch decision.
  - This instruction is discarded with penalty, except when conditional branch is not taken.
- The location immediately following the branch is called the *branch delay slot.*

The Branch Delay Slot

- Instead of conditionally discarding instruction in delay slot, *always* let it complete execution.
- Let compiler find an instruction *before* branch to move into slot, if data dependencies permit.
- Called *delayed branching* due to reordering.
- If useful instruction put in slot, penalty is *zero.*
- If not possible, insert explicit NOP in delay slot for one-cycle penalty, whether or not taken.

	Add	R7, R8, R9
	Branch_if_[R3]=0	TARGET
	I_{j+1}	
	⋮	
TARGET:	I_k	

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

	Branch_if_[R3]=0	TARGET
	Add	R7, R8, R9
	I_{j+1}	
	⋮	
TARGET:	I_k	

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

Quiz (1)

1. A(n) ____ is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason.

   or

   _____ is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline.

   A. control hazard B. data hazard

   C. structural hazard D. instruction hazard

Quiz (2)

2. About the data dependency in a pipeline, which is ***not*** true in the following?

   A. Operand forwarding can handle all data dependencies without the penalty of stalling the pipeline.

   B. Compiler can detect data dependencies and deal with them by analyzing instructions.

   C. Compiler puts *explicit* NOP instructions between instructions having a dependency.

   D. Even with a cache hit and operand forwarding, the data dependency between “Load R2, (R3)” and “Subtract R9, R2, #30” will also cause one-cycle stall.

Quiz (3)

3. Consider the following sequence of instructions being processed on the pipelined 5-stage RISC processor:

Load R1, 4(R2)

Sub R4, R1, R5

And R6, R1, R7

Or R8, R3, R4

Quiz (4)

(1) Identify all the data dependencies in the above instruction sequence. For each dependency, indicate the two instructions and the register that causes the dependency.

There are three data dependencies in this instruction sequence:

- Sub instruction depends on Load instruction for register R1
- And instruction depends on Load instruction for register R1
- Or instruction depends on Sub instruction for register R4

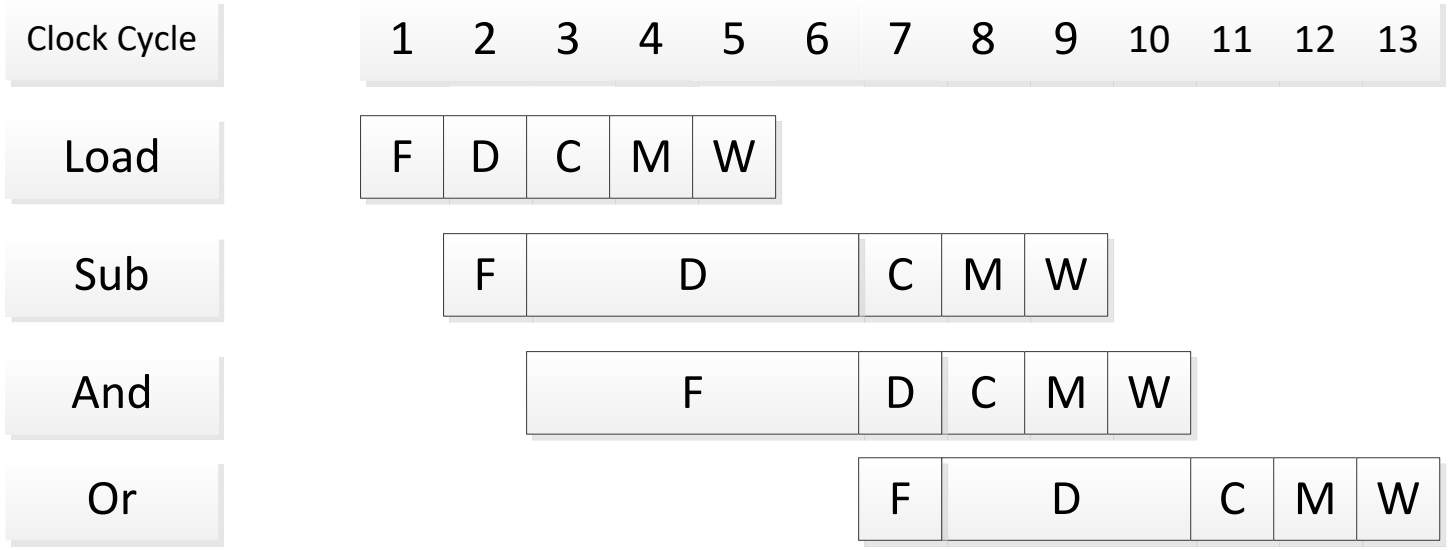
Load R1, 4(R2)
Sub R4, R1, R5
And R6, R1, R7
Or R8, R3, R4

Load R1, 4(R2)
Sub R4, R1, R5
And R6, R1, R7
Or R8, R3, R4

Quiz (5)

(2) Assume that the pipeline does not use operand forwarding. Also assume that the only sources of pipeline stalls are the data hazards. Draw a diagram that represents instruction flow through the pipeline during each clock cycle.

The following diagram shows the instruction flow through the pipeline.



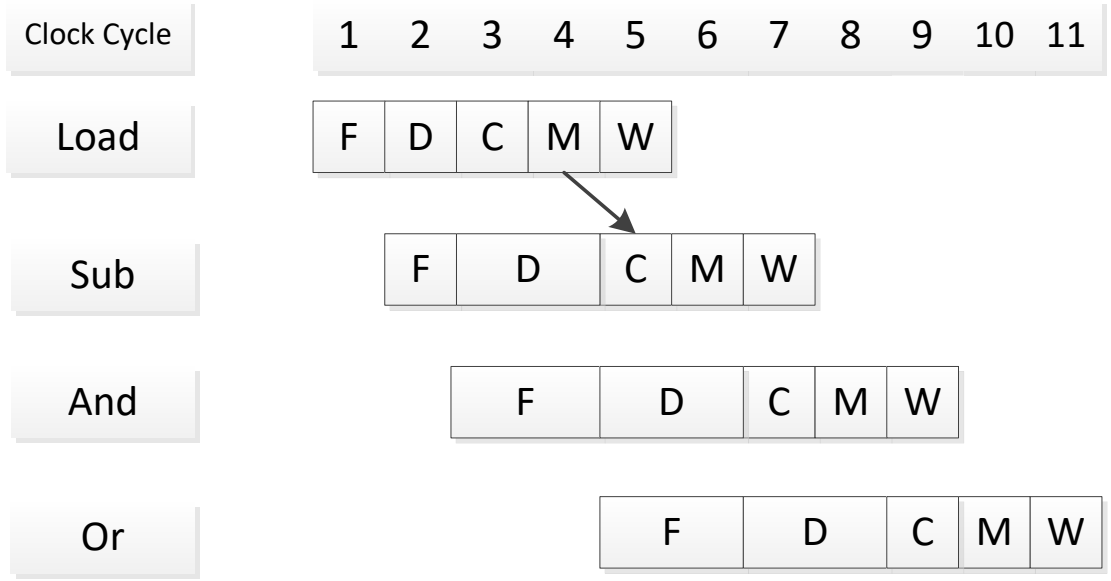
Quiz (6)

(3) Assume that the pipeline uses operand forwarding. The pipeline hardware is similar to figure 2 above, but add separate forwarding paths from the outputs of stage-3 and stage-4 to the input of stage-3. Draw a diagram that represents the flow of instructions through the pipeline during each clock cycle. Indicate operand forwarding by arrows.

Load R1, 4(R2)
Sub R4, R1, R5
And R6, R1, R7
Or R8, R3, R4

Quiz (7)

The following diagram shows the instruction flow through the pipeline in the presence of operand forwarding:



Homework

- 6.2 (see Example 6.1)