Chapter 6
Pipelining

陈俊颖

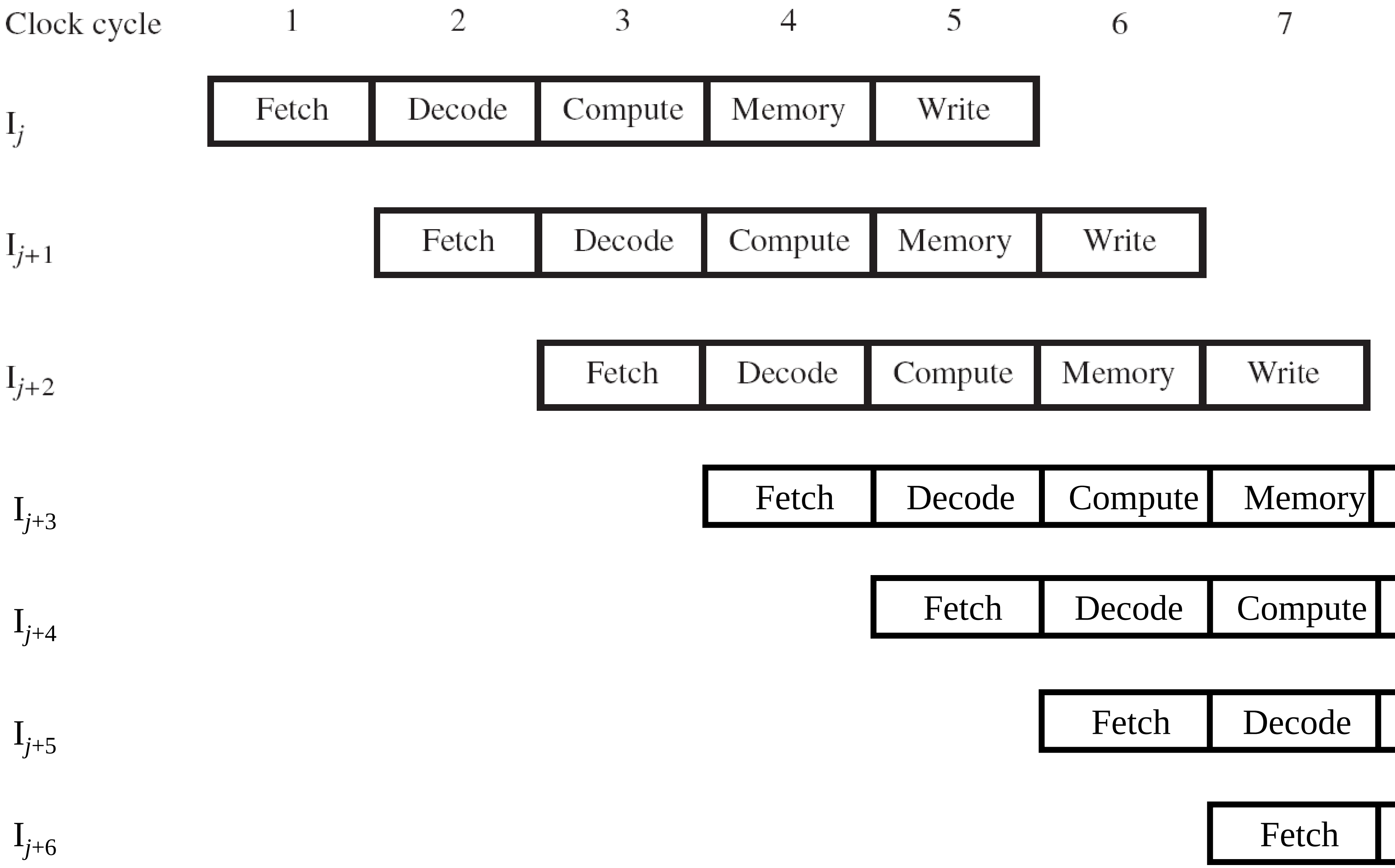
Contents

6.1 Basic Concept of Pipelining

- Circuit technology and hardware arrangement influence the speed of execution for programs.
- All computer units benefit from faster circuits.
- Pipelining involves arranging the hardware to *perform multiple operations simultaneously.*
- Similar to assembly line where product moves through stations that perform specific tasks.
- Same total time for each item, but *overlapped.*

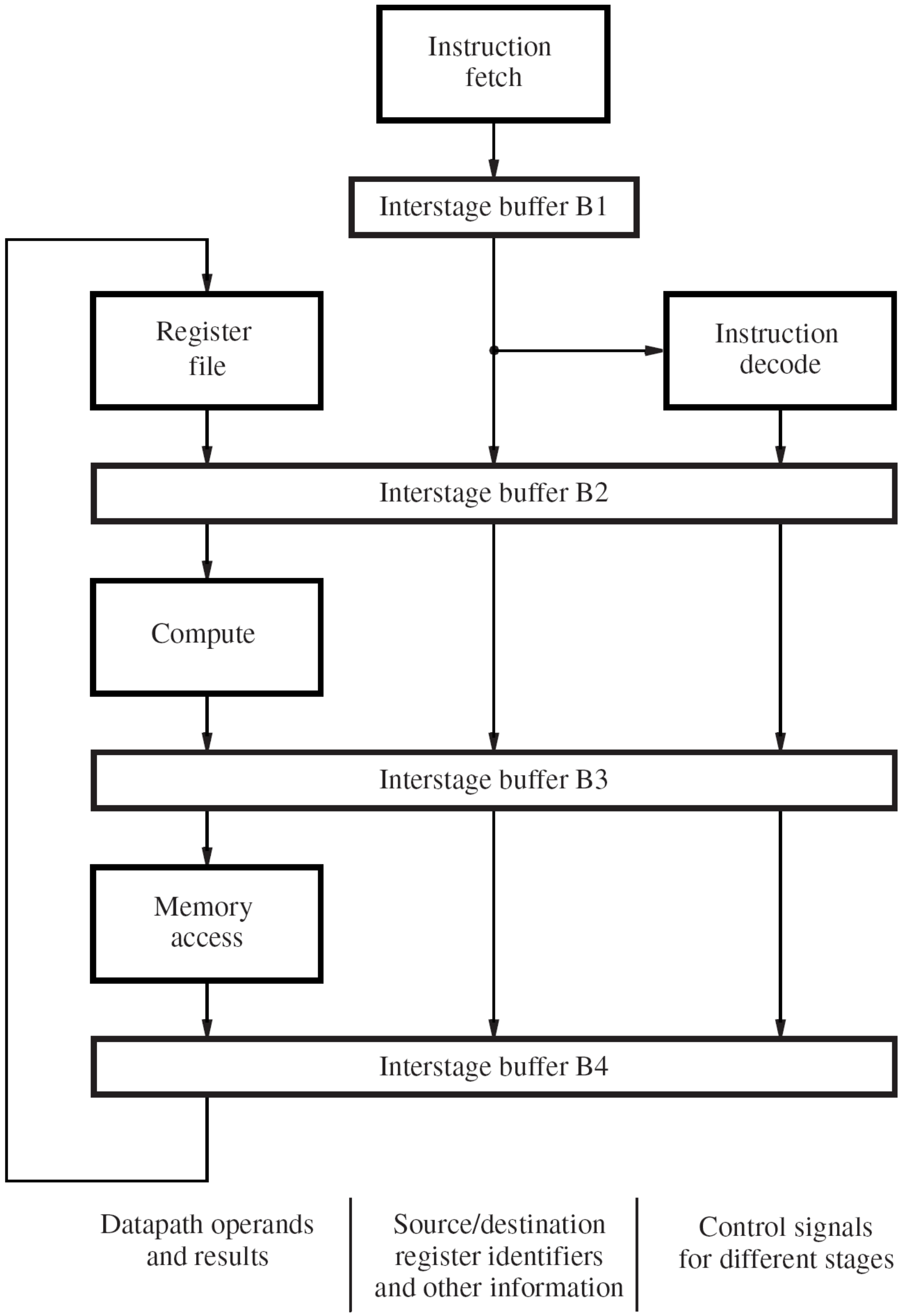
Pipelining in a Computer

- Focus on pipelining of *instruction execution.*
- Multistage datapath in Chapter 5 consists of: Fetch, Decode, Compute, Memory, Write.
- Instructions fetched & executed one at a time with only one stage active in any cycle.
- *With pipelining*, multiple stages are active simultaneously for different instructions.
- Still 5 cycles to execute one instruction, but *rate* is 1 per cycle.


Clock cycle
1
2
3
4
5
6
7
I_j
Fetch
Decode
Compute
Memory
Write
I_{j+1}
Fetch
Decode
Compute
Memory
Write
I_{j+2}
Fetch
Decode
Compute
Memory
Write
I_{j+3}
Fetch
Decode
Compute
Memory
I_{j+4}
Fetch
Decode
Compute
I_{j+5}
Fetch
Decode
I_{j+6}
Fetch

6.2 Pipeline Organization

- Use program counter (PC) to fetch instructions
- A new instruction enters pipeline every cycle
- Carry along instruction-specific information as instructions flow through the different stages
- Use *interstage buffers* to hold this information
- These buffers incorporate RA, RB, RM, RY, RZ, IR, and PC-Temp registers from Chapter 5
- The buffers also hold control signal settings


Instruction fetch
Interstage buffer B1
Register file
Instruction decode
Interstage buffer B2
Compute
Interstage buffer B3
Memory access
Interstage buffer B4
Datapath operands and results
Source/destination register identifiers and other information
Control signals for different stages

6.3 Pipelining Issues

- It is not possible to have a new instruction enter the pipeline in every cycle.
- Consider two successive instructions I_j and I_{j+1}
  - Assume that the destination register of I_j matches one of the source registers of I_{j+1}.
  - Result of I_j is written to destination in cycle 5.
  - But I_{j+1} reads *old* value of register in cycle 3.
  - Due to pipelining, I_{j+1} computation is incorrect.
  - So *stall* (delay) I_{j+1} until I_j writes the new value.
  - Condition requiring this stall is a *data hazard.*

Pipelining Issues

- Any condition that causes the pipeline to stall is called a *hazard*.
- Besides data hazard (6.4), other hazards arise from
  - Memory delays (6.5)
  - Branch instructions (6.6)
  - Resource limitations (6.7)

6.4 Data Dependencies

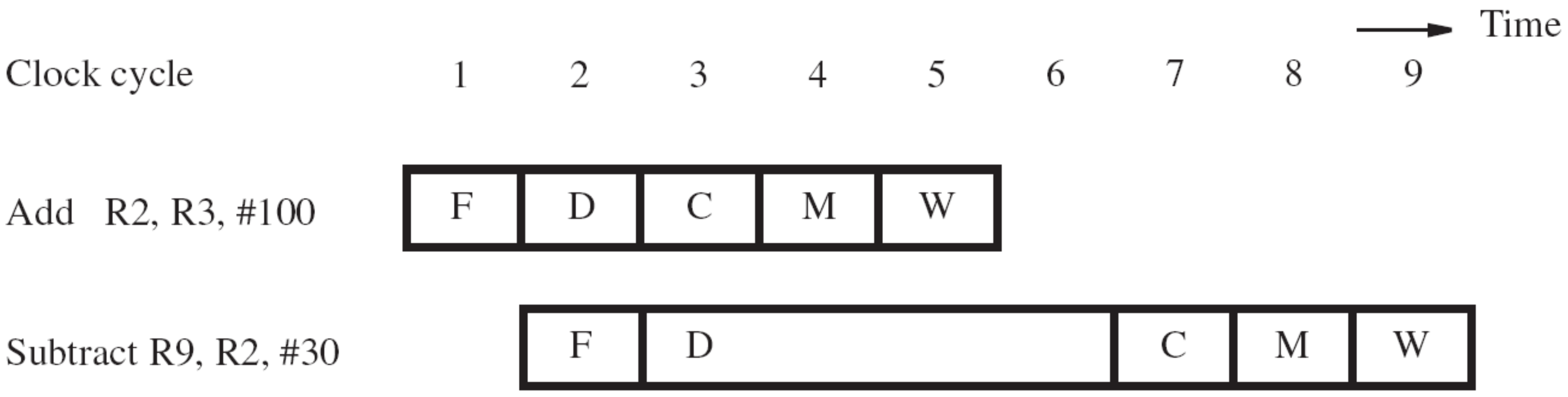
- Now consider the specific instructions

```
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

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



Details for Stalling the Pipeline

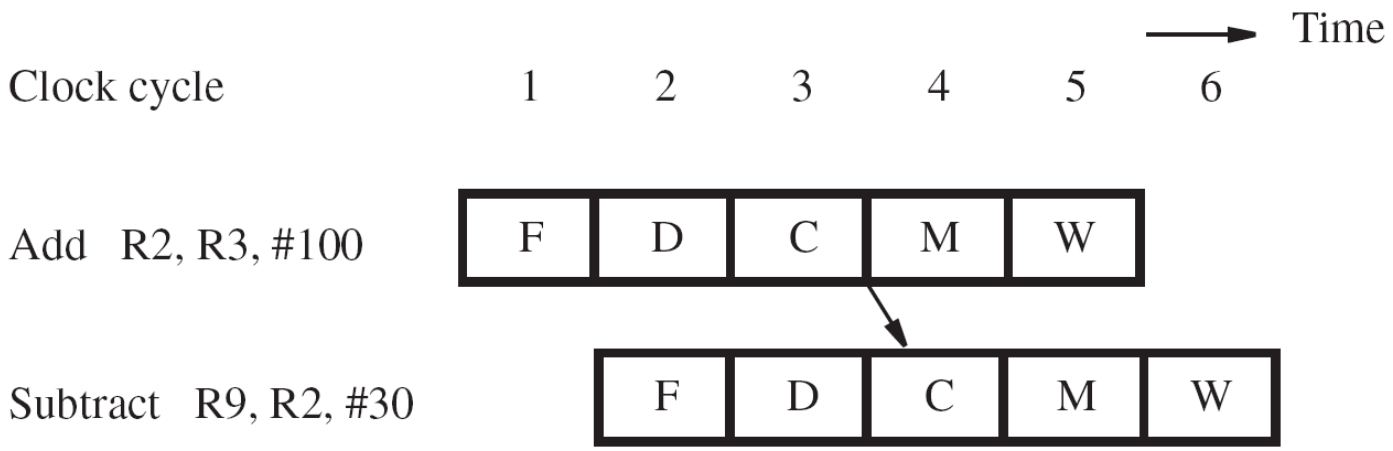
- Control circuitry must recognize dependency while Subtract is being decoded in cycle 3.
- Interstage buffers carry register identifiers for source(s) and destination of instructions.
- In cycle 3, compare destination identifier in Compute stage against source(s) in Decode.
- R2 matches, so Subtract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

Details for Stalling the Pipeline

- Stall the Subtract instruction for 3 cycles by keeping contents of interstage buffer B1.
- What happens after Add leaves Compute?
  - Control signals are set in cycles 3 to 5 to create an *implicit* NOP (No-operation) in Compute.
  - NOP control signals in interstage buffer B2 create a cycle of idle time in each later stage.
  - The idle time from each NOP is called a *bubble*.

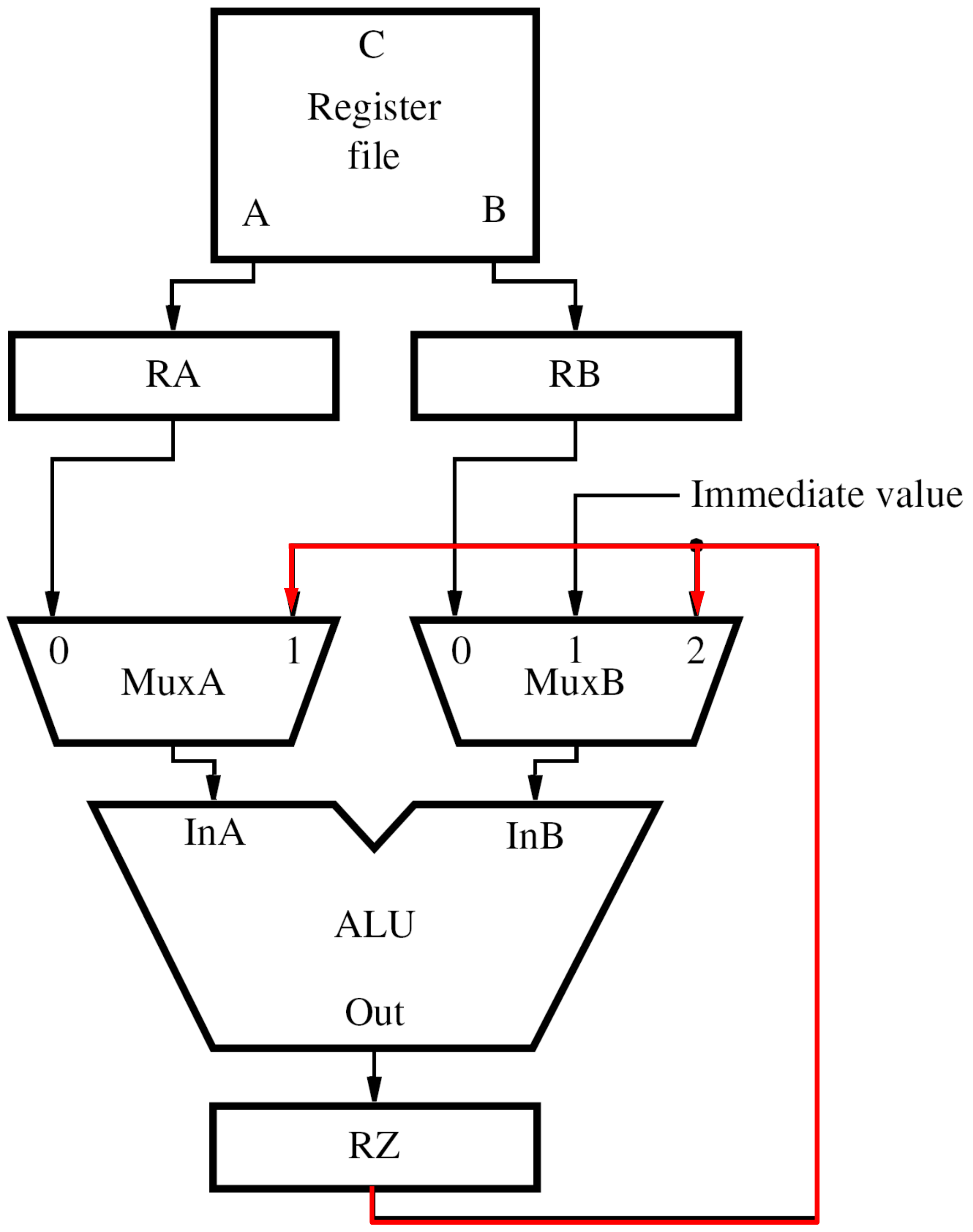
Operand Forwarding

- 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.



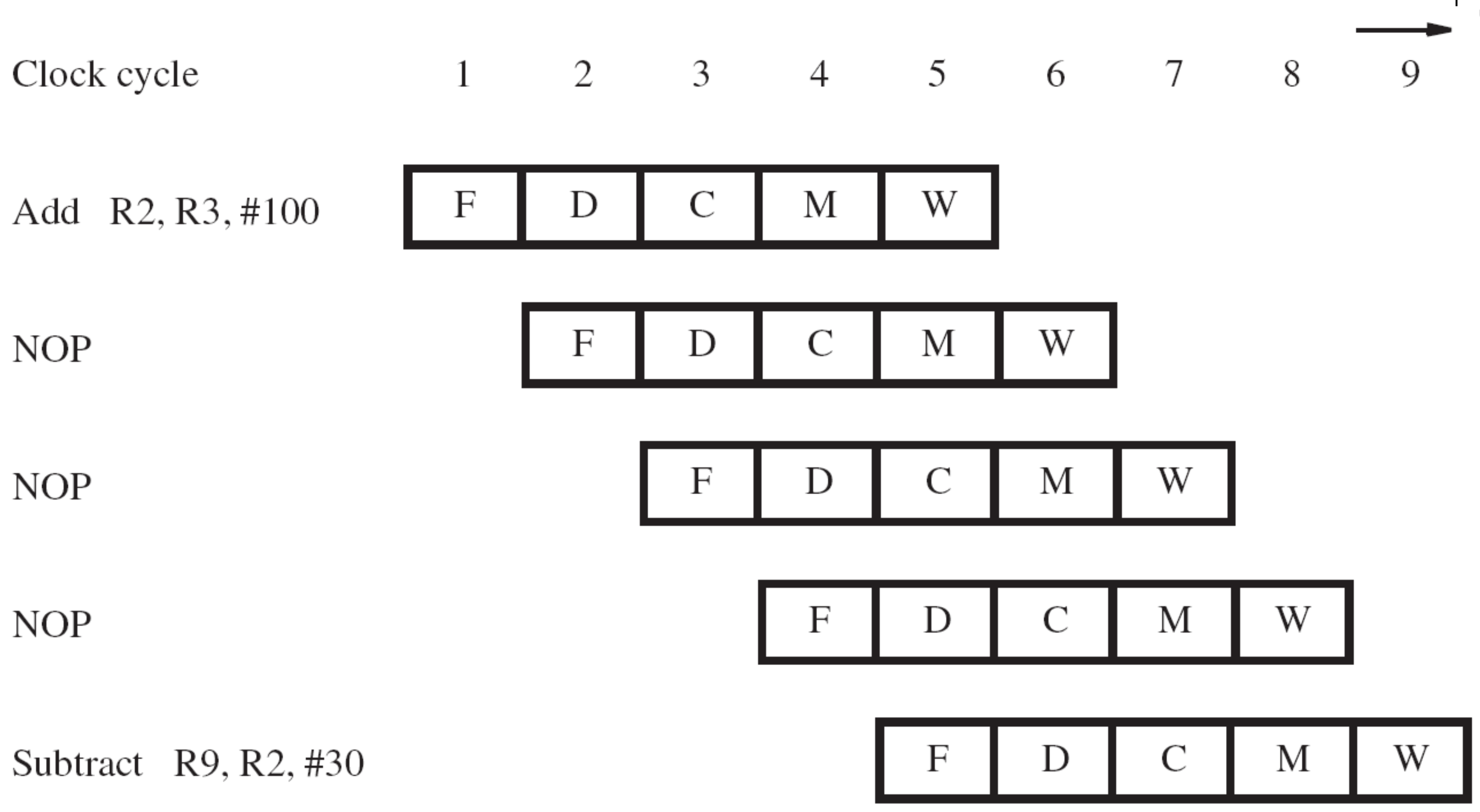
Details for Operand Forwarding

- 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.


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

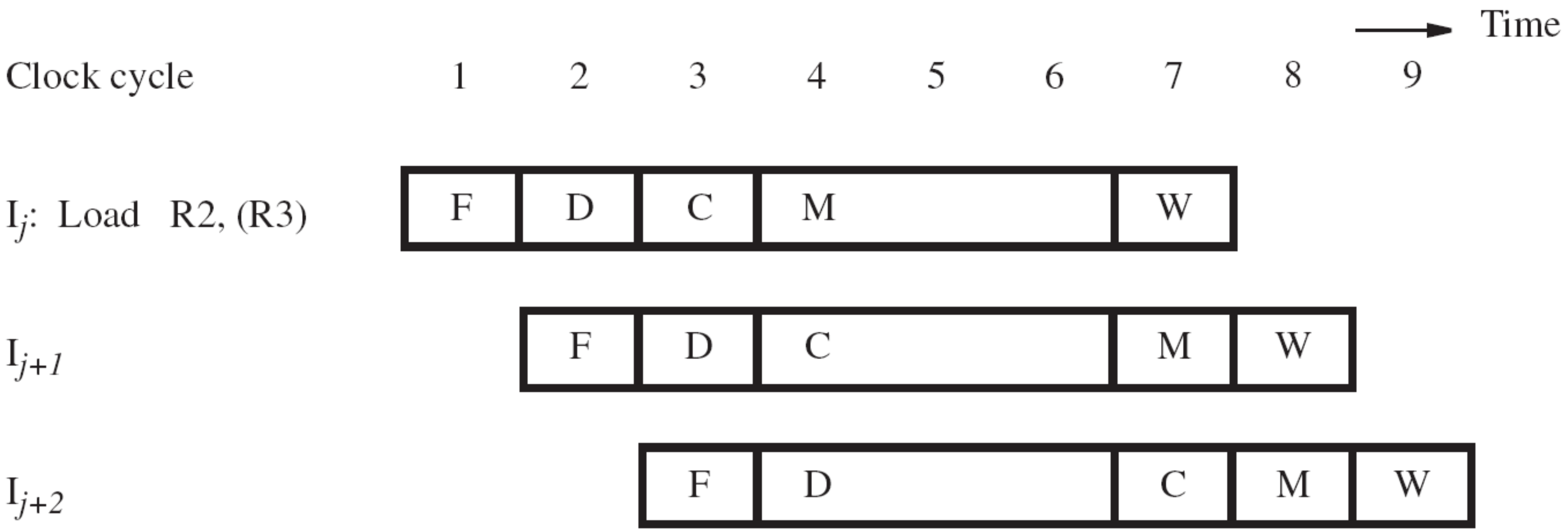
Software Handling of Dependencies

- Compiler can detect data dependencies and deal with them by analyzing 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.
  - Simplifies the hardware implementation of the pipeline, but increases the code size.
- Compiler can *optimize* by moving instructions into NOP slots (if data dependencies permit).


Time
Clock cycle
1
2
3
4
5
6
7
8
9
Add R2, R3, #100
F D C M W
NOP
F D C M W
NOP
F D C M W
NOP
F D C M W
Subtract R9, R2, #30
F D C M W

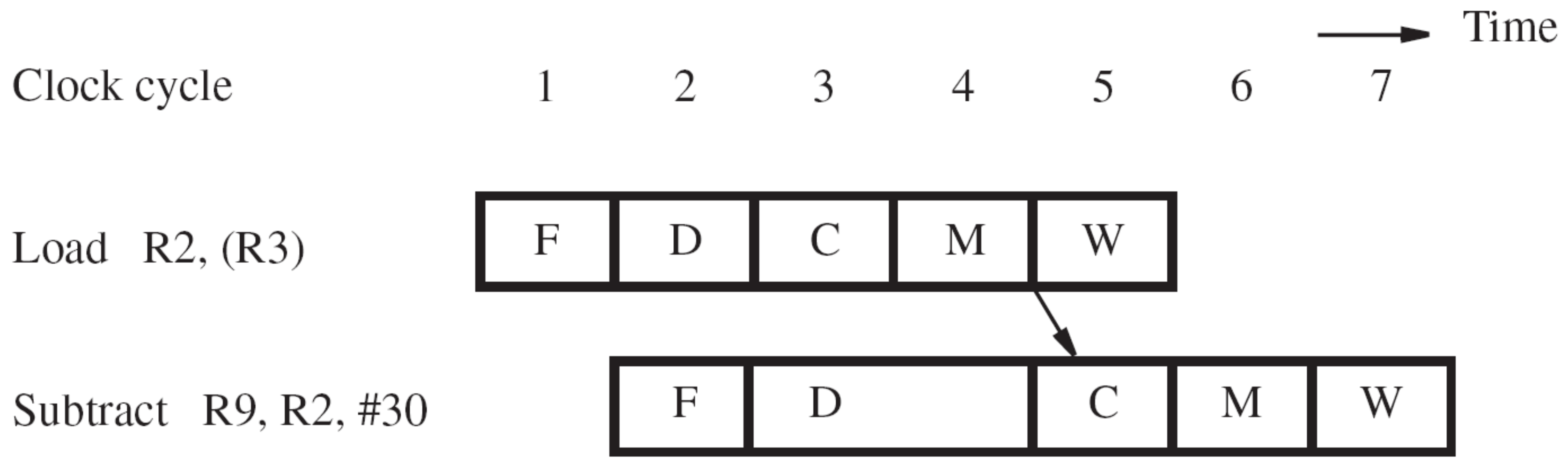
6.5 Memory Delays

- 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.


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

- 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.

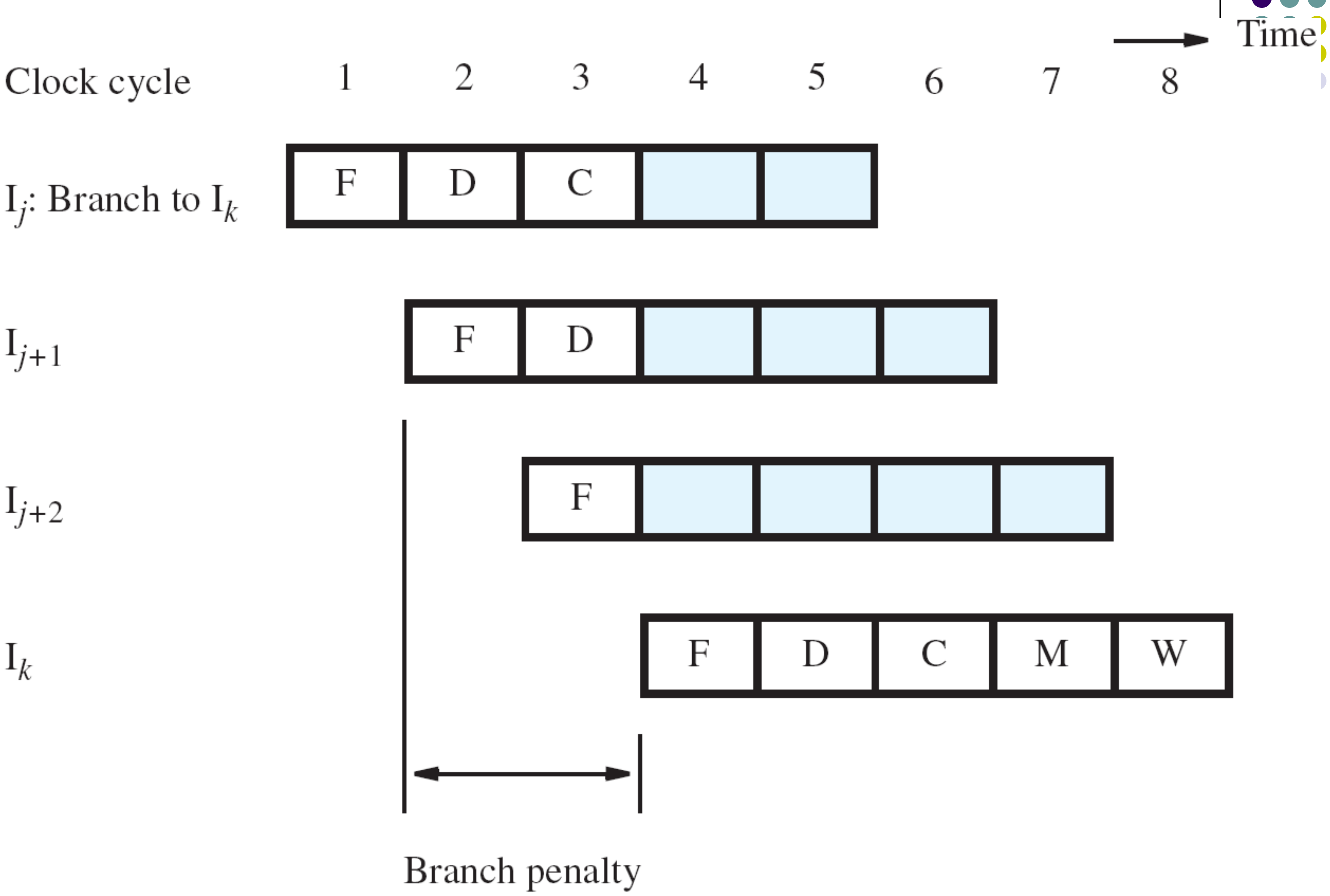


6.6 Branch Delays

- 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.
- Techniques to mitigate this effect are desired.
- Understand branch behavior to find solutions.

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*.

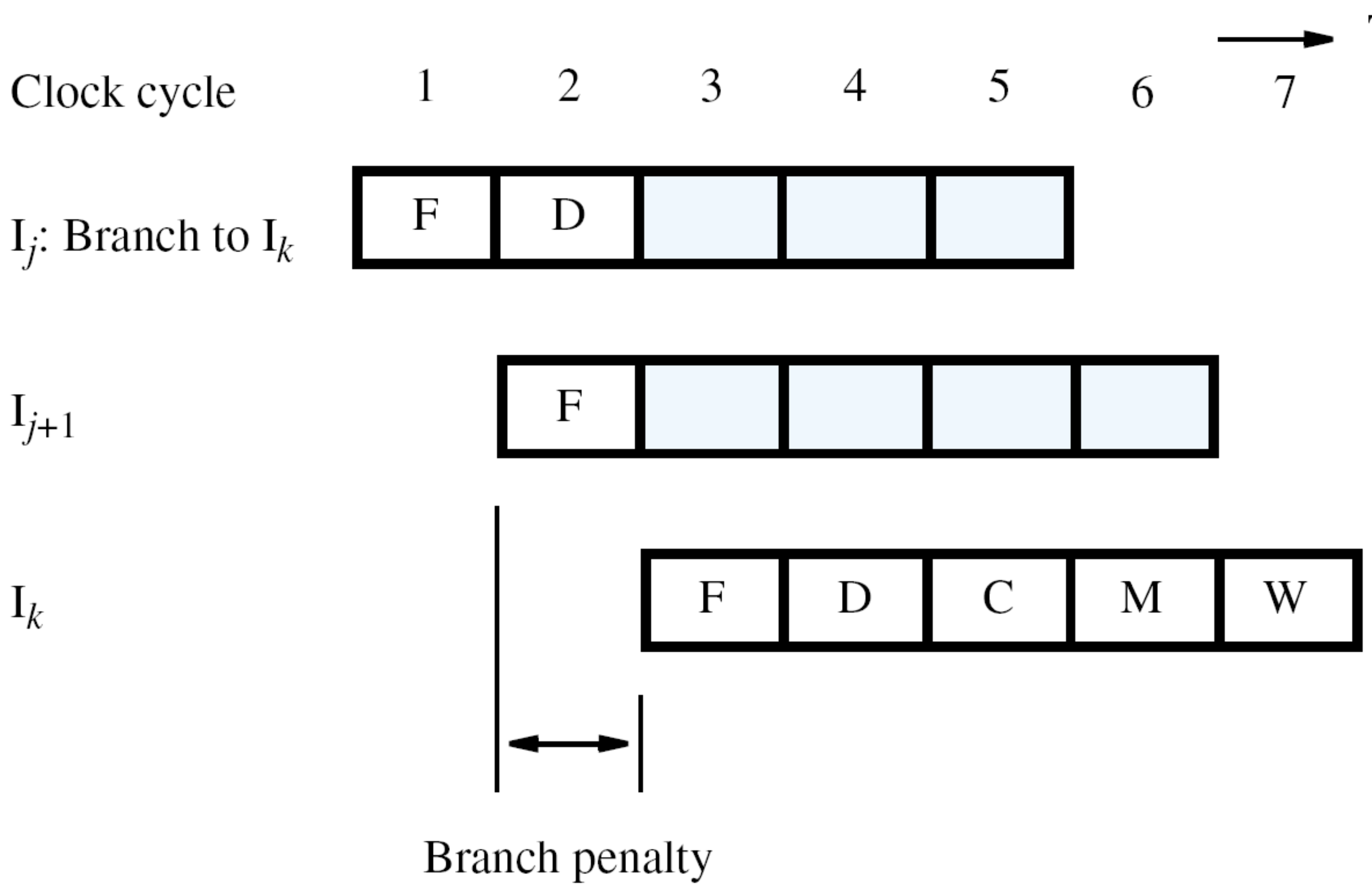

Time
Clock cycle
1
2
3
4
5
6
7
8
I_j: Branch to I_k
F
D
C
I_{j+1}
F
D
I_{j+2}
F
I_k
F
D
C
M
W
Branch penalty

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.


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

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

Branch Prediction

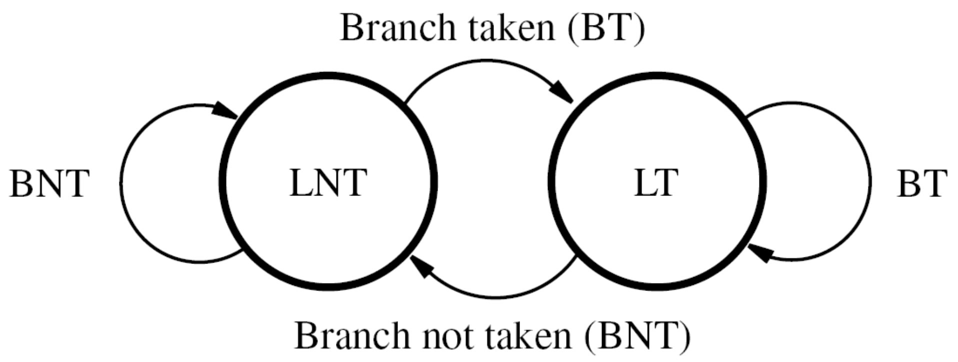
- A branch is decided in Decode stage (cycle <u>2</u>) while following instruction is *always* fetched.
- Following instruction may require discarding (or with delayed branching, it may be a NOP).
- Instead of discarding the *following* instruction, can we anticipate the *actual* next instruction?
- Two aims:(a) *predict* the branch decision
  (b) use prediction *earlier* in cycle <u>1</u>

Static Branch Prediction

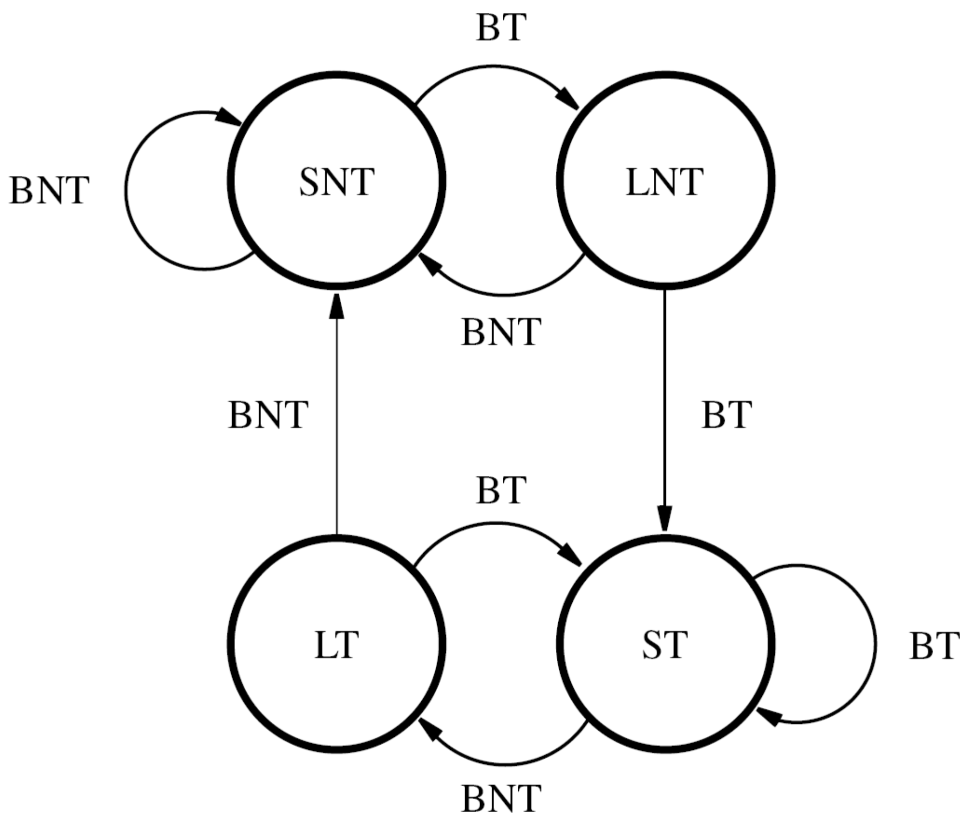
- Simplest approach: assume branch *not* taken.
- Penalty if prediction disproved during Decode.
- If branches are random, accuracy is 50%.
- But a branch at end of a loop is usually taken.
  - So for backward branch, always predict *taken.*
  - Use target address as soon as it is available.
  - Expect higher accuracy for this special case, but what about accuracy for other branches?

Dynamic Branch Prediction

- To improve prediction accuracy further, use actual branch behavior to prediction.
  - Idea: track branch decisions during execution for *dynamic* prediction.
- Simplest approach: use most recent outcome for likely taken (LT) or likely not-taken (LNT)
- For branch at end of loop, we mispredict in last pass, and in first pass if loop is *re-entered.*
- Avoid misprediction for loop re-entry with four states (ST, LT, LNT, SNT) for strongly/likely.



(a) A 2-state algorithm



(b) A 4-state algorithm

Branch Target Buffer

- Prediction only provides a presumed *decision*
- Decode stage computes *target* in cycle 2.
- But we need target (and prediction) in cycle 1.
- *Branch target buffer* stores more information about the history from *last* execution of each branch:
  - The address of the branch instruction
  - One or two state bits for the branch prediction algorithm
  - The branch target address

Branch Target Buffer

- The branch target buffer identifies the branch instructions by their addresses.
- In cycle 1, use branch instruction address to look up the buffer for target address and use history for prediction.
- Fetch in cycle 2 using prediction; if mispredict detected during Decode, correct it in cycle 3.

6.7 Resource Limitations

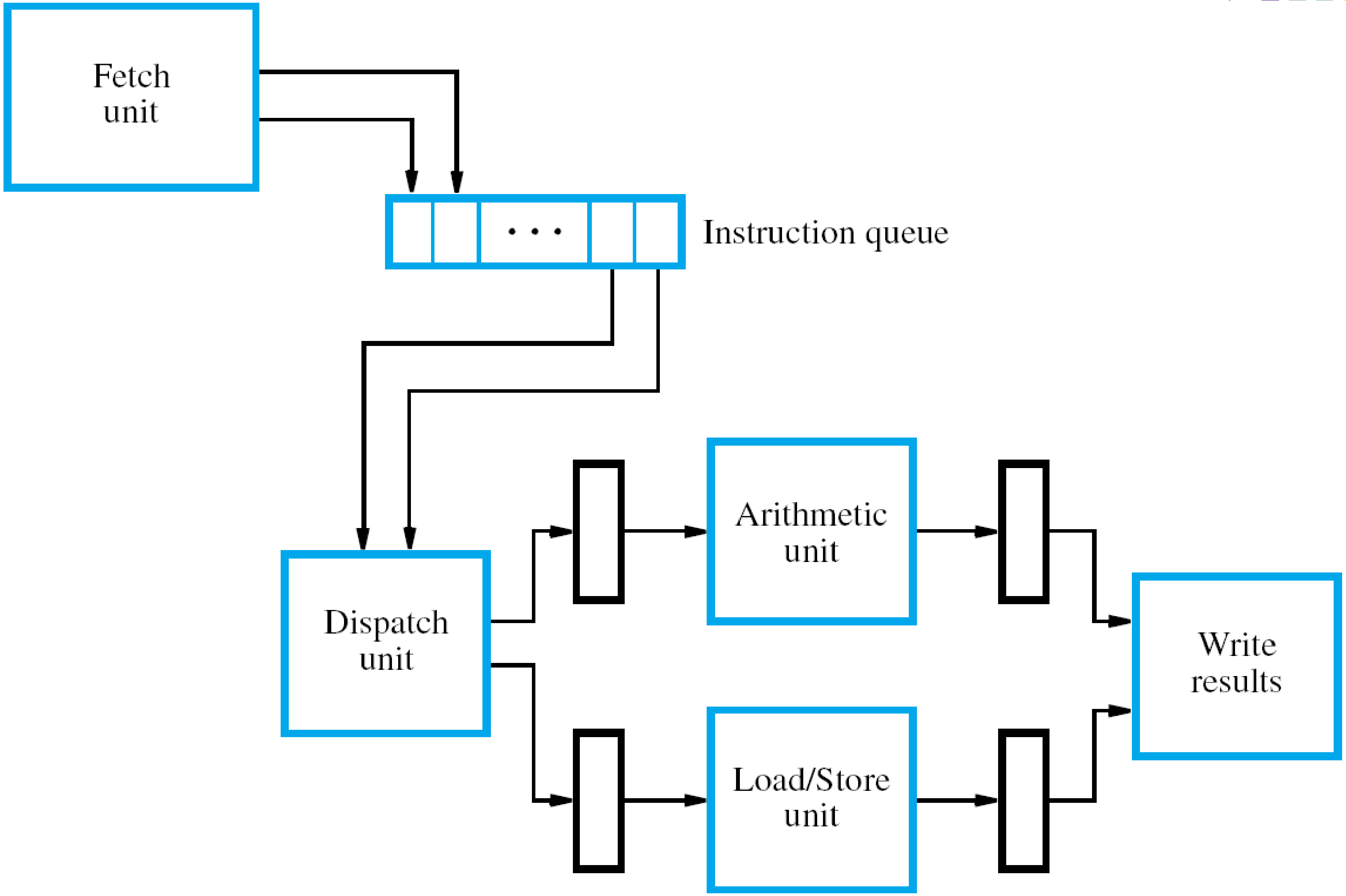
- Two instructions may need to access the same resource in the same clock cycle.
- One instruction must be stalled to allow the other instruction to use the resource.
- This can be prevented by providing additional hardware.

Resource Limitations

- 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.

6.9 Superscalar Operation

- Introduce multiple execution units to enable *multiple instruction issue* for > 1 instr./cycle.
- This organization is for a superscalar processor.
- An elaborate *fetch unit* brings 2+ instructions into an instruction queue in every cycle.
- A *dispatch unit* takes 2+ instructions from the head of queue in every cycle, decodes them, sends them to appropriate *execution units*.
- A *completion unit* writes results to registers.


Fetch unit
Instruction queue
Dispatch unit
Arithmetic unit
Load/Store unit
Write results

Superscalar Operation

- Minimum superscalar arrangement consists of a Load/Store unit and an arithmetic unit
  - Because of Index mode address calculation, Load/Store unit has a two-stage pipeline.
  - Arithmetic unit usually has one stage.
- For two execution units, how many operands?
  - Up to 4 inputs, so register file has 4 read ports.
  - Up to 2 results, so also need 2 write ports.
    (and methods to prevent write to same register)

→ Time

Clock cycle	1	2	3	4	5	6
Add R2, R3, #100	F	D	C	W		
Load R5, 16(R6)	F	D	C	M	W	
Subtract R7, R8, R9		F	D	C	W	
Store R10, 24(R11)		F	D	C	M	W

Branches and Data Dependencies

- With no branches or data dependencies, interleave arithmetic & memory instructions to obtain maximum throughput (2 per cycle).
- But branches do occur and must be handled.
- Branches processed entirely by fetch unit to determine which instructions enter queue.
- Fetch unit uses prediction for all branches.
- Necessary because decisions may need values produced by other instructions in progress.

Branches and Data Dependencies

- *Speculative execution*: results of instructions not committed until prediction is confirmed.
- Requires extra hardware to track speculation and to recover in the event of misprediction.

- For data dependencies between instructions, the execution units have *reservation stations*.
- They buffer register identifiers and operands for dispatched instructions awaiting execution.
- Broadcast results for stations to capture & use.

Out-of-Order Execution

- With instructions buffered at execution units, should execution reflect original sequencing?
- If two instructions have no dependencies, there are no actual ordering constraints.
- This enables *out-of-order execution*, but then leads to *imprecise exceptions* in program state.
- For *precise exceptions*, must commit results in strictly in original order with extra hardware.

Execution Completion

- To commit results in original program order, superscalar processors can use 2 techniques:
  - *Register renaming* uses temporary registers to hold new data before it is safe for final update.
  - *Reorder buffer* in commitment unit is where dispatched instructions placed in strict order.
- Update the actual destination register only for instruction at head of queue in reorder buffer.
- Ensures instructions *retired* in original order.

Dispatch Operation

- Dispatch of instruction proceeds only when all needed resources available (temp. register, space in reservation station & reorder buffer).
- If instruction has some but not all resources, should a subsequent instruction proceed?
- Decisions must avoid *deadlock* conditions (two instructions need each other's resources).
- More complex, so easier to use original order, particularly with more than 2 execution units.

Summary

- Pipelining
  - Overlaps activity for 1 instruction per cycle.
- Pipeline issues
  - Data dependencies
  - Memory delays
  - Branch delays
  - Resource limitations
- Superscalar operation
  - Enable multiple instruction issue for more than 1 instruction per cycle.

Homework

- 6.2 (see Example 6.1)
- 6.13