SISTEMI EMBEDDED

Computer Organization
Pipelining

Federico Baronti

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 different tasks are *overlapped* for different items (or instructions in processors)

Pipelining in a Computer

- Focus on pipelining of *instruction execution*
- 5-stage organization consists of:
  Fetch, Decode, Compute, Memory, Write
- We've seen how instructions are 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, but *rate* reaches 1 instruction per cycle
  - Ideally, one instruction is completed at each cycle

"Ideal situation"

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
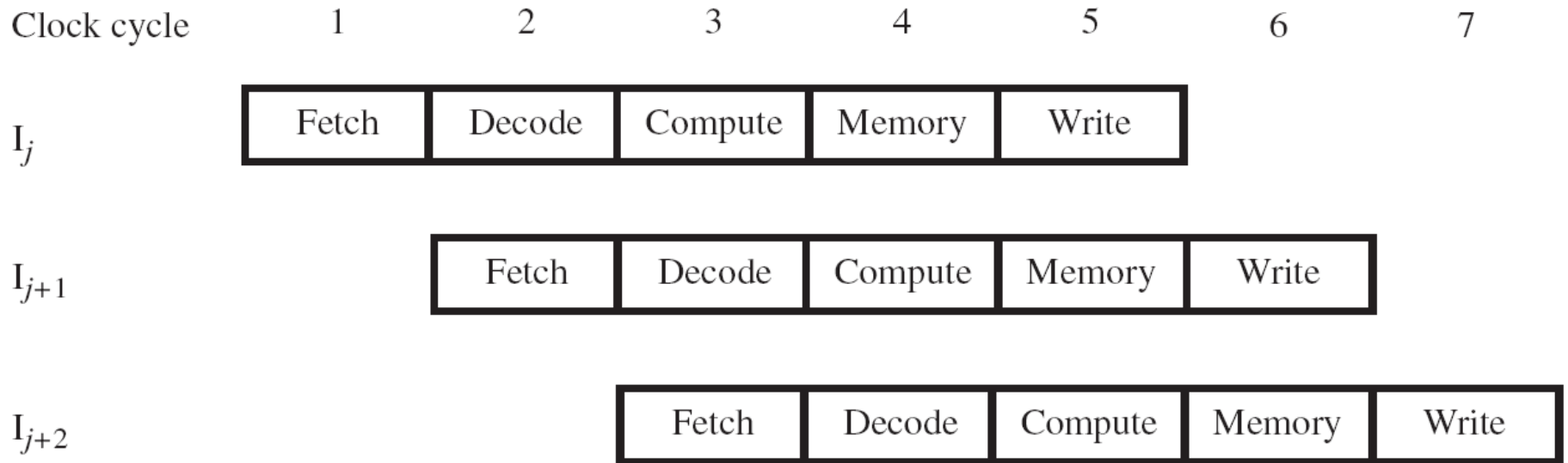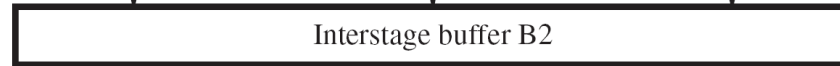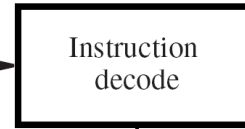
Pipeline Organization

- 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, RZ, RM, RY, IR, and PC-Temp registers
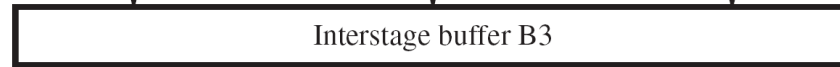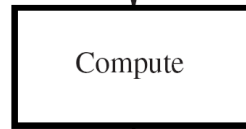 - The buffers also hold control signal settings

1 Fetch

Instruction fetch

↓

Interstage buffer B1

2 Decode

Register file

Instruction decode

↓

Interstage buffer B2

3 Compute

Compute

↓

Interstage buffer B3

4 Memory

Memory access

↓

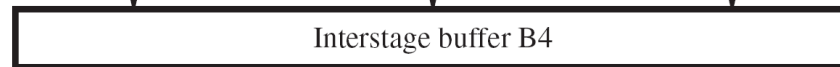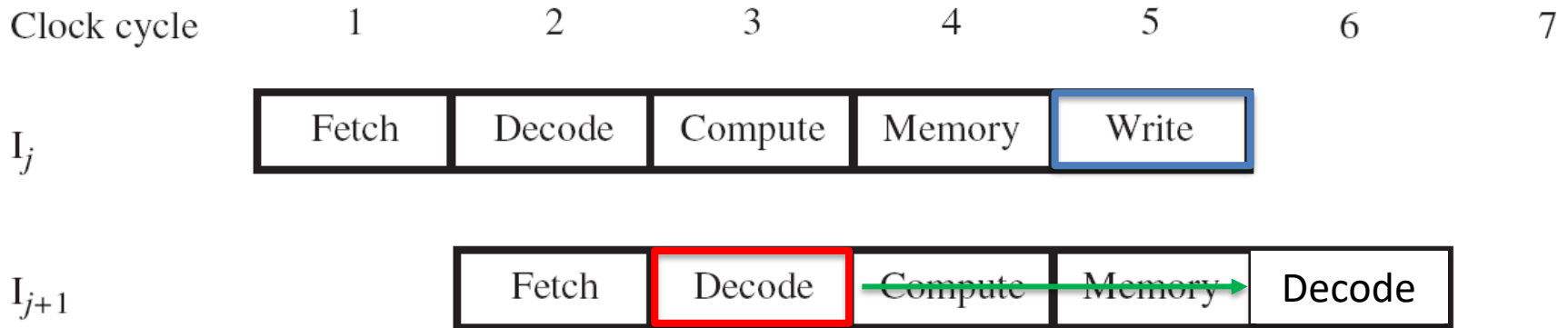Interstage buffer B4

5 Write

Datapath operands and results | Source/destination register identifiers and other information | Control signals for different stages

Pipelining Issues

- 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***

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

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

									Time →
Clock cycle	1	2	3	4	5	6	7	8	9
Add R2, R3, #100	F	D	C	M	W				
Subtract R9, R2, #30		F	D				C	M	W
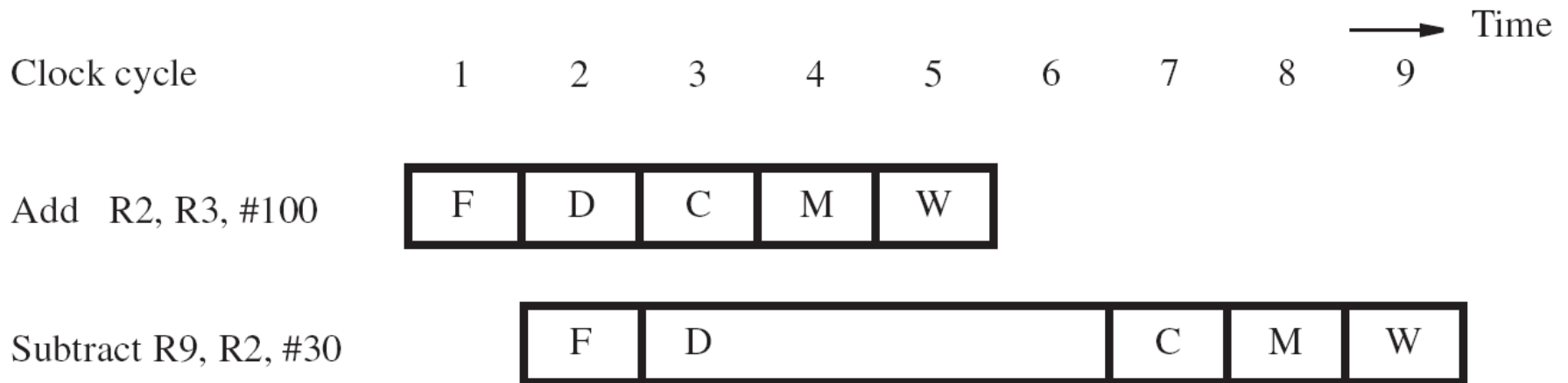
Details for Stalling the Pipeline (1)

- 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 is allowed to continue normally
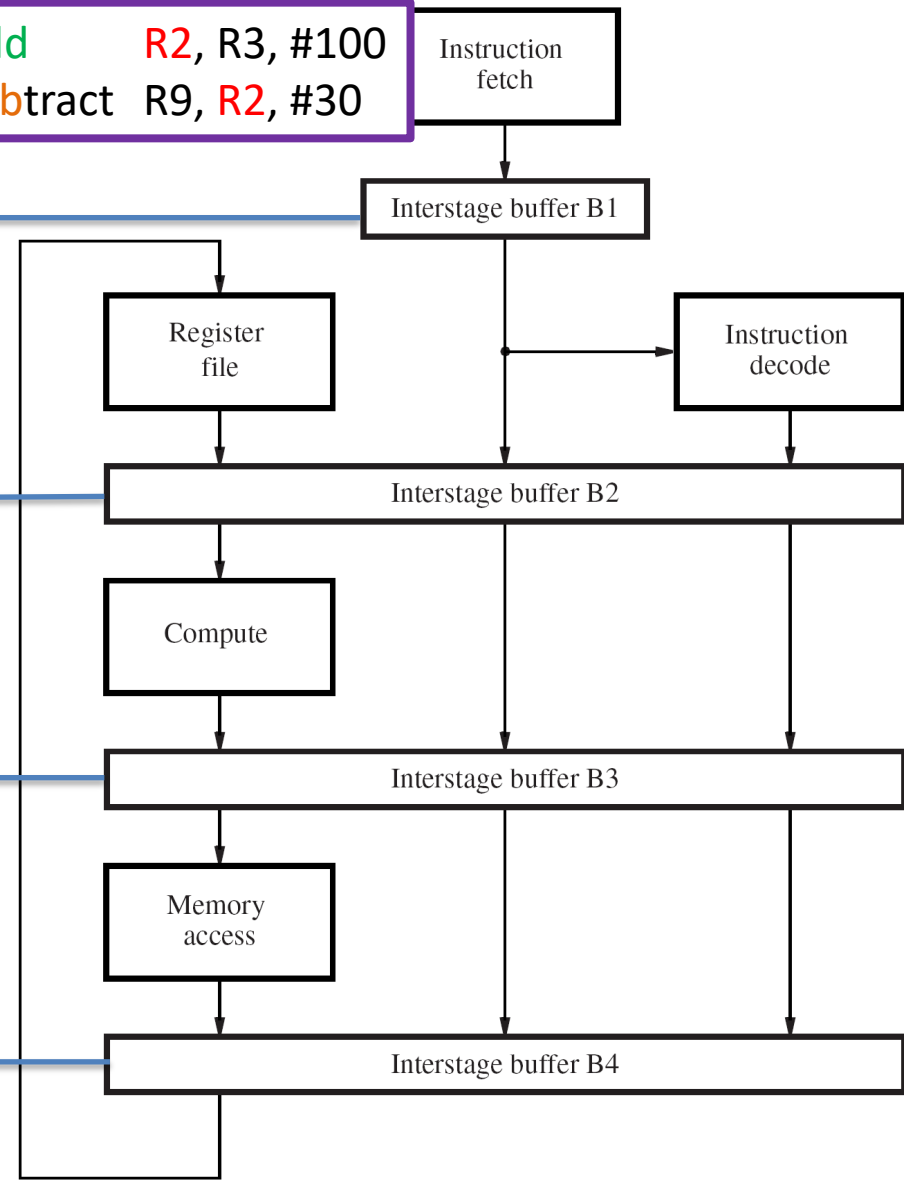
Details for Stalling the Pipeline (2)

- Stall the Subtract instruction for 3 cycles by holding interstage buffer B1 contents steady

- But 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*

clock cycle

1	2	3	4	5	6	7	8	9
	Add	Sub	Sub	Sub	Sub			
		Add	Nop	Nop	Nop	Sub		
			Add	Nop	Nop	Nop	Sub	
				Add	Nop	Nop	Nop	Sub

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

Instruction fetch

Interstage buffer B1

Register file

Instruction decode

Interstage buffer B2

Compute

Interstage buffer B3

Memory access

Interstage buffer B4

No write in the memory

No write in the register
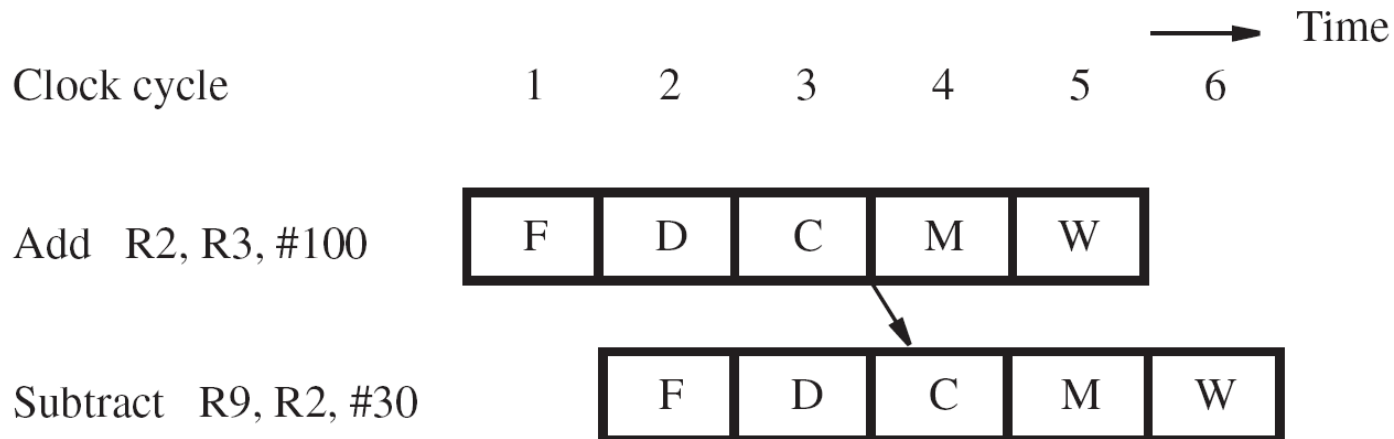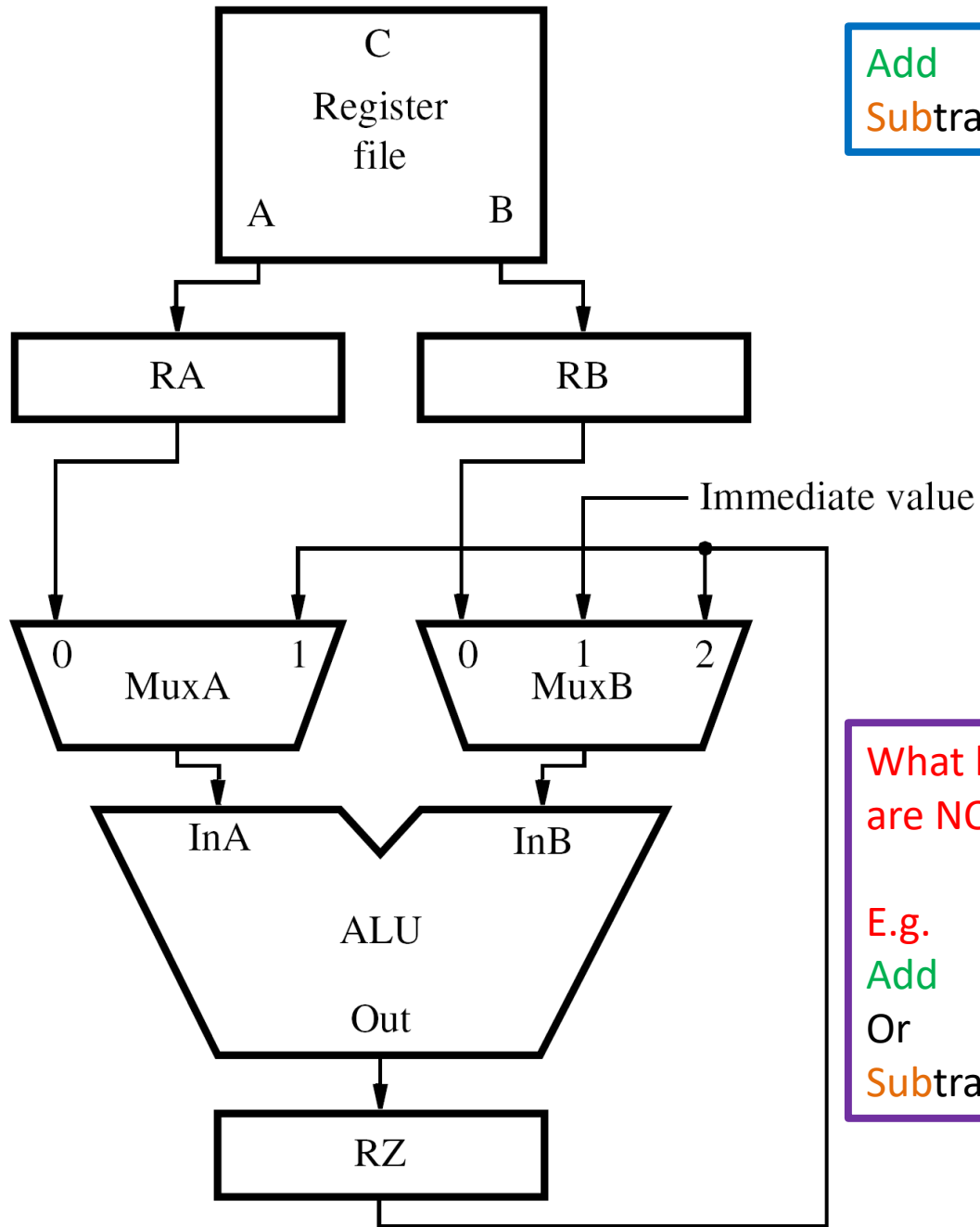
Data hazard is detected

R2 is written

R9 is written

Datapath operands and results

Source/destination register identifiers and other information

Control signals for different stages

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

Time

Clock cycle	1	2	3	4	5	6

Add R2, R3, #100

F	D	C	M	W

Subtract R9, R2, #30

F	D	C	M	W

C
Register file
A   B

RA   RB

Immediate value

0   MuxA   1   0   1   MuxB   2

InA   InB

ALU

Out

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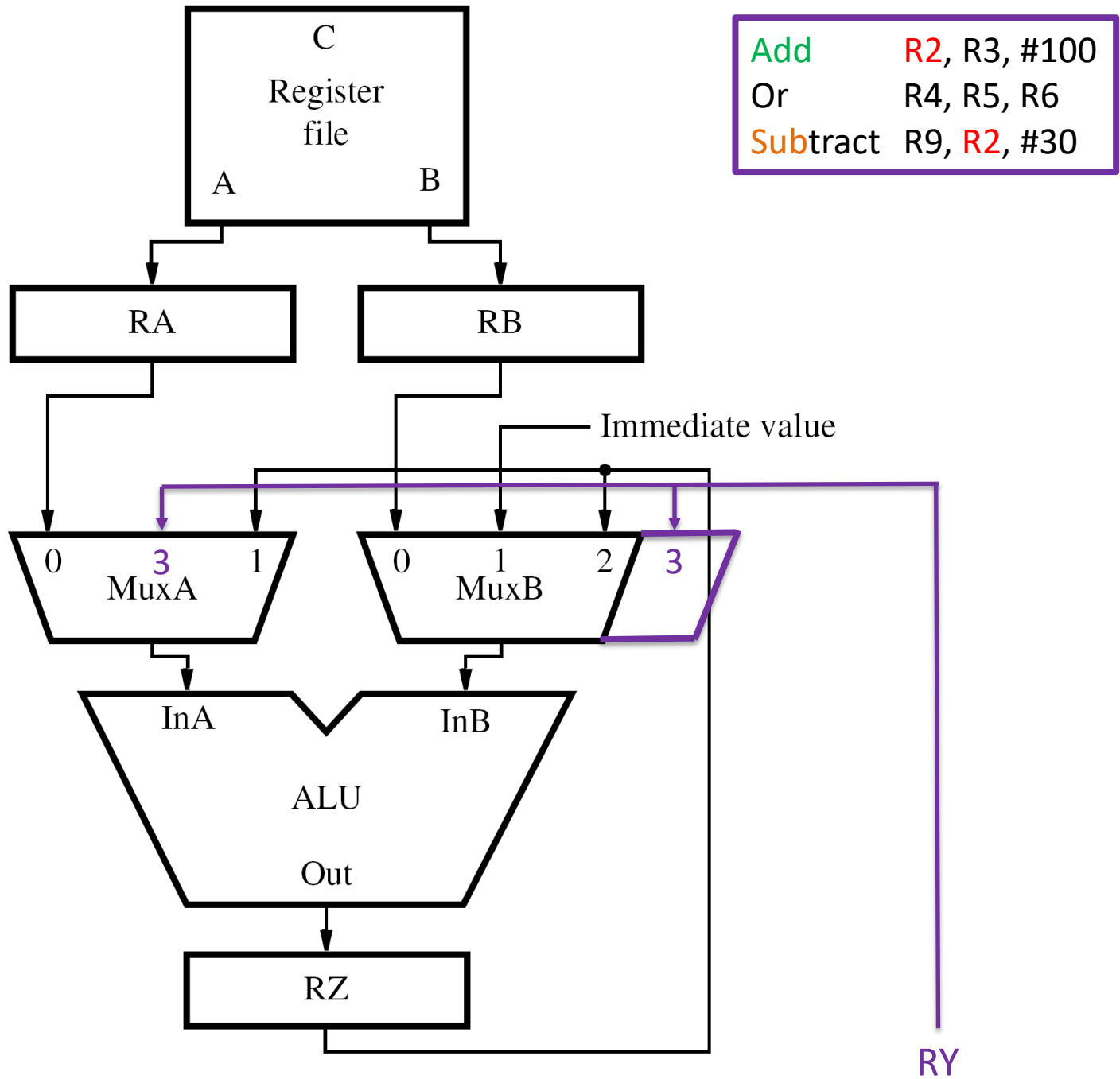
Add   R2, R3, #100
Subtract R9, R2, #30

What happens if instr. are NOT consecutive?

E.g.
Add   R2, R3, #100
Or   R4, R5, R6
Subtract R9, R2, #30

C
Register file

A   B

RA   RB

Immediate value

0   **3**   1   0   1   2   **3**
MuxA   MuxB

InA   InB

ALU

Out

RZ

RY

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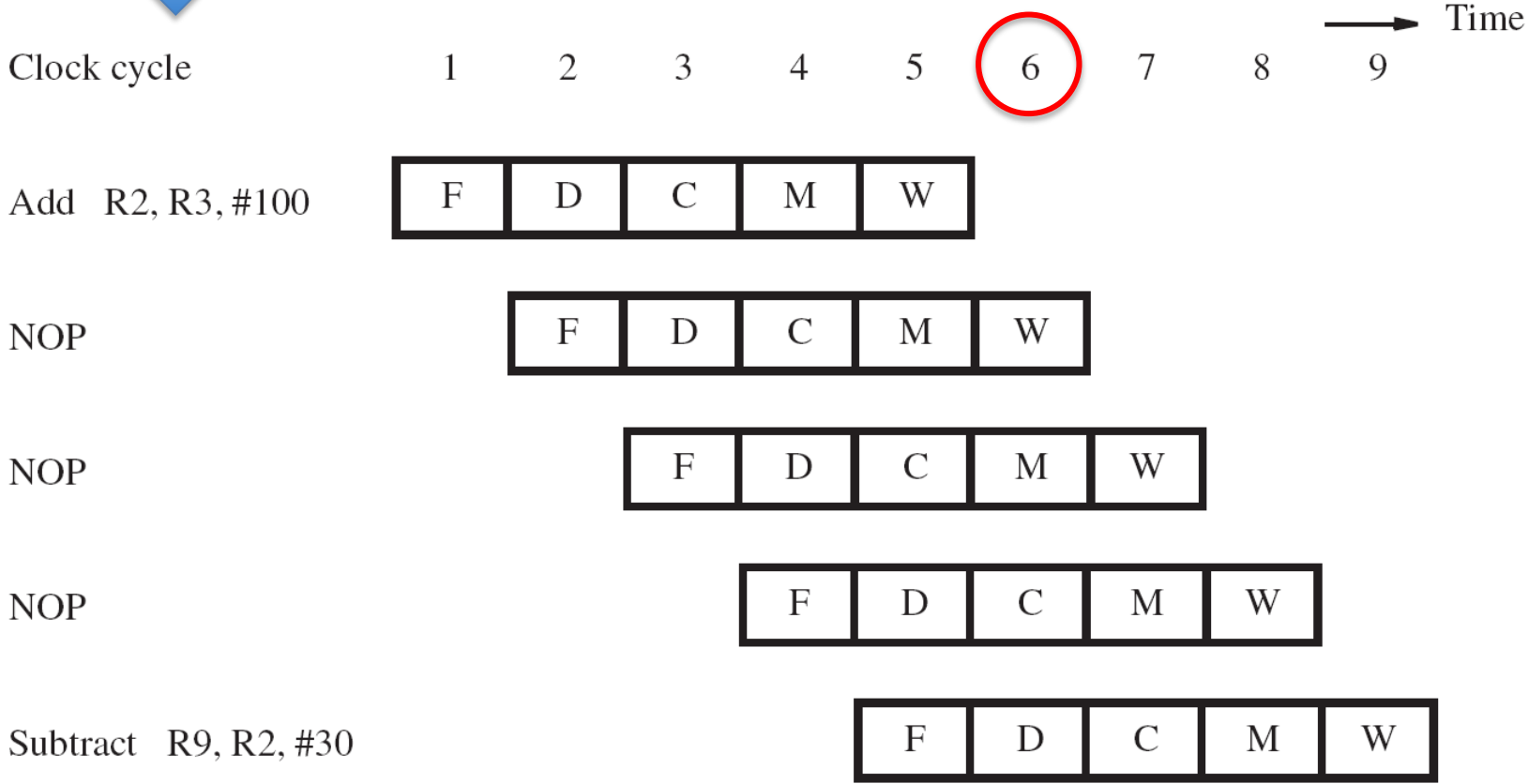Or   R4, R5, R6
Subtract R9, R2, #30

Software Handling of Dependencies

- 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)
  - Leading to a smaller code size and shorter execution time

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

Compiler

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 |

Memory Delays (1)

- Memory delays can also cause pipeline stalls
- A cache memory holds instructions and data from the main memory, but 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
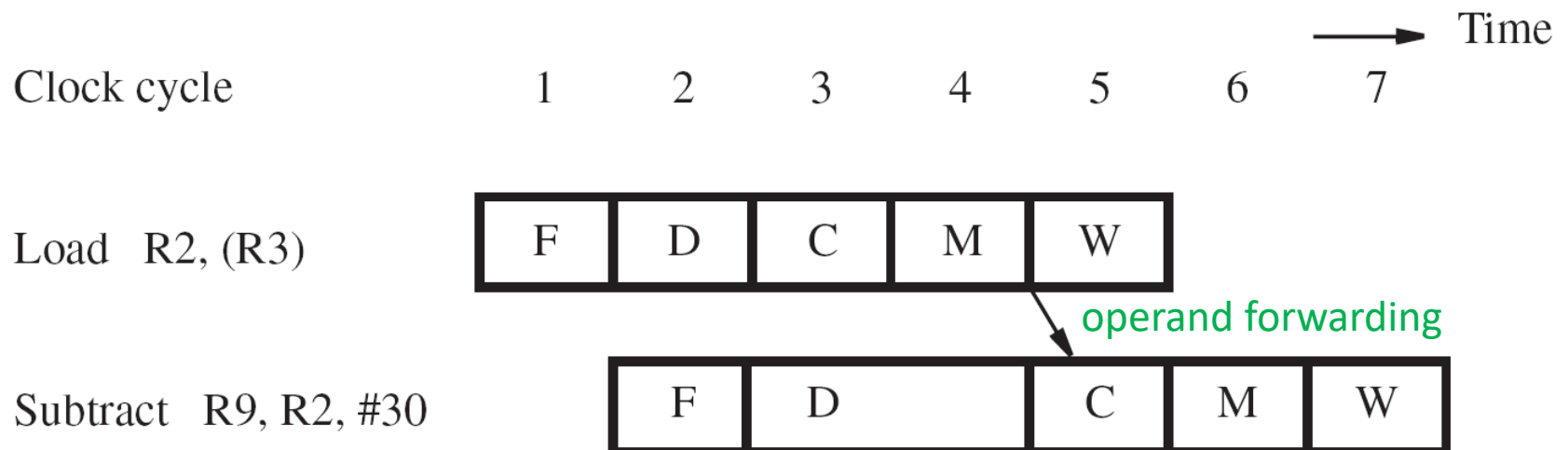
Memory Delays (2)

- Example considering a memory access time of 3 clock cycles
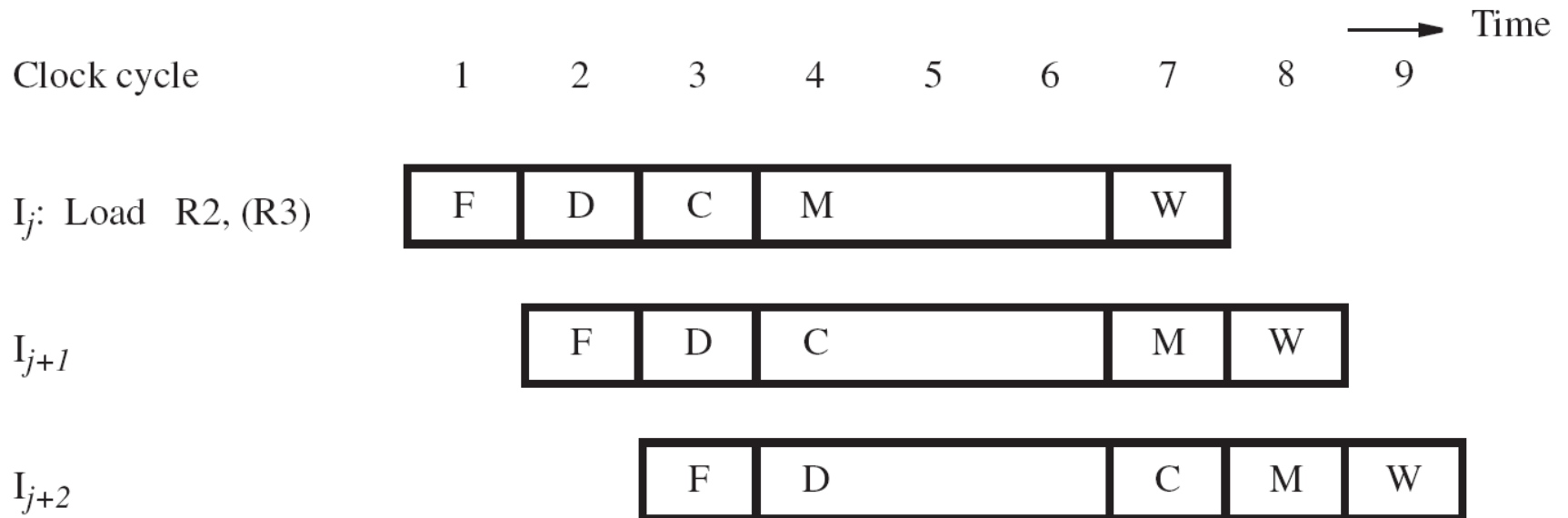
Memory Delays (3)

- 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

Time

Clock cycle	1	2	3	4	5	6	7

Load   R2, (R3) | F | D | C | M | W |

operand forwarding

Subtract   R9, R2, #30 | F | D | | C | M | W |

Branch Delays

- Ideal pipelining: fetch each new instruction while the previous instruction is being decoded

	1	2	3	4	5	6	7
I_j	F	D	C	M	W		
I_{j+1}		F	D	C	M	W	
I_{j+2}			F	D	C	M	W

- Branch instructions may alter execution sequence, but they must be processed to know the effect
- Any delay for determining branch outcome may lead to an increase in total execution time
- Techniques to mitigate this effect are desired
- Understand branch behavior to find solutions

Unconditional Branches (1)

- Consider instructions I_j, I_{j+1}, I_{j+2} in sequence
  - I_j is an unconditional branch with target I_k
- The *target address,* calculated using **offset** (which is available after the Decode stage) and [PC]+4, is known after the Compute stage
- 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 into a two-cycle *penalty*

Unconditional Branches (2)
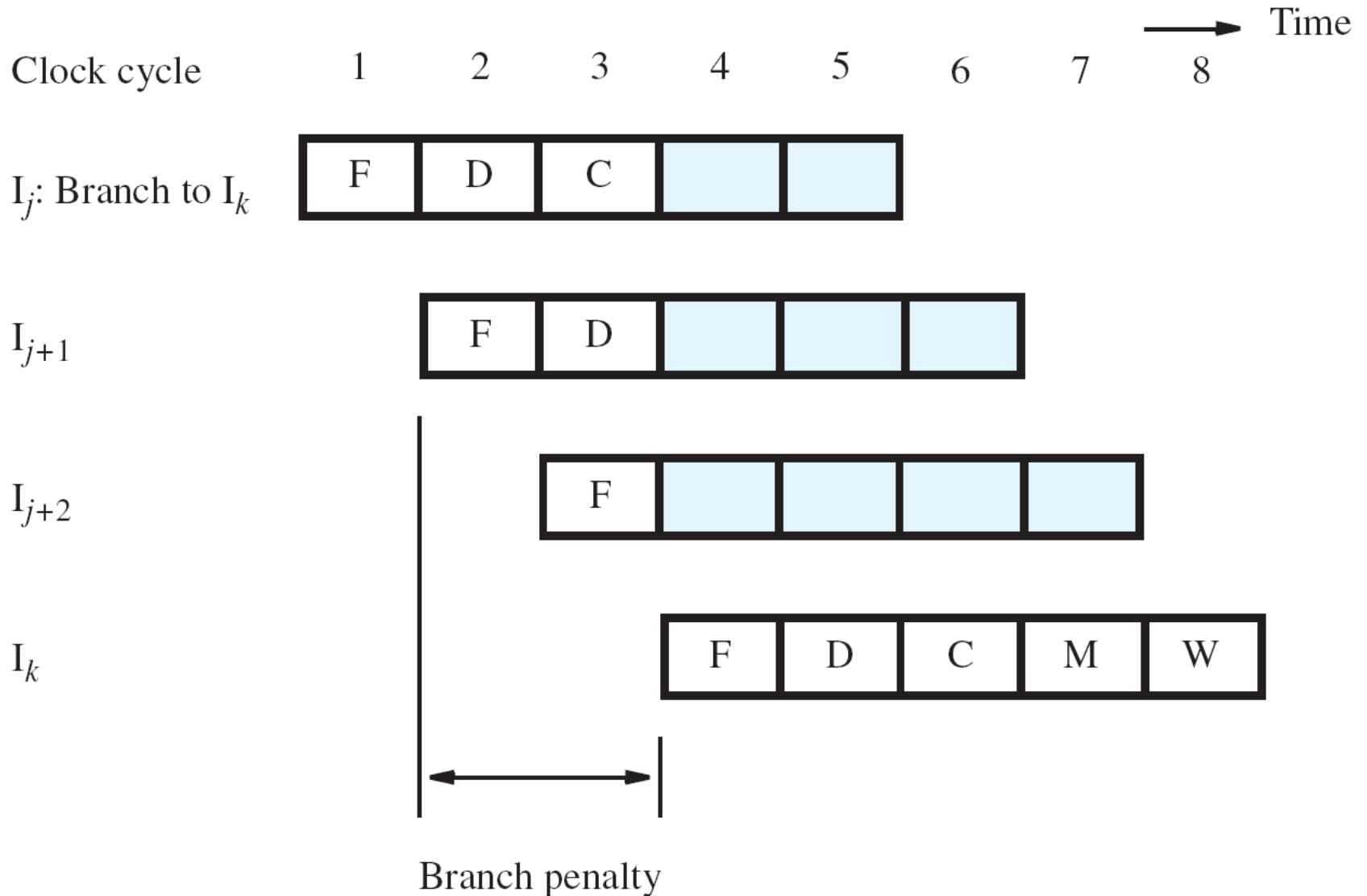
Effect of branch penalty

- Ideally, 1 instr. per *cycle* w/ pipelining
- Branch instr. are executed frequently
  - Roughly p_{BR} = 20 % of the instr. executed by the processor (may be significantly larger than the number of branch instr. in the code, because of loops)
  - A 2-cycle branch penalty increases the average **number of cycles S per instr.** by 40 %
    - $S = (1 - p_{BR}) \times 1 + p_{BR} \times (1 + 2) = 1 + 2 \times p_{BR} = 1.4$
    - $S = 1 + \delta_{BR}, \delta_{BR} = 2 \times p_{BR} = 0.4$
  - Things are a little bit better, as not all the conditional branches will be taken

Reducing the Branch Penalty (1)

- In pipeline, adder for PC in the Instruction Address Generator block is used every cycle, so it cannot calculate the branch target address

Needs to be duplicate for pipeline execution
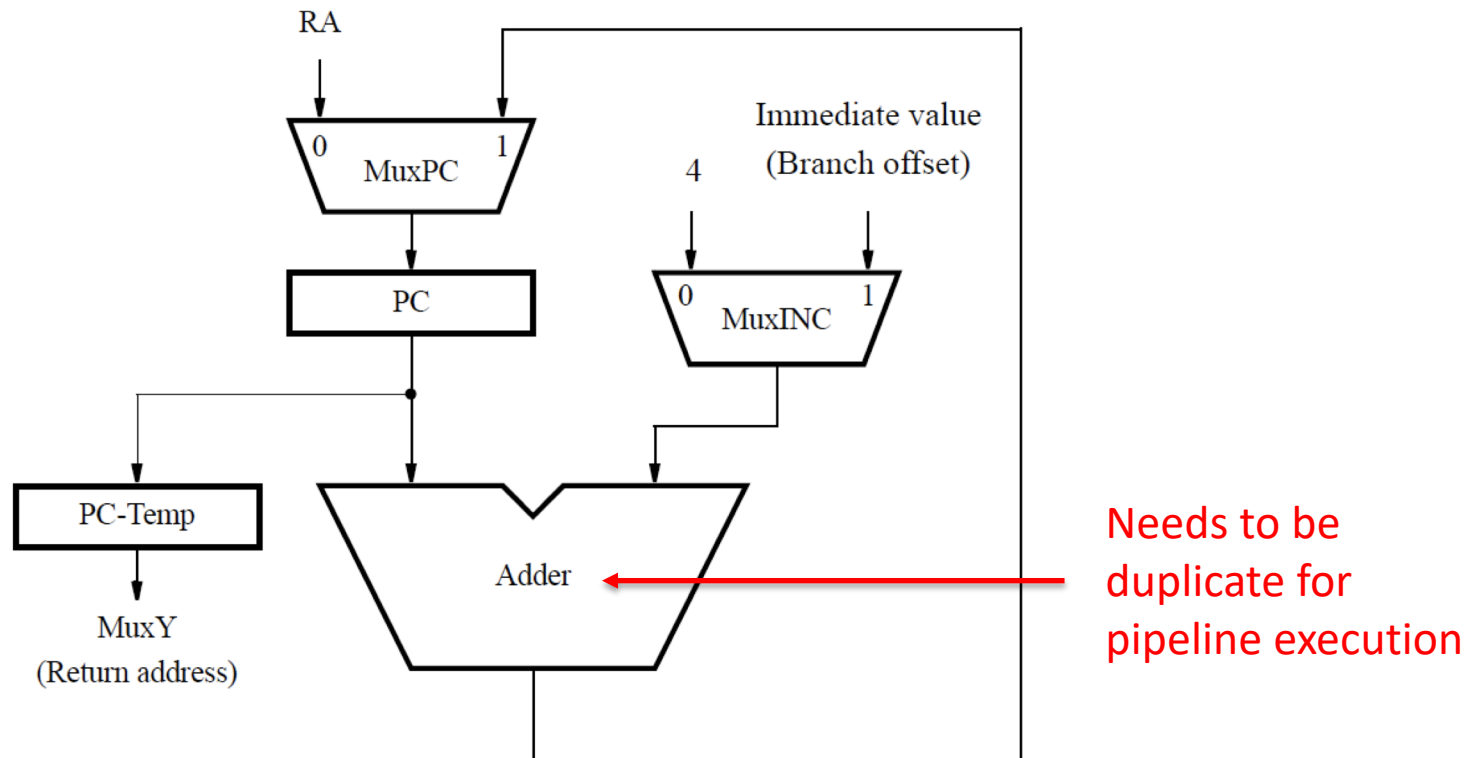
Reducing the Branch Penalty (2)

- In pipeline, adder for PC in the Instruction Address Generator block is used every cycle, so it cannot calculate the branch target address

- So introduce a second adder just for branches and place this second adder in the Decode stage to enable earlier determination of target address

- For previous example, now only I_{j+1} is fetched

- Only one instruction needs to be discarded

- The branch penalty for **UNCONDITIONAL** branches is reduced to one cycle

Reducing the Branch Penalty (3)

Unconditional branch
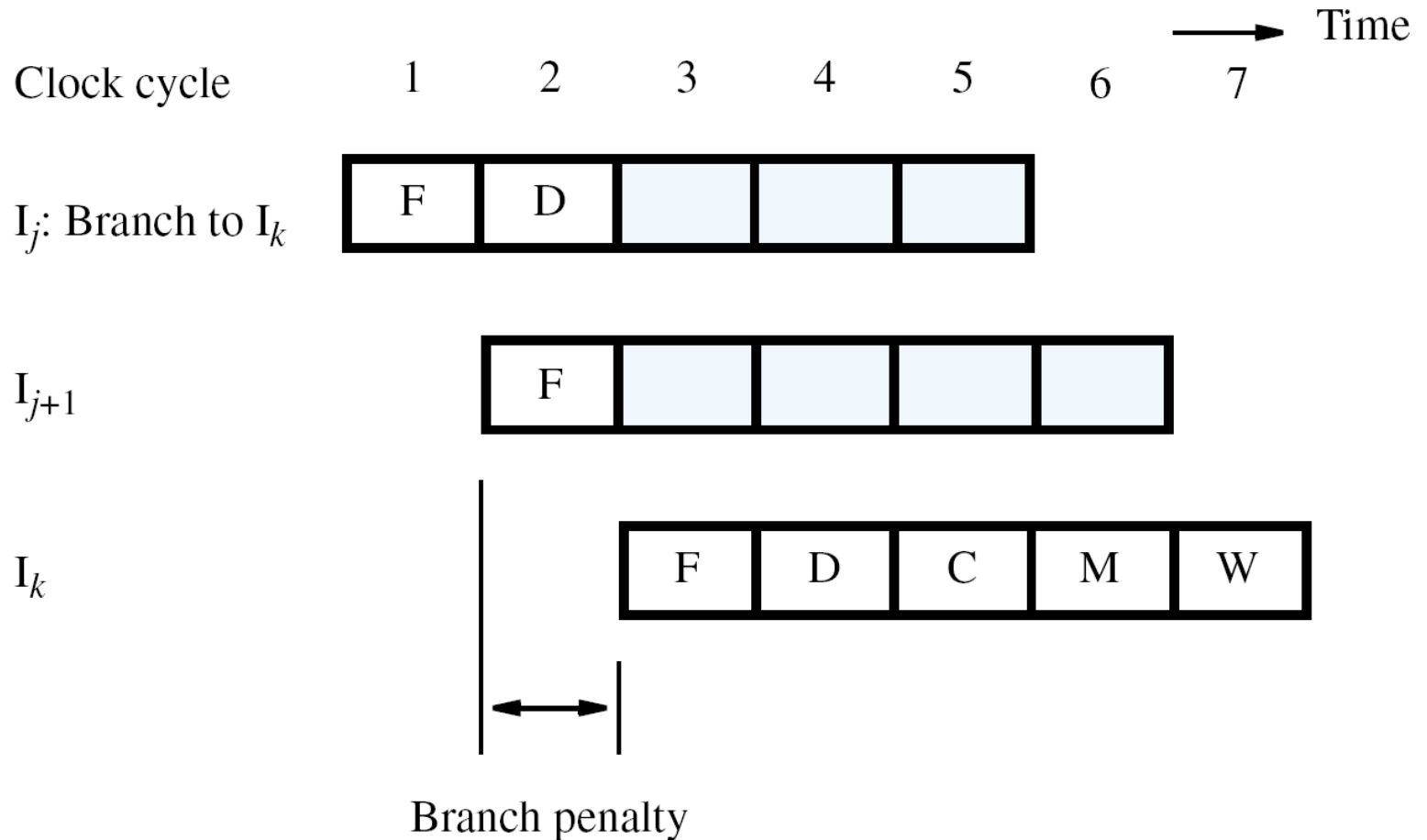
Conditional Branches

- Consider a conditional branch instruction:
  - Branch_if_[R5]=[R6]   LOOP
- Requires not only target address calculation, but also requires comparison for condition
- In the 5-stage architecture, ALU performs the comparison
- Target address now calculated in Decode stage
- To maintain one-cycle penalty, a comparator just for branches must be inserted in Decode stage
  - Higher hardware complexity

The Branch Delay Slot (1)

- 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

- That next instruction is discarded with penalty, except when conditional branch is not taken
  - Non-uniform behaviour between the two cases

- The location immediately following the branch is called the *branch delay slot*

The Branch Delay Slot (2)

- 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
- Benefits of delayed branching depends on the possibility for the compiler to fill the branch delay slot (this happens for more than 70 % of the cases in many programs)

```
            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
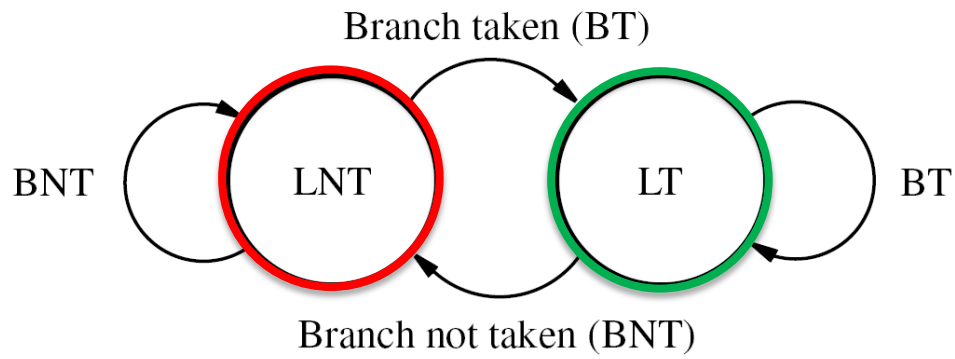
Branch Prediction

- 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>
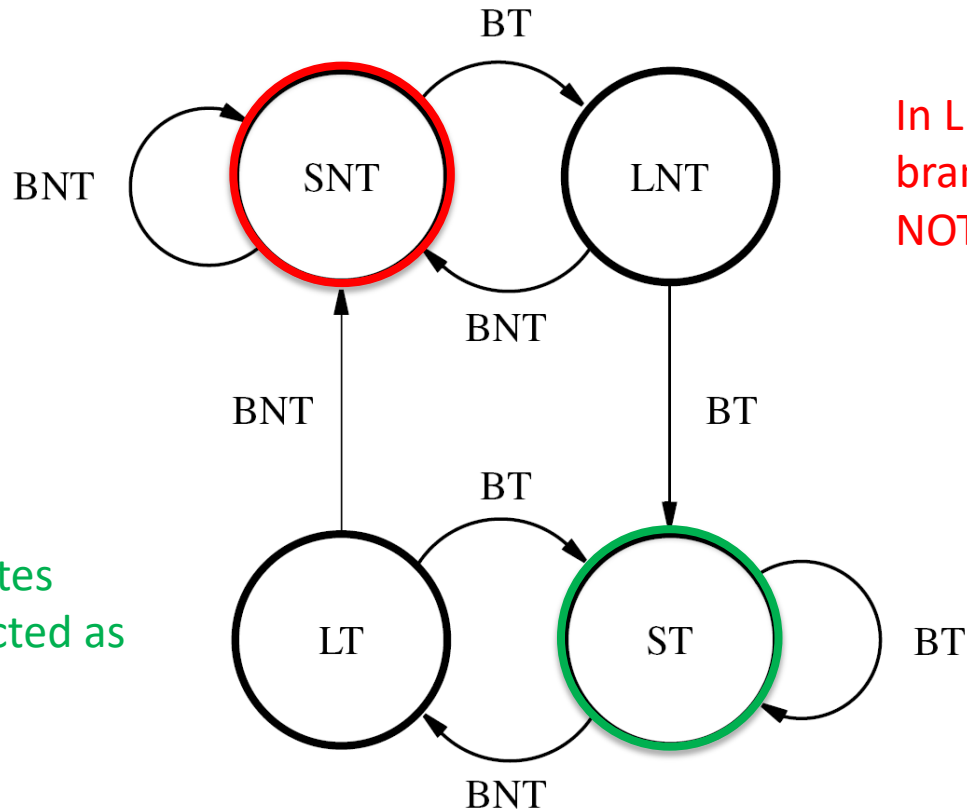
Static Branch Prediction

- 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*
  - Instead, always predict *not taken* for **forward** branch
- Expect higher accuracy for this special case, but what about accuracy for other branches?
  - For the last iteration of loops, the static prediction is wrong

Dynamic Branch Prediction

- Idea: track branch decisions during execution for *dynamic* prediction to improve accuracy

- 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

- Must be wrong *twice* to change prediction

Branch taken (BT)

BNT — LNT — LT — BT

Branch not taken (BNT)

(a) A 2-state algorithm

BT

BNT — SNT — LNT

BNT

BNT — SNT

BT

BT

LT — ST — BT

BNT

In LNT and SNT states branch is predicted NOT TAKEN

In LT and ST states branch is predicted as TAKEN

(b) A 4-state algorithm

Branch Target Buffer

- Prediction only provides a presumed *decision*
- Decode stage computes *target* in cycle <u>2</u>
- But we need target (and prediction) in cycle <u>1</u>
- *Branch target buffer* stores target address and history from *last* execution of each branch. Each element of the buffer contains: **the address of the branch instr., the state of the branch prediction alg and the branch target address**
- In cycle 1, use branch instruction address to look up target and use history for prediction
- Fetch in cycle 2 using prediction; if mispredict detected during Decode, correct it in cycle 3

Performance Evaluation

- For a **non-pipelined proc.**, the instr. throughput (#instr. per second) is:
  - $F_{np} = R/S$, where R is the clock freq. and S is the average number of cycles to execute one instr. For the 5-stage architecture, $S = 5$ assuming that every memory access can be performed in one cycle (no cache misses).

- For a **pipelined proc.**, throughput F is increased by instr. execution overlapping. Ideally, S can be reduced to 1 (one instr. per cycle). This implies no pipeline stalls.
  - How can we quantify the effect of pipeline stalls and branch penalties on achieved instr. throughput?

Effect of Stalls

- Let's consider a proc. w/ *operand forwarding* (in hardware). Stalls occur when data dependency is related to a Load instr., which causes a 1-cycle stall
  - E.g. if freq. of Load instr. p_{LD} = 25 %, freq. of data dependency after a Load p_{LD-dep} = 40 %, then throughput F is reduced to:
    - $F = R/(1 + \delta_{stall})$ where $\delta_{stall} = 1 \times p_{LD} \times p_{LD-dep}$ = 0.1
    - Thus, $F = R/1.1 = 0.91R$
  - The compiler can improve performance by trying to reduce p_{LD-dep}

Effect of Branch Penalty

- Let's consider a proc. w/ *branch decision and branch target address evaluation in the Decode stage*. When branch is mispredicted, there is 1-cycle penalty.
  - E.g. if freq. of *branch* instr. p_{BR} = 20 %, prediction accuracy $p_{BR-pred}$ = 90 %, then throughput F is reduced to:
    - F = R/(1 + $\delta_{BR\_penalty}$) where
      $\delta_{BR\_penalty}$ = 1xp_{BR}x (1- $p_{BR-pred}$) = 0.02
    - Thus, *$F = R/1.02 = 0.98R$*
  - $\delta_{BR\_penalty}$ adds to δ_{stall}

Effect of Cache Misses

- When a cache miss occurs, there is a penalty due to the access of a slower memory which stalls the pipeline for N_{miss} cycles.
  - E.g. if freq. of *cache* misses during fetch $p_{miss\text{-}fetch}$ = 5 %, freq. of *cache* misses during mem access $p_{miss\text{-}mem}$ = 10 %, freq. of Load and Store instr. $p_{LD\text{-}ST}$ = 30 %, N_{miss} = 15, then throughput F is reduced to:
    - $F = R/(1 + \delta_{cache\text{-}miss})$ where
      $\delta_{cache\text{-}miss} = N_{miss} (p_{miss\text{-}fetch} + p_{LD\text{-}ST} \times p_{miss\text{-}mem})$
      = 15x(0.05 + 0.03) = 1.2
    - Thus $F = R/2.2 = 0.45R$
  - $\delta_{cache\text{-}miss}$ adds to $\delta_{BR\_penalty}$ and δ_{stall}.
  - Thus, overall $F = R/2.32 = 0.42R$
   **Cache misses are the dominant factor.**

Number of Pipeline Stages n

$$F = \frac{R}{1 + \delta_{\text{penalty}}}, \; R \text{ clock frequency}$$

- R increases with n ($R \propto n$, if n is low)
- However, also δ_{penalty} increases with n because of higher probability of stalls, later branch decisions, higher cycle penalty,…
- Choose so that ALU determins R (the other stages need similar times)
  - To increase R further piplenine also ALU
  - Up to 20 stages to have R in the order of GHz

Superscalar Operation

- Introduce **multiple execution units** to enable *multiple instruction issue* for **higher than 1** instr./cycle throughput

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

Superscalar Processor (1)
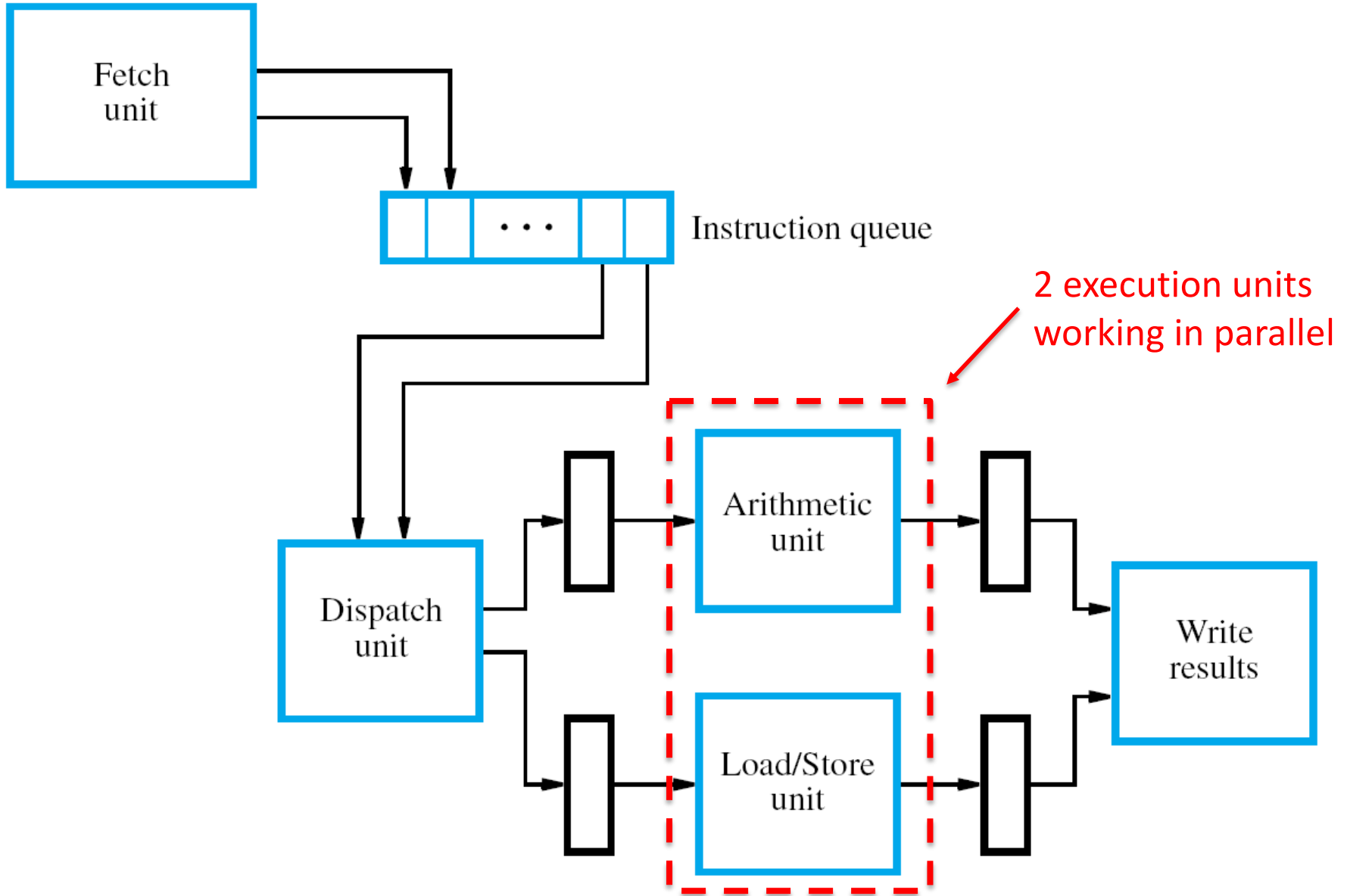
Superscalar Processor (2)

- 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 the same reg.)

Superscalar Proc. Instr. Exec. Example
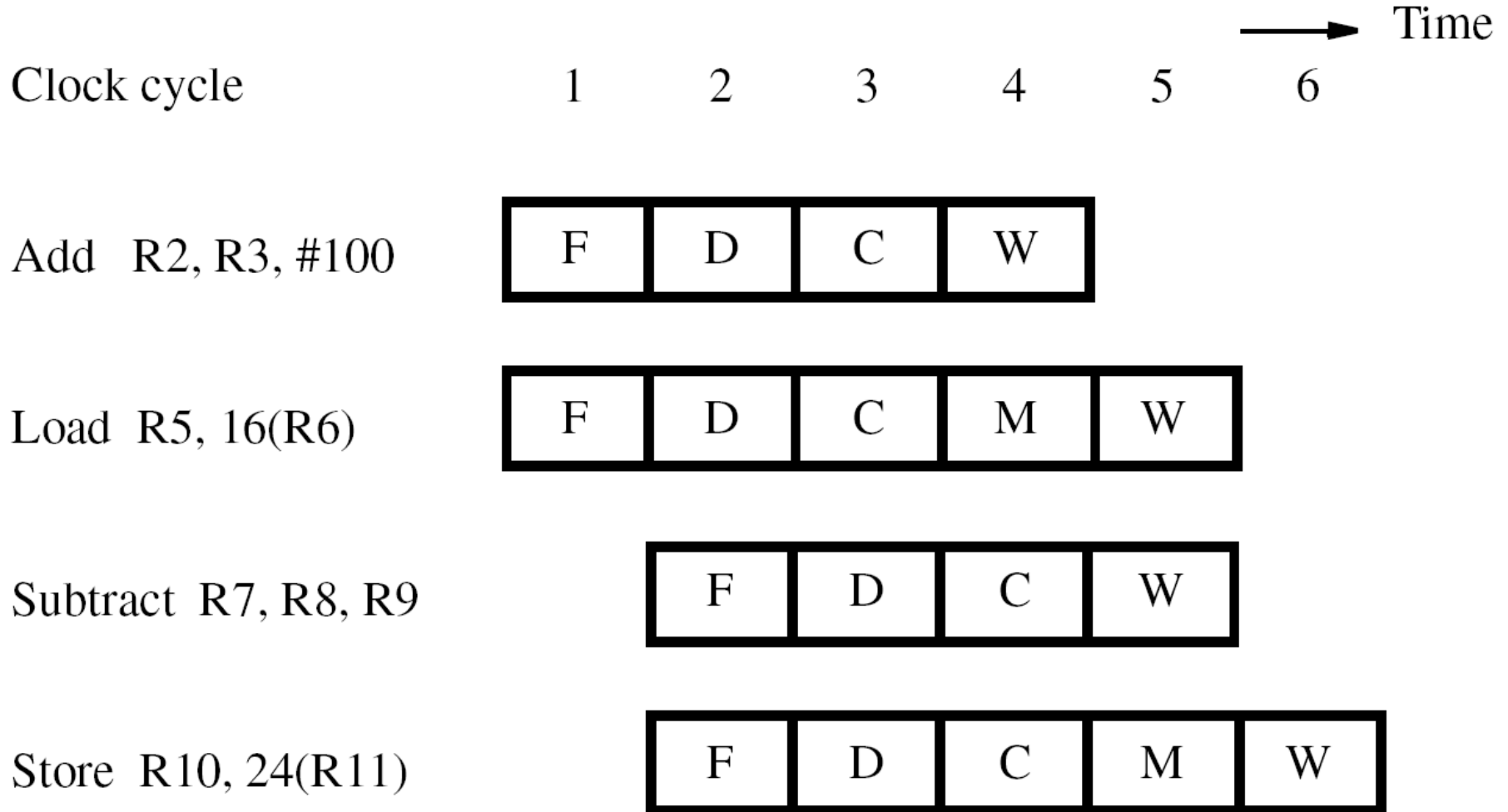
Branches and Data Dependencies (1)

- 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 (2)

- *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*

- For *precise exceptions*, results must strictly be committed in original order. This requires additional 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 to commit them in the register file
  - *Reorder buffer* in commitment unit is where dispatched instructions are placed strictly in the program order
    - Update the actual destination register only for instruction at the head of *reorder buffer* queue
    - 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

Pipelining in CISC Processors

- Load/Store architecture simplifies pipelining; influenced development of RISC processors

- CISC processors introduce complications from instructions with multiple memory operands and side effects (autoincrement, cond. codes)

- But existing CISC architectures later pipelined (with more effort) after development of RISC

- Examples: Freescale ColdFire and Intel IA-32

Concluding Remarks

- Pipelining overlaps activity for 1 instr./cycle
- Combine it with multiple instruction issue in superscalar processors for +1 instr./cycle
- Potential performance gains depend on:
  - Instruction set characteristics
  - Design of pipeline hardware
  - Ability of compilers to optimize code
- Interaction of these aspects is a key factor

References

- C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian "Computer Organization and Embedded Systems," *McGraw-Hill International Edition*
  - Chapter VI: Pipelining