Chapter 6
Pipelining

Chapter Outline

- Pipelining: overlapped instruction execution
- Hazards that limit pipelined performance gain
- Hardware/software implications of pipelining
- Influence of pipelining on instruction sets

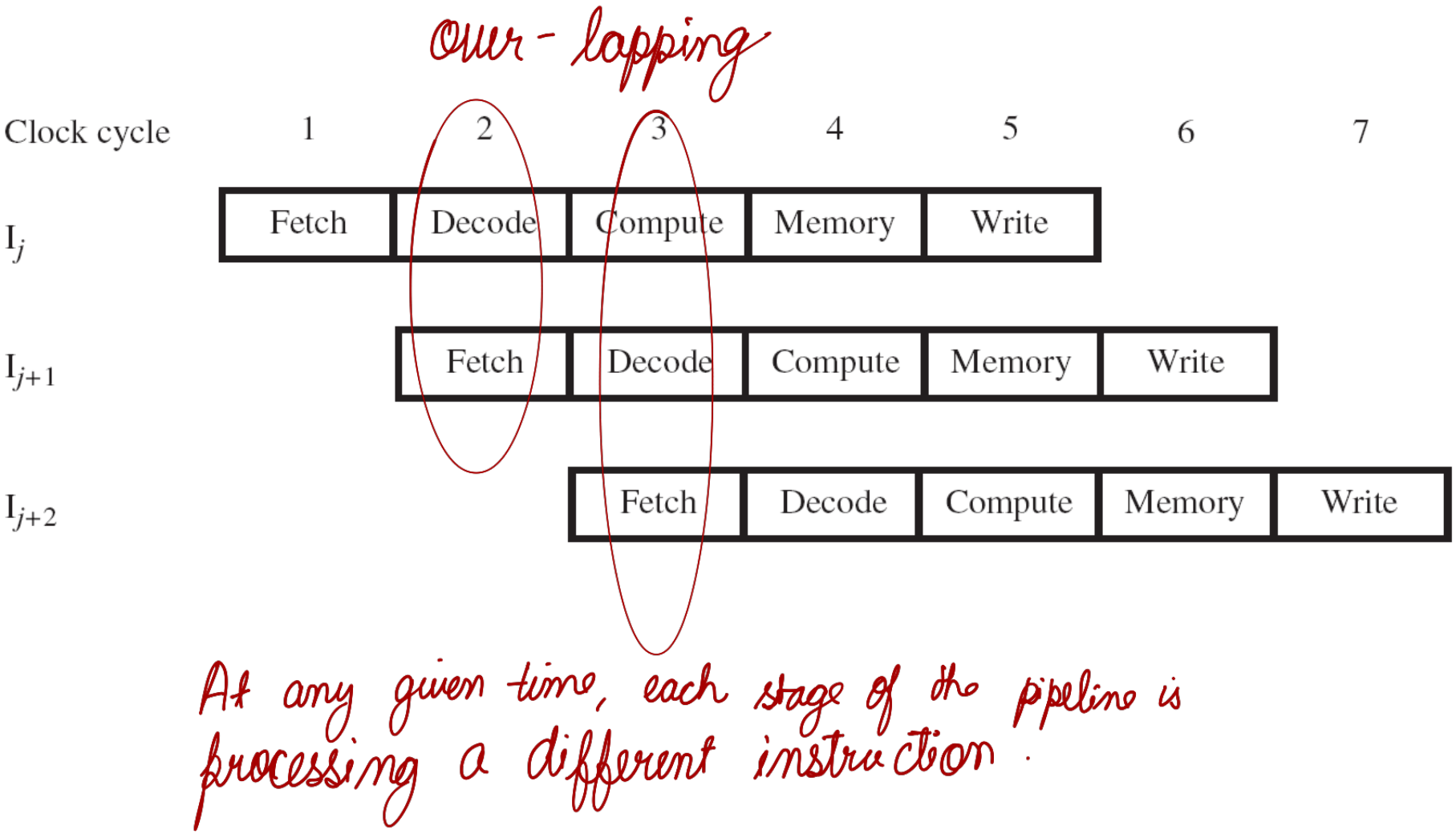
Basic Concept of Pipelining

- 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 consists of: Fetch, Decode, Compute, Memory and 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, but *rate* is 1 per cycle

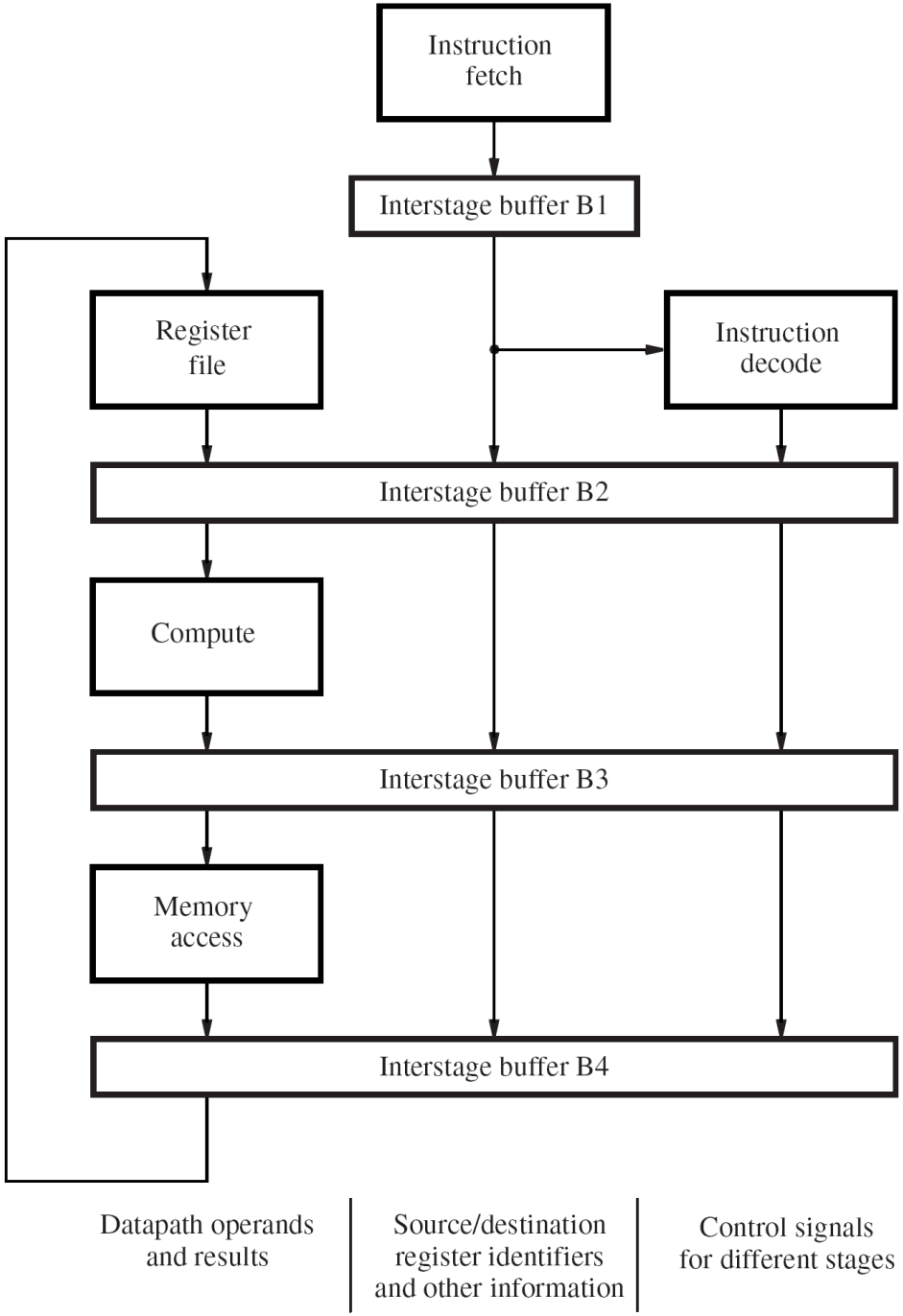
Pipelined Execution – Ideal case



At any given time, each stage of the pipeline is processing a different instruction.

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 discussed earlier
- 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

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*

Data Dependencies

- Now consider the specific instructions

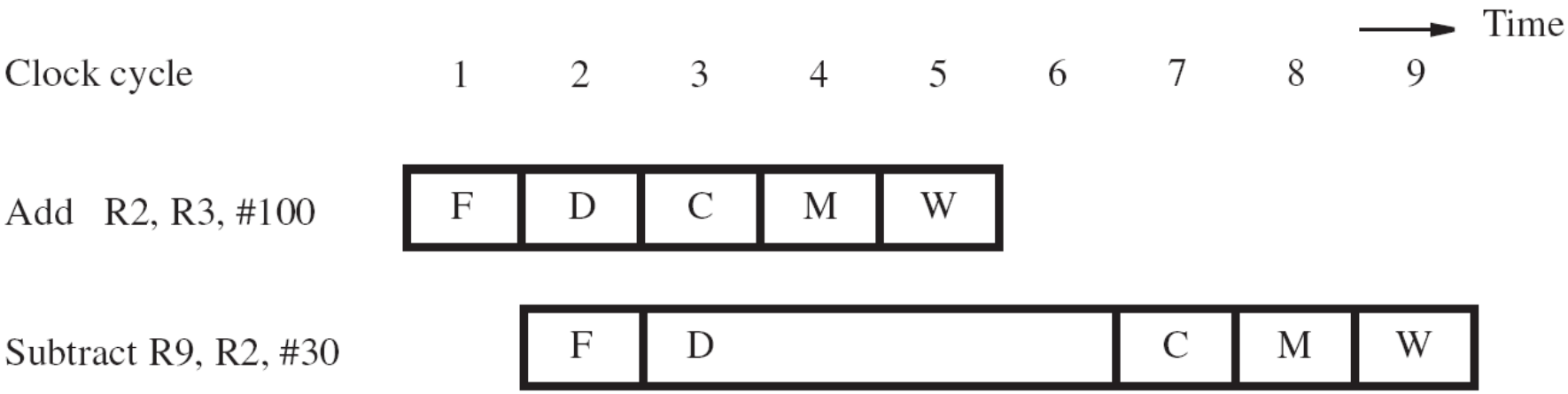
  Add R2, R3, #100

  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

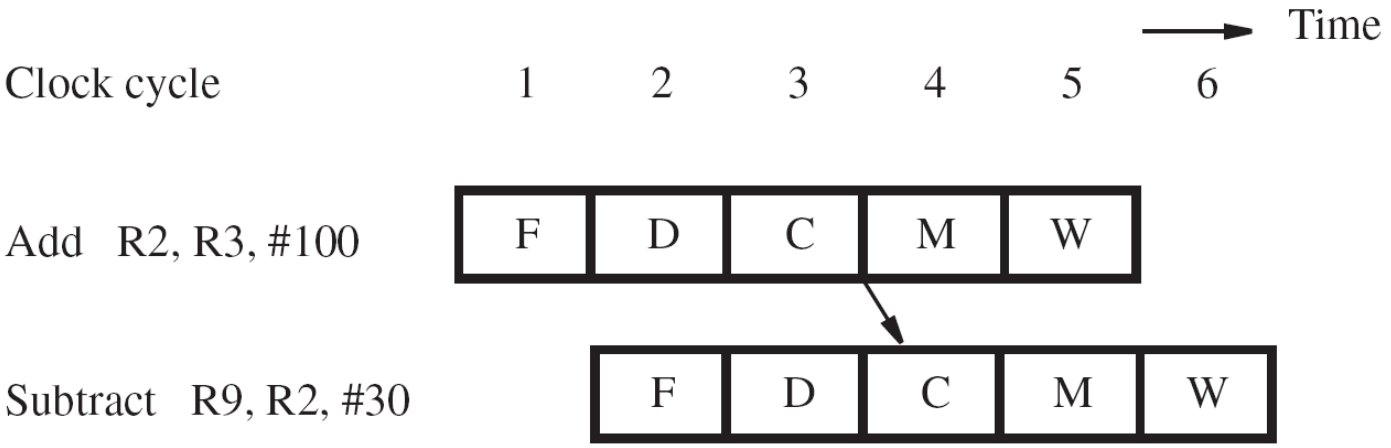
- Control circuitry must recognize dependency while Subtract is being decoded in cycle 3
- Interstage buffers carry register identifiers for source(s) and destination of instructions
- In cycle 3, compare destination identifier in Compute stage against source(s) in Decode
- R2 matches, so Subtract kept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

Details for Stalling the Pipeline

- Stall the Subtract instruction for 3 cycles by 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*

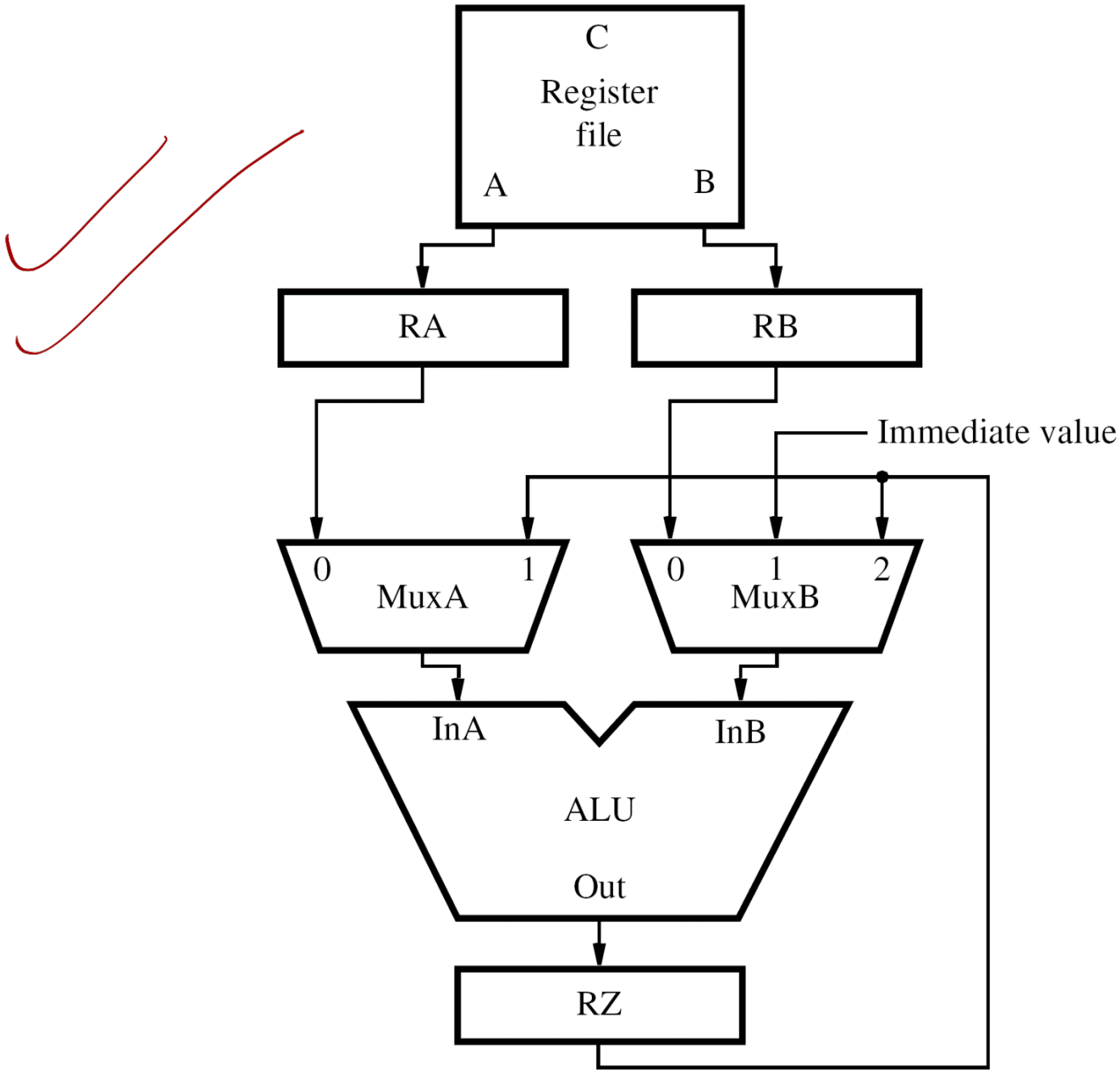
Operand Forwarding

- 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

More on Operand Forwarding

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

- Consider another situation as above
- Introduce multiplexers before ALU inputs to use contents of register RY as forwarded value
- Extend MuxA and MuxB with this input as well

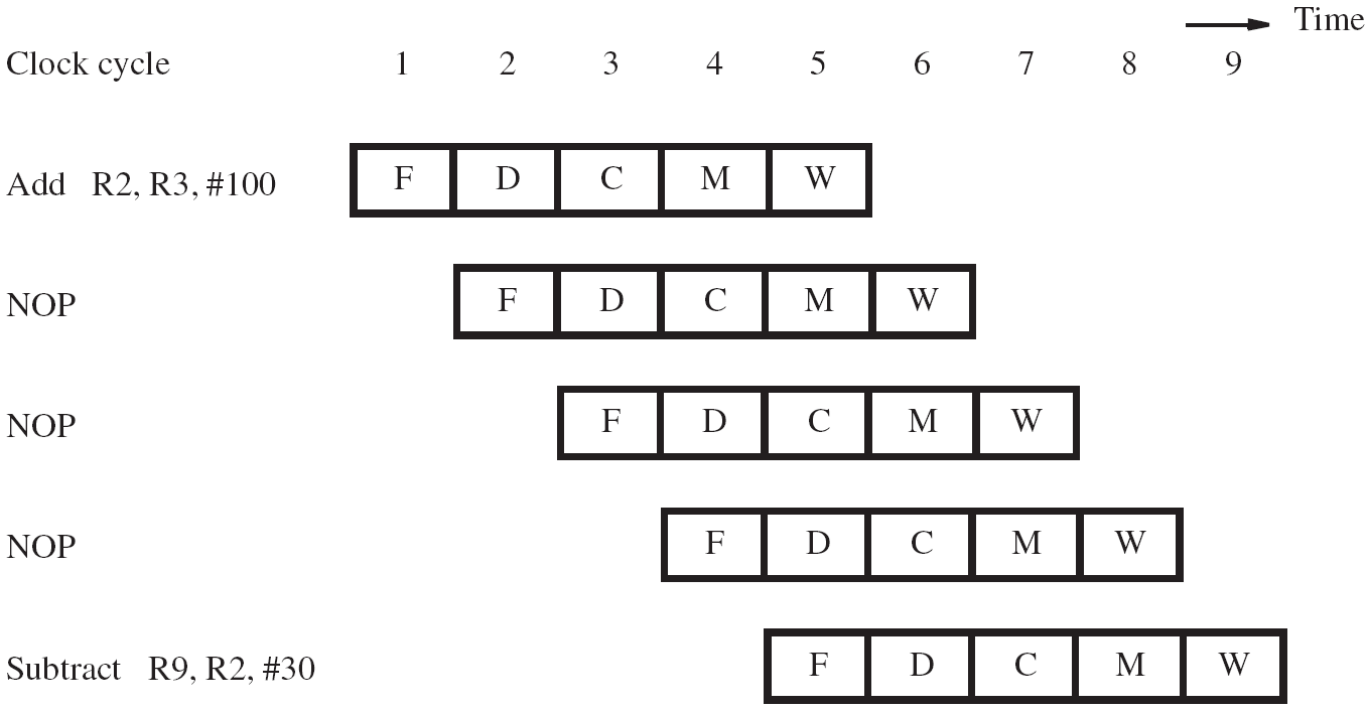
Software Handling of Dependencies

- 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)

Software Handling of Dependencies

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

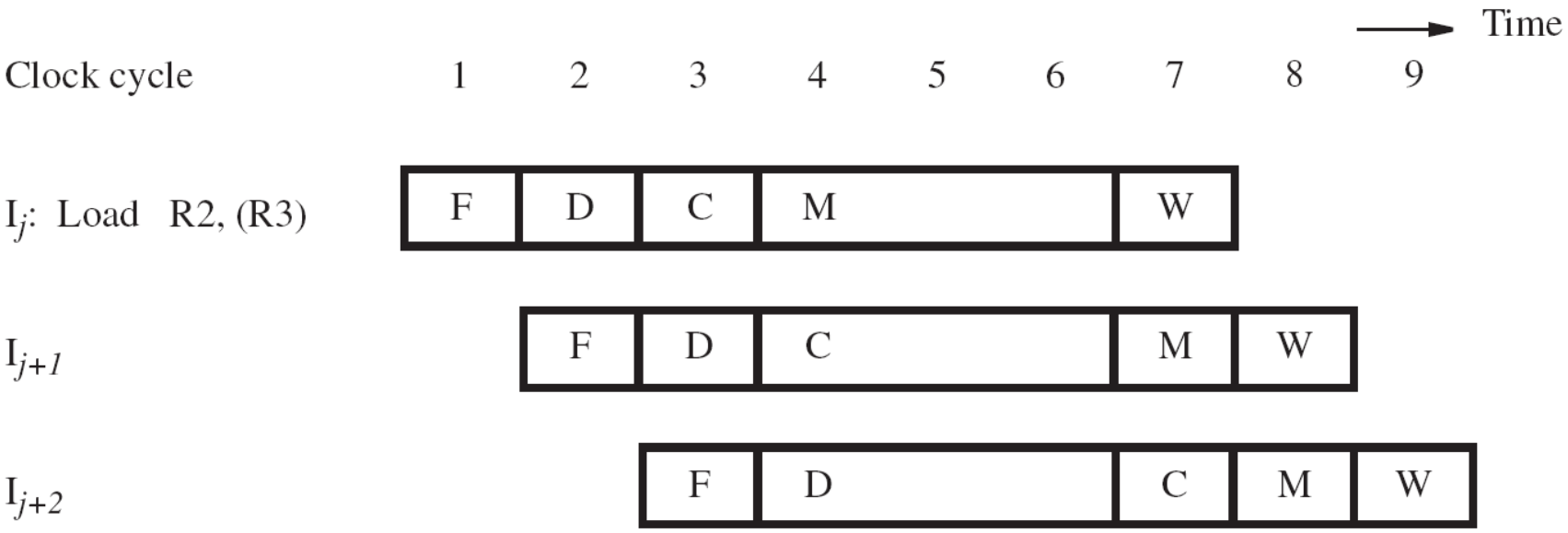
Insertion of NOP instructions for a data dependency



Memory Delays

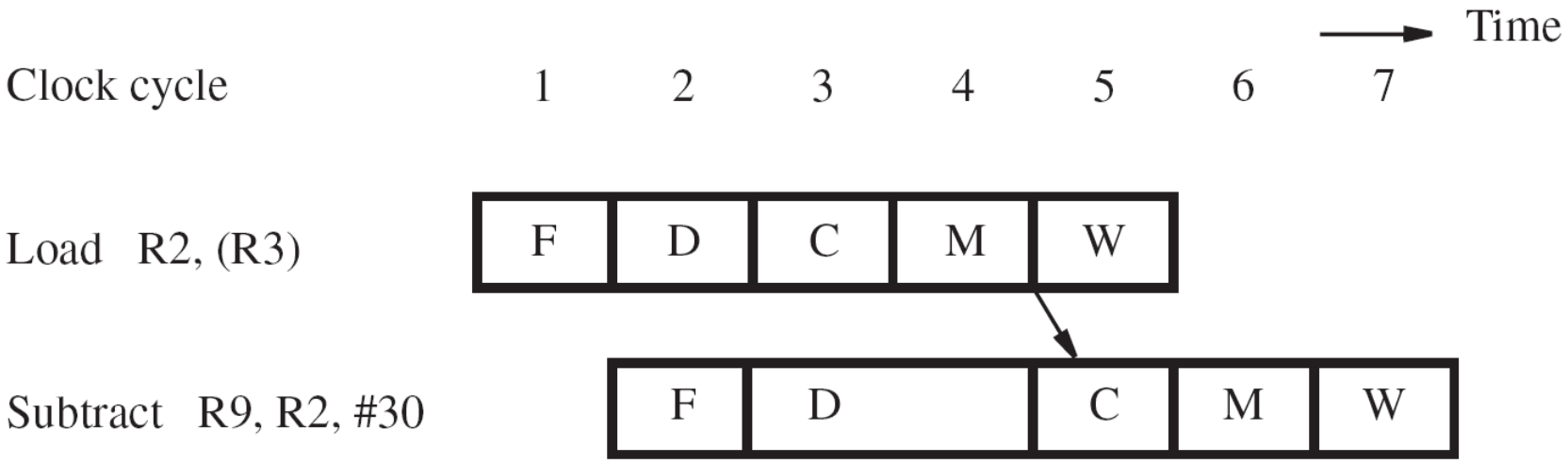
- Memory delays can also cause pipeline stalls
- A cache memory holds instructions and data from the main memory, but is faster to access
- With a cache, typical access time is one cycle
- But a cache *miss* requires accessing slower main memory with a much longer delay
- In pipeline, memory delay for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

Memory Delays



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



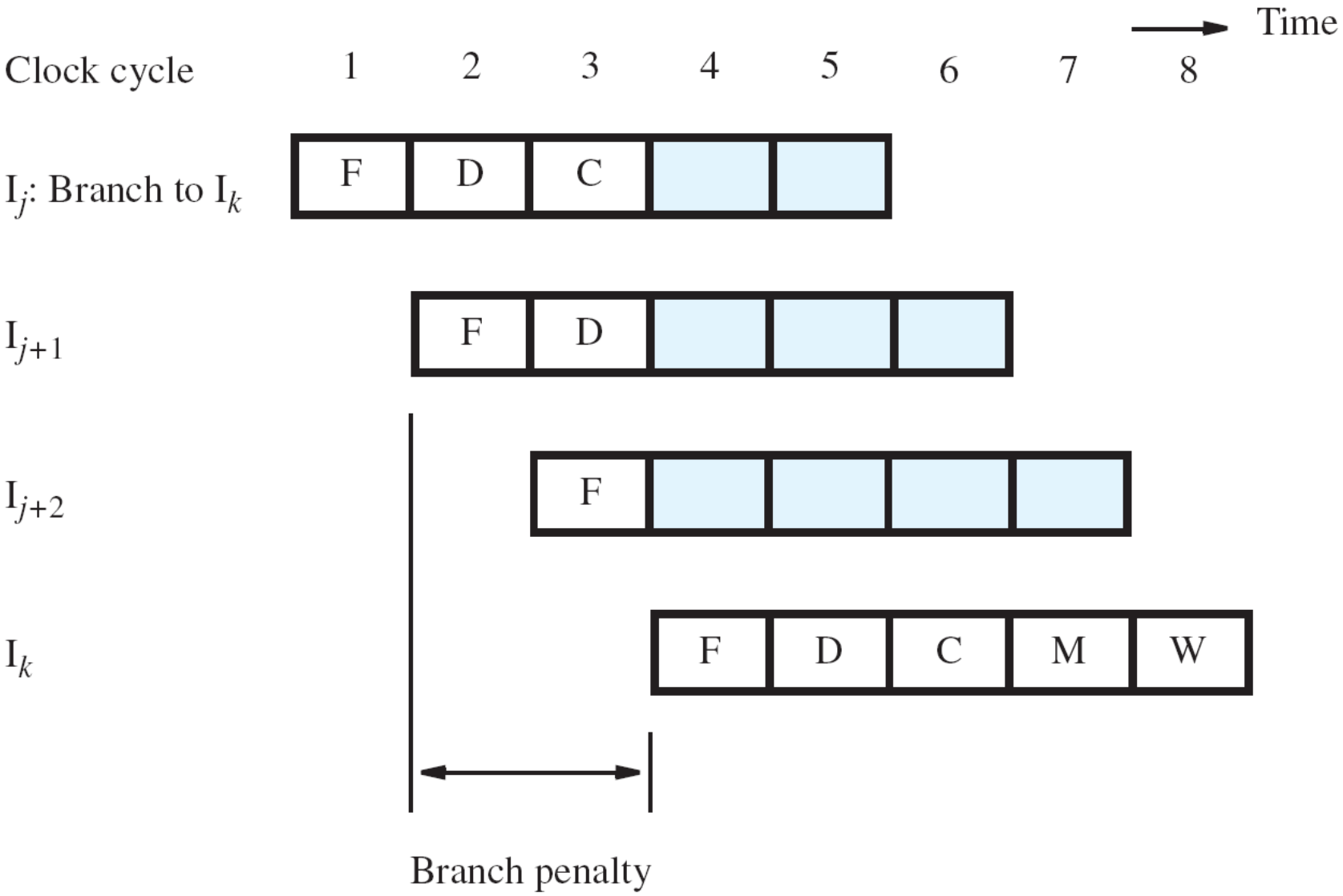
Branch Delays – Control Hazard

- Ideal pipelining: fetch each new instruction while previous instruction is being decoded
- Branch instructions alter execution sequence, but they must be processed to know the effect
- 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
- Compute stage normally determines target address using offset and PC+4 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 *penalty*

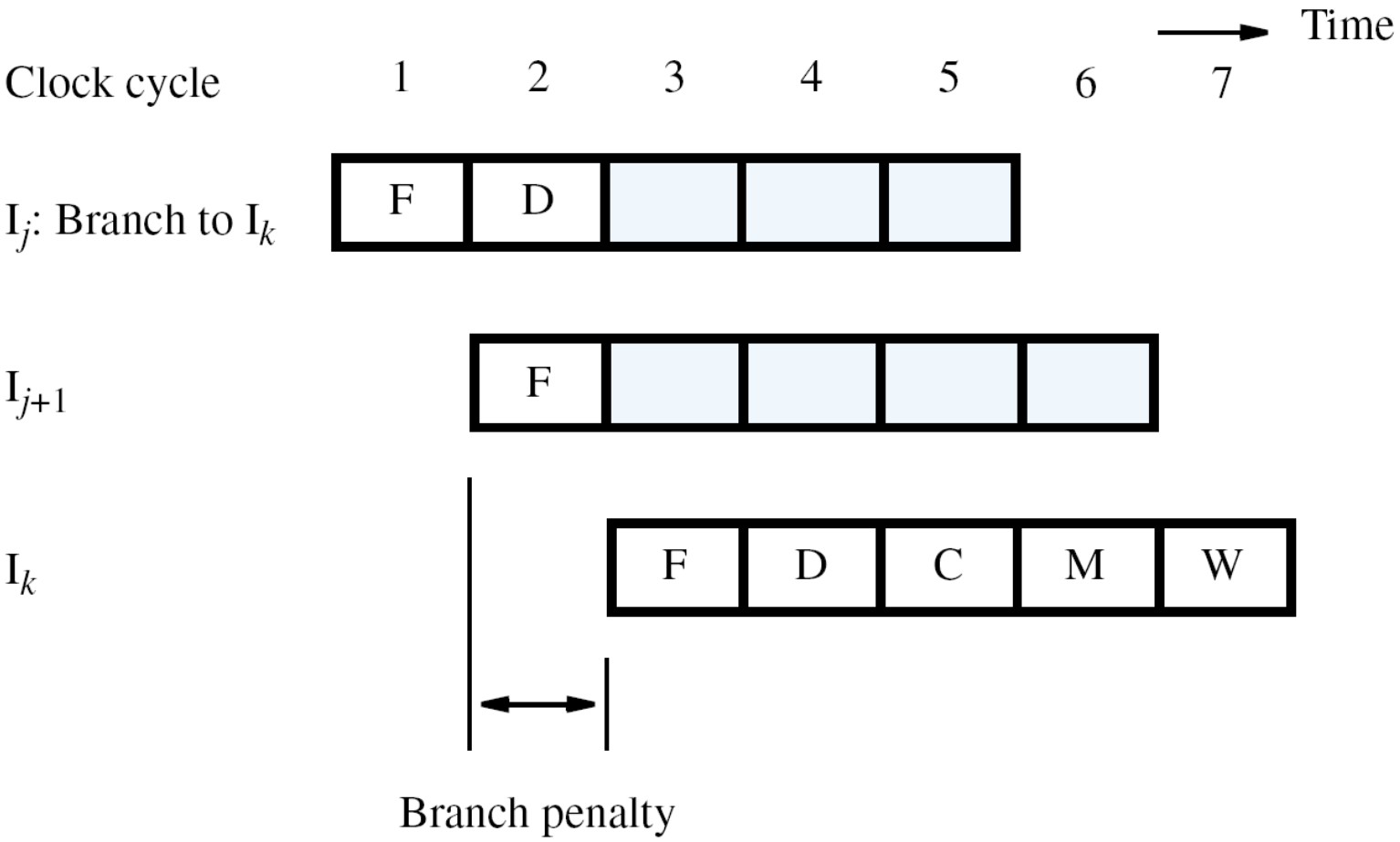
Unconditional Branches



Reducing Branch Penalty

- In pipeline, adder for PC is used every cycle, so it cannot calculate the branch target address
- So introduce a second adder just for branches
- 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 is reduced to one cycle

Reducing 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
- ALU normally performs the comparison
- Target address now calculated in Decode stage
- To maintain one-cycle penalty, we introduce a comparator just for branches in Decode stage

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
- That next instruction is discarded with penalty, except when conditional branch is not taken
- The location immediately following the branch is called the *branch delay slot*

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

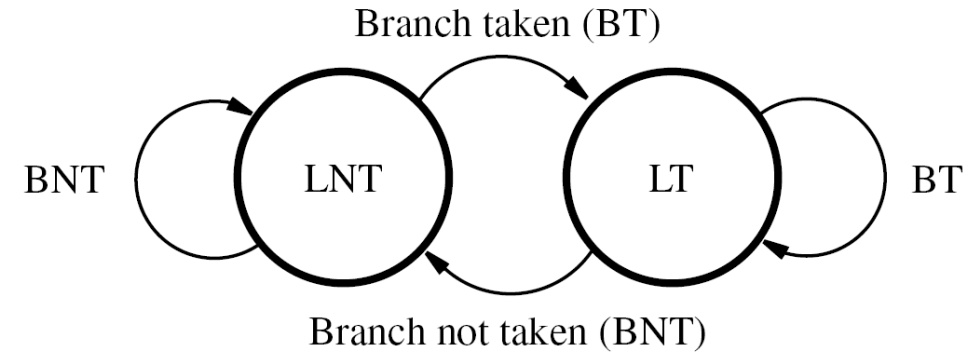
- A branch is decided in Decode stage (cycle 2) while following instruction is *always* fetched
- Following instruction may require discarding (or with delayed branching, it may be a NOP)
- Instead of discarding the *following* instruction, can we anticipate the *actual* next instruction?
- Two aims: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

Static Branch Prediction

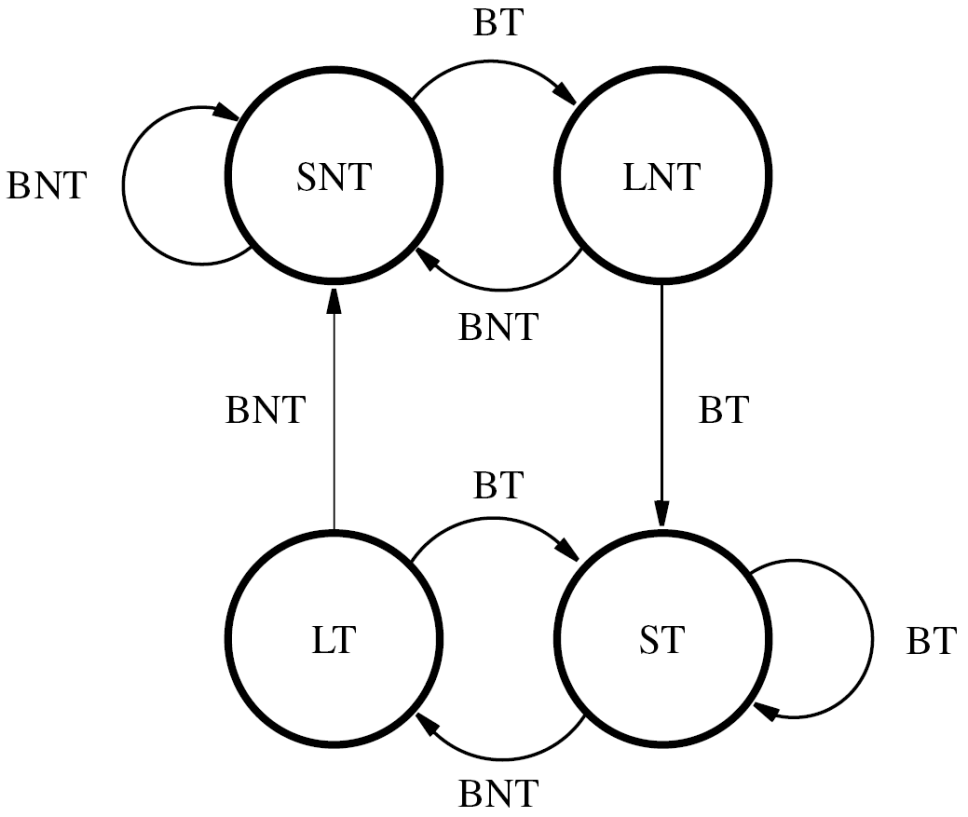
- Simplest approach: assume branch *not* taken
- Penalty if prediction disproved during Decode
- If branches are random, accuracy is 50%
- But a branch at end of a loop is usually taken
- So for backward branch, always predict *taken*
- Use target address as soon as it is available
- Expect higher accuracy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

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



(a) A 2-state algorithm



(b) A 4-state algorithm

Resource Limitations

- Effect of Resource Limitation
  - Structural hazards
  - Both instruction fetch and operand fetch from memory at the same time
  - Use separate L1 cache for instruction and data

Performance Evaluation

- For non-pipelined processor, execution time T of a program with N instructions is given by

  $T = (N \times S)/R$

  S is the average number of clock cycles per instruction

  R is the clock rate in cycles per second

- Throughput $P_{np} = R/S$
- For 5 stage RISC processor, with no pipelining, S=5
- For pipelined processor with no stalls, S=1 and hence, throughput $P_p = R$

Performance Evaluation

- Effect of Data Hazard
- δ_{stall} = Additional number of cycles per instruction due to stall over S=1
- Consider one-cycle stall due to data hazard (With Load instructions) with 25% of instructions being Load instructions and 40% of Load instructions followed by dependent instruction.
- $\delta_{stall} = 0.25 \times 0.40 \times 1 = 0.10$
- $P_p = R/(1 + \delta_{stall}) = R/1.1 = 0.91R$

Performance Evaluation

- Effect of mis-predicting branch
- Consider one-cycle branch penalty with 20% of instructions being Branch instructions and average branch prediction accuracy is 90%.
- $\delta_{\text{branch\_penalty}} = 0.20 \times 0.10 \times 1 = 0.02$
- Number of Pipeline Stages
- Large value increases throughput but also causes more hazards due to increased dependencies
- ALU is also pipelined for faster clock. Modern technology uses deep pipelines (More than 20)

Performance Evaluation

- Effect of cache miss
- Slower main memory stalls the pipeline for p_m cycles in the event of cache miss.
- Fraction m_i of all **instructions** that are fetched incur a cache miss.
- Fraction d of all instructions are Load or Store instructions, i.e., **data** to be loaded/stored.
- Fraction m_d of them incur cache miss.
- Increase over the ideal case of $S = 1$ due to cache misses is $\delta_{miss} = (m_i + d \times m_d) \times p_m$

Performance Evaluation

- Suppose that 5% of all fetched instructions incur a cache miss
- 30% of all instructions executed are Load or Store instructions, and 10% of their data-operand accesses incur a cache miss.
- Assume that the penalty to access the main memory for a cache miss is 10 cycles.
- Increase over the ideal case of $S = 1$ due to cache misses in this case is given by
- $\delta_{miss} = (0.05 + 0.30 \times 0.10) \times 10 = 0.8$

Performance Evaluation

- When all factors are combined, S is increased from the ideal value of 1 to

$$1 + \delta_{\text{stall}} + \delta_{\text{branch\_penalty}} + \delta_{\text{miss}}$$

- Contribution of cache misses is often the dominant one.

Sections to Read (From Hamacher's Book)

- Chapter on Pipelining
  - All sections and sub-sections upto and including Sub-section 6.8.2.
  - Not needed: Complete Sections 6.9 and 6.10