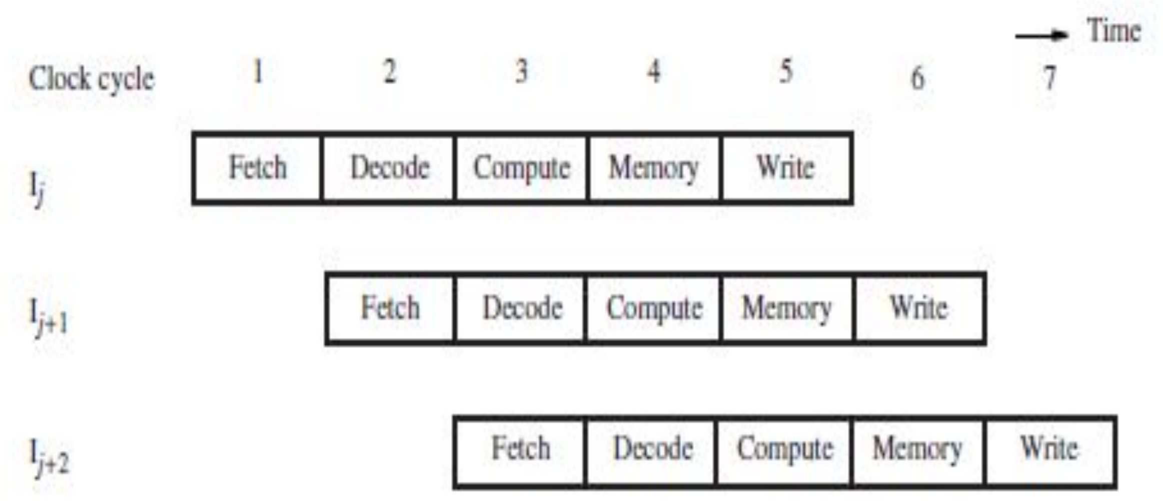
Pipelining

- The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to implement the processor and the main memory.
- Another possibility is to arrange the hardware so that more than one operation can be performed at the same time.
- Pipelining is a particularly effective way of organizing concurrent activity in a computer system



Pipelined execution

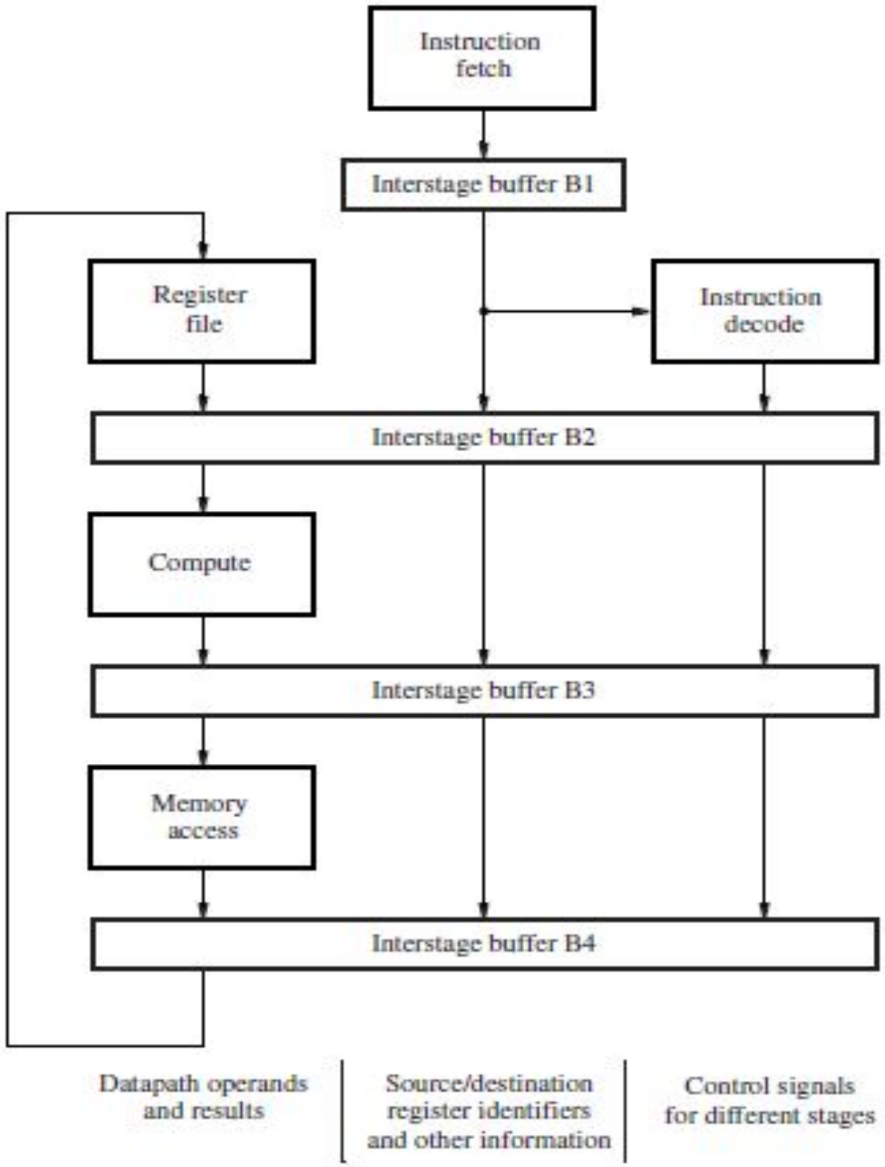
Pipeline Organization

In the first stage of the pipeline, the program counter (PC) is used to fetch a new instruction.

As other instructions are fetched, execution proceeds through successive stages.

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

Information such as register addresses, immediate data, and the operations to be performed must be carried through the pipeline as each instruction proceeds from one stage to the next. This information is held in *interstage buffers*.



A five-stage pipeline

Pipeline Hazards

Structural hazards: attempt to use the same resource two different ways at the same time

- E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)

Data hazards: attempt to use item before it is ready

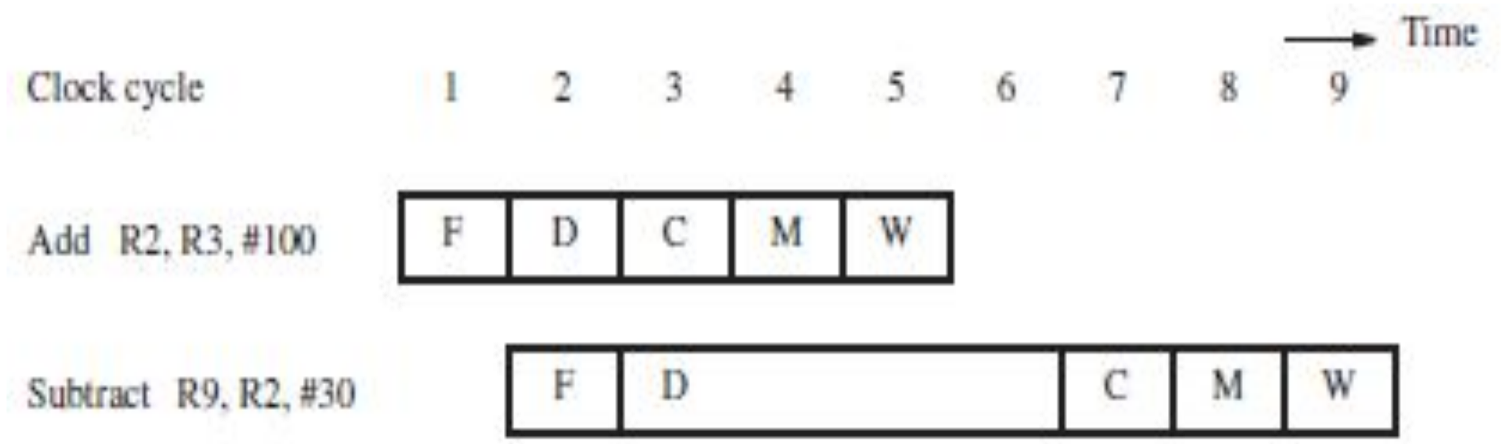
- E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
- instruction depends on result of prior instruction still in the pipeline

Control hazards: attempt to make a decision before condition is evaulated

- E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
- branch instructions

Data Dependencies

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



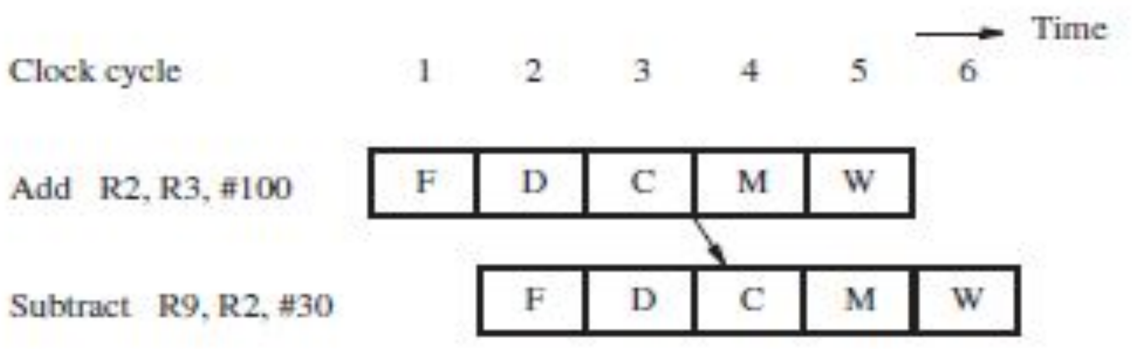
Pipeline stall due to data dependency.

Each NOP creates one clock cycle of idle time, called a bubble, as it passes through the Compute, Memory, and Write stages to the end of the pipeline.

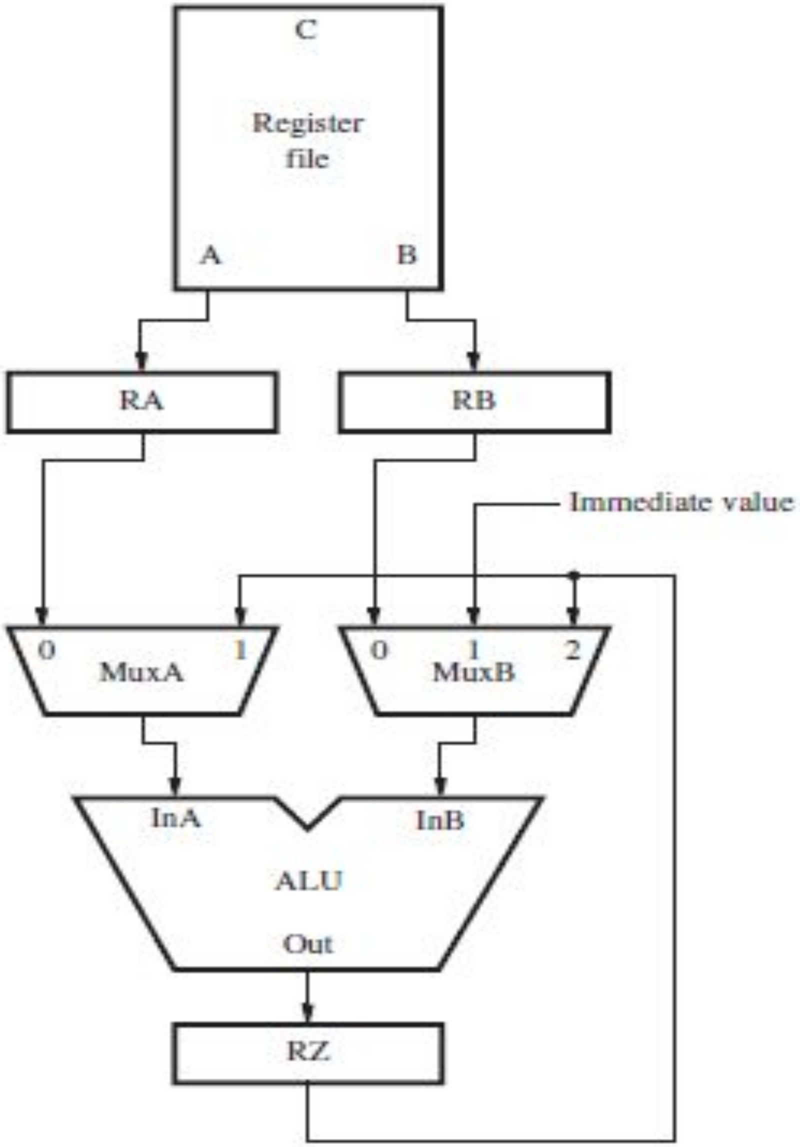
Operand Forwarding

Pipeline stalls due to data dependencies can be alleviated through the use of *operand forwarding*.

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



Avoiding a stall by using operand forwarding.


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

Handling Data Dependencies in Software

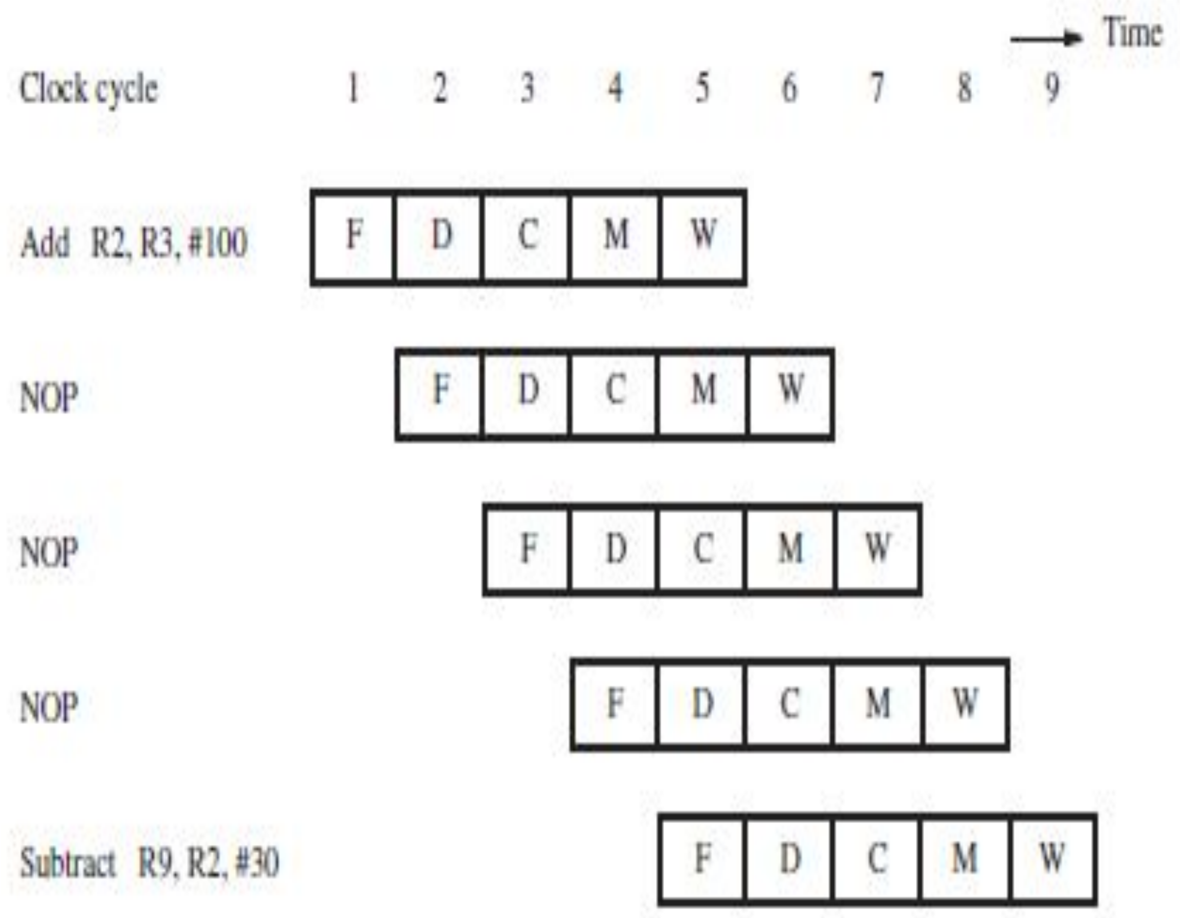
An alternative approach is to leave the task of detecting data dependencies and dealing with them to the compiler.

When the compiler identifies a data dependency between two successive instructions Ij and Ij+1, it can insert three explicit NOP instructions between them.

The NOPs introduce the necessary delay to enable instruction Ij+1 to read the new value from the register file after it is written.

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

Insertion of NOP instructions for a data dependency



Pipelined execution of instructions

Memory Delays

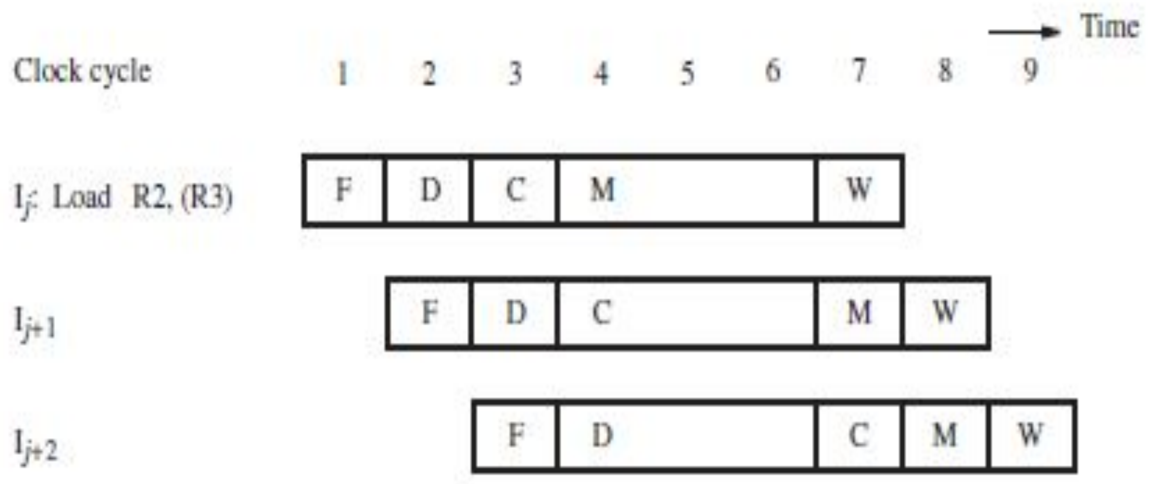
Delays arising from memory accesses are another cause of pipeline stalls. For example, a Load instruction may require more than one clock cycle to obtain its operand from memory.

This may occur because the requested instruction or data are not found in the cache, resulting in a *cache miss*.

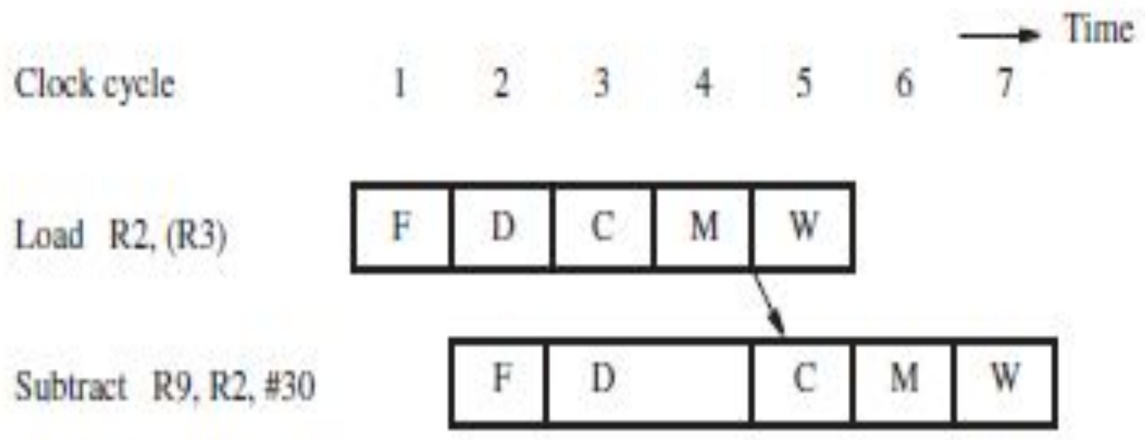
A memory access may take ten or more cycles.

 A cache miss causes all subsequent instructions to be delayed. A similar delay can be caused by a cache miss when fetching an instruction.

```
Load R2, (R3)
Subtract R9, R2, #30
```



Stall caused by a memory access delay for a Load instruction.



Stall needed to enable forwarding for an instruction that follows a Load instruction.

Branch Delays

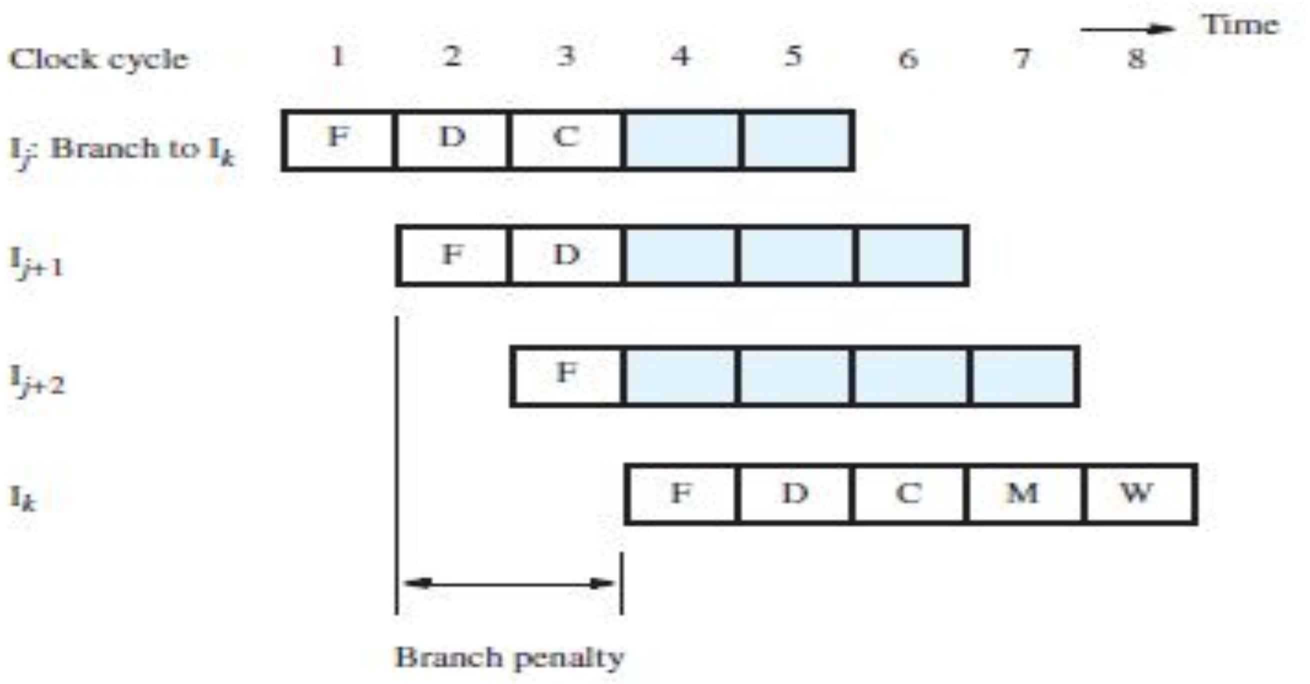
- Branch instructions can alter the sequence of execution, but they must first be executed to determine whether and where to branch.

- Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs.

- Reducing the branch penalty requires the branch target address to be computed earlier in the pipeline.

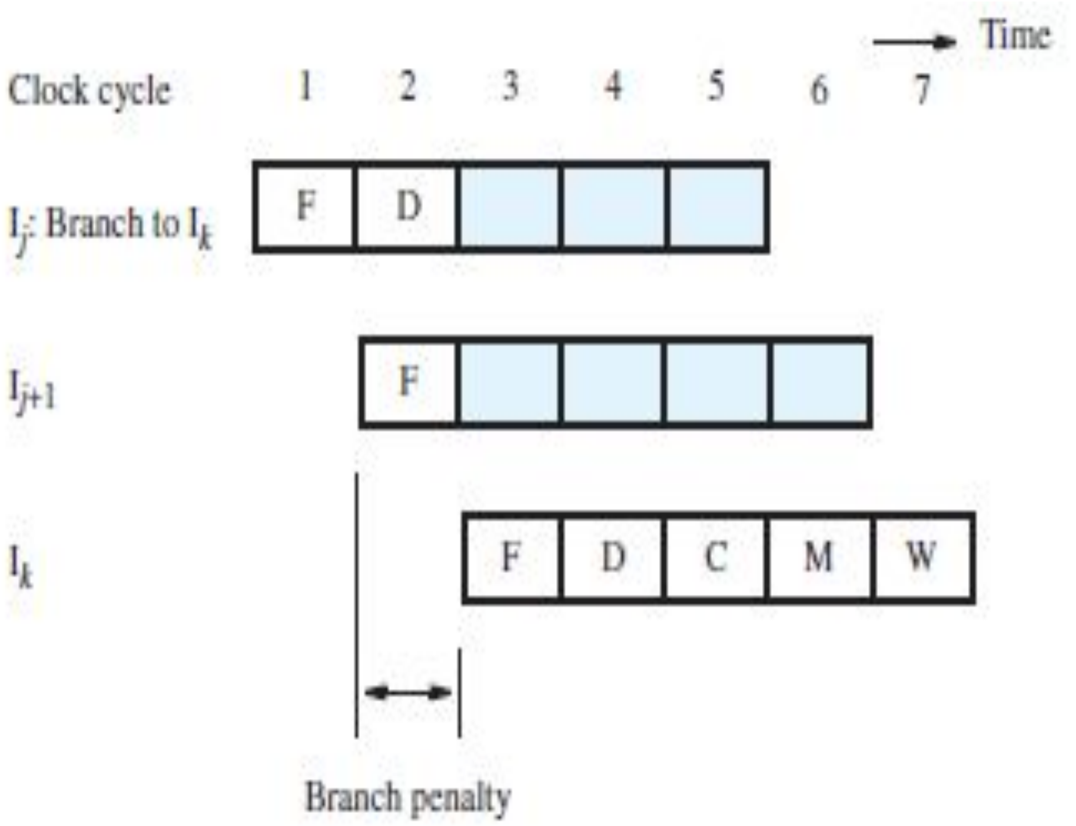
Unconditional branches

Reducing the branch penalty requires the branch target address to be computed earlier in the pipeline



Branch penalty when the target address is determined in the Compute stage of the pipeline.

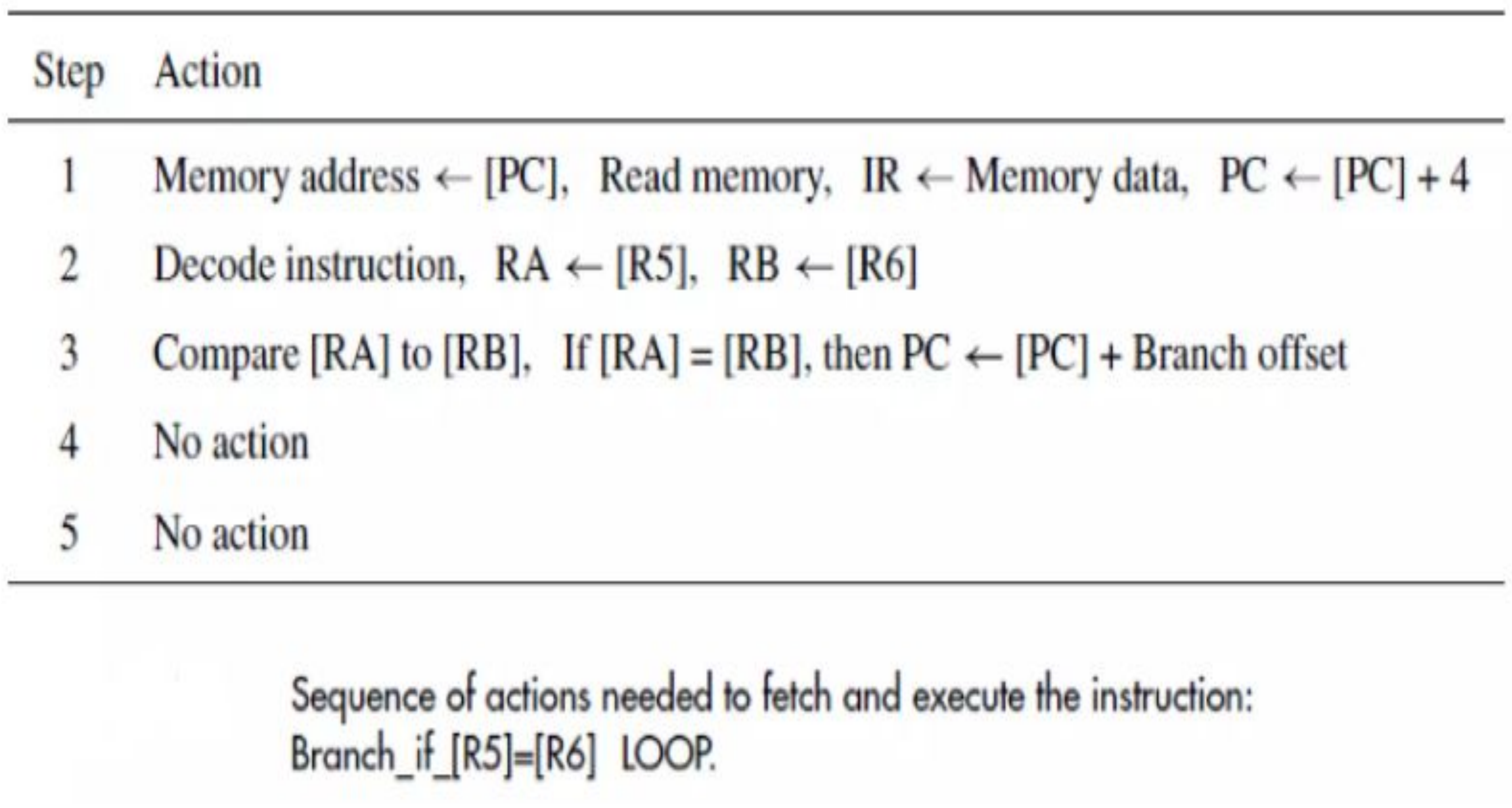
Target address is determined in the decode stage of the pipeline



Branch penalty when the target address is determined in the Decode stage of the pipeline

Conditional branches

For pipelining, the branch condition must be tested as early as possible to limit the branch penalty



Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R5], RB ← [R6]
3	Compare [RA] to [RB], If [RA] = [RB], then PC ← [PC] + Branch offset
4	No action
5	No action

Sequence of actions needed to fetch and execute the instruction:
Branch_if_[R5]=[R6] LOOP.

The Branch Delay Slot

- The location that follows a branch instruction is called the branch delay slot

- Rather than conditionally discard the instruction in the delay slot, we can arrange to have the pipeline always execute this insttruction, whether or not the branch is 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

Filling the branch delay slot with a useful instruction.

Branch prediction

To reduce the branch penalty further, the processor needs to anticipate that an instruction being fetched is a branch instruction

Predict its outcome to determine which instruction should be fetched.

Static branch prediction

The simplest form of branch prediction is to assume that the branch will not be taken and to fetch the next instruction in sequential address order.

If the prediction is correct, the fetched instruction is allowed to complete and there is no penalty.

Misprediction incurs the full branch penalty

Dynamic branch prediction

The processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that a branch instruction is executed.

Dynamic prediction algorithm can use the result of the most recent execution of a branch instruction

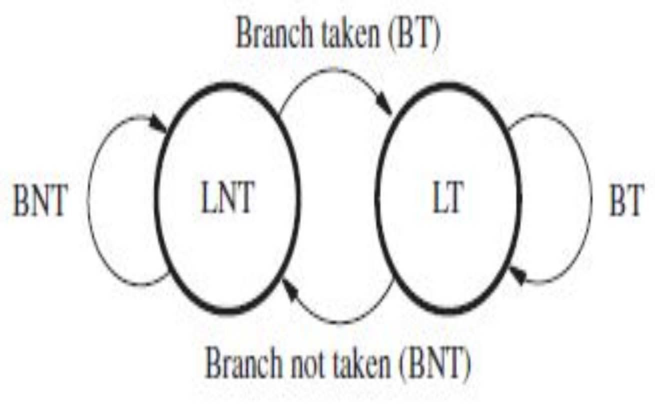
The processor assumes that the next time the instruction is executed, the branch decision is likely to be the same as the last time.

LT - Branch is likely to be taken
LNT - Branch is likely not to be taken

A 2-state algorithm

When the branch instruction is executed and the branch is taken, the machine moves to state LT. otherwise, it remains in state LNT

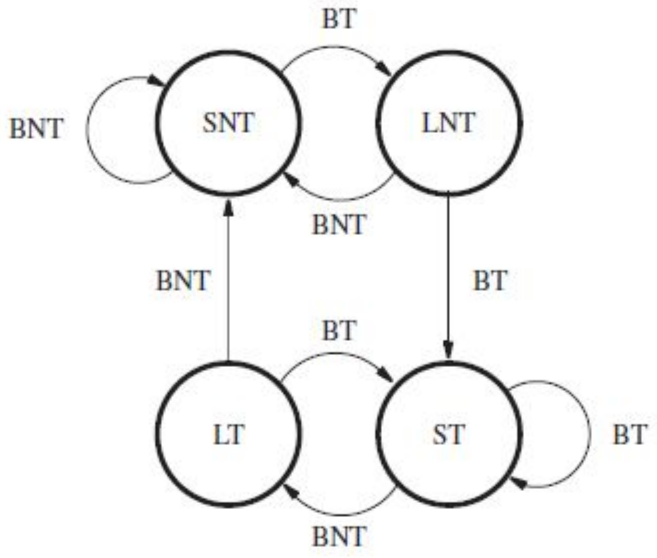
The next time the same instruction is encountered, the branch is predicted as taken if the state machine is in state LT. Otherwise it is predicted as not taken.



A 4-state algorithm

After the branch instruction is executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT.

The branch is predicted as taken if the state is either ST or LT. Otherwise, the branch is predicted as not taken.



Performance evaluation
Basic performance equation

- For a non-pipelined processor, the execution time, T, of a program that has a dynamic instruction count of N is given by

$$T = \frac{N \times S}{R}$$

- S is a average number of clock cycles it takes to fetch and execute one instruction

- R is the clock rate in cycles per second

Instruction throughput

Number of instructions executed per second. For non-pipelined execution, the throughput, Pnp, is given by

$$P_{np} = \frac{R}{S}$$

The processor with five cycles to execute all instructions.

If there are no cache misses, S is equal to 5

For the five-stage pipeline, each instruction is executed in five cycles, but a new instruction can ideally enter the pipeline every cycle.

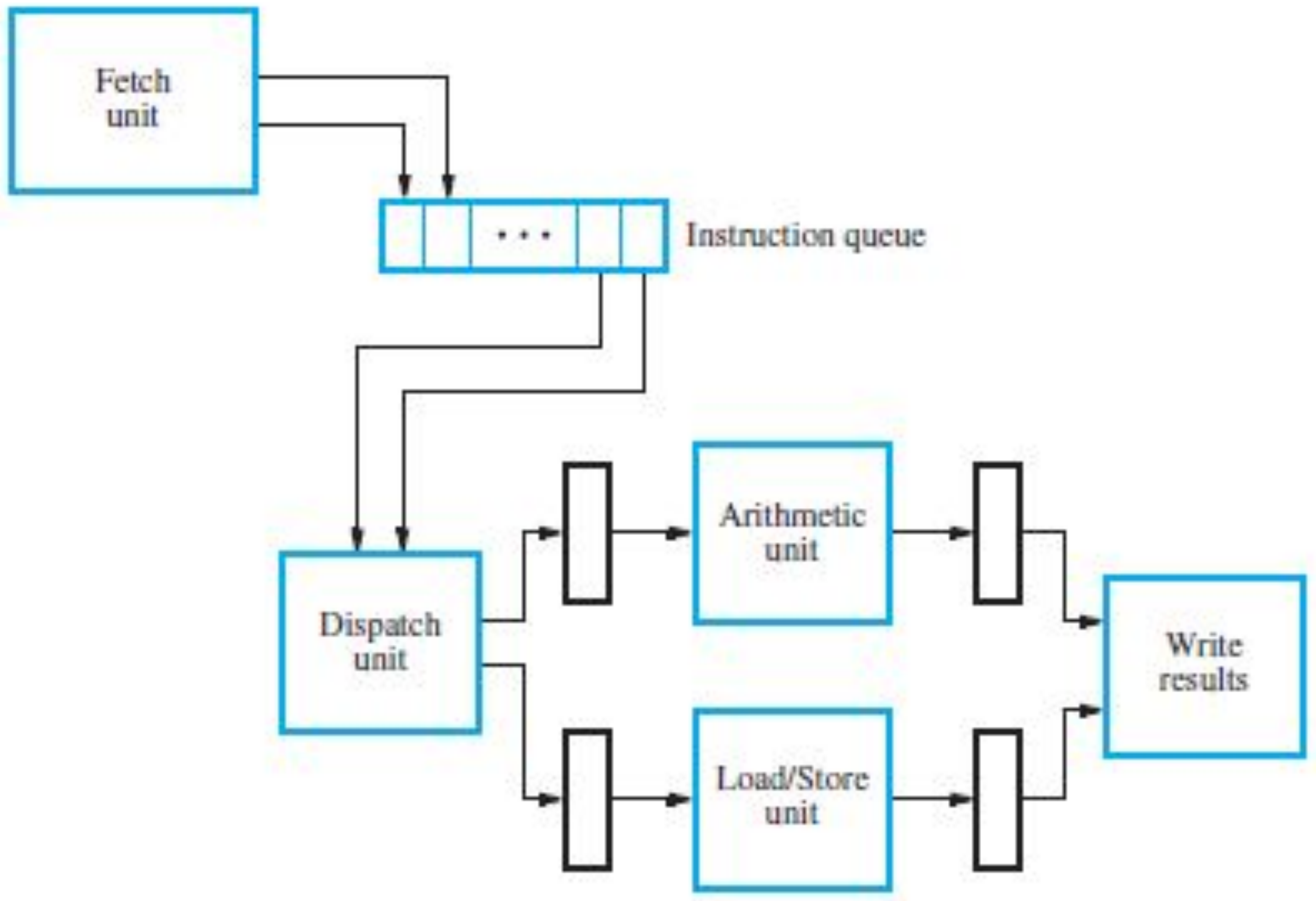
Thus, in the absence of stalls, S is equal to 1, and the ideal throughput with pipelining is

$$P_p = R$$

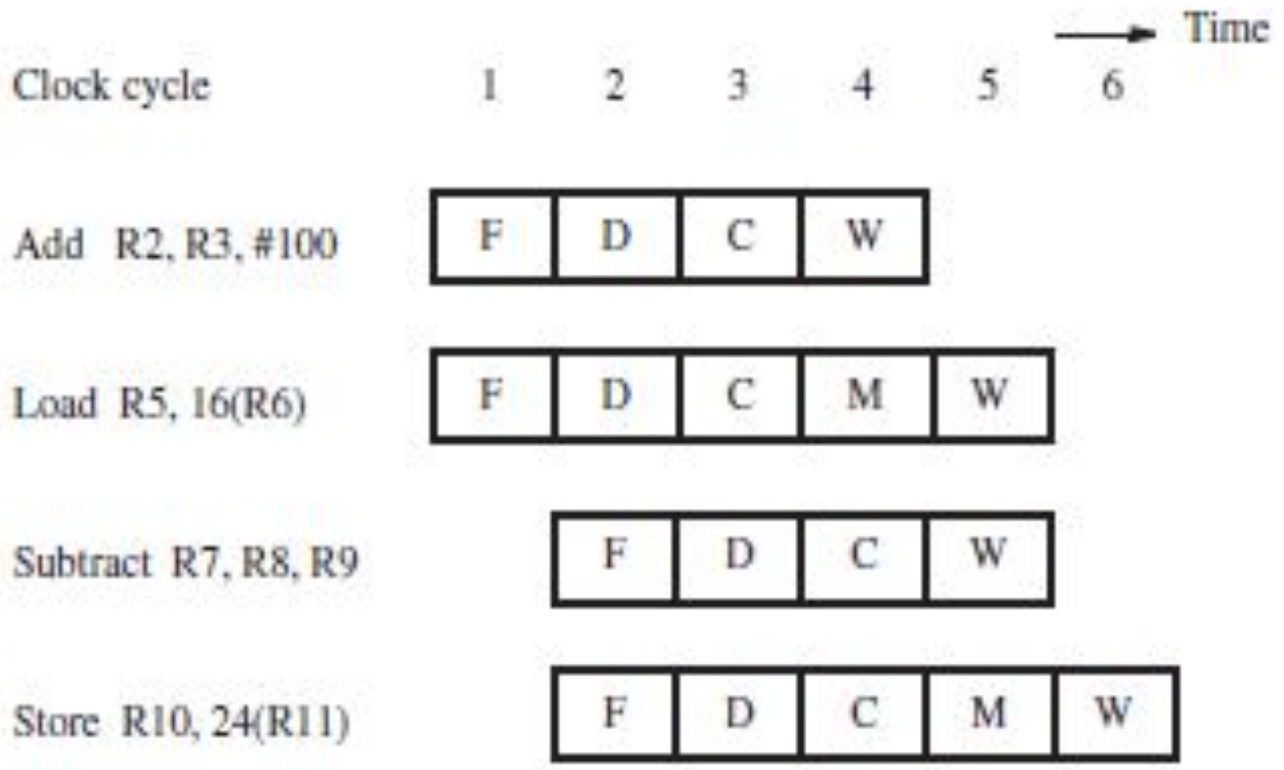
A five-stage pipeline can potentially increase the throughput by a factor of five.

In general, an n-stage pipeline has the potential to increase throughput n times.

Superscalar Operation



```
Add        R2, R3, #100
Load       R5, 16(R6)
Subtract   R7, R8, R9
Store      R10, 24(R11)
```



Pipelining in CISC Processors

The instruction set of a RISC processor makes pipelining relatively easy to implement.
All instructions are one word in size, and operand information is typically located in the same position within a word for different instructions.
No instruction requires more than one memory operand. Only Load and Store instructions access memory operands, typically using only indexed addressing
The availability of more complex addressing modes such as Autoincrement or Autodecrement introduces *side effects when executing instructions. Aside effect occurs when* a location other than that of the destination operand is also affected.

For example, the instruction

Move R5, (R8)+