Pipelining

www.sarithdivakar.info

Objectives

- Basics of pipelining and how it can lead to improved performance.
- Hazards that cause performance degradation and techniques to alleviate their effect on performance
- Role of *optimizing compilers*, which rearrange the sequence of instructions to maximize the benefits of pipelined execution
- Replicating hardware units in a *superscalar* processor so that multiple pipelines can operate concurrently.

Basic Concepts

- Pipelining overlaps the execution of successive instructions to achieve high performance
- The speed of execution of programs is improved by
  - using faster circuit technology
  - arranging the hardware so that more than one operation can be performed at the same time
- The number of operations performed per second is increased, even though the time needed to perform any one operation is not changed
- **Pipelining is a particularly effective way of organizing concurrent activity in a computer system.**

Pipelined Execution

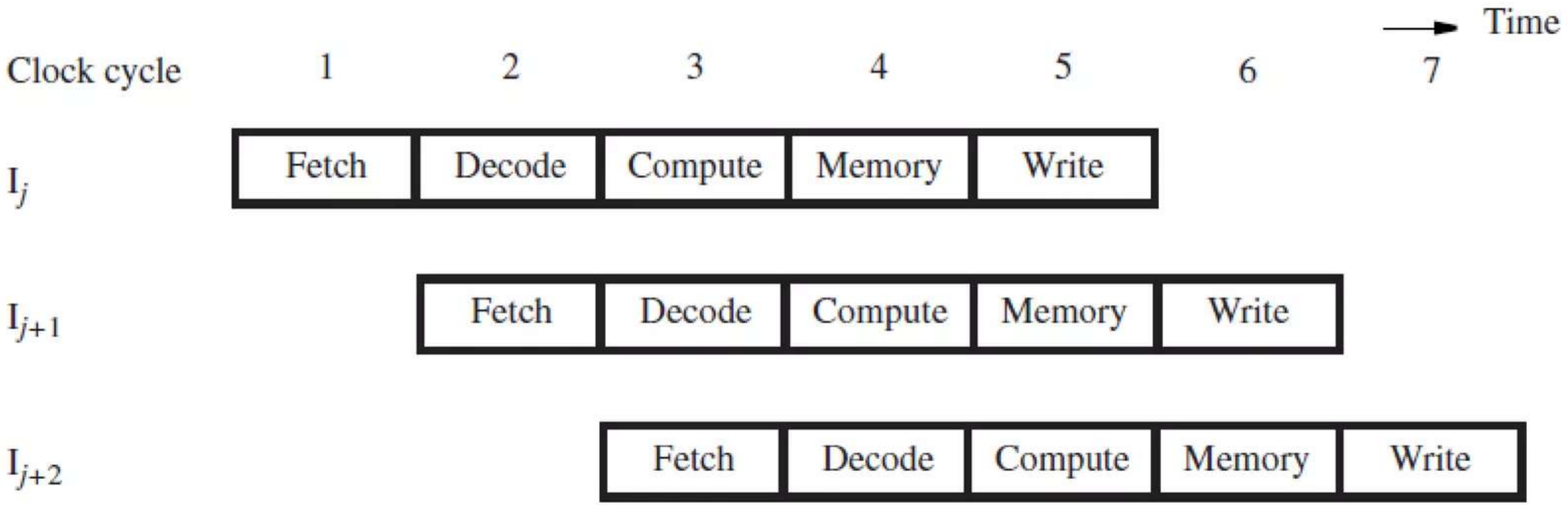
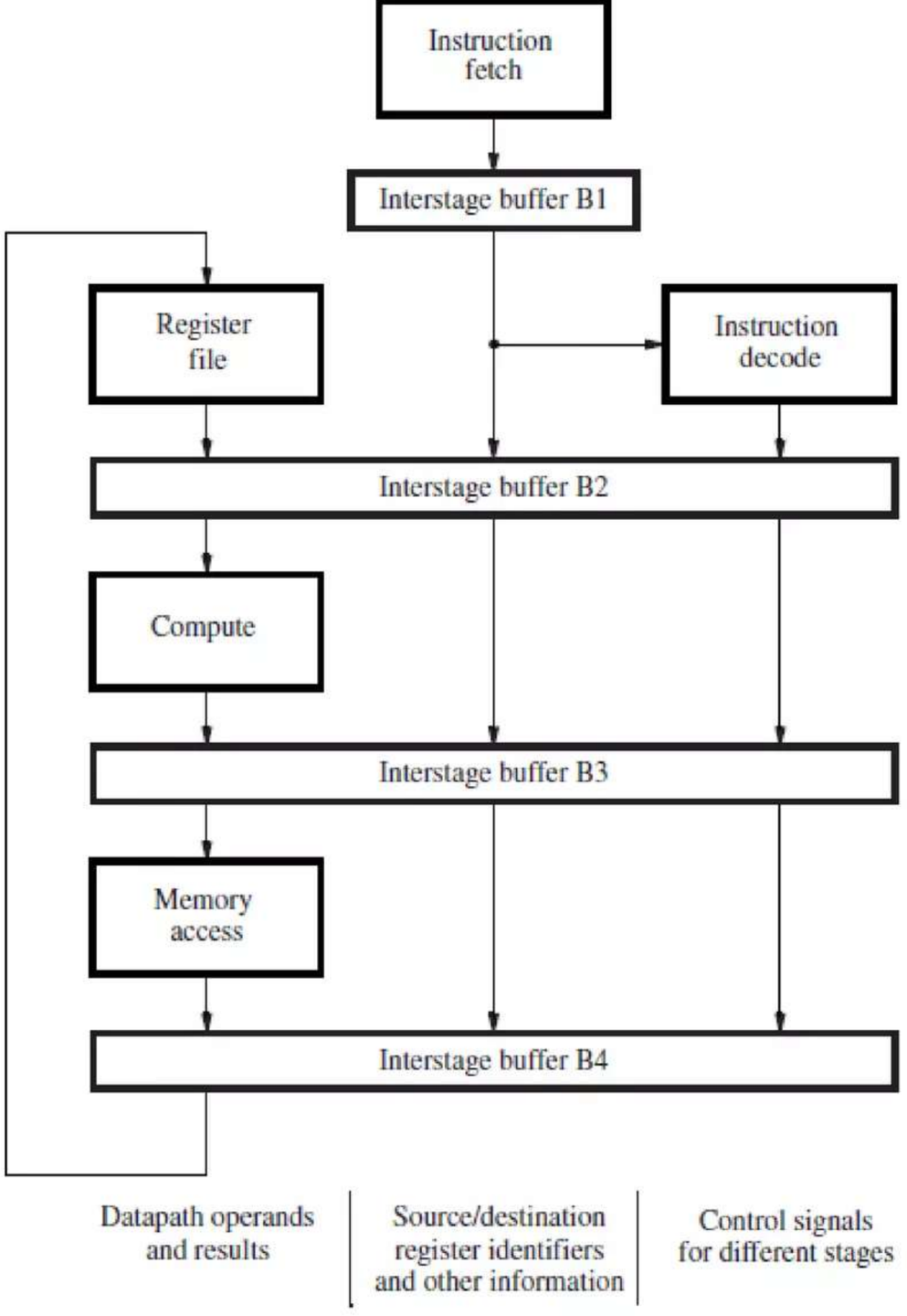


Figure 6.1 Pipelined execution—the ideal case.

Pipeline Organization


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

- Any condition that causes the pipeline to stall is called a *hazard*
- *Data hazard*: Value of a source operand of an instruction is not available when needed
- Other hazards arise from memory delays, branch instructions, and resource limitations

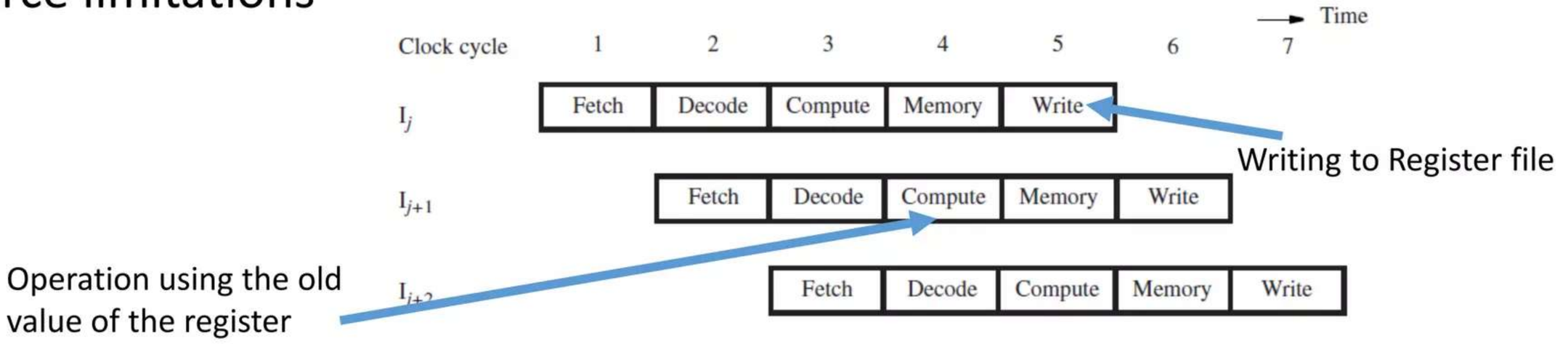


Figure 6.1 Pipelined execution—the ideal case.

Data Dependencies

- Add R2, R3, #100
- Subtract R9, R2, #30
- The Subtract instruction is stalled for three cycles to delay reading register R2 until cycle 6 when the new value becomes available.

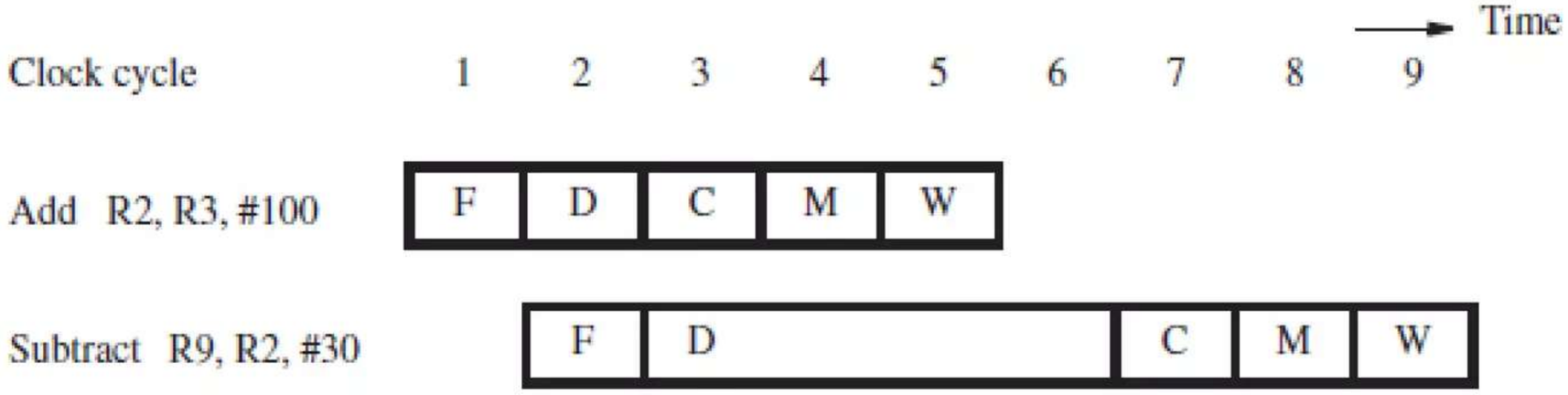


Figure 6.3 Pipeline stall due to data dependency.

Operand Forwarding

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

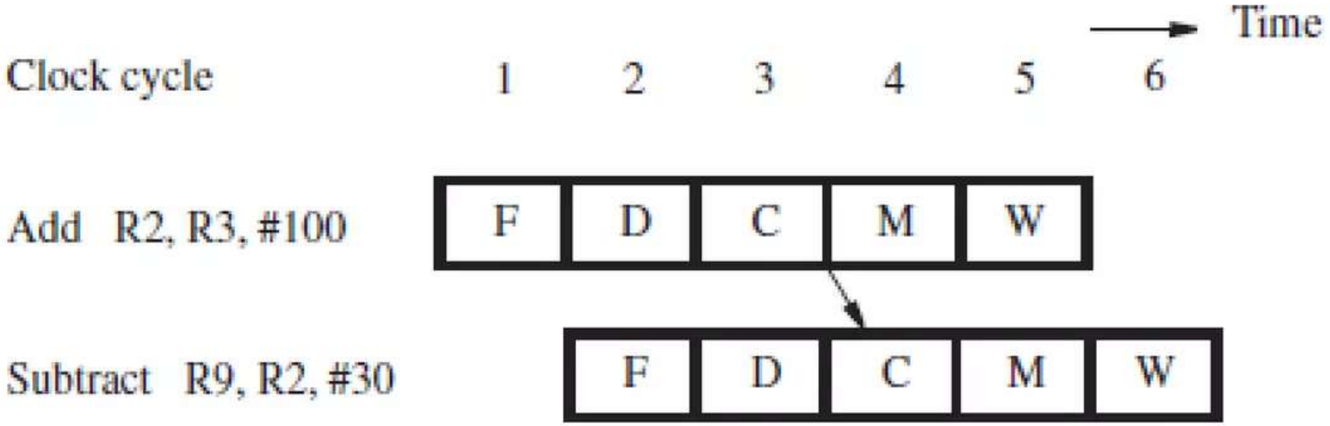


Figure 6.4 Avoiding a stall by using operand forwarding.

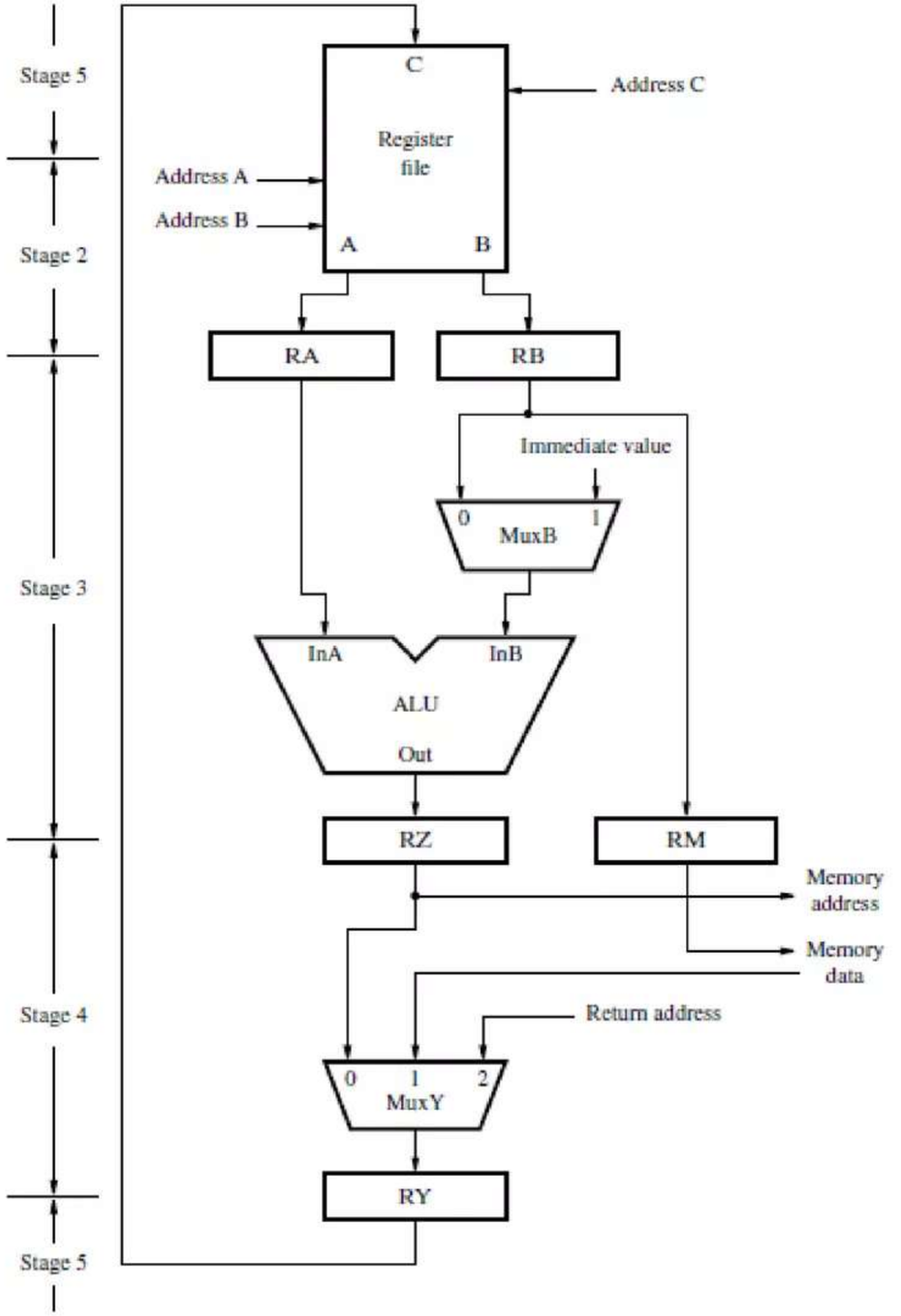


Figure 5.8 Datapath in a processor.

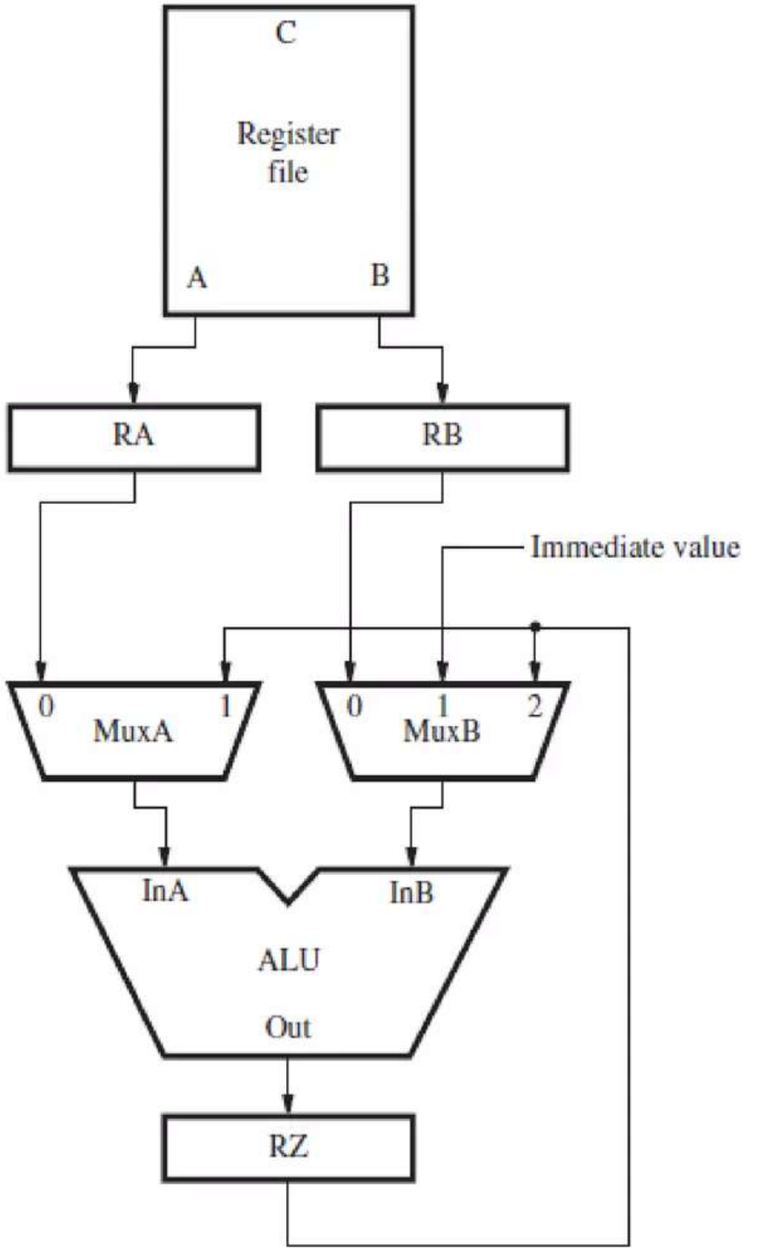
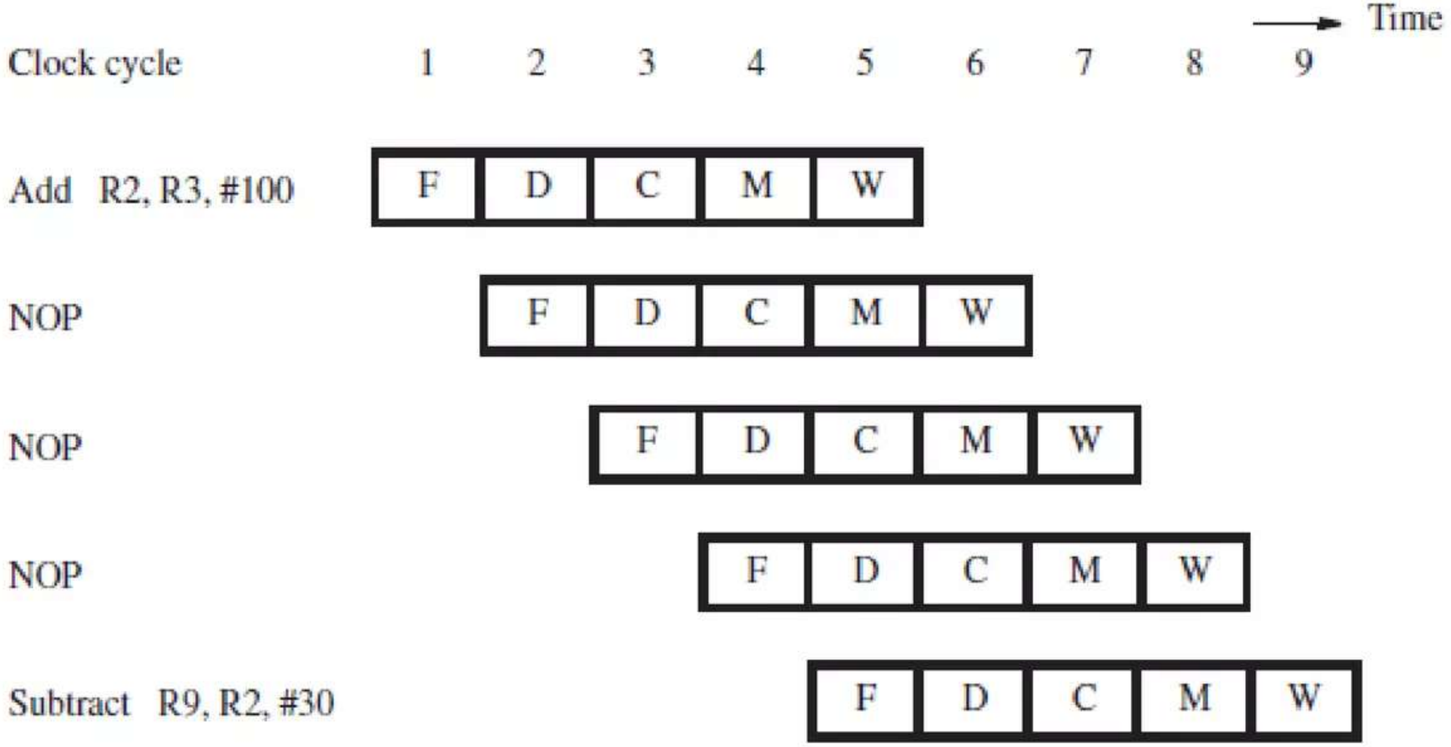


Figure 6.5 Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

Handling Data Dependencies in Software

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

- Code size increases
- Execution time not reduced



(b) Pipelined execution of instructions

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

The compiler can attempt to *optimize* the code to improve performance and reduce the code size by reordering instructions to move useful instructions into the NOP slots

Memory Delays

- Load instruction may require more than one clock cycle to obtain its operand from memory.
- Load R2, (R3)
- Subtract R9, R2, #30

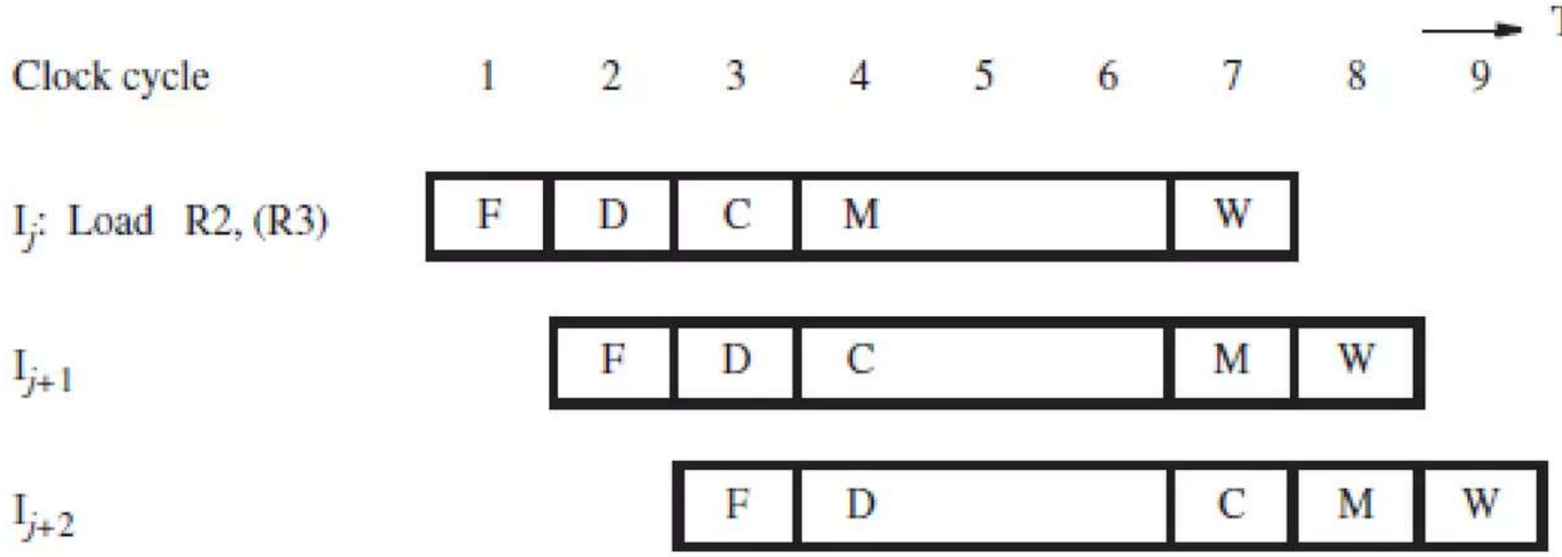


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

Operand forwarding

- Operand forwarding cannot be done because the data read from memory (the cache, in this case) are not available until they are loaded into register RY at the beginning of cycle 5

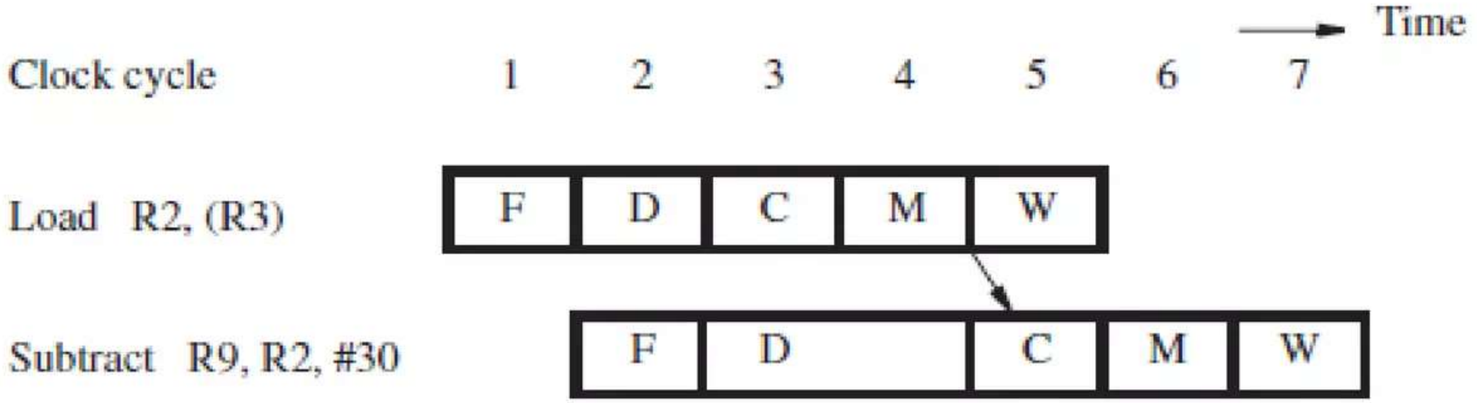


Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

The memory operand, which is now in register RY, can be forwarded to the ALU input in cycle 5.

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

With a two-cycle branch penalty, the relatively high frequency of branch instructions could increase the execution time for a program by as much as 40 percent.

Unconditional Branches

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

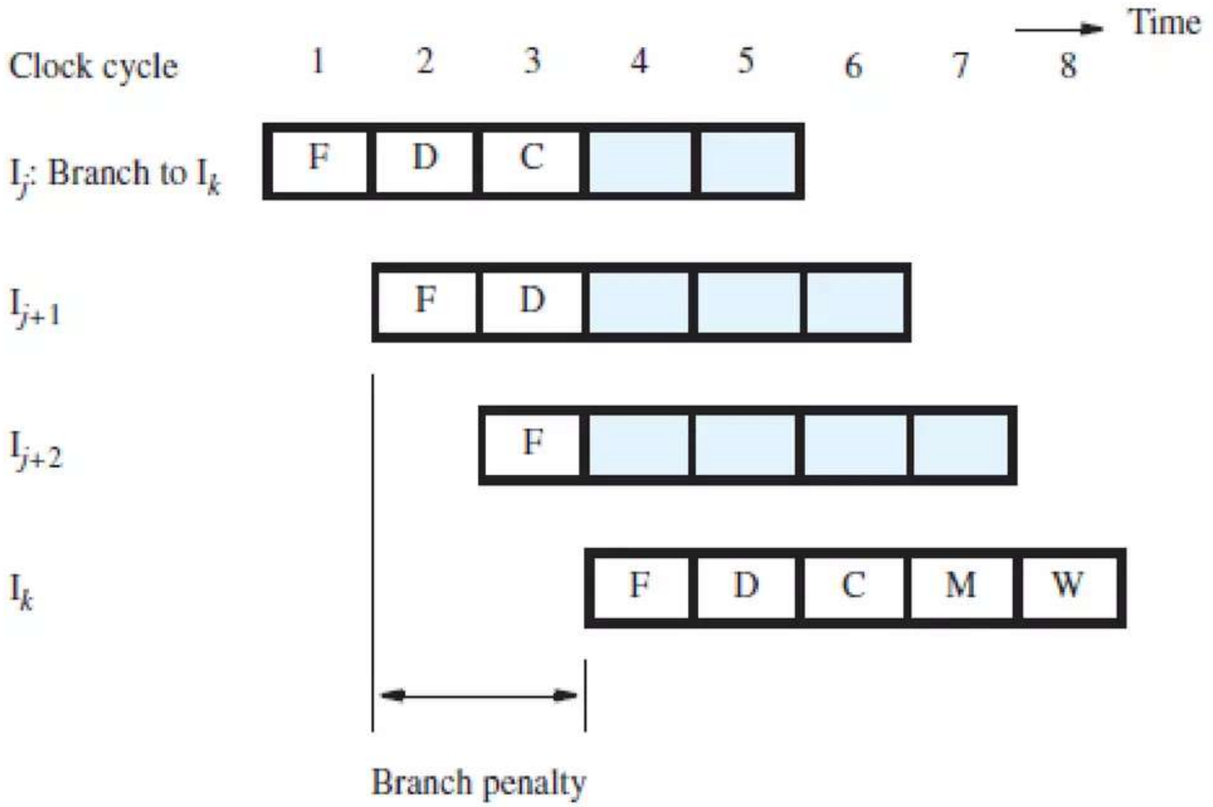


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

Target address is determined in the Decode stage of the pipeline.

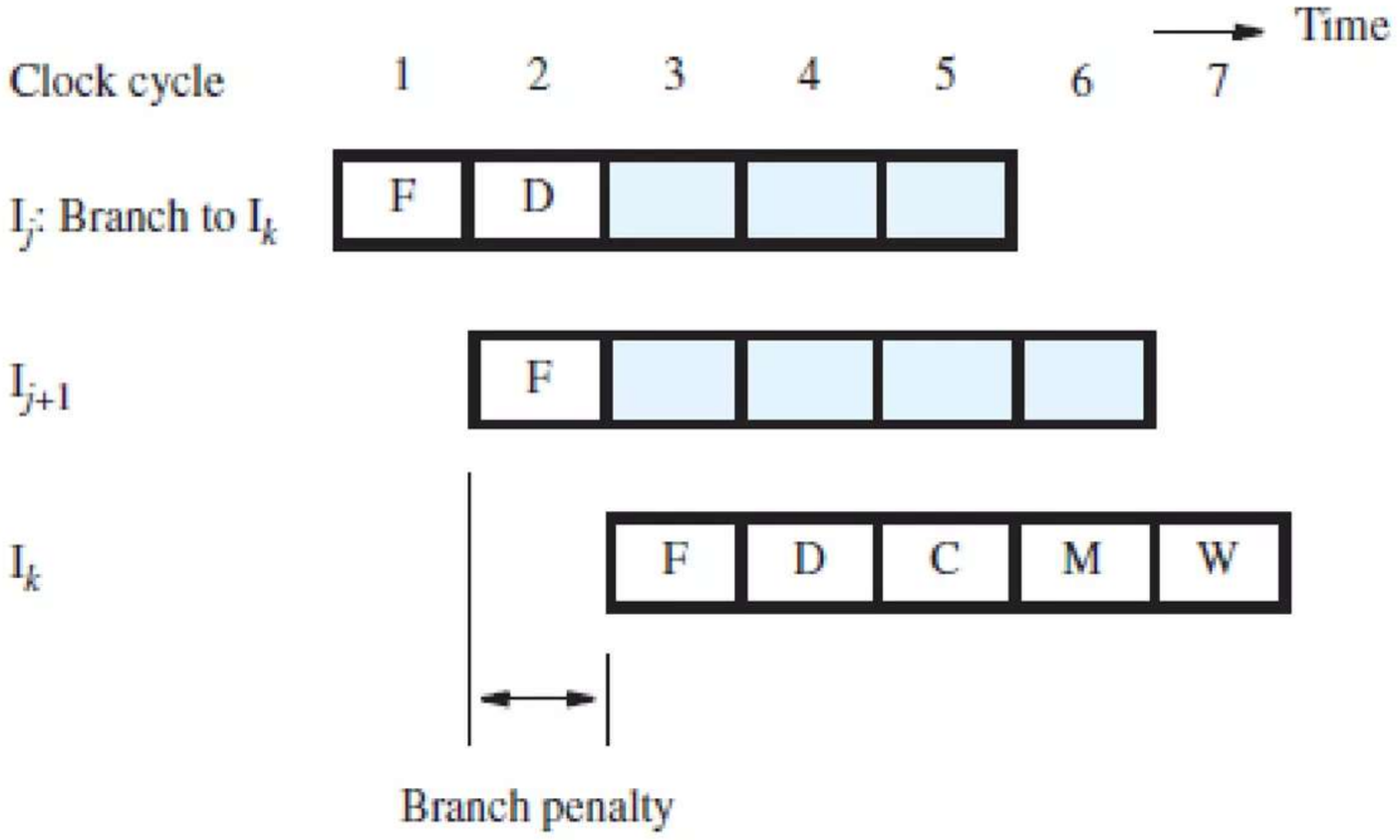


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

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

Figure 5.16 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 instruction, 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

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**
- **Dynamic Branch Prediction**

Static Branch Prediction

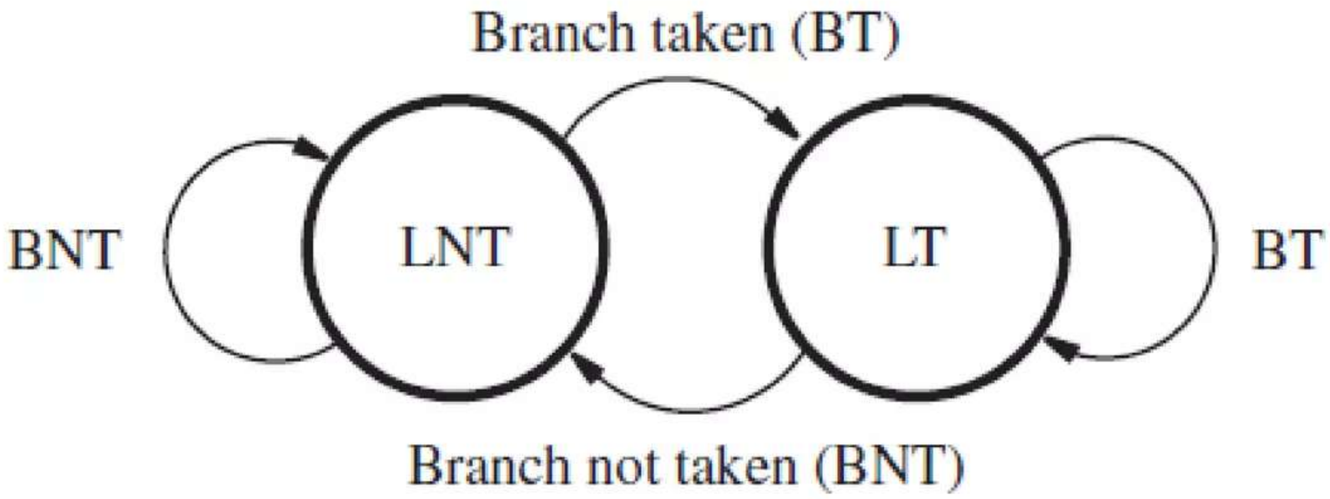
- The simplest form of branch prediction is to assume that the branch will not be taken and to fetch the next instruction in sequential address order
- If the prediction is correct, the fetched instruction is allowed to complete and there is no penalty
- Misprediction incurs the full branch penalty

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.

A 2-state algorithm

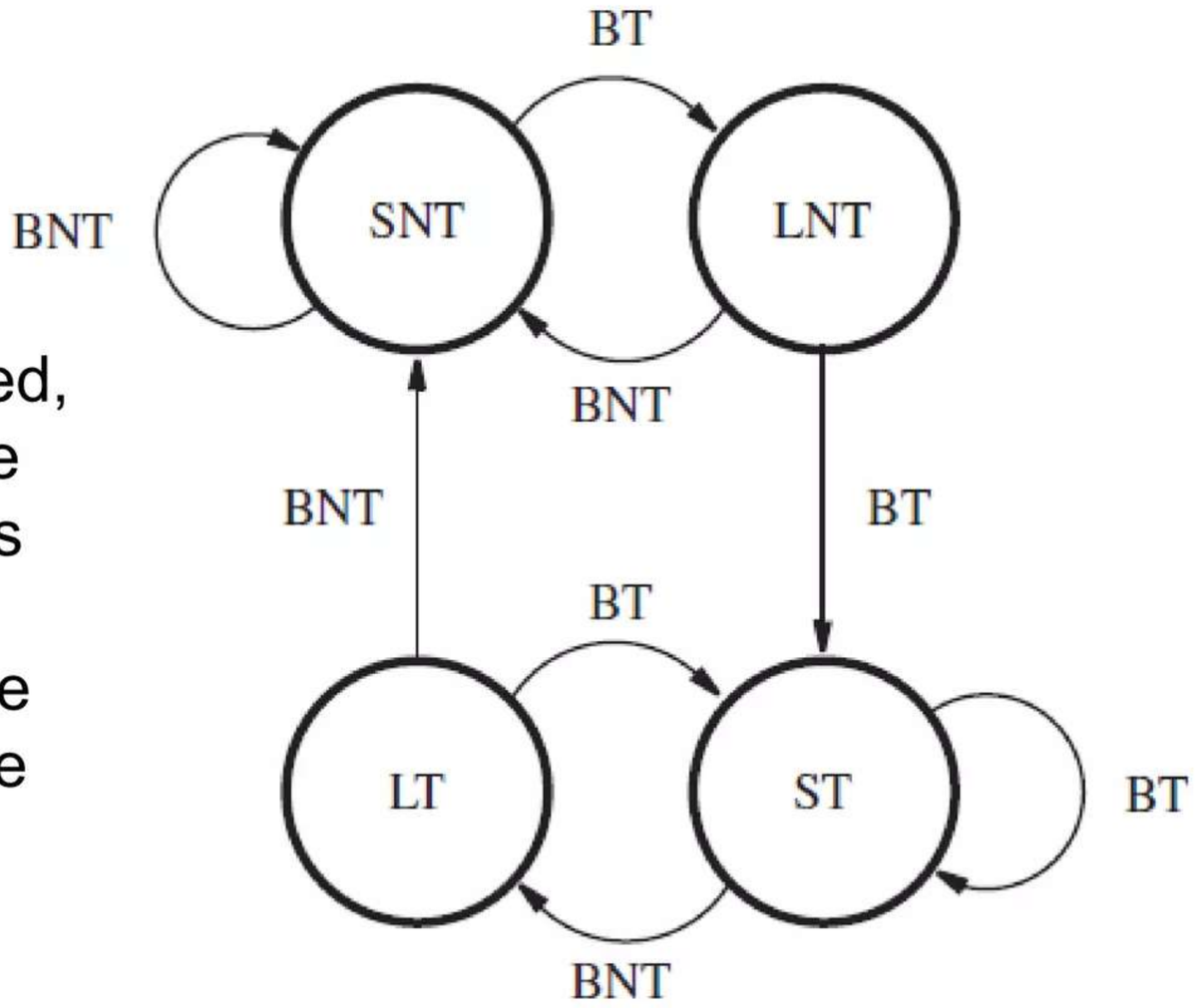
- When the branch instruction is executed and the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT.
- The next time the same instruction is encountered, the branch is predicted as taken if the state machine is in state LT. Otherwise it is predicted as not taken.



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



ST - Strongly likely to be taken
LT - Likely to be taken
LNT - Likely not to be taken
SNT - Strongly likely not to be taken

Performance Evaluation

- ***Basic performance equation***
- For a non-pipelined processor, the execution time, *T*, of a program that has a dynamic instr $T = \frac{N \times S}{R}$ t of *N* is given by

- *S* is the 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

n-stage pipeline has the potential to increase throughput *n* times

Assignment
Date of Submission: 27-Jun-2016

- *How much of this potential increase in instruction throughput can actually be realizedin practice?*
- *What is a good value for n?*

Superscalar Operation

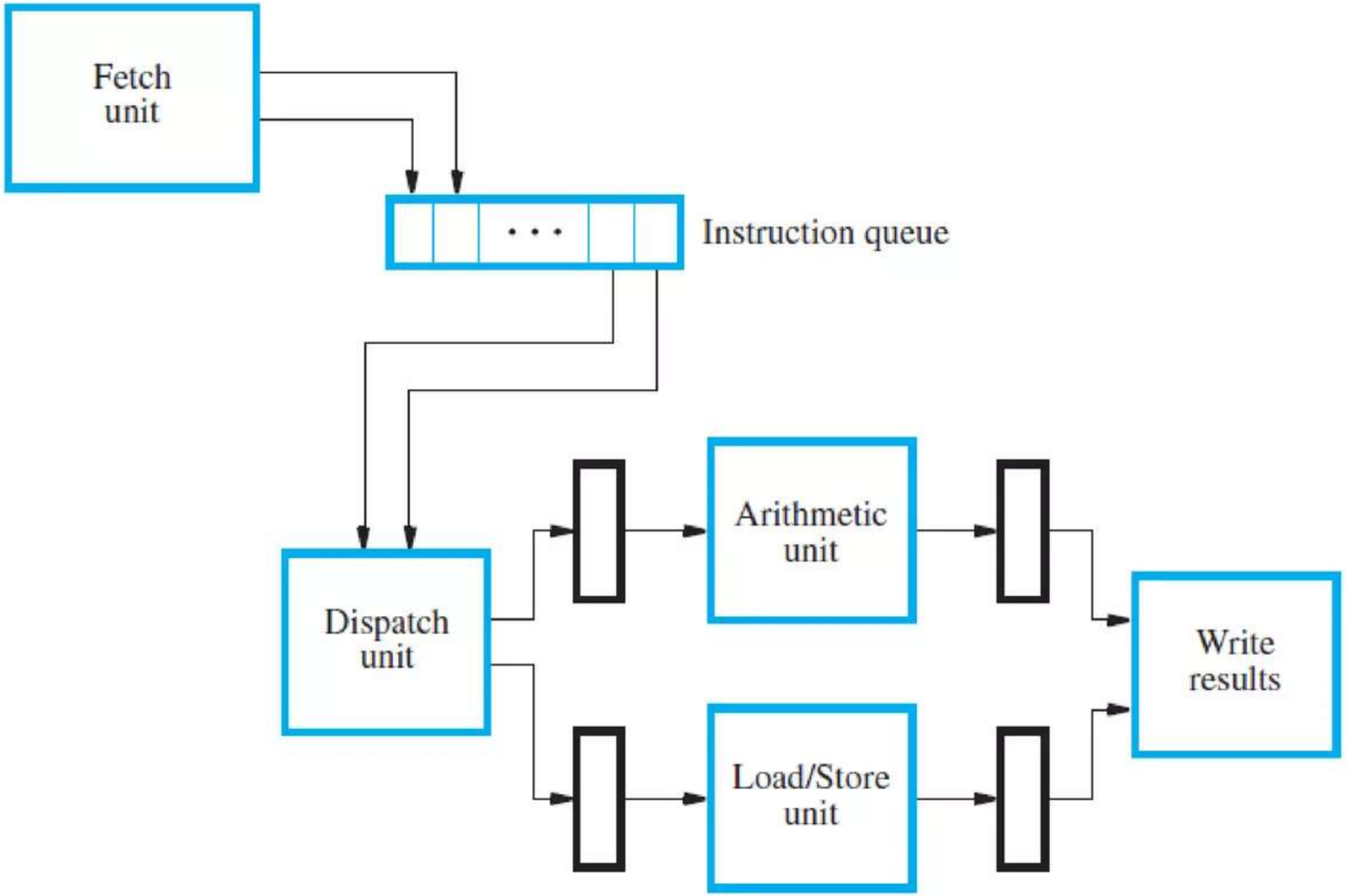


Figure 6.13 A superscalar processor with two execution units.

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

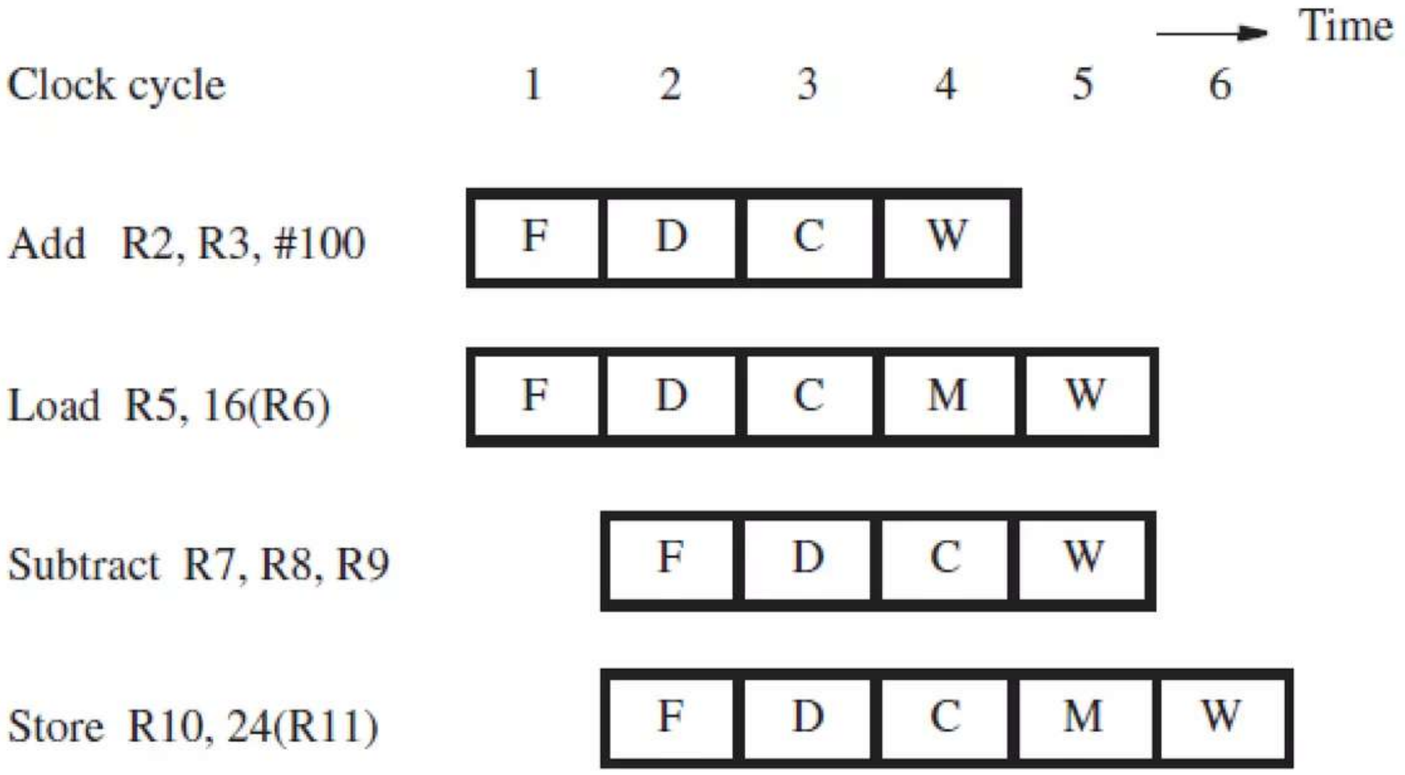


Figure 6.14 An example of instruction flow in the processor of Figure 6.13.

References

- "Computer Organization", Carl Hamacher Safwat Zaky Zvonko Vranesic, McGraw-Hill, 2002