Chapter-2

Pipelining

Pipelining

1.1 Basic Concept—The Ideal Case

1.2 Pipeline Organization

1.3 Pipelining Issues

1.4 Data Dependencies

- 1.4.1 Operand Forwarding
- 1.4.2 Handling Data Dependencies in Software

1.5 Memory Delays

1.6 Branch Delays

- 1.6.1 Unconditional Branches
- 1.6.2 Conditional Branches
- 1.6.3 The Branch Delay Slot
- 1.6.4 Branch Prediction

1.7 Resource Limitations

Pipelining

Pipelining is the process of storing and prioritizing computer instructions that the processor executes. The pipeline is a "logical pipeline" that lets the processor perform an instruction in multiple steps. The processing happens in a continuous, orderly, somewhat overlapped manner.

Basic Concept—The Ideal Case(Overview)

So far, we have assumed that only one instruction is being processed by the multi-stage hardware at any point of time

The speed of execution of programs is influenced by many factors

• How do we decrease the execution time of a program?

One possibility is to use faster circuits to implement the processor

– This approach will decrease the execution time of each instruction

• Another possibility is to arrange the processor hardware in such a way that multiple instructions can be processed at the same time. This approach is called pipelining

Pipelining does not change the time needed to perform a single instruction, but it increases the number of instructions performed per second (instruction completion rate or throughput)

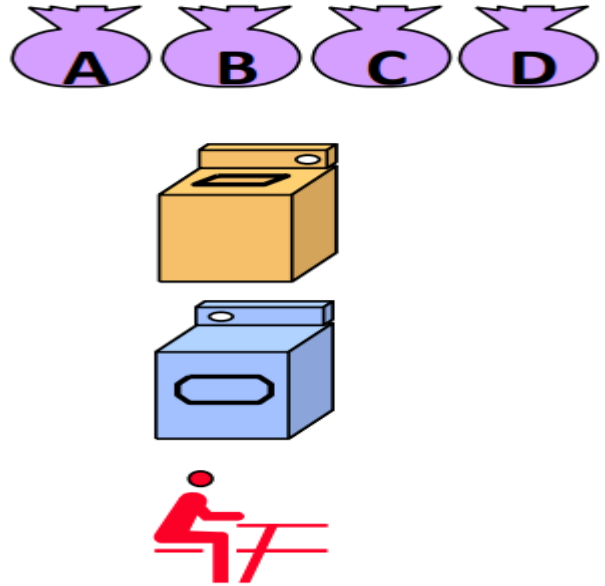
Pipelining is a particularly effective way of organizing concurrent activity in a computer system.

The basic idea is very simple. It is frequently encountered in manufacturing plants or Laundry, where pipelining is commonly known as an assembly-line operation.

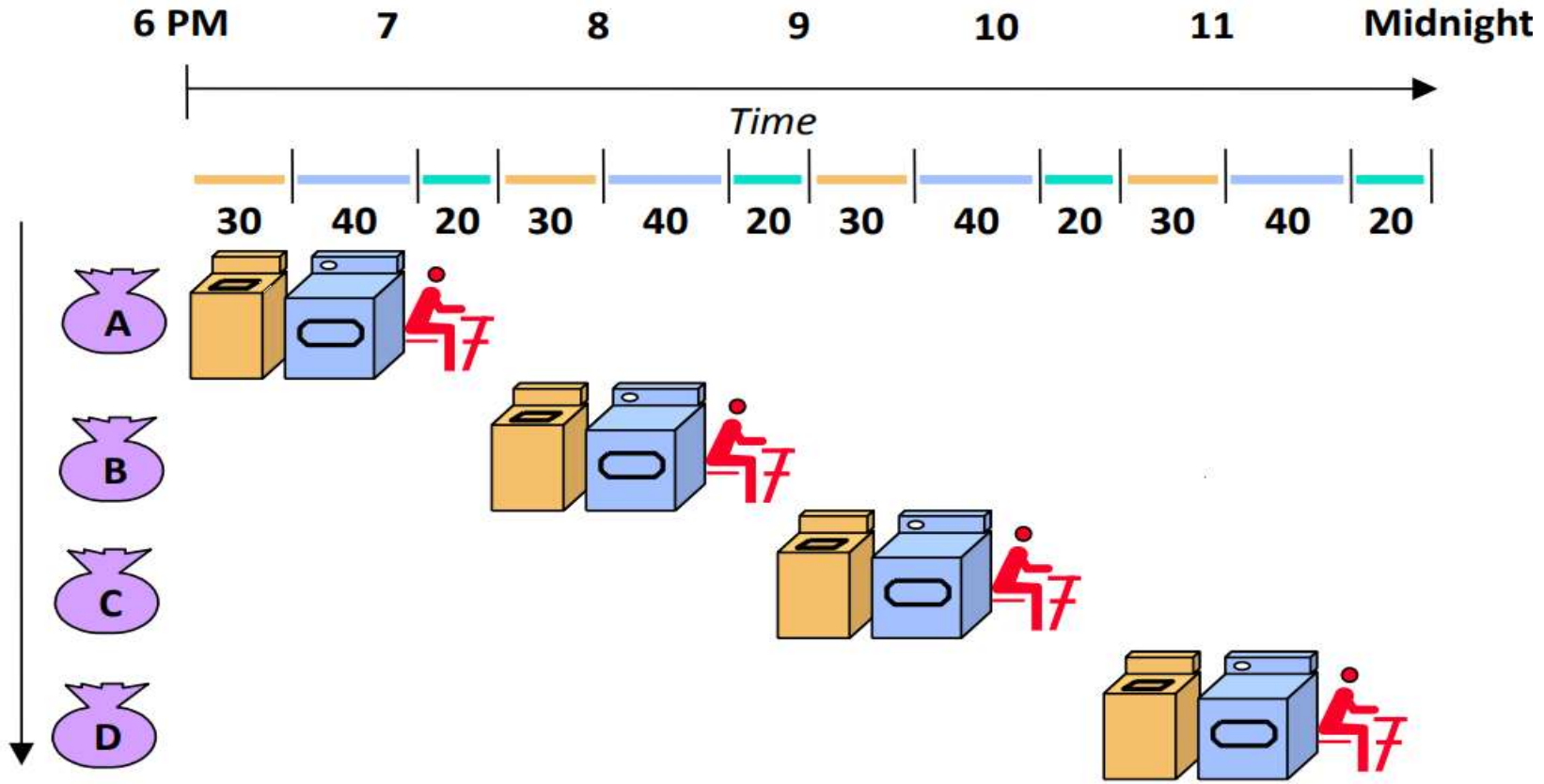
Traditional Pipelining Example

Laundry Example

- Four loads of laundry need to be washed, dried and folded
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

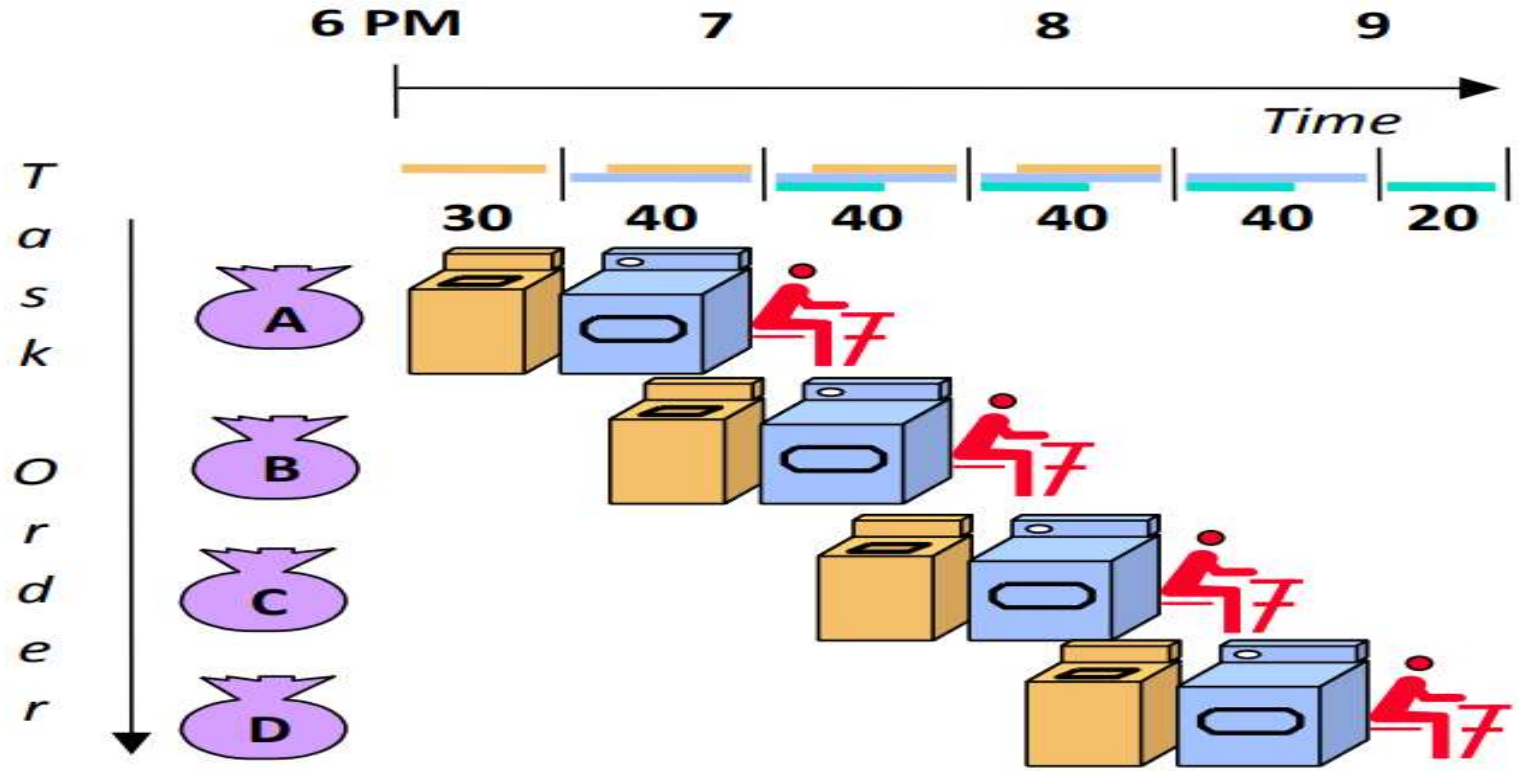


Sequential laundry takes 6 hours for 4 loads



If pipelining is used, how long would laundry take?

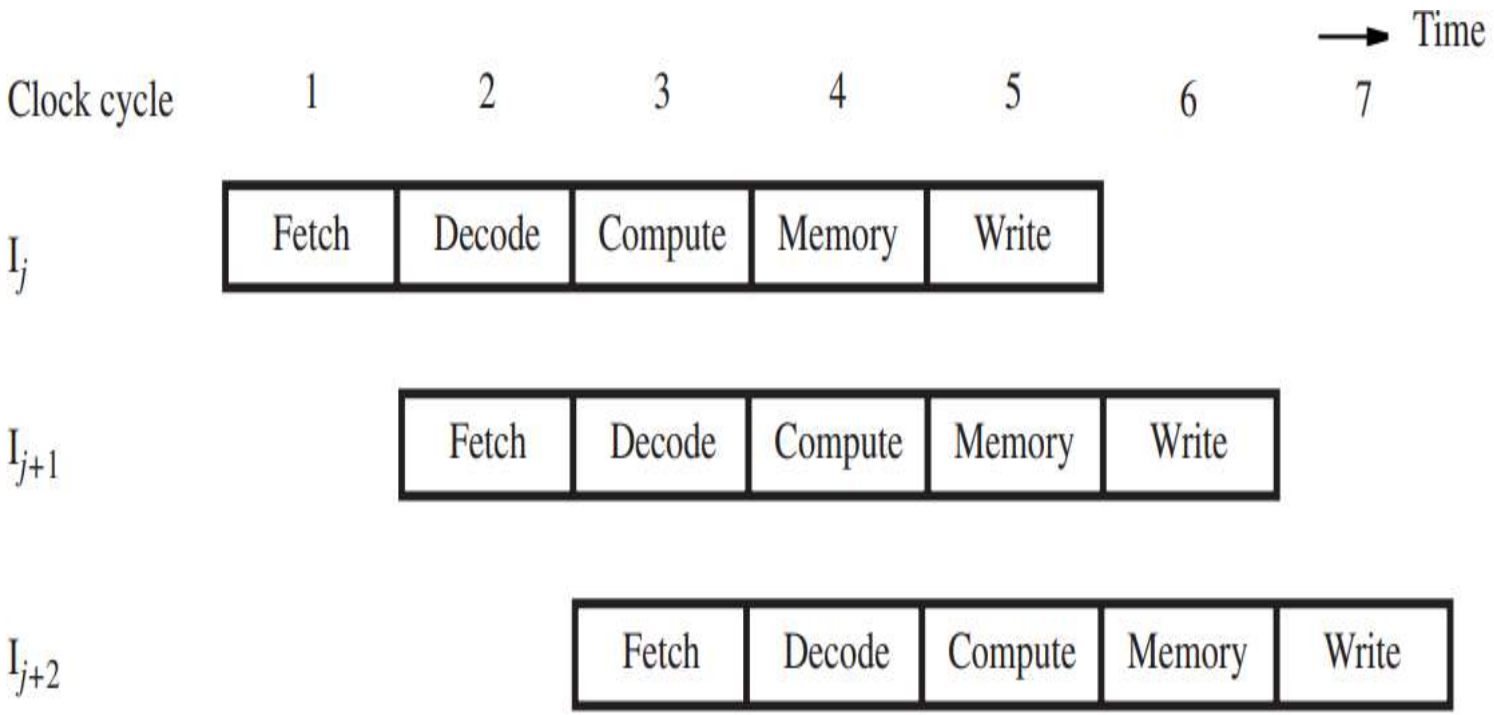
- Pipelined laundry takes 3.5 hours for 4 loads

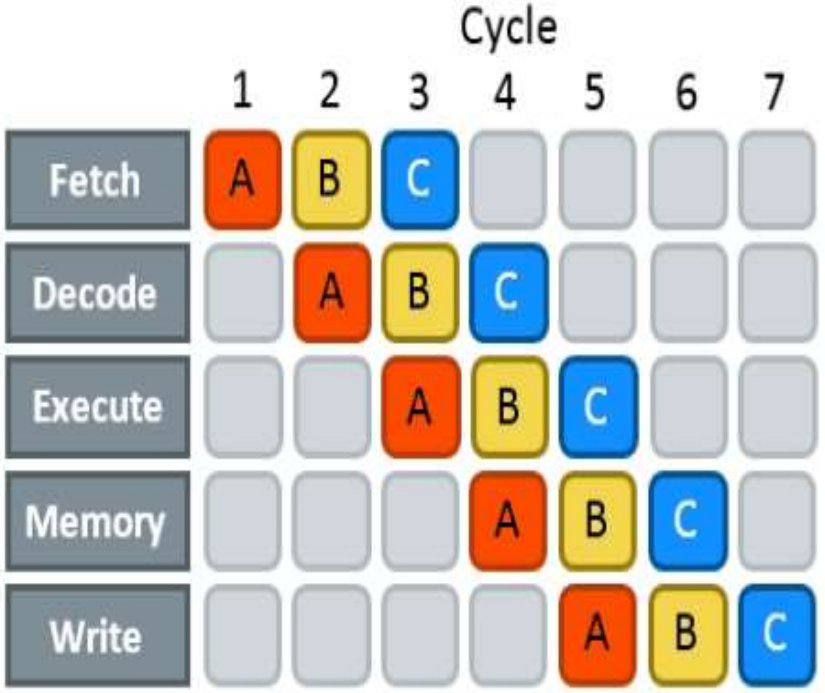


Pipelining doesn't reduce the time taken by an individual task, it improves the throughput of entire workload

- Task completion rate limited by slowest pipeline stage
- Potential speedup = Number of pipeline stages
- Unbalanced lengths of pipeline stages reduces speedup
- Time to "fill" pipeline and time to "drain" it further reduces speedup

Pipelined execution Fig 6.1





The five stages corresponding to the labeled as Fetch, Decode, Compute, Memory, and Write.

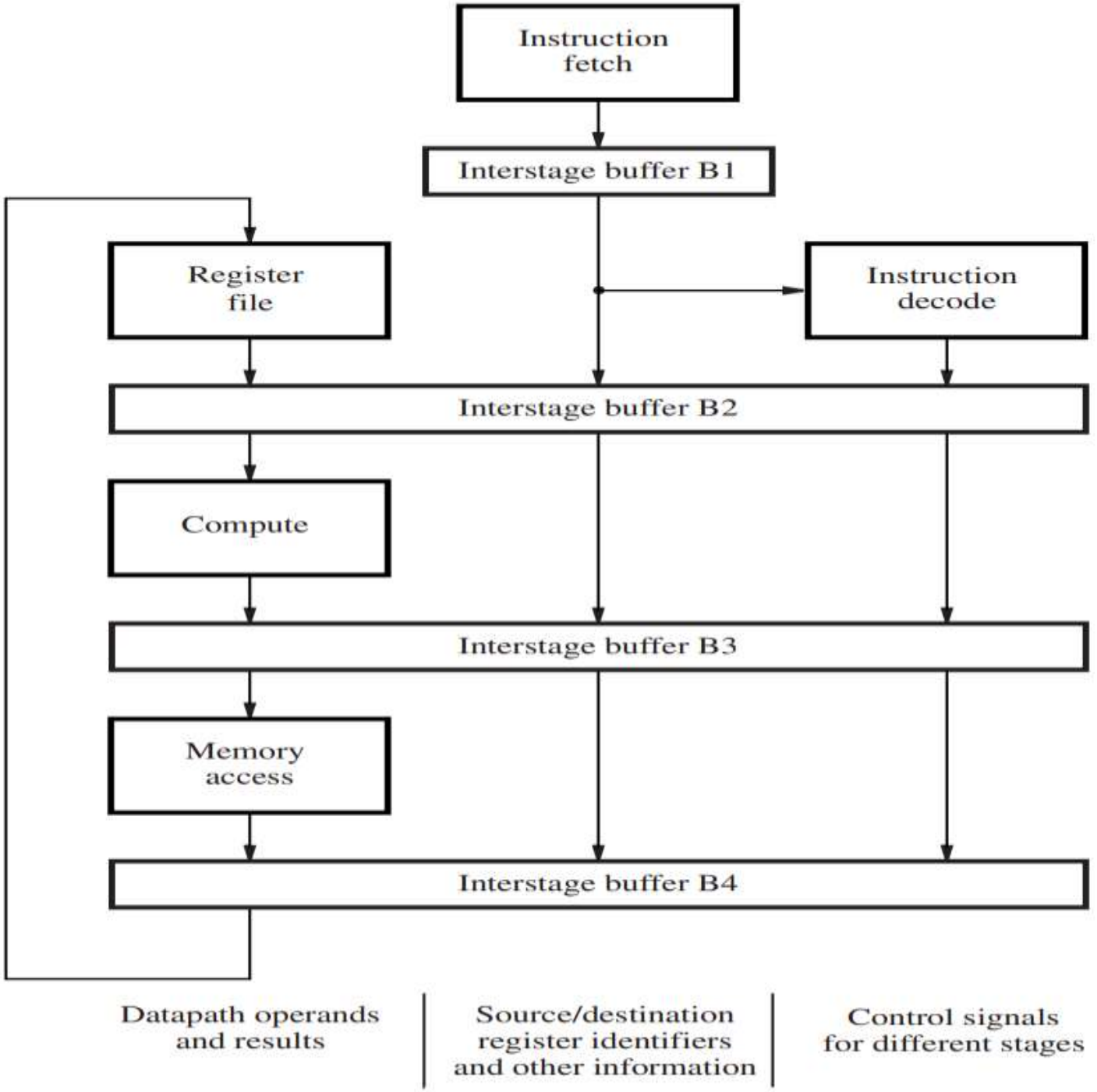
Instruction Ij is fetched in the first cycle and moves through the remaining stages in the following cycles.

In the second cycle, instruction Ij+1 is fetched while instruction Ij is in the Decode stage where its operands are also read from the register file.

In the third cycle, instruction Ij+2 is fetched while instruction Ij+1 is in the Decode stage and instruction Ij is in the Compute stage where an arithmetic or logic operation is performed on its operands.

Ideally, this overlapping pattern of execution would be possible for all instructions. Although any one instruction takes five cycles to complete its execution, instructions are completed at the rate of one per cycle

Pipeline Organization



The interstage buffers are used as follows:

• Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.

• Interstage buffer B2 feeds the Compute stage with the two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder.

The settings for control signals move through the pipeline to determine the ALU operation, the memory operation, and a possible write into the register file.

• Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage.

In the case of a write access to memory, buffer B3 holds the data to be written. These data were read from the register file in the Decode stage. The buffer also holds the incremented PC value passed from the previous stage, in case it is needed as the return address for a subroutine-call instruction.

• Interstage buffer B4 feeds the Write stage with a value to be written into the register file.

This value may be the ALU result from the Compute stage, the result of the Memory access stage, or the incremented PC value that is used as the return address for a subroutine-call instruction.

Pipelining Issues

- Figure 6.1 depicts the ideal overlap of three successive instructions. But, there are times when it is not possible to have a new instruction enter the pipeline in every cycle.
- Consider the case of two instructions, Ij and Ij+1, where the destination register for instruction Ij is a source register for instruction Ij+1.
- The result of instruction Ij is not written into the register file until cycle 5, but it is needed earlier in cycle 3 when the source operand is read for instruction Ij+1.
- If execution proceeds as shown in Figure 6.1, the result of instruction Ij+1 would be incorrect because the arithmetic operation would be performed using the old value of the register in question.
- To obtain the correct result, it is necessary to wait until the new value is written into the register by instruction Ij.
- Hence, instruction Ij+1 cannot read its operand until cycle 6, which means it must be stalled in the Decode stage for three cycles.
- While instruction Ij+1 is stalled, instruction Ij+2 and all subsequent instructions are similarly delayed. New instructions cannot enter the pipeline, and the total execution time is increased.

- Any condition that causes the pipeline to stall is called a hazard.
- Pipeline stalls cause degradation in pipeline performance. We need to identify all pipeline hazards and find ways to minimize their impact

Types of Pipeline Hazards There are three types of pipeline hazards:

- Data hazard – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline

– Instruction processing needs to be delayed until the operands become available

- Instruction (control) hazard – a delay in the availability of an instruction or the memory address needed to fetch the instruction

- Structural hazard – a situation where two (or more) instructions require the use of a given hardware resource at the same time.

Data Dependencies (Data Hazards)

Consider the two instructions :

Add R2, R3, #100

Subtract R9, R2, #30

The destination register (R2) for the first instruction is a source register for the second instruction

- Register R2 carries data from the first instruction to the second instruction --> There is a data dependency between these two instructions
- First instruction writes to register R2 in stage-5 of the pipeline (Writeback stage)
- Second instruction reads register R2 in stage-2 of the pipeline (decode stage)
- If second instruction reads R2 before the first instruction writes R2, the result of second instruction would be incorrect, as it would be based on R2's old value
- To obtain the correct result, second instruction needs to wait until the first instruction has written to R2

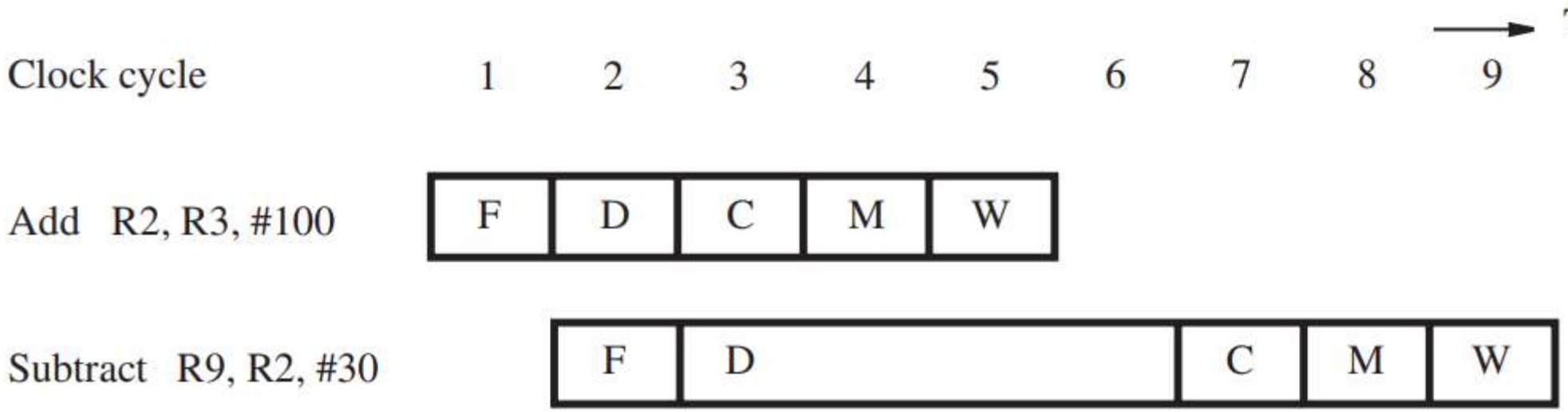


Figure 6.3 Pipeline stall due to data dependency.

- Second instruction is stalled in decode stage for three additional cycles to delay reading R2 until the new value of R2 has been written by first instruction
- How is this pipeline stall implemented in hardware?
- Control circuitry recognizes the data dependency when it decodes the second instruction (comparing source register ids in inter-stage buffer B1 with destination register id in inter-stage buffer B2)

During cycles 3 to 5:

• first instruction proceeds through the pipe

• second instruction is held in inter-stage buffer B1 • control circuitry inserts NOP (no-operation) instructions in buffer B2.

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 (Alleviating Data Hazards)

- Pipeline stalls due to data dependencies can be alleviated through the use of operand forwarding.
- Consider the pair of instructions Add R2, R3, #100 Subtract R9, R2, #30, where the pipeline is stalled for three cycles to enable the Subtract instruction to use the new value in register R2.
- The desired value is actually available at the end of cycle 3, when the ALU completes the operation for the Add instruction. This value is loaded into register RZ, which is a part of interstage buffer B3.
- Rather than stall the Subtract instruction, the hardware can forward the value from register RZ to where it is needed in cycle 4, which is the ALU input.

The arrow in below figure shows data being forwarded from C stage of first instruction to C stage of the second instruction

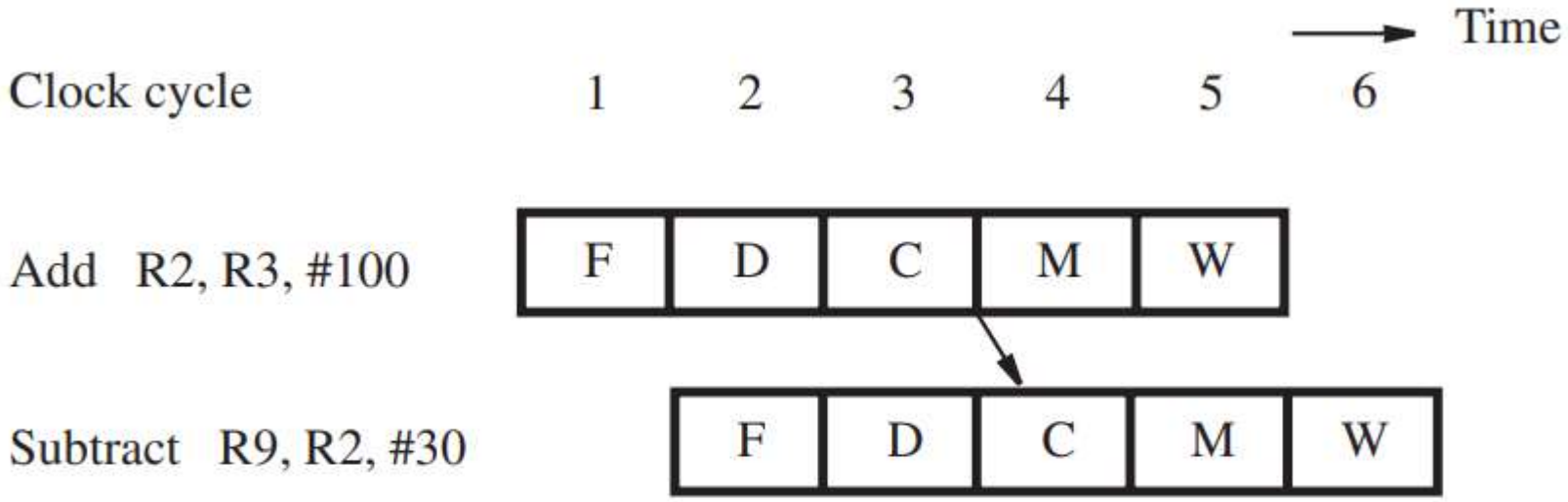


Figure 6.4 Avoiding a stall by using operand forwarding.

Figure 6.5 shows the modification needed in the datapath to make this forwarding possible.

A new multiplexer, MuxA, is inserted before input InA of the ALU, and the existing multiplexer MuxB is expanded with another input.

The multiplexers select either a value read from the register file in the normal manner, or the value available in register RZ.

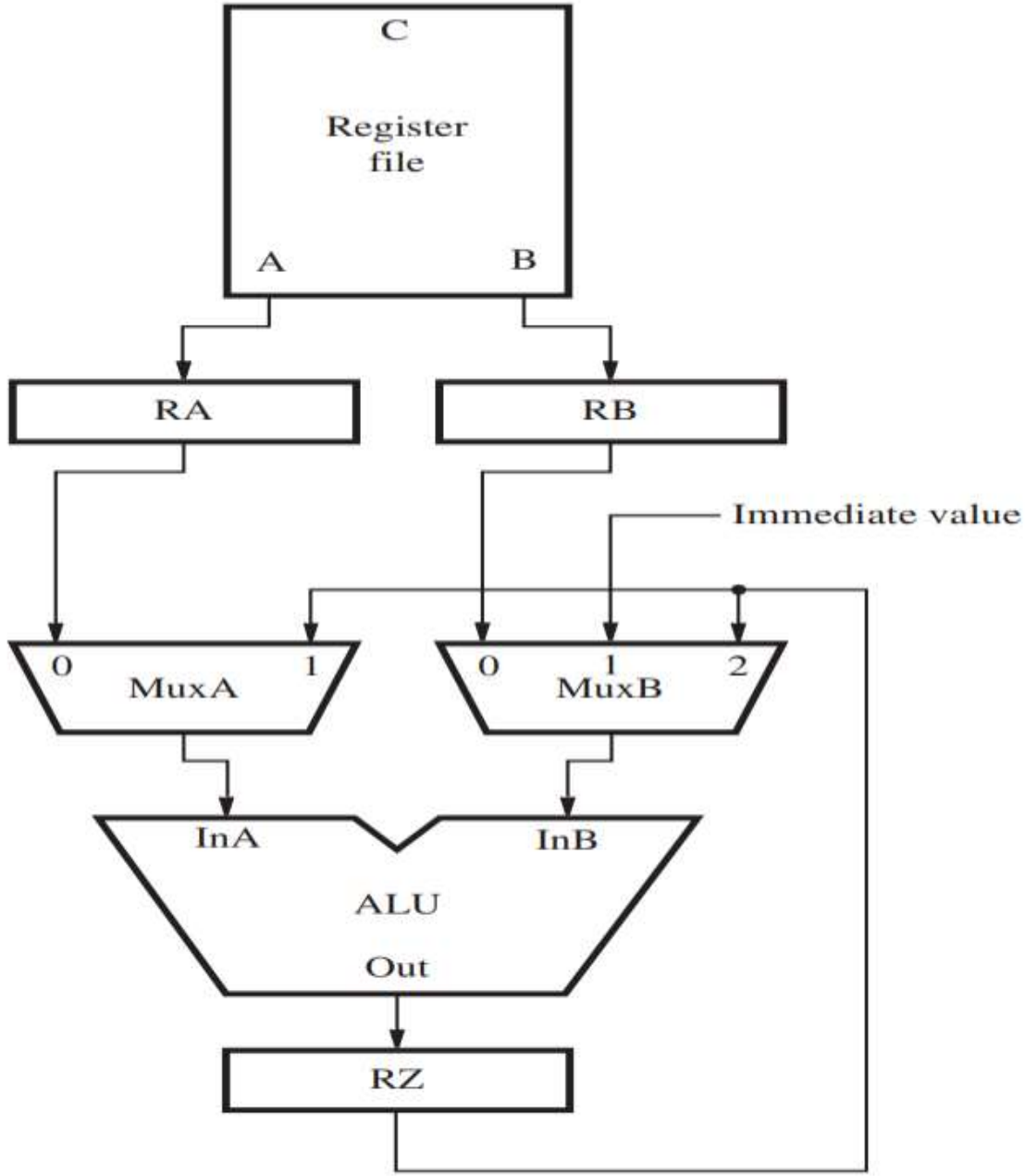


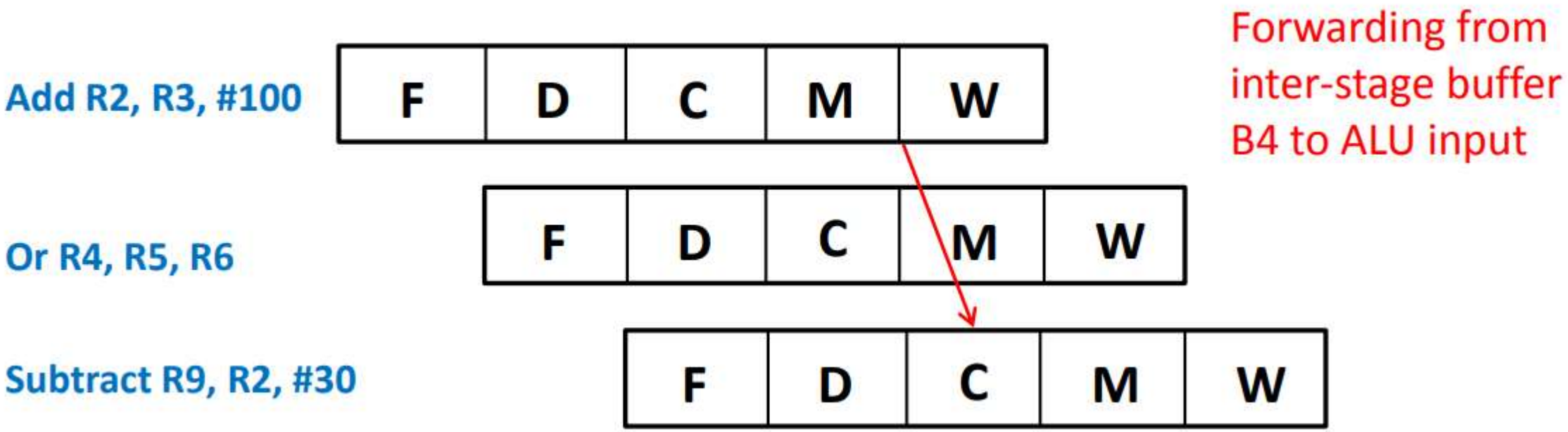
Figure 6.5 Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

- So far, we have looked at forwarding from C stage to C stage
- Such forwarding mitigates data hazards between successive instructions

How can we avoid stalls when instruction Ij+2 depends on instruction Ij ?

Solution: Forward operands from M stage to C stage

Example:



Handling Data Dependencies in Software

Figures 6.3 and 6.4 show how data dependencies may be handled by the processor hardware, either by stalling the pipeline or by forwarding data.

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 (No-operation) instructions between them.

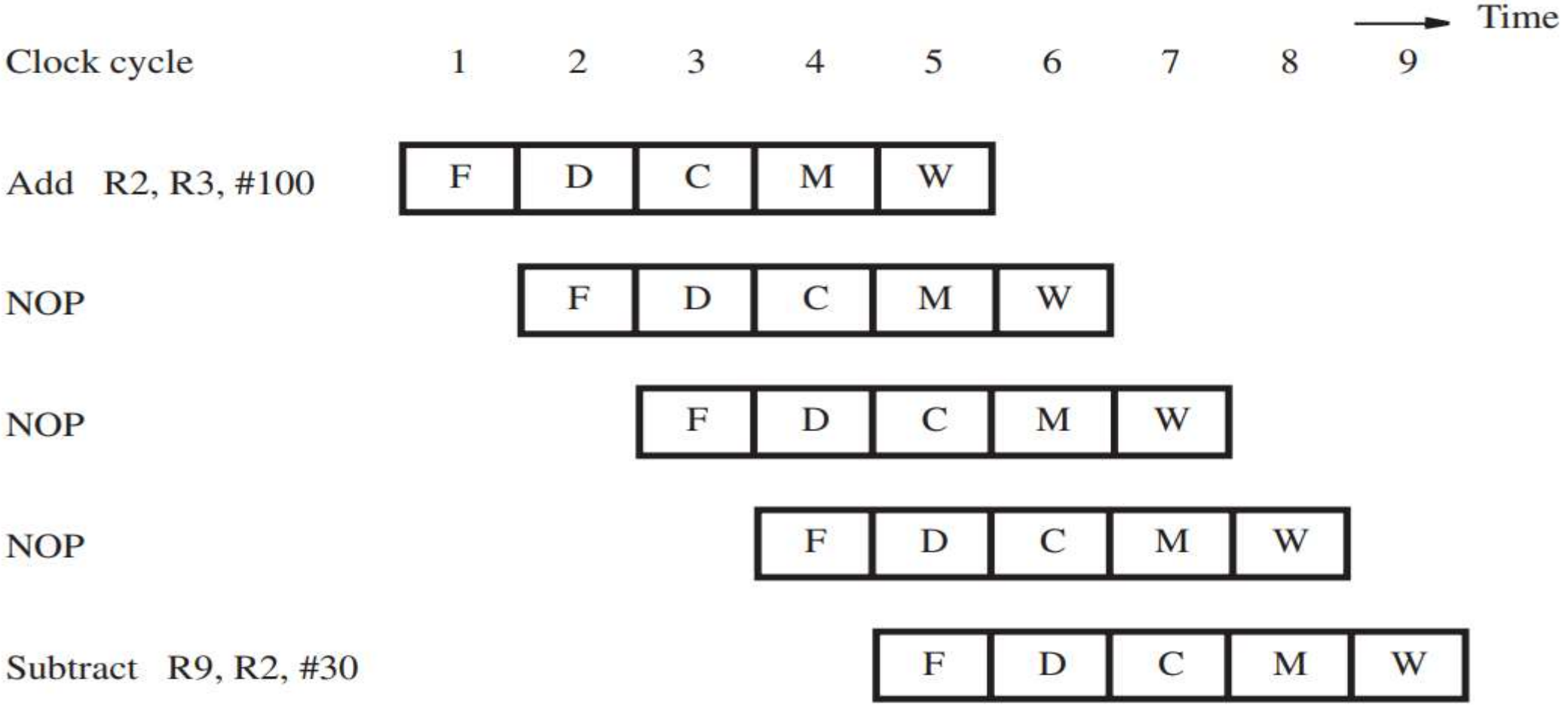
The NOPs introduce the necessary delay to enable instruction Ij+1 to read the new value from the register file after it is written.

For the instructions in Figure 6.4, the compiler would generate the instruction sequence in

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

(a) Insertion of NOP instructions for a data dependency

Figure 6.6b shows that the three NOP instructions have the same effect on execution time as the stall in Pipeline stall due to data dependency.



(b) Pipelined execution of instructions

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

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.

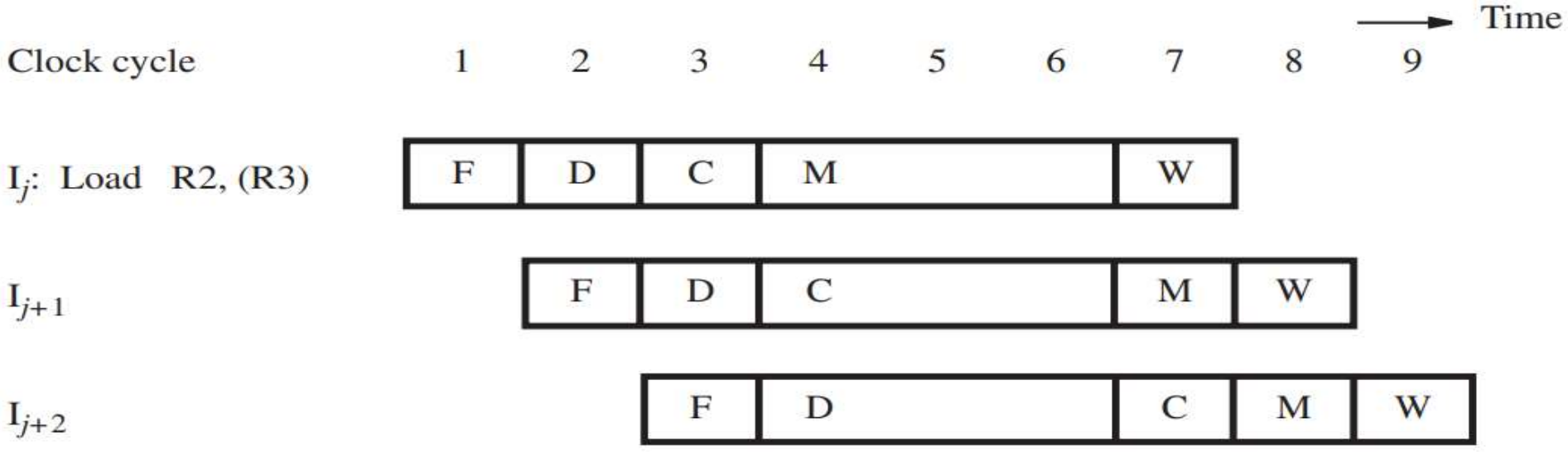


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

Figure shows the effect of a delay in accessing data in the memory on pipelined execution. A memory access may take ten or more cycles. For simplicity, the figure shows only three 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.

There is an additional type of memory-related stall that occurs when there is a data dependency involving a Load instruction. Consider the instructions:

Load R2, (R3)

Subtract R9, R2, #30

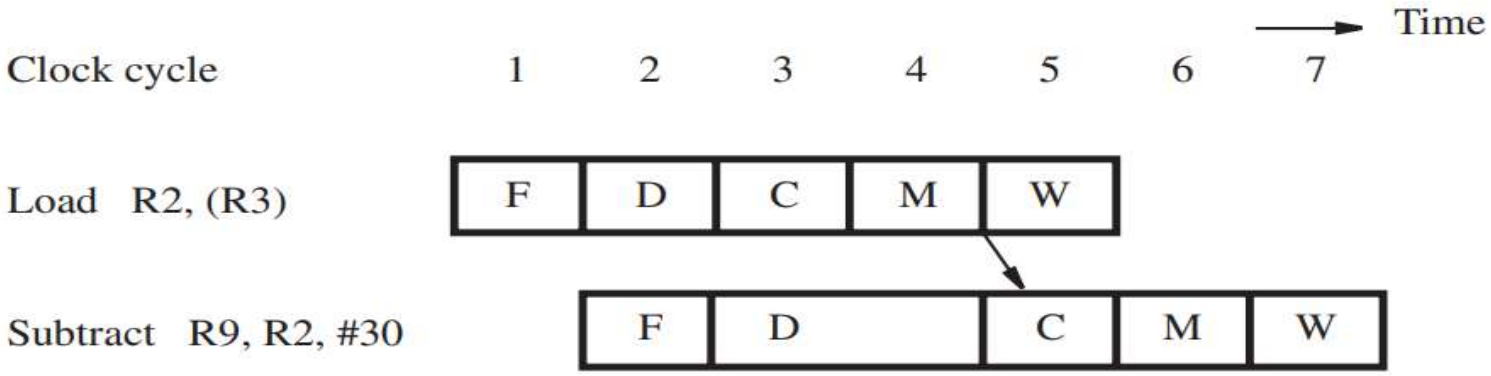


Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

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

The compiler can eliminate the one-cycle stall for this type of data dependency by reordering instructions to insert a useful instruction between the Load instruction and the instruction that depends on the data read from the memory.

The inserted instruction fills the bubble that would otherwise be created.

Branch Delays

Instruction hazards (also called control hazards) are caused by a delay in the availability of an instruction or the memory address needed to fetch the instruction

- In ideal pipelined execution, a new instruction is fetched every cycle, while the previous instruction is being decoded
- This will work fine as long as the instruction addresses follow a predetermined sequence (e.g., PC <-- [PC]+4)
- However, branch instructions can alter this sequence
- Branch instructions first need to be executed to determine whether and where to branch
- Pipeline needs to be stalled before the branch outcome is decided

Unconditional Branches

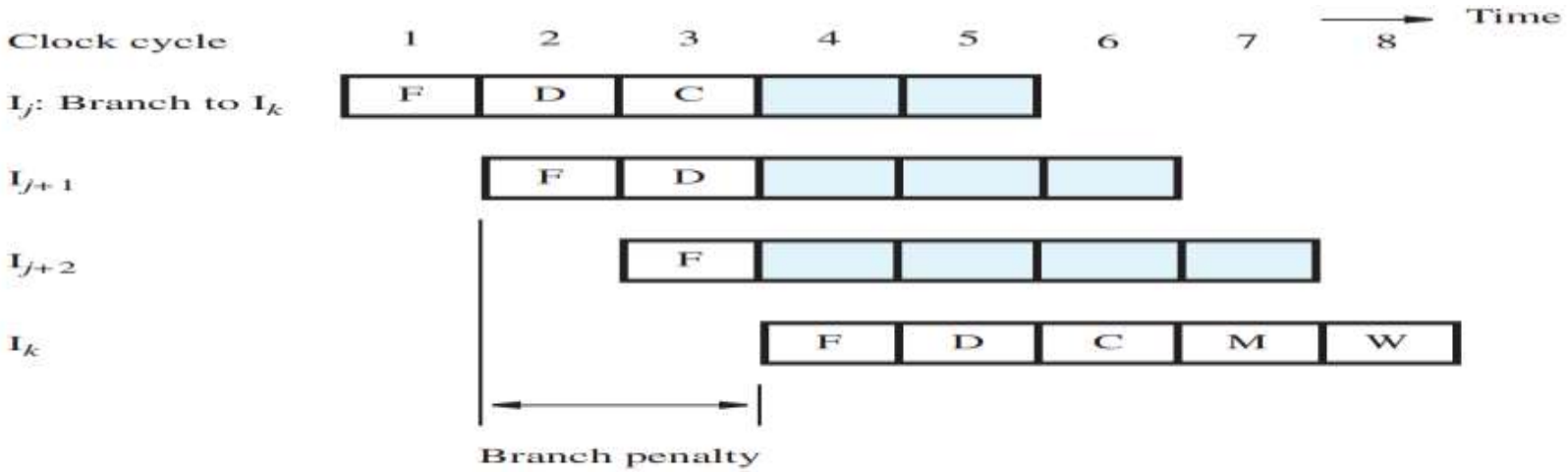


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

The pipelined execution of a sequence of instructions, beginning with an unconditional branch instruction, Ij.

The next two instructions, Ij+1 and Ij+2, are stored in successive memory addresses following Ij.

The target of the branch is instruction Ik .

The branch instruction is fetched in cycle 1 and decoded in cycle 2, and the target address is computed in cycle 3.

Hence, instruction Ik is fetched in cycle 4, after the program counter has been updated with the target address.

In pipelined execution, instructions Ij+1 and Ij+2 are fetched in cycles 2 and 3, respectively, before the branch instruction is decoded and its target address is known.

They must be discarded. The resulting two-cycle delay constitutes a branch penalty.

Reducing Branch Penalty

Branch instructions occur frequently (~ 20% of dynamic instruction count in a typical program), 2-cycle branch penalty can increase execution time by 40%

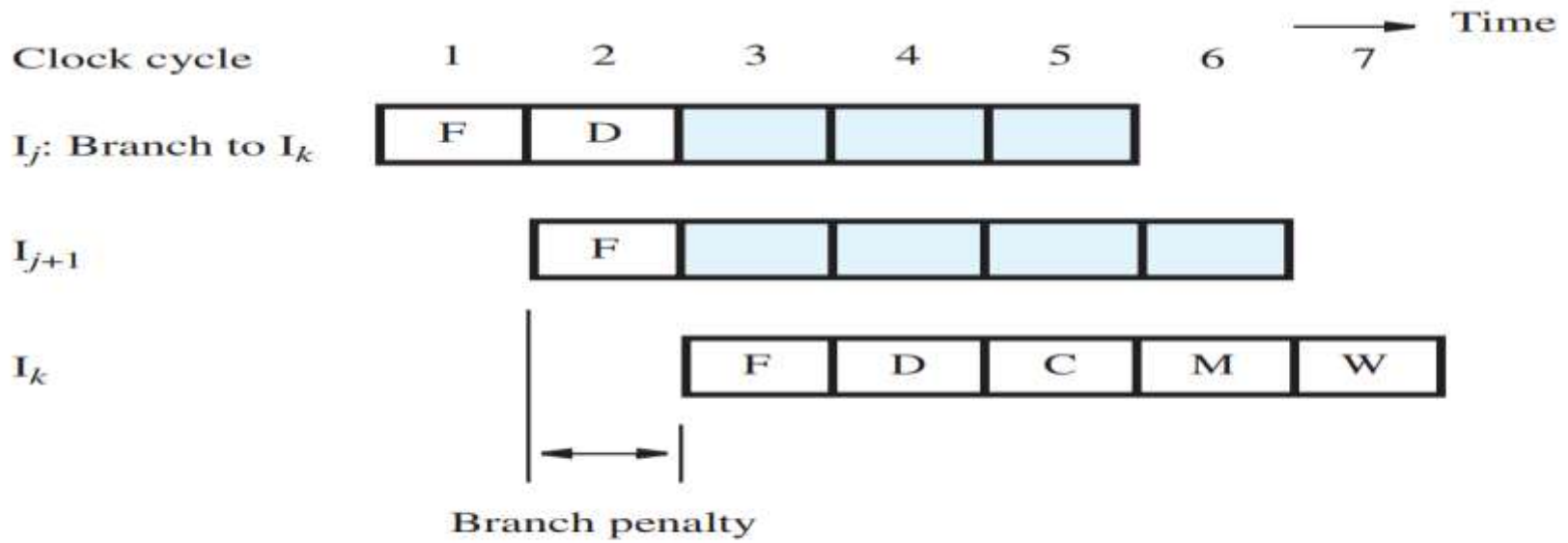


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

How to reduce branch penalty for unconditional branches?

- Compute branch target address earlier in the pipeline
- Include an adder in the "decode" stage to compute target address
- When the decoder determines that the instruction is an unconditional branch, the computed target address is available before the end of the decode stage

Branch target address computed in cycle # 2

- Branch penalty reduced to one cycle
- Only one instruction (Ij+1) is fetched incorrectly and needs to be discarded

Conditional Branches

• Consider a conditional branch instruction such as Branch_if_[R5]=[R6] LOOP

• Testing the branch condition (comparison between [R5] and [R6]) determines whether the branch is taken or not taken

• The execution steps for this instruction are shown in datapath. The result of the comparison in the third step determines whether the branch is taken.

• If the branch is taken, the penalty will be 2 cycles

How to reduce the branch penalty for conditional branches?

• Test the branch condition in the “Decode” stage

– Move the comparator from “Compute” stage to “Decode” stage

In this stage, the comparator uses the values from outputs A and B of the register file directly.

– Branch condition tested in parallel with target address computation

– Branch penalty reduced to one cycle

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.

Moving the branch decision and effective address calculation to the “Decode” stage reduces the branch penalty from two to one cycle

• Are there ways to reduce the branch penalty to zero?

• Yes. The basic idea behind Branch delay slot is to find an instruction that appears before the branch in program order but the branch outcome is independent of that instruction

• If the compiler finds such an instruction, it places this instruction in the branch delay slot (the location immediately after the branch instruction)

• This instruction is always executed irrespective of whether the branch is taken or not taken => no discarding and no branch penalty

Consider the program fragment

```
                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

Without the delay slot optimization:

• If the branch condition is evaluated to b n
Ij+1 needs to be discarded => penalty = 1 cycle

• If the branch condition is false ([R3] ! = 0), there is no penalty

After the delay slot optimization:

```
                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

• The “Add” instruction is always executed, even if the branch is taken • Instruction Ij+1 is fetched only if the branch is not taken

• Branch penalty if zero irrespective of the branch outcome

Logically, execution proceeds as though the branch instruction were placed after the Add instruction.

That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence. This technique is called delayed branching.

The effectiveness of delayed branching depends on how often the compiler can reorder instructions to usefully fill the delay slot.

Experimental data collected from many programs indicate that the compiler can fill a branch delay slot in 70 percent or more of the cases.

Branch Prediction

The decision to fetch this instruction is actually made in cycle 1, when the PC is incremented while the branch instruction itself is being fetched.

Thus, to reduce the branch penalty further, the processor needs to anticipate that an instruction being fetched is a branch instruction and predict its outcome to determine which instruction should be fetched in cycle 2

Static Branch Prediction

• In static branch prediction, the prediction made for a conditional branch remains constant (static) throughout the execution of a program

Example 1: Always-predict-not-taken

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.

However, if it is determined that the branch is to be taken, the instruction that has been fetched is discarded and the correct branch target instruction is fetched.

Misprediction incurs the full branch penalty. This simple approach is a form of static branch prediction.

The same choice (assume not-taken) is used every time a conditional branch is encountered.

In this case, always assuming that branches will not be taken results in a prediction accuracy of 50 percent for better prediction, The processor can determine the static prediction of taken or not-taken by checking the based on execution history of program.

Dynamic Branch Prediction

Dynamic Branch Prediction To improve prediction accuracy further, we can use actual branch behavior to influence the prediction, resulting in 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.

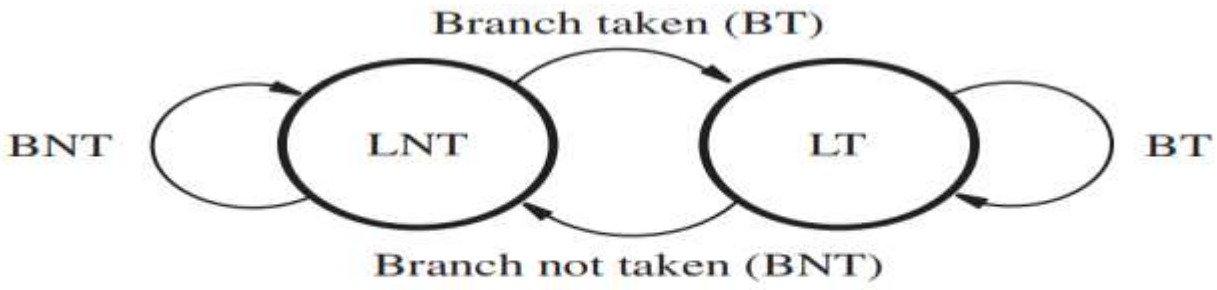
The prediction by undelaying H/W is not fixed ,rather than it changes dynamically. This technique has high accuracy than static prediction.

This result can be captured in a single bit (e.g., “0” if the branch was taken and “1” if the branch was not taken)

The two states are:

LT - Branch is likely to be taken

LNT - Branch is likely not to be taken



(a) A 2-state algorithm

Example1: Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = Taken and "N" = Not taken. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LNT	NT	T	LT
2	LT	T	T	LT
3	LT	T	NT	LNT
4	LNT	NT	T	LT
5	LT	T	T	LT
6	LT	T	NT	LNT

Prediction Accuracy = 2/6

2-bit perdition when 4 states.

Better prediction accuracy can be achieved by keeping more information about execution history.

An algorithm that uses four states is shown in Figure.

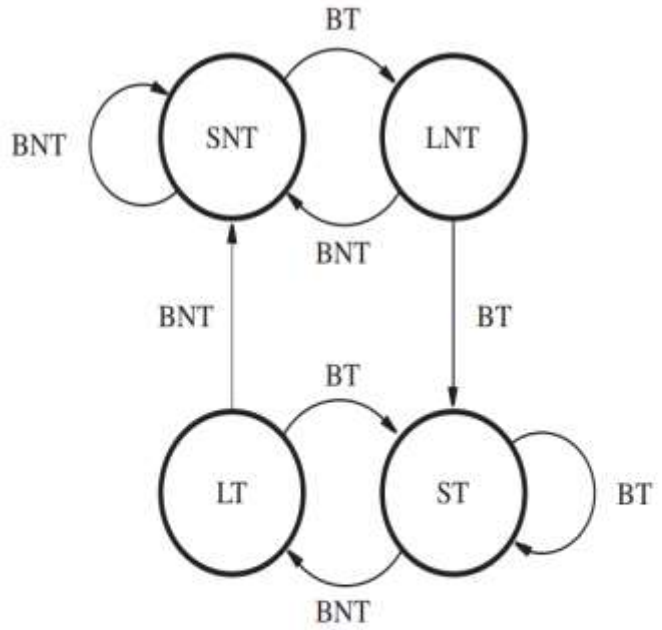
The four states are:

ST - Strongly likely to be taken

LT - Likely to be taken

LNT - Likely not to be taken

SNT - Strongly likely not to be taken



(b) A 4-state algorithm

Figure 6.12 State-machine representation of branch prediction algorithms.

Example 2 :Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = Taken and "N" = Not taken. Assume that the 2-bit branch predictor starts in the LT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LT	T	T	ST
2	ST	T	T	ST
3	ST	T	NT	LT
4	LT	T	T	ST
5	ST	T	T	ST
6	ST	T	NT	LT

Prediction Accuracy = 4/6

Resource Limitations

Pipelining enables overlapped execution of instructions, but the pipeline stalls when there are insufficient hardware resources to permit all actions to proceed concurrently.

If two instructions 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.

Such stalls can occur in a computer that has a single cache that supports only one access per cycle.

If both the Fetch and Memory stages of the pipeline are connected to the cache, then it is not possible for activity in both stages to proceed simultaneously.

Normally, the Fetch stage accesses the cache in every cycle. However, this activity must be stalled for one cycle when there is a Load or Store instruction in the Memory stage also needing to access the cache.