Pipelining

Carl Hamacher

Overview

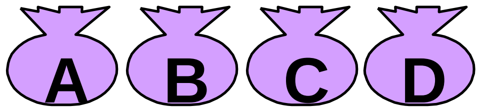
- Pipelining is widely used in modern processors.
- It improves system performance in terms of throughput.
- It requires sophisticated compilation techniques.

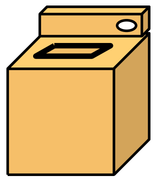
Making the Execution of Programs Faster

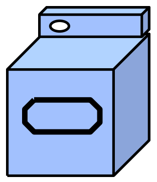
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- No. of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

Traditional Pipeline Concept

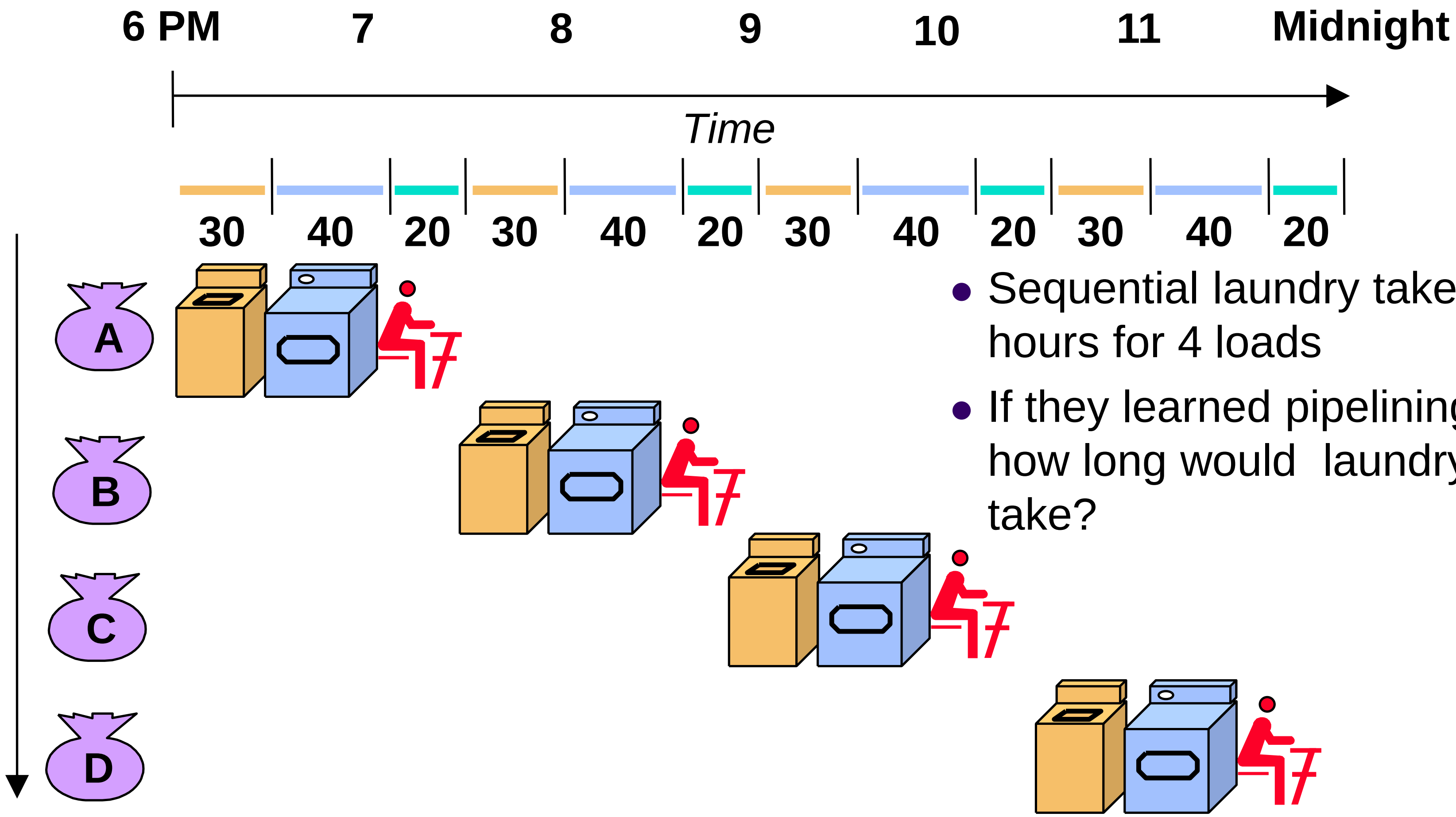
- Laundry Example
- A, B, C, D
  each have one load of clothes
  to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes





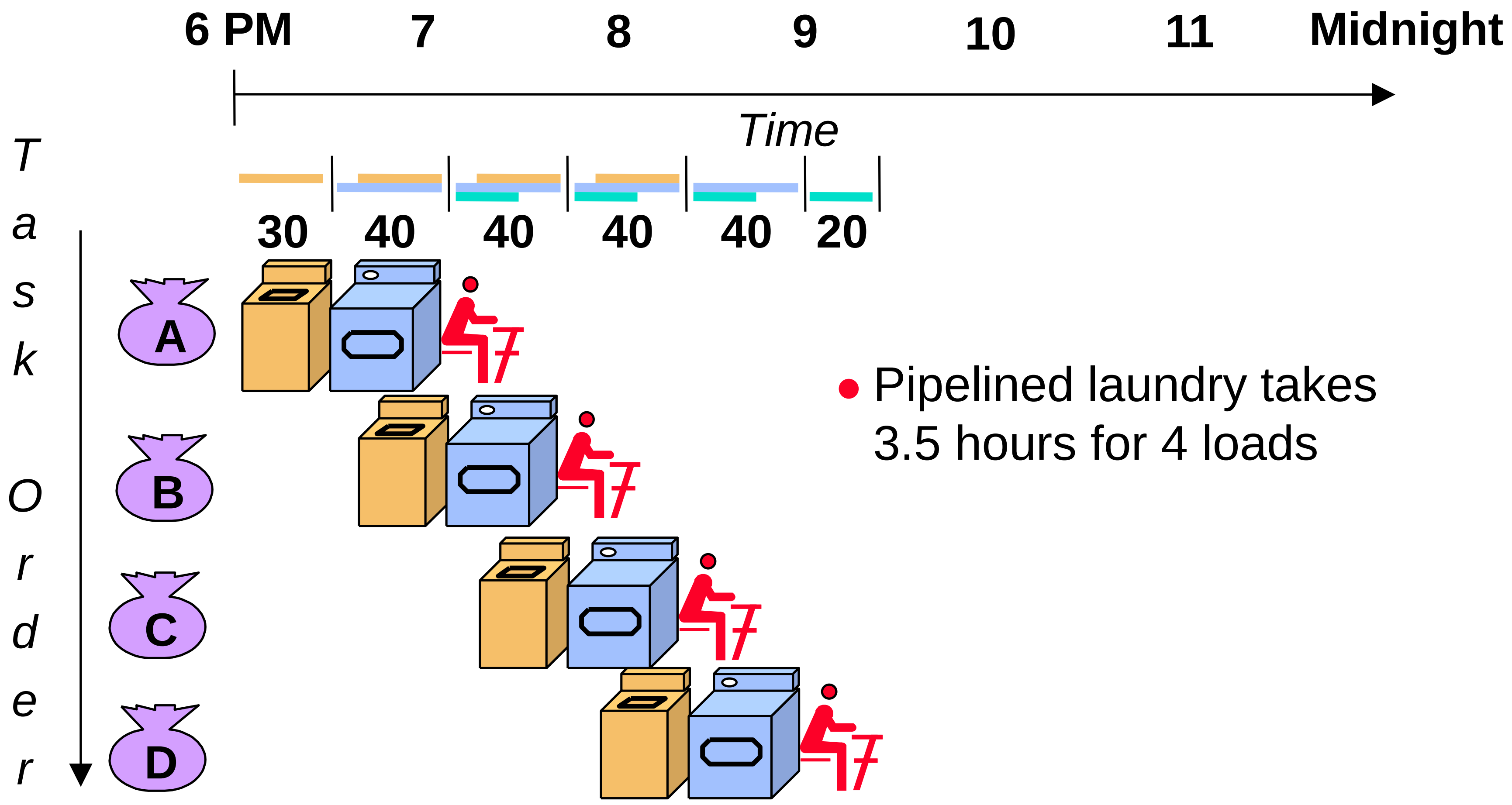


Traditional Pipeline Concept



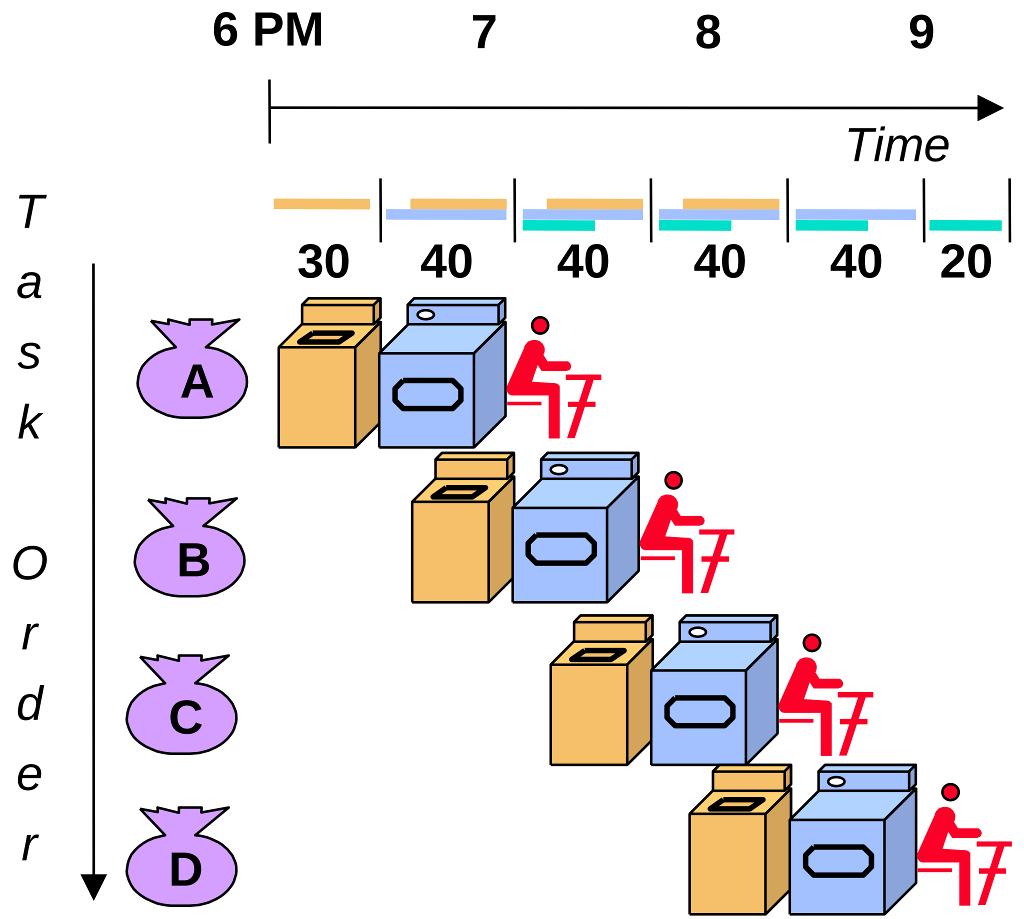
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



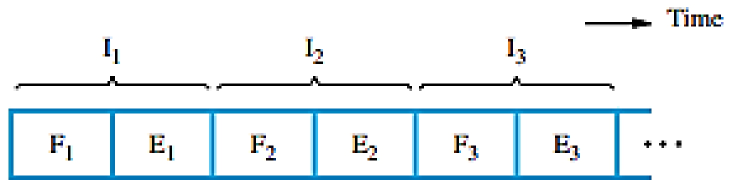
- Pipelined laundry takes 3.5 hours for 4 loads

Traditional Pipeline Concept

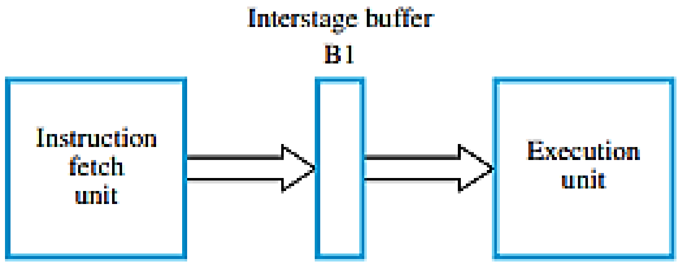


- It doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = No. of pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for dependences

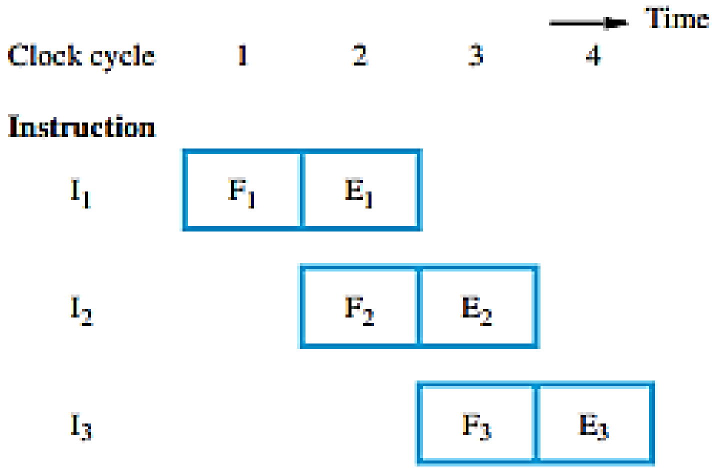
Pipelining in a Computer



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

Figure 8.1 Basic idea of instruction pipelining.

- Clock period: fetch and execute steps can each be completed in one clock cycle.

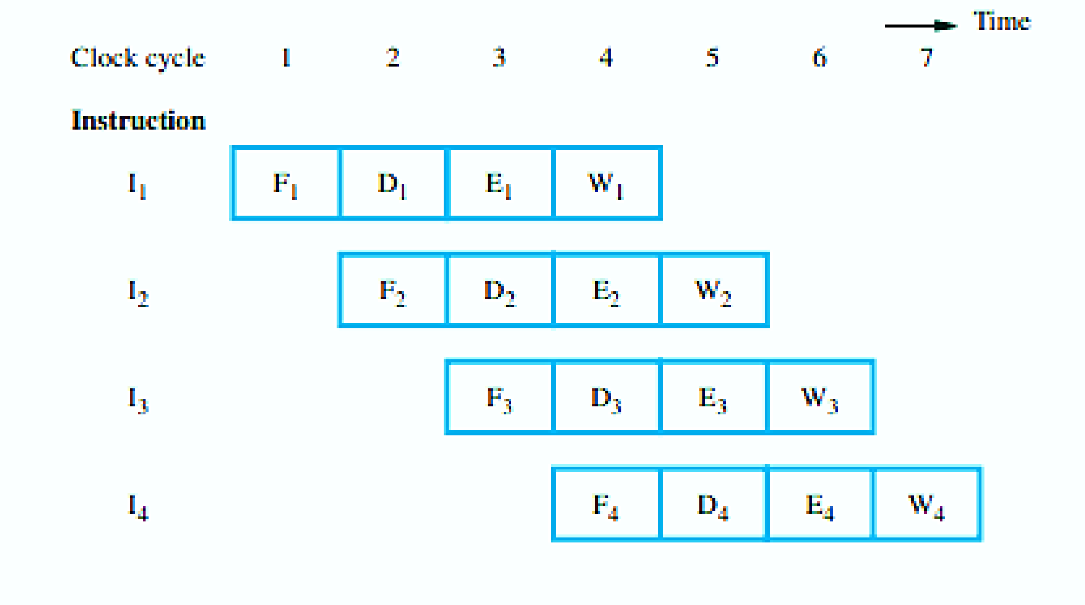
- In the first clock cycle, the fetch unit fetches an instruction I_1 (step F_1) and stores it in buffer B1 at the end of the clock cycle.

- In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2).

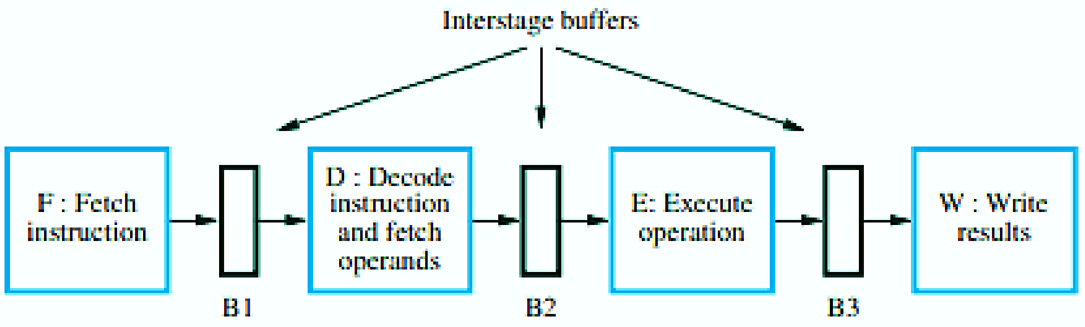
- Meanwhile, the execution unit performs the operation specified by instruction I_1, which is available to it in buffer B1 (step E_1).
- By the end of the second clock cycle, the execution of instruction I_1 is completed and instruction I_2 is available.
- Instruction I_2 is stored in B1, replacing I_1, which is no longer needed.
- Step E_2 is performed by the execution unit during the third clock cycle, while instruction I_3 is being fetched by the fetch unit.
- Hence, both the fetch and execute units are kept busy all the time

Pipelining in a Computer

Fetch + Decode
+ Execution + Write


Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4

(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 8.2 A 4-stage pipeline.

- Clock cycle 4: information in the buffers is as follows:
  - Buffer B1 holds instruction I_3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
  - Buffer B2 holds both the source operands for instruction I_2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I_2 (step W_2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
  - Buffer B3 holds the results produced by the execution unit and the destination information for instruction I_1.

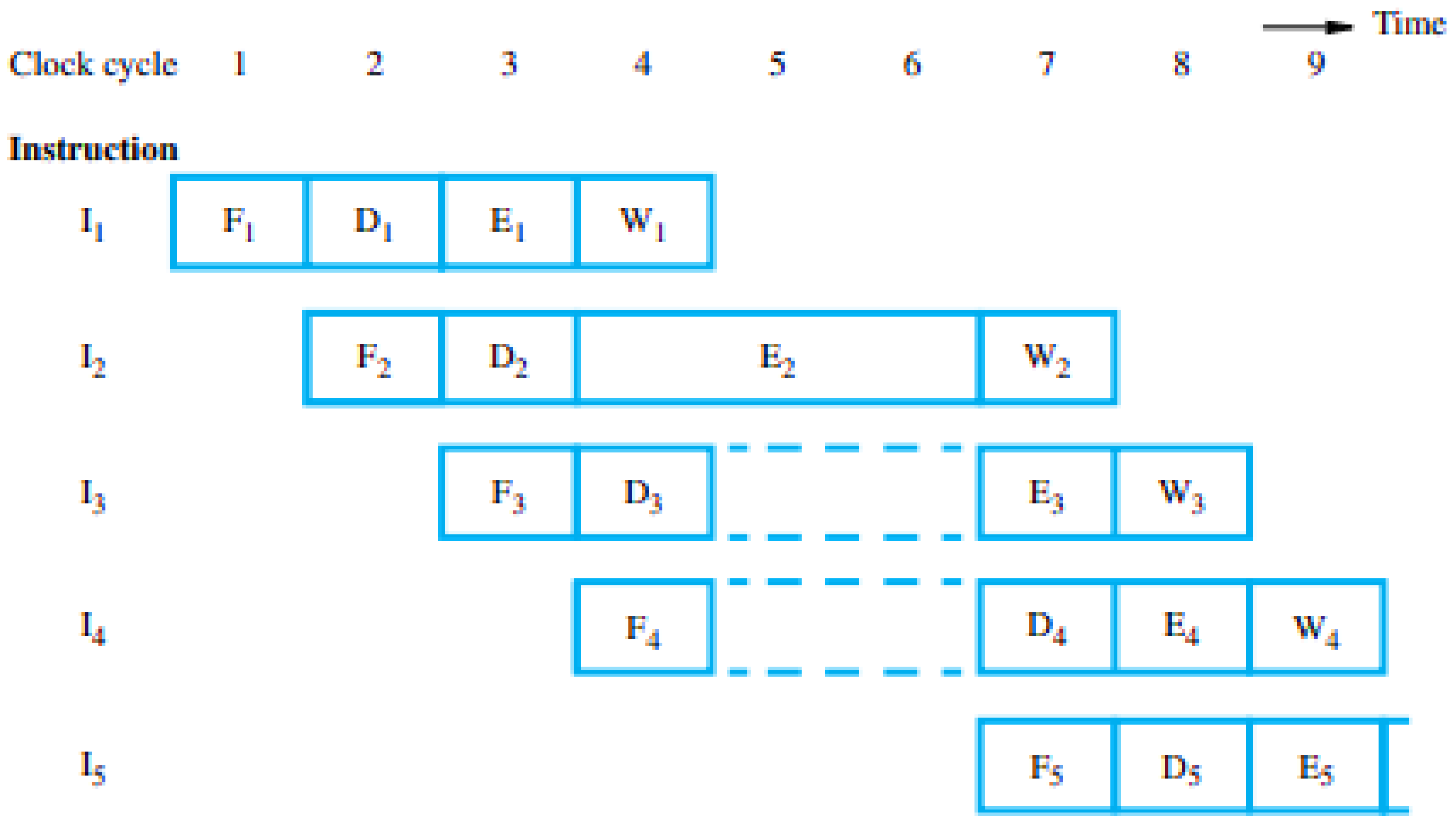
Role of Cache Memory

- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is slow, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Solution: We have cache.

Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

Pipeline Performance



Time
Clock cycle 1 2 3 4 5 6 7 8 9
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4
I_5 F_5 D_5 E_5

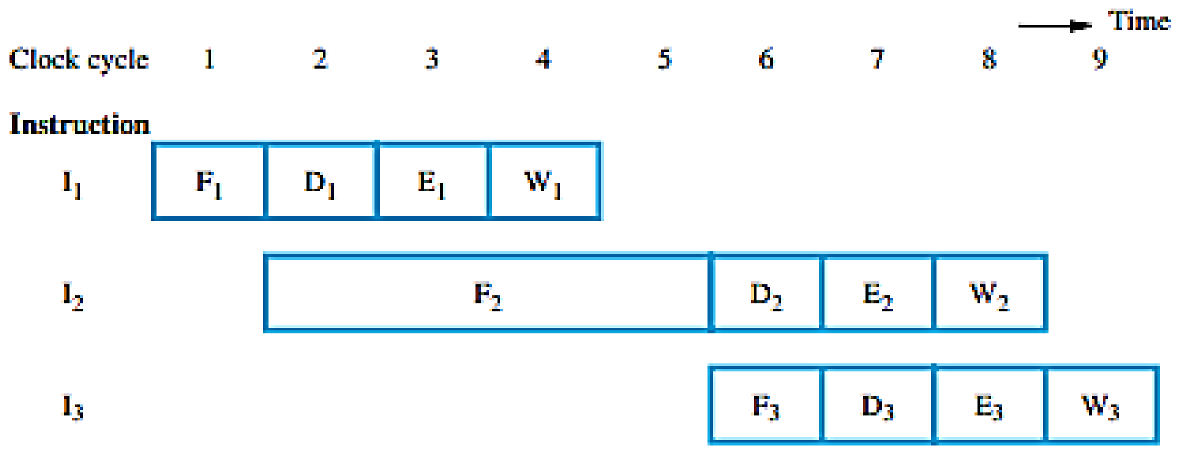
Figure 8.3 Effect of an execution operation taking more than one clock cycle.

Pipeline Performance

- Previous one is said to have been stalled for two clock cycles. Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

Pipeline Performance

Instruction hazard



(a) Instruction execution steps in successive clock cycles

→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

Idle periods – stalls (bubbles)

(b) Function performed by each processor stage in successive clock cycles

Figure 8.4 Pipeline stall caused by a cache miss in F2.

A **cache miss (Control hazard)**:

- I_1 is fetched from the cache in cycle 1, and its execution proceeds.
- However, fetching of I_2, started in cycle 2, results in a cache miss.
- The fetch unit must now suspend further fetch requests and wait for I_2 to arrive.
- Let's suppose I_2 is received and loaded into buffer B1 at the end of cycle 5.
- The pipeline resumes its normal operation at that point.

Structural hazard:

A situation when two instructions require a resource at the same time.

Example: Access to memory.

- One instruction may need to access memory as part of the **Execute** or **Write** stage while another instruction is being **fetched**.

- If instructions and data are in the **same cache** unit, only one instruction can *proceed* and the other instruction is *delayed*.

Pipeline Performance

Structural hazard   Load X(R1), R2

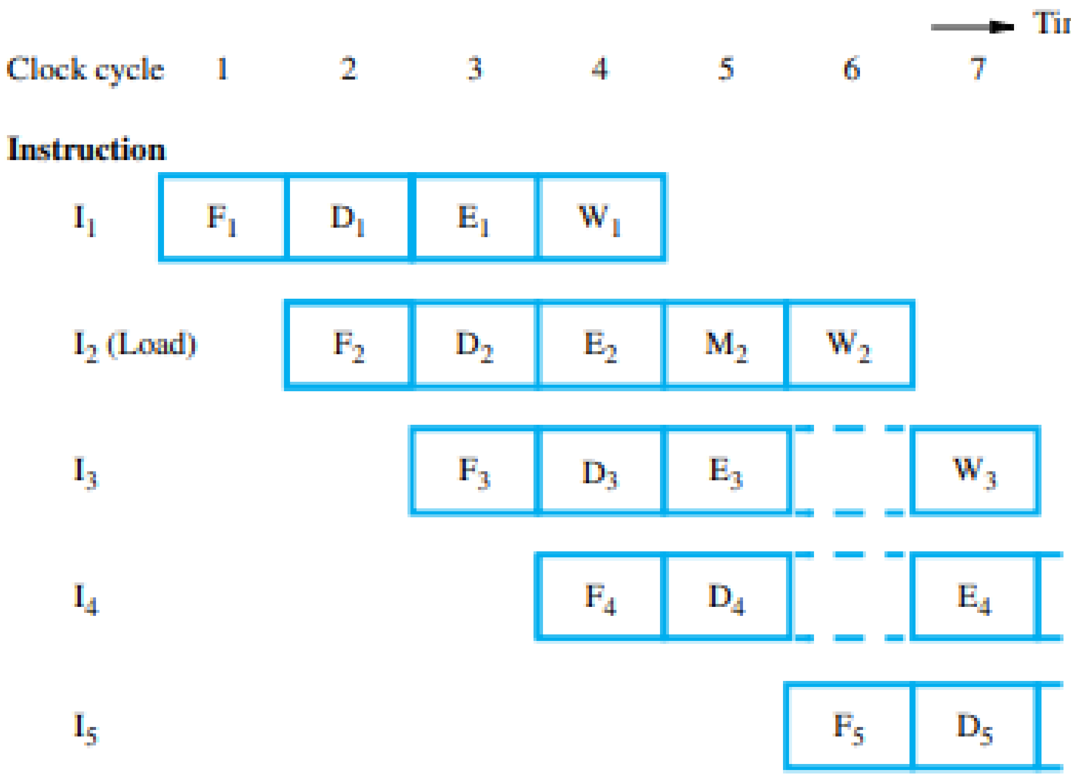


Figure 8.5 Effect of a Load instruction on pipeline timing.

Load of data:

- Memory address, X [R1], is computed in step E_2 in cycle 4, then memory access takes place in cycle 5.
- The operand read from memory is written into register R2 in cycle 6.
- So, execution takes 2 clock cycles (cycles 4 and 5), and causes the pipeline to stall for 1 cycle, because both I_2 and I_3 require the register file in cycle 6.

Load of data:

- Even though the instructions and data are all available, the pipeline is stalled because one hardware resource, the register file, cannot handle two operations at once.
- If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled.
- Generally, structural hazards are avoided by providing sufficient hardware resources on the processor chip.

Question

- Four instructions, the I2 takes two clock cycles for execution. Draw the figure for 4-stage pipeline, and figure out the total cycles needed for the four instructions to complete.

Data Hazards

- Must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.
- Hazard occurs

  $A \leftarrow 3 + A$

  $B \leftarrow 4 \times A$
- No hazard

  $A \leftarrow 5 \times C$

  $B \leftarrow 20 + C$
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:

```
Mul  R2, R3, R4
Add  R5, R4, R6
```

Data Hazards

Figure 8.6. Pipeline stalled by data dependency between D_2 and W_1.

Data dependency:

As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand.

Hence, the D step cannot be completed until the W step of the multiply has been completed.

Completion of step D_2 is delayed to clock cycle 5 (see D_{2A})

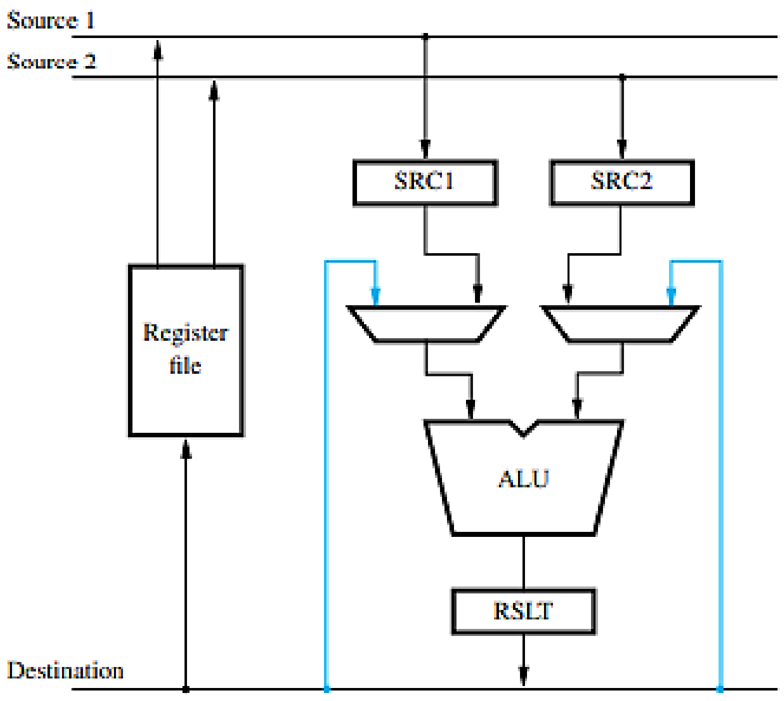
I_3 is fetched in cycle 3, but its decoding is delayed as D_3 cannot precede D_2.

Hence, pipelined is stalled for 2 cycles.

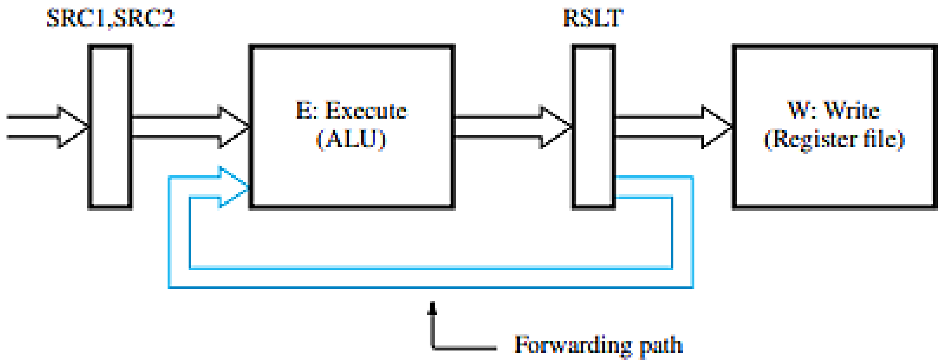
Operand Forwarding

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.

- A special arrangement needs to be made to “forward” the output of ALU to the input of ALU.



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Figure 8.7 Operand forwarding in a pipelined processor.

Operand Forwarding

- Processor datapath involves ALU and register file along with registers SRC1, SRC2, and RSLT.
- These registers constitute the interstage buffers needed for pipelined operation. SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3.
- Two multiplexers connected at the inputs to ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

Handling Data Hazards in Software

- Let the compiler detect and handle the hazard:

```
I1: Mul  R2, R3, R4
    NOP
    NOP
I2: Add  R5, R4, R6
```

- Illustrates a close link between the compiler and the hardware

- Compiler can reorder the instructions to perform some useful work during the NOP slots.

Handling Data Hazards in Software

Limitations:

- Insertion of NOP leads to larger code size.
- A given processor architecture may have several hardware implementations, offering different features.
- NOP inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

Side Effects

- Sometimes an instruction changes the contents of a register other than the one named as the destination.
- When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect.
- Example: Stack instructions, such as push and pop.
- Conditional code flags:

```
Add  R1, R3
AddWithCarry  R2, R4      //R4 ← [R2] + [R4] + carry
```

- Instructions designed for execution on pipelined hardware should have few side effects.

Instruction Hazards

- Whenever the stream of instructions supplied by the instruction fetch unit is interrupted, the pipeline stalls.
- Cache miss
- Branch

Unconditional Branches

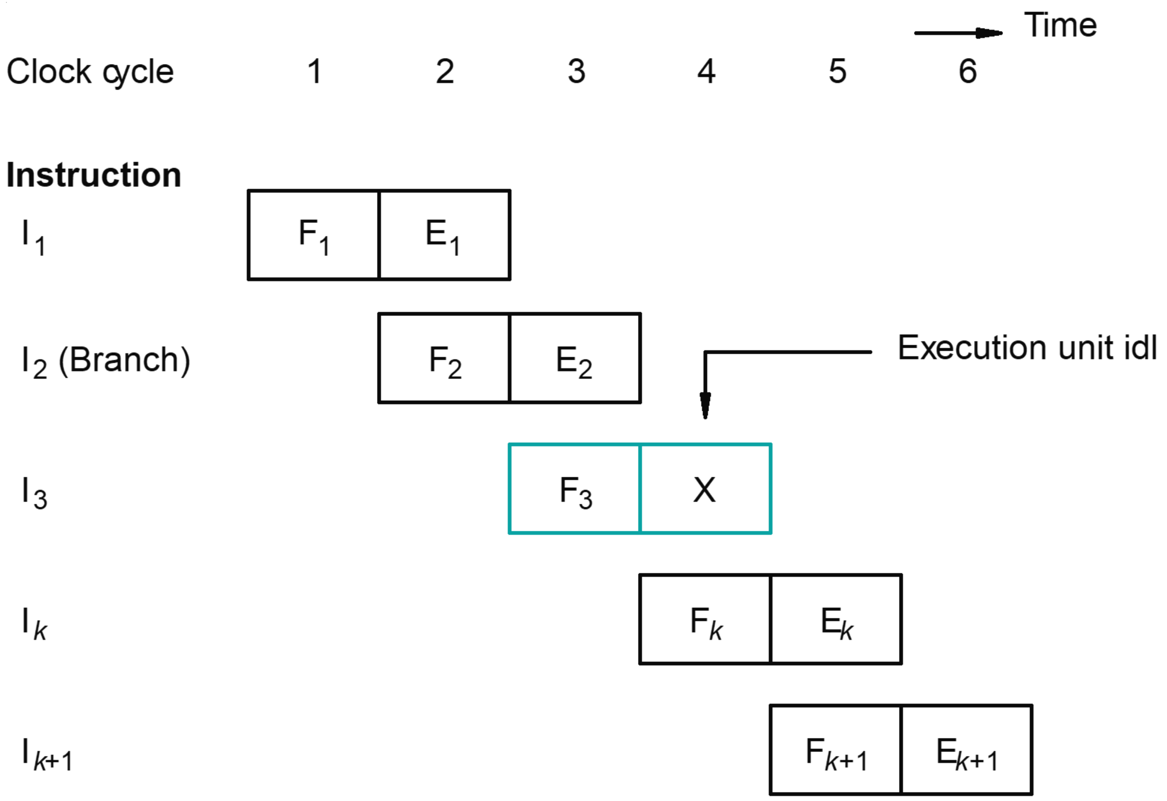


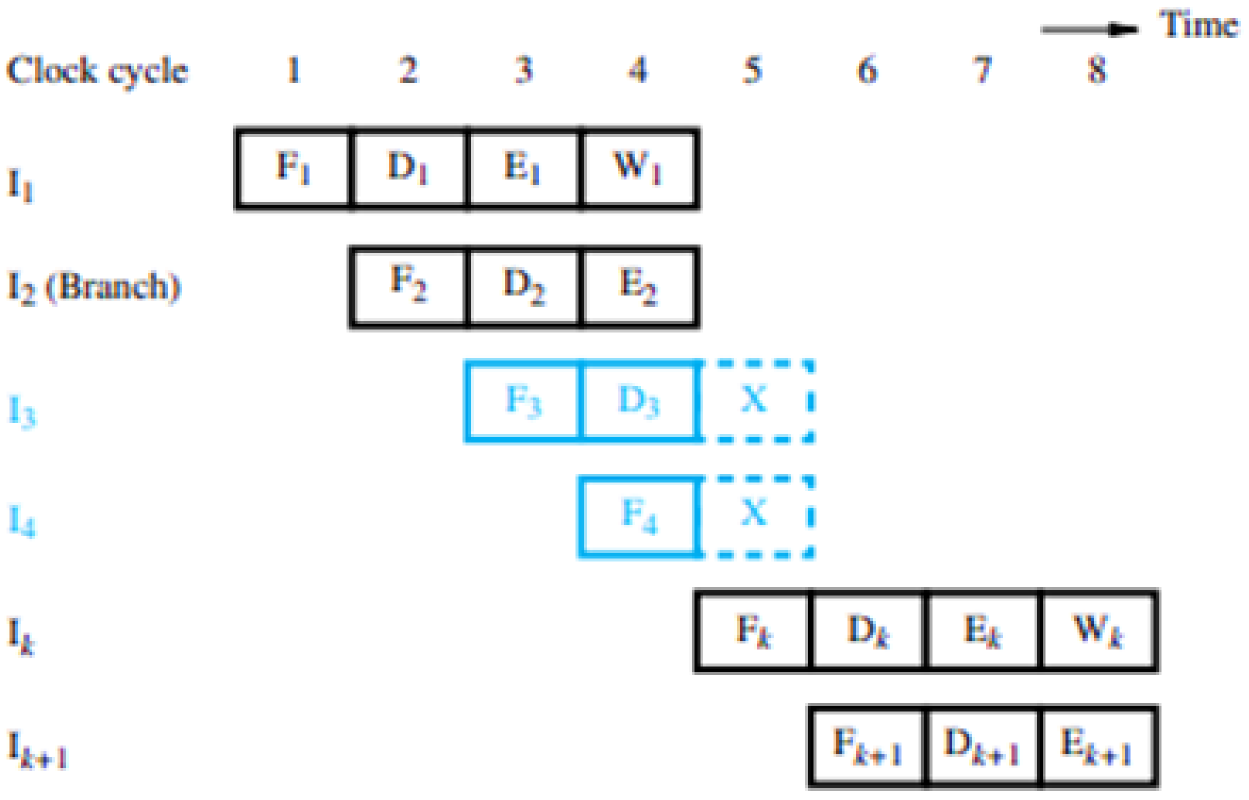
Figure 8.8. An idle cycle caused by a branch instruction.

Unconditional braches

- I_1 to I_3 are stored at successive memory addresses, and I_2 is a branch instruction, and the branch target is I_k.
- In clock cycle 3, the fetch operation for I_3 is in progress at the same time that the branch instruction is being decoded and the target address computed.
- In clock cycle 4, the processor must discard I_3, which has been incorrectly fetched, and fetch I_k.
- In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.
- The time lost as a result of a branch instruction is often referred to as the *branch penalty*.

Branch Timing

- Branch penalty: 2 clock cycles



(a) Branch address computed in Execute stage

Branch Timing

- Reducing the penalty

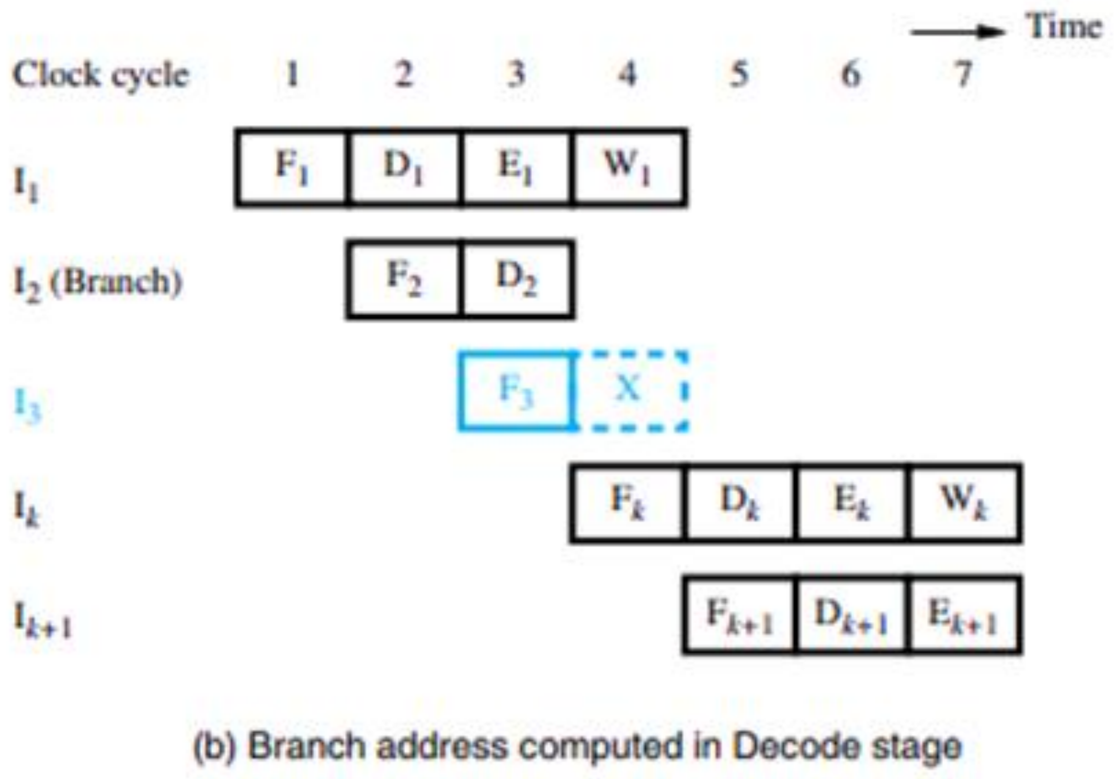


Figure 8.9 Branch timing.

Reducing the penalty

- Reducing the branch penalty requires the branch address to be computed earlier in the pipeline.
- Instruction fetch unit has dedicated hardware to identify a branch instruction which computes the branch target address quickly after an instruction is fetched.
- With this additional hardware, both of these tasks can be performed in step D_2 (Figure 8.9*b*).
- Now the branch penalty is only one clock cycle.

Instruction Queue and Prefetching

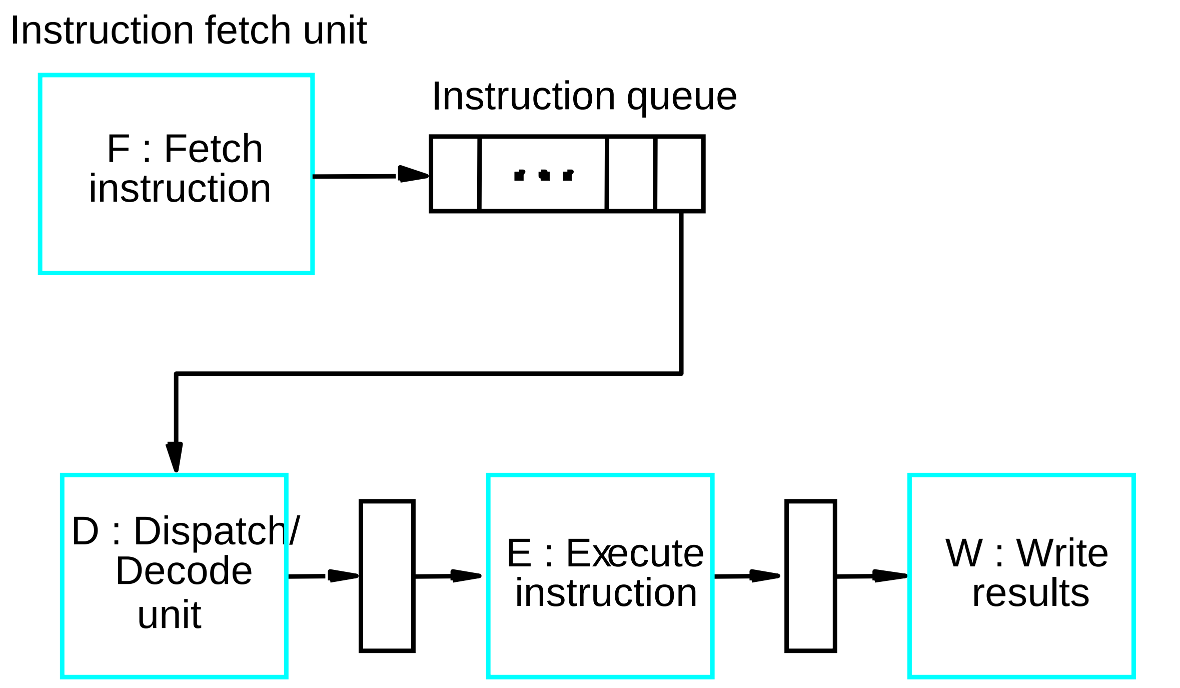


Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2*b*.

Instruction Queue and Prefetching

- Cache miss or a branch instruction stalls the pipeline.
- Many processors employ sophisticated fetch units that fetch instructions before they are needed and put them in a queue which can store several instructions.
- Another unit, *dispatch unit*, takes instructions from the front of the queue and sends them to the execution unit.
- The dispatch unit also performs the decoding function.

Instruction Queue and Prefetching

- Fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions.
- It keeps the instruction queue filled at all times to reduce the impact of delays when fetching instructions.
- If the pipeline stalls due to data hazard, e.g., the dispatch unit is not able to issue instructions from the instruction queue, however, the fetch unit continues to fetch instructions and add them to the queue.
- Also, if there is a delay in fetching instructions because of a branch/cache miss, the dispatch unit continues to issue instructions from the instruction queue.

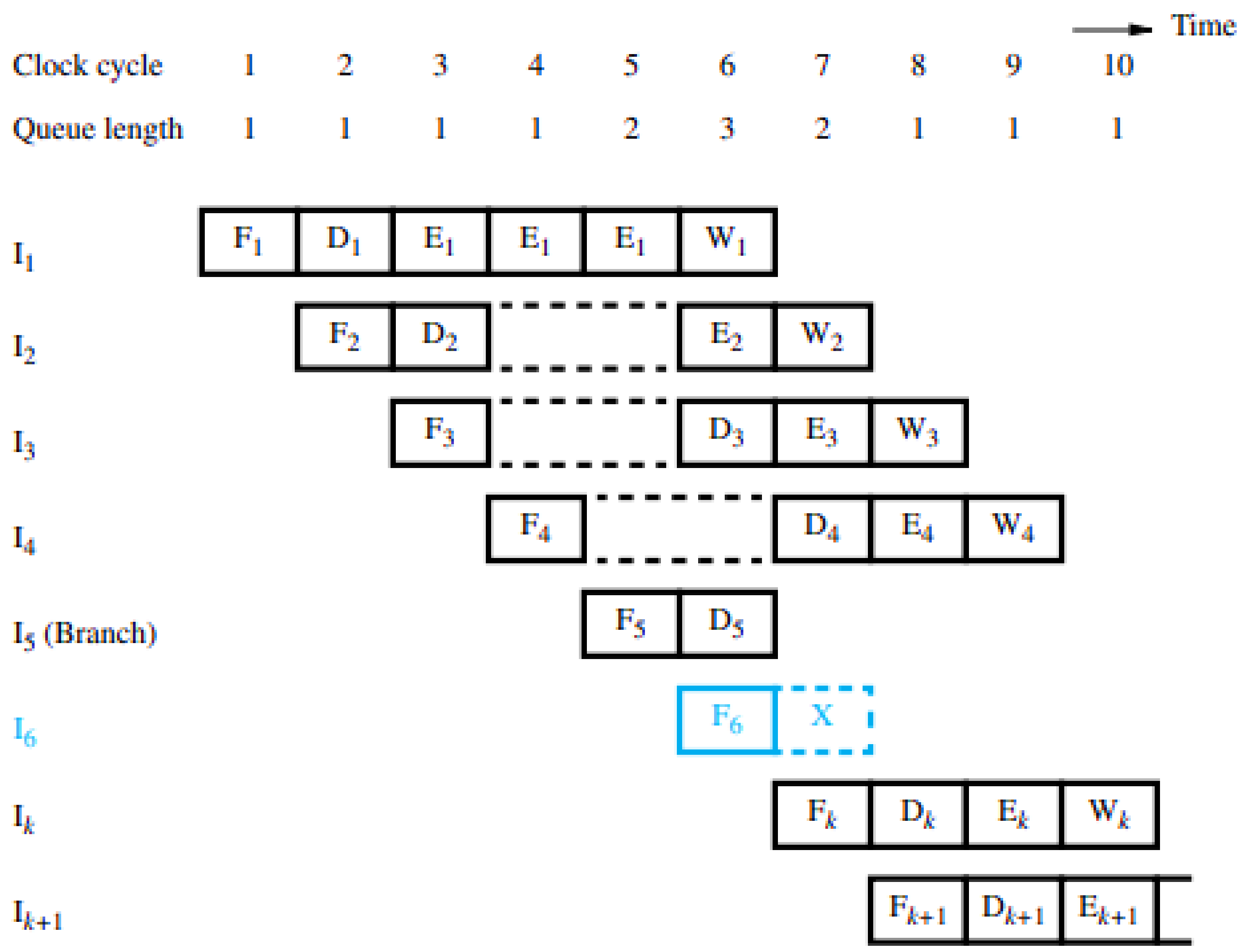


Figure 8.11 Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

Instruction Queue and Prefetching

- Assume that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one.
- Hence, the queue length remains the same for the first four clock cycles.
- Suppose that I_1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.
- I_5 is a branch instruction, and its target I_k is fetched in cycle 7, and I_6 is discarded.

Instruction Queue and Prefetching

- Branching causes a stall in cycle 7 and I_6 is discarded. Instead, I_4 sent to the decoding stage. After discarding I_6, the queue length drops to 1 in cycle 8.
- Queue length remains the same until there is another stall.
- Instructions I_1, I_2, I_3, I_4, and I_k complete execution. Hence, the branch instruction does not increase the overall execution time.
- Reason: Instruction fetch unit executes the branch instruction (by computing the branch address) concurrently with the execution of other instructions.
- This technique is referred to as *branch folding*.

Delayed Branch

- The location following a branch instruction is called a *branch delay slot*. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction.
- *Delayed branching* can minimize the penalty incurred due to conditional branch instructions.
- The instructions in the delay slots are always fetched. Hence, arrange for them to be fully executed whether or not the branch is taken.

Delayed Branch

- If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions

- Objective: place useful instructions in these slots.

- Effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions.

Delayed Branch

```
LOOP      Shift_left     R1
          Decrement      R2
          Branch=0       LOOP
NEXT      Add            R1,R3
```

(a) Original program loop

```
LOOP      Decrement      R2
          Branch=0       LOOP
          Shift_left     R1
NEXT      Add            R1,R3
```

(b) Reordered instructions

Figure 8.12. Reordering of instructions for a delayed branch.

Delayed Branch

- For a processor with one delay slot, the instructions can be reordered.
- The shift instruction is fetched while the branch instruction is being executed.
- After evaluating the branch condition, the processor fetches the instruction at LOOP/NEXT, depending on whether the condition is true/false.
- In either case, it completes execution of the shift instruction.

Delayed Branch

- Pipelined operation is not interrupted at any time, and there are no idle cycles.
- Logically, the program is executed as if the *branch instruction were placed* after the shift instruction.
- That means branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name "delayed branch."

Delayed Branch

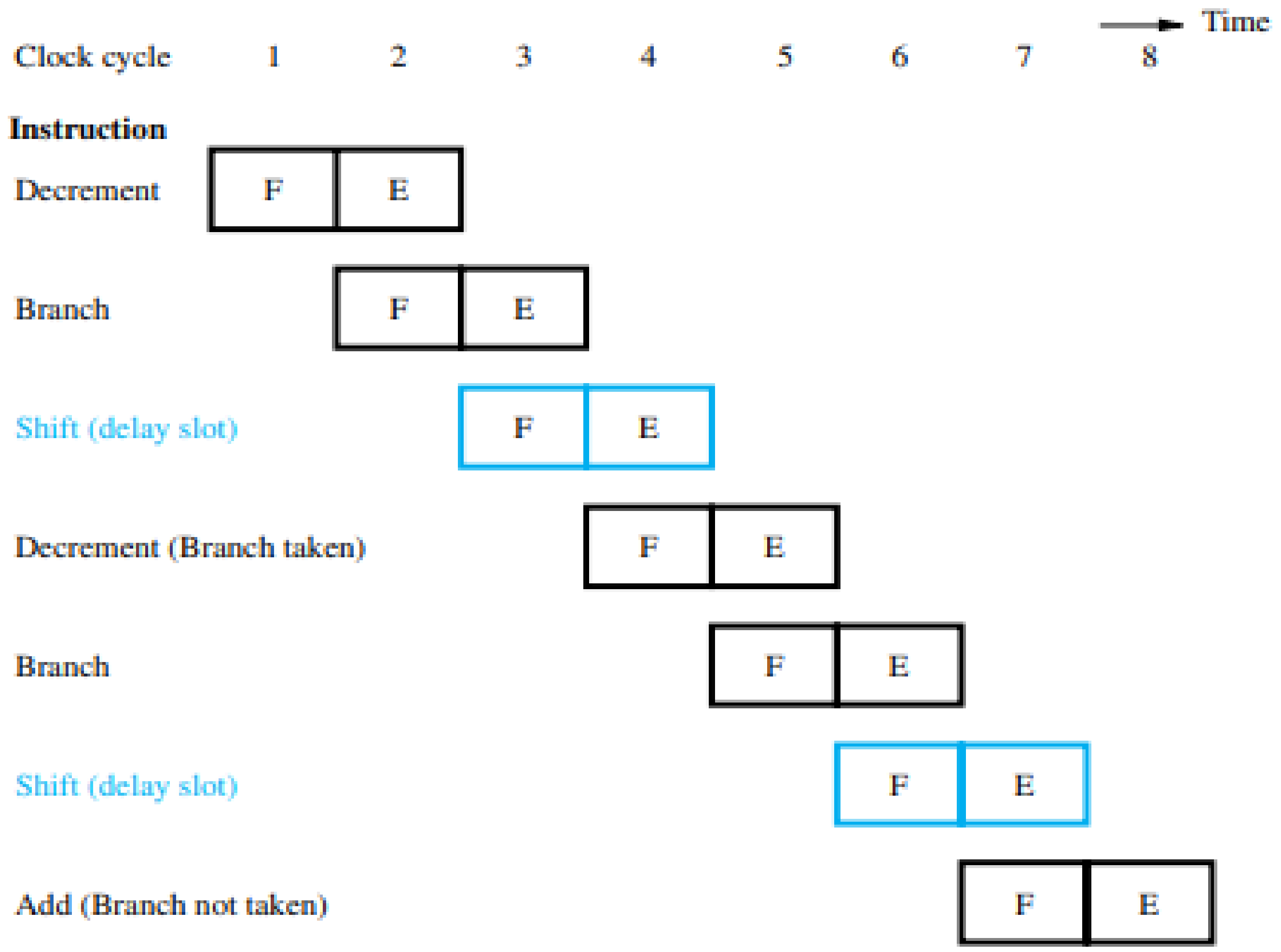


Figure 8.13 Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12*b*.

Delayed Branch

- Study indicates that sophisticated compilation techniques can use one branch delay slot in around 85% cases.
- For a processor with two branch delay slots, the compiler attempts to find two instructions preceding the branch instruction that it can move into the delay slots without introducing a logical error.
- The chances of finding two such instructions are considerably less than the chances of finding one.
- Thus, if increasing the number of pipeline stages involves an increase in the number of branch delay slots, the potential gain in performance may not be fully realized.

Branch Prediction

- To predict whether or not a particular branch will be taken.
- Simplest form: assume branch will not take place and continue to fetch instructions in sequential address order.
- Until the branch is evaluated, instruction execution along the predicted path must be done on a speculative basis.
- Speculative execution: instructions are executed before the processor is certain that they are in the correct execution sequence.
- Caution: No processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.

Incorrectly Predicted Branch

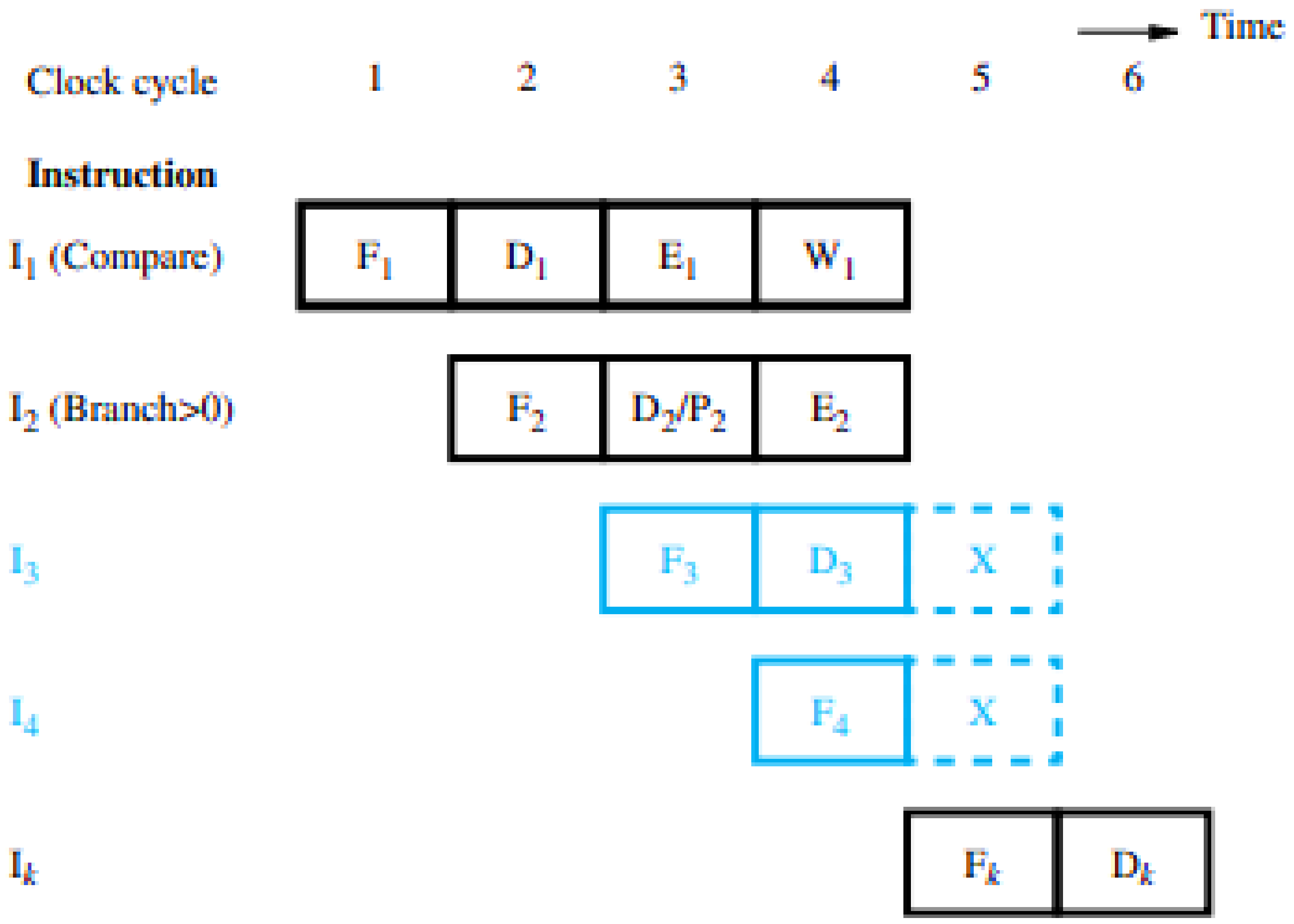


Figure 8.14 Timing when a branch decision has been incorrectly predicted as not taken.

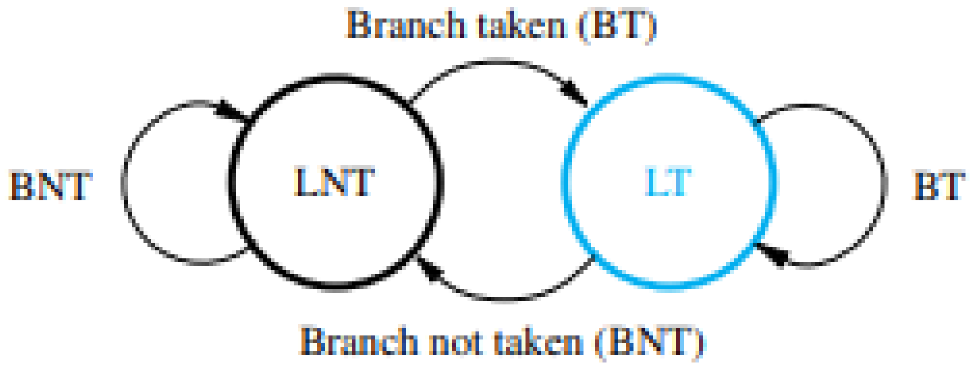
Branch Prediction

- Better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.
- Use hardware to observe whether the target address is lower or higher than that of the branch instruction.
- Let compiler include a branch prediction bit.
- So far the branch prediction decision is always the same every time a given instruction is executed – static branch prediction.

- Objective: To reduce the probability of making a wrong decision and to avoid fetching instructions that eventually have to be discarded.

- Dynamic branch prediction: Processor hardware assesses the likelihood of a given branch being taken by *keeping track a branch of decisions* every time that instruction is executed.

- Instruction history is used (i.e. most recent execution of the instruction)
- Consider a two-state machine: LT (branch is likely to be taken), and LNT (branch is likely not to be taken)
- One bit of history information for each branch instruction.
- Will work well for Loop (will be incorrect in the last pass)

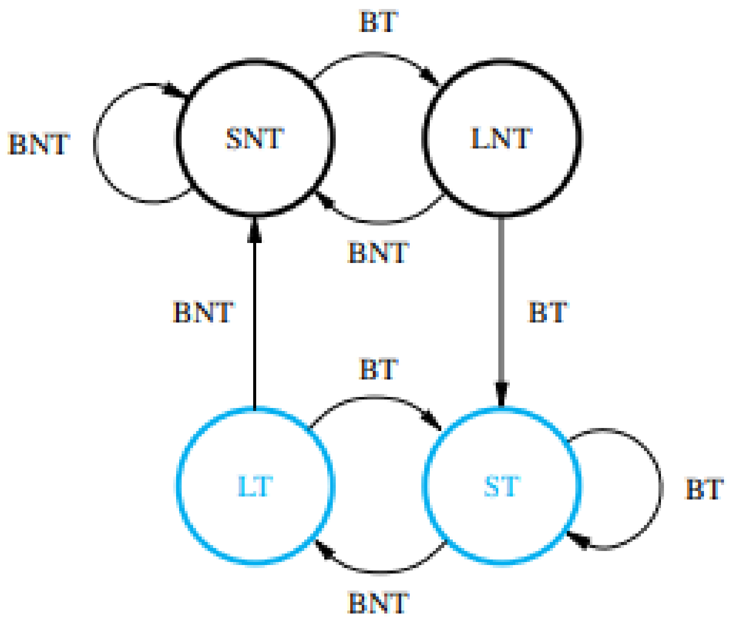


(a) A 2-state algorithm

A 4-state algorithm

BN

- 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.



(b) A 4-state algorithm

Figure 8.15 State-machine representation of branch-prediction algorithms.

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

- For better performance keep more information about execution history like a four-stage machine (2 bits)

- SNT to LNT: if twice incorrect in a row, then state change to ST.

- State information may be kept in a look-up table.

- Store the history bits as a tag associated with branch instruction in the instruction cache.

Influence on Instruction Sets

Overview

- Some instructions are much better suited to pipeline execution than others.
- Addressing modes
- Conditional code flags

Addressing Modes

- Addressing modes: simple vs. complex.

- When we choose the addressing modes, we must consider the effect of each addressing mode on instruction flow in the pipeline:
  - Side effects
  - Extent to which complex addressing modes cause the pipeline to stall
  - Whether a given mode is likely to be used by compilers

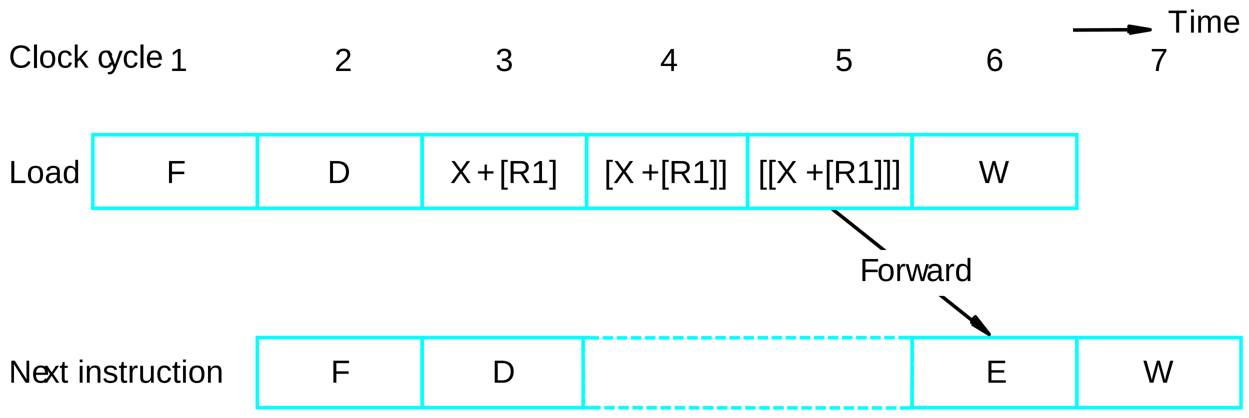
Recall

Load X(R1), R2

Load (R1), R2

Complex Addressing Mode

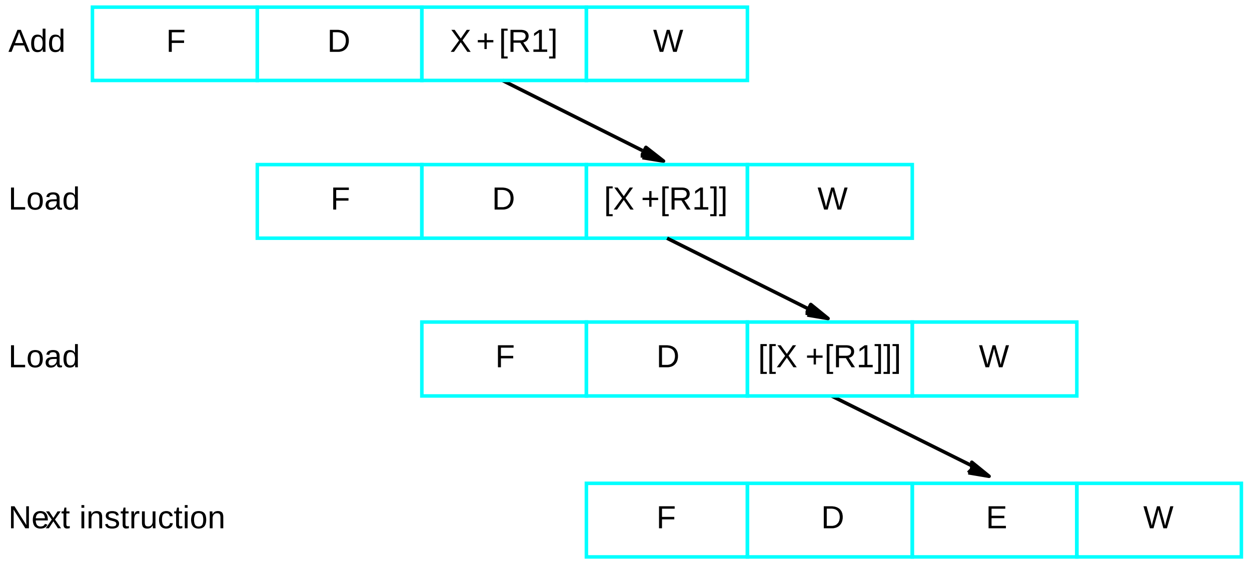
Load (X(R1)), R2



(a) Complex addressing mode

Simple Addressing Mode

Add #X, R1, R2
Load (R2), R2
Load (R2), R2



(b) Simple addressing mode

Addressing Modes

- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- Advantage: reducing the number of instructions / program space
- Disadvantage: cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- Conclusion: complex addressing modes are not suitable for pipelined execution.

Addressing Modes

- Good addressing modes should have:
  - Access to an operand does not require more than one access to the memory
  - Only load and store instruction access memory operands
  - The addressing modes used do not have side effects
- Register, register indirect, index

Conditional Codes

- If an optimizing compiler attempts to reorder instruction to avoid stalling the pipeline when branches or data dependencies between successive instructions occur, **it must ensure that reordering does not cause a change in the outcome of a computation.**
- **Dependency** introduced by the condition-code flags **reduces** the **flexibility** available for the compiler to reorder instructions.

Conditional Codes

```
Add            R1,R2
Compare        R3,R4
Branch=0       . . .
```

(a) A program fragment

```
Compare        R3,R4
Add            R1,R2
Branch=0       . . .
```

(b) Instructions reordered

Figure 8.17. Instruction reordering.

Conditional Codes

- Two conclusions:
  - To provide flexibility in reordering instructions, the condition-code flags should be affected by as few instruction as possible.
  - Compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not.

Datapath and Control Considerations

Original Design

Pipelined Design

- Separate instruction and data caches
- PC is connected to IMAR
- DMAR
- Separate MDR
- Buffers for ALU
- Instruction queue
- Instruction decoder output

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two regs
- Writing into one register in the reg file
- Performing an ALU operation

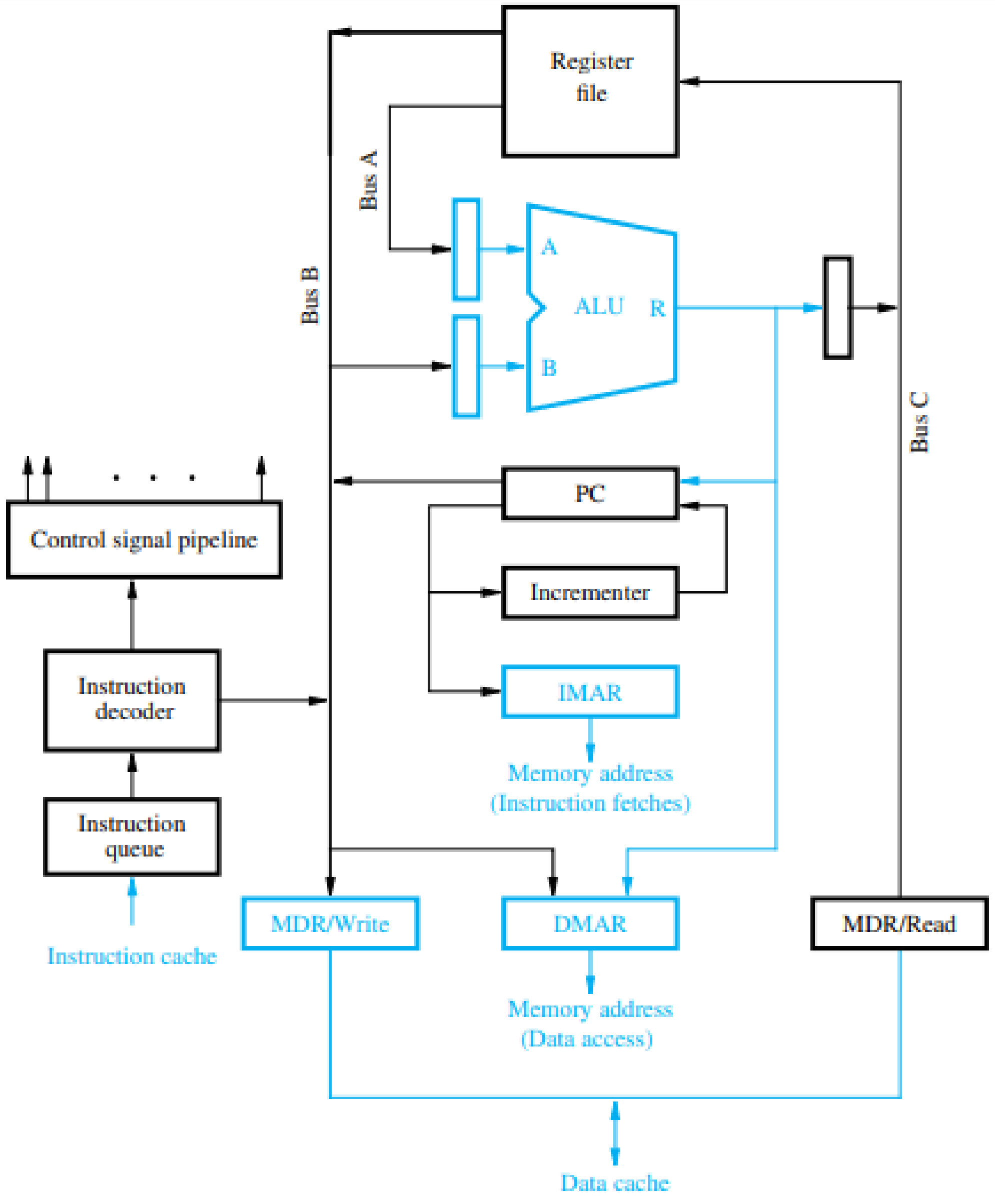


Figure 8.18 Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

Superscalar Operation

Overview

- The maximum throughput of a pipelined processor is one instruction per clock cycle.
- If we equip the processor with multiple processing units to handle several instructions in parallel in each processing stage, several instructions start execution in the same clock cycle – multiple-issue.
- Processors are capable of achieving an instruction execution throughput of more than one instruction per cycle – superscalar processors.
- Multiple-issue requires a wider path to the cache and multiple execution units.

Superscalar

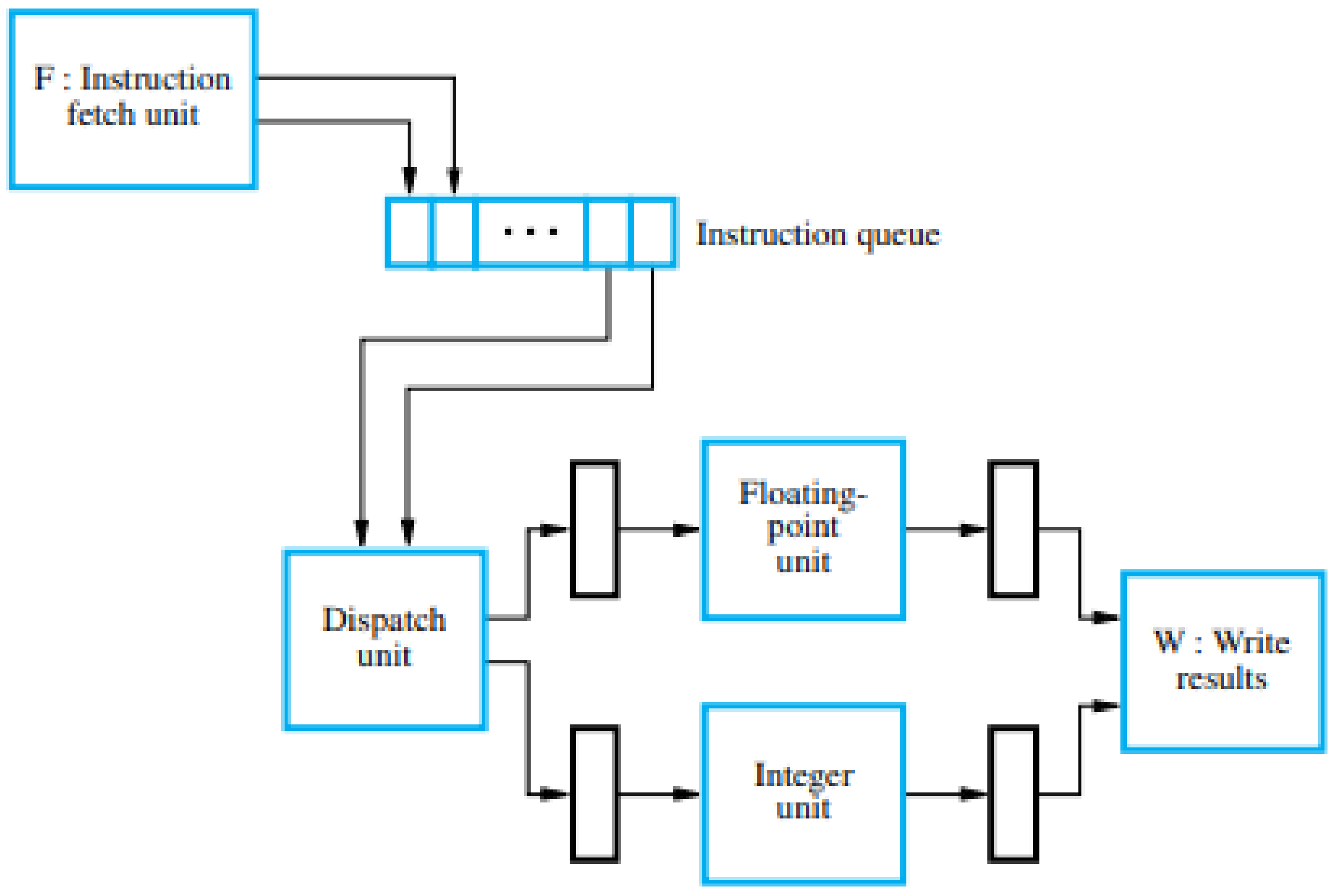


Figure 8.19 A processor with two execution units.

Timing

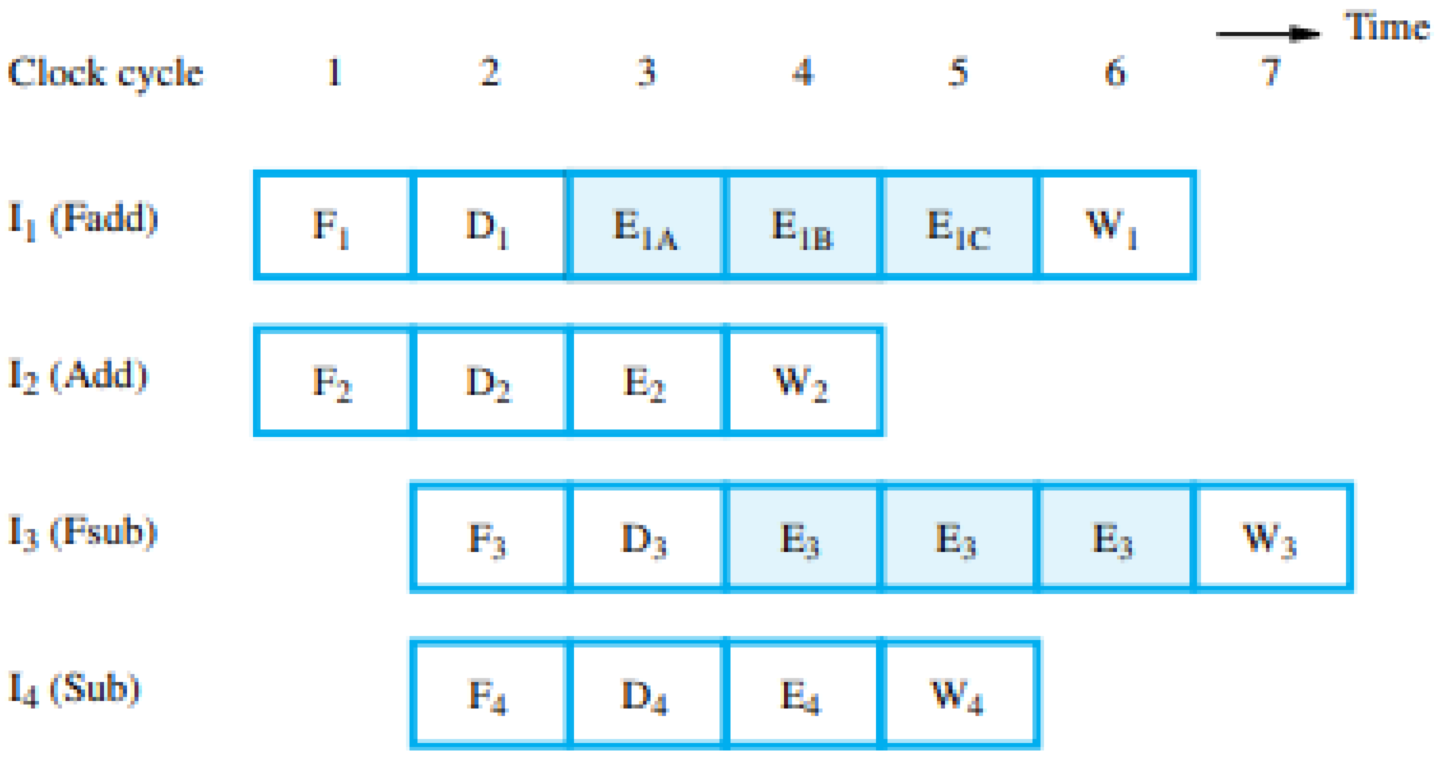
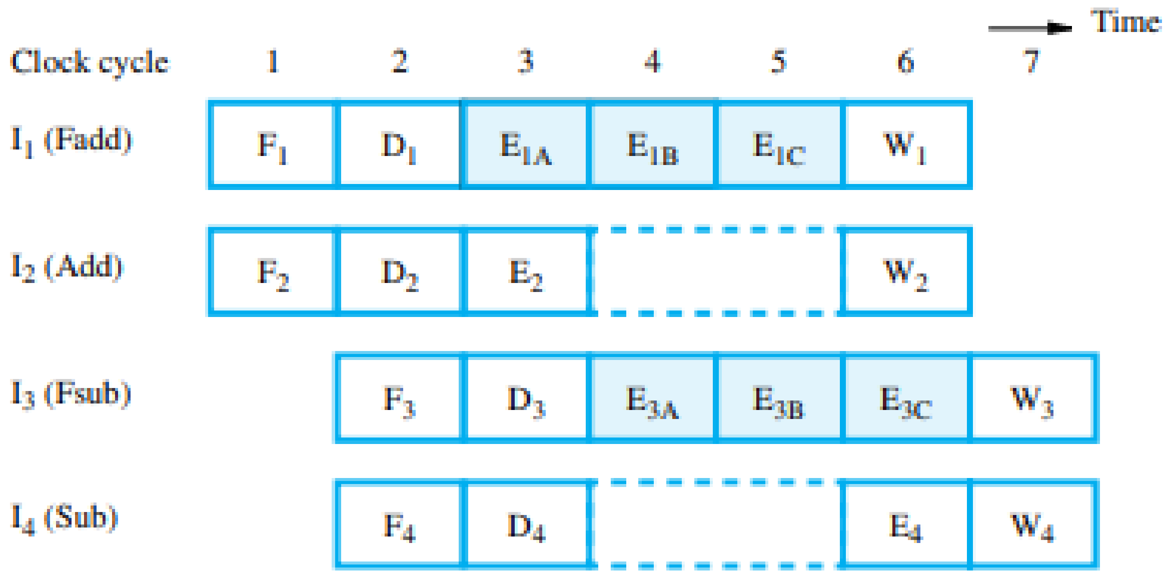


Figure 8.20 An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

Out-of-Order Execution

- Hazards
- Exceptions
- Imprecise exceptions
- Precise exceptions



(a) Delayed write

- If the I2 depends on I1, then I2 will be delayed.
- Complication due to exception such as bus error or illegal operation.
- If result of I2 is written back to register and I1 causes an exception: **Inconsistent state.**
- PC may point to instruction for which exception occurred.
- One or more of the succeeding instructions have been executed to the completion. If such a situation is permitted the processor is said to have **imprecise exceptions.**

- To guarantee a consistent state, the results must be written into the destination strictly in program order.
- If an exception occurs during an instruction execution, all subsequent instructions that may have been partially executed are discarded. This is called a **precise exception.**

- When an external interrupt is received, the Dispatch unit stops receiving new instructions from the instruction queue, and the instructions remaining in the queue are discarded.
- All instructions whose execution is pending continue to completion.

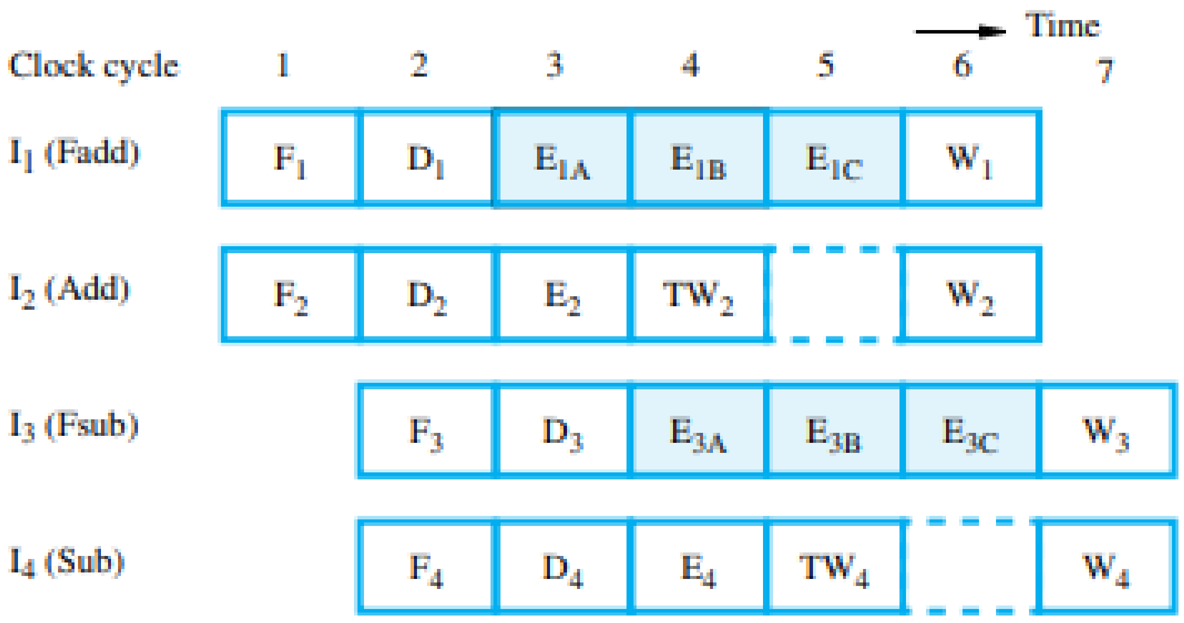
- Results are written into temporary registers, and later transferred to the permanent registers in correct program order. This is called **commitment step** because the effect of the instruction cannot be reversed after that point.

- During an exception, results of any subsequent instruction that has been executed would still be in temporary registers and can be safely discarded.

- A temporary register assumes the role of the permanent register whose data it is holding and given the same name. **Register renaming.**

Execution Completion

- It is desirable to use out-of-order execution, so that an execution unit is freed to execute other instructions ASAP.
- At the same time, instructions must be completed in program order to allow precise exceptions.
- The use of temporary registers
- Commitment unit



Time
Clock cycle 1 2 3 4 5 6 7
I1 (Fadd) F1 D1 E1A E1B E1C W1
I2 (Add) F2 D2 E2 TW2 W2
I3 (Fsub) F3 D3 E3A E3B E3C W3
I4 (Sub) F4 D4 E4 TW4 W4

(b) Using temporary registers

- **Commitment unit** is used as a special control unit to solve the problem of out-of-order execution by using a queue called **reorder buffer**.

- When an instruction reaches the head of the queue and execution of that instruction has been completed, results are transferred from temporary register to permanent register, and the instruction is removed from the queue.

- All the resources that were assigned to the instruction are released. – **Retired**. An instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired.

- Dispatch unit must ensure that all the resources needed for the execution of an instruction are available.

- In principle, I5 can be dispatched before I4, provided that a place is reserved in the reorder buffer for I4 and all instructions are retired in the correct order.
-
- A **deadlock** situation can arise when two units use a shared resource.
- Issuing instruction out-of-order may likely increase the complexity of the Dispatch unit.

Performance Considerations

Overview

- The execution time T of a program that has a dynamic instruction count N is given by:

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

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate.

- Instruction throughput is defined as the number of instructions executed per second.

$$P_s = \frac{R}{S}$$

Overview

- An *n*-stage pipeline has the potential to increase the throughput by *n* times.
- However, the only real measure of performance is the total execution time of a program.
- Higher instruction throughput will not necessarily lead to higher performance.
- Two questions regarding pipelining
  - How much of this potential increase in instruction throughput can be realized in practice?
  - What is good value of *n*?

Number of Pipeline Stages

- Since an *n*-stage pipeline has the potential to increase the throughput by *n* times, how about we use a 10,000-stage pipeline?
- As the number of stages increase, the probability of the pipeline being stalled increases.
- The inherent delay in the basic operations increases.
- Hardware considerations (area, power, complexity,…)