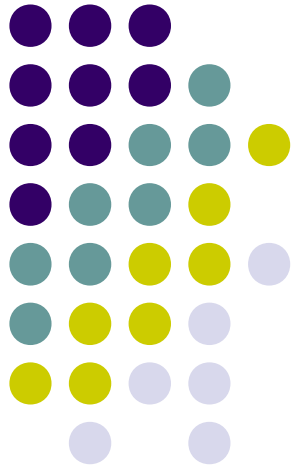
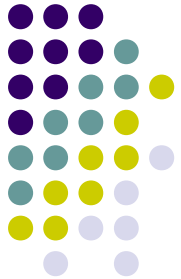
# Chapter 8. Pipelining

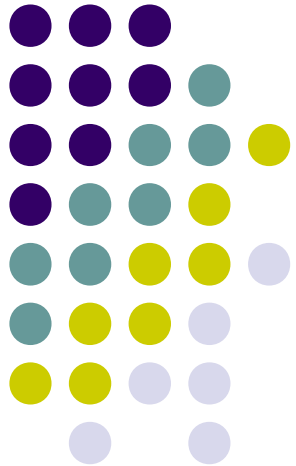


# Overview

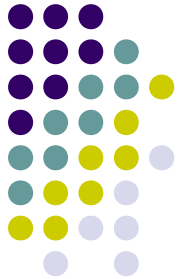


- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

# Basic Concepts



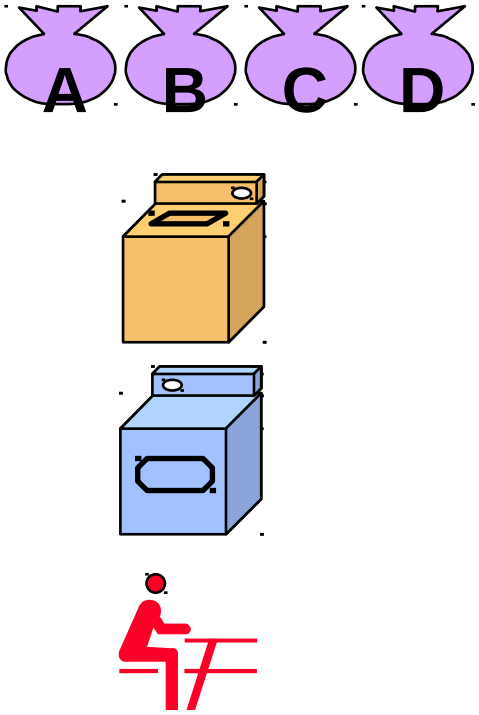
# Making the Execution of Programs Faster



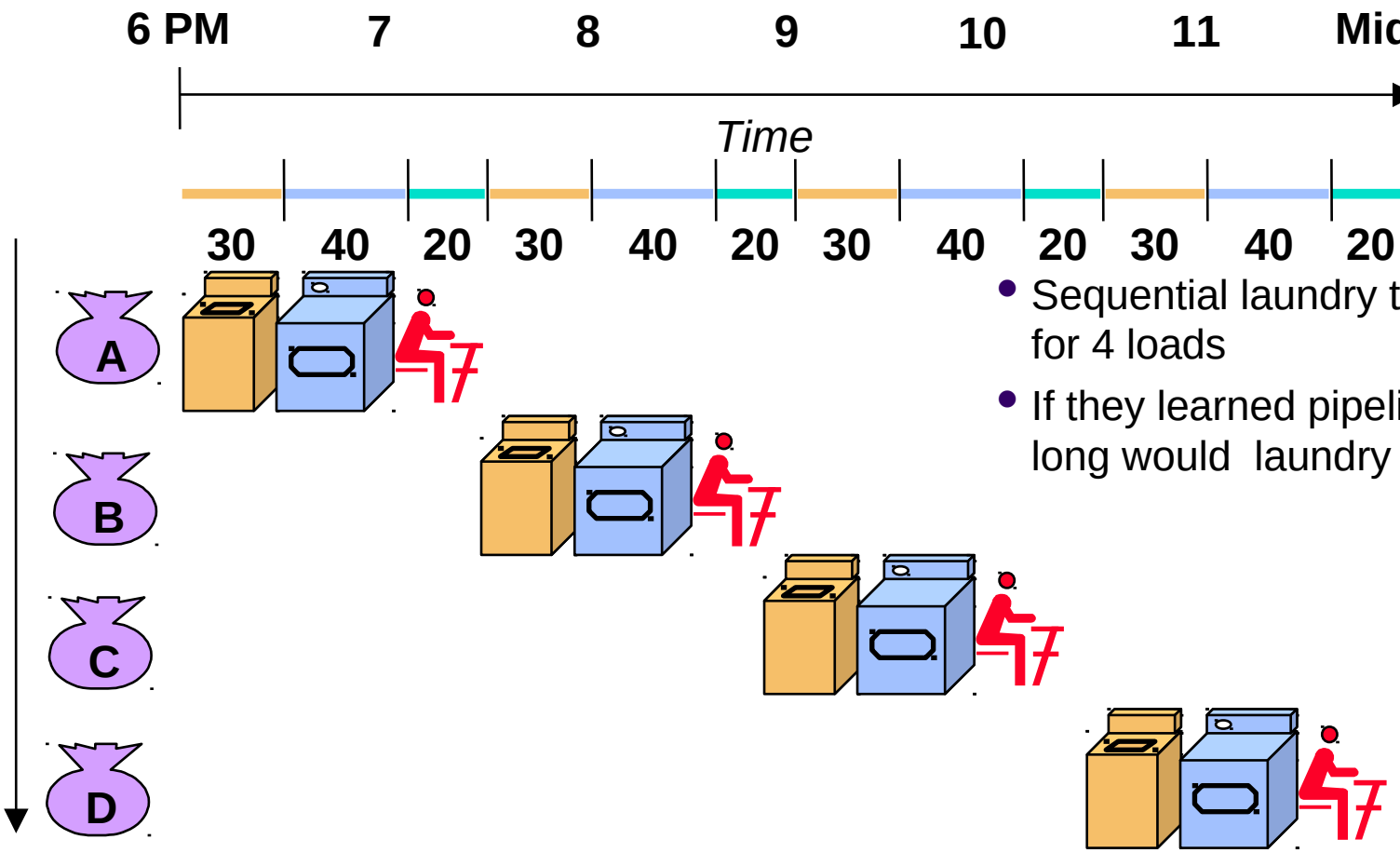
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- The number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

# Traditional Pipeline Concept

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

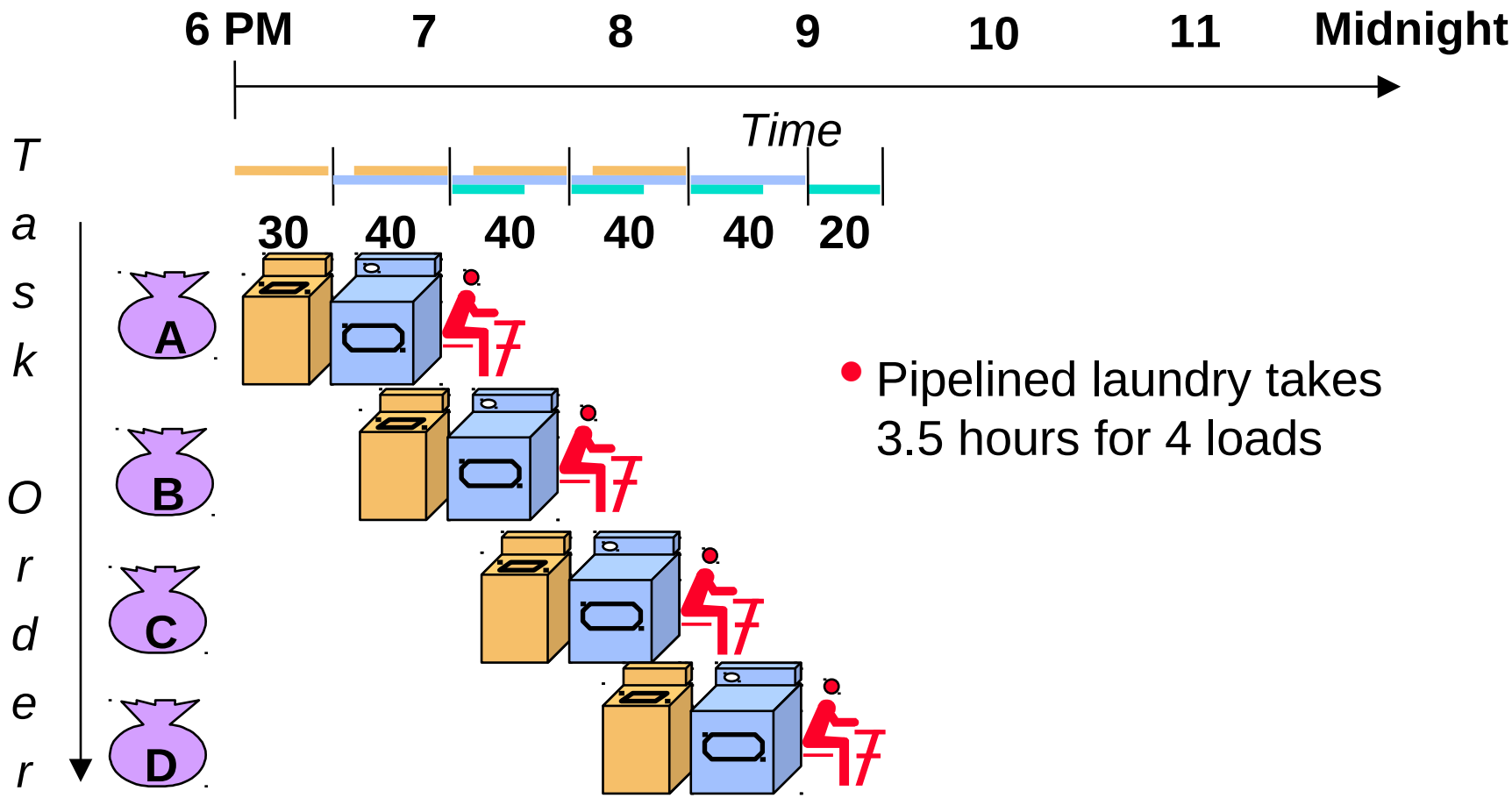


# Traditional Pipeline Concept



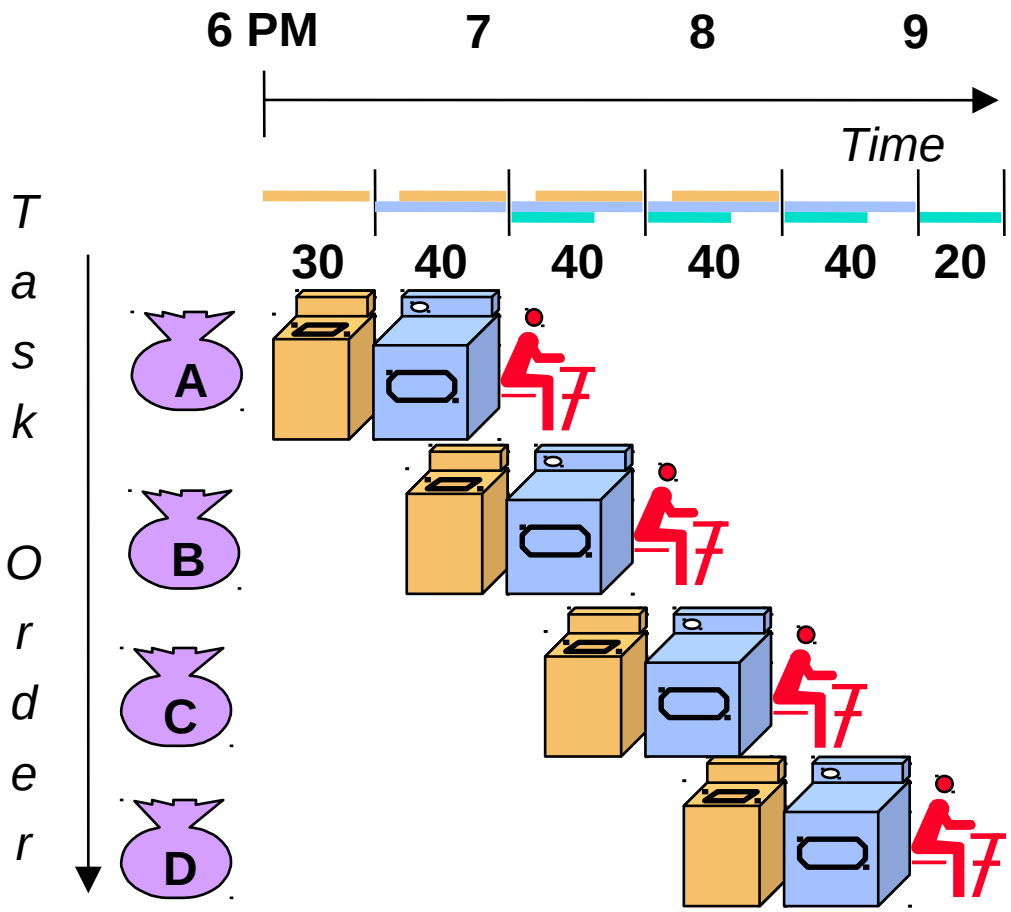

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

# Traditional Pipeline Concept




- Pipelined laundry takes 3.5 hours for 4 loads

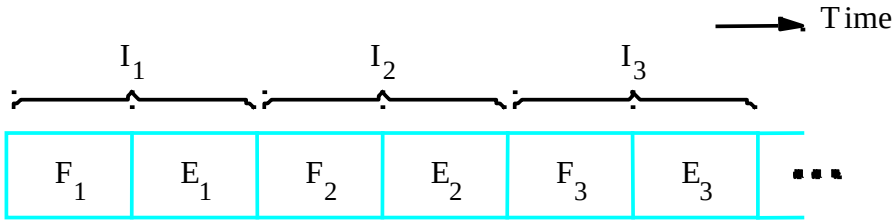
# Traditional Pipeline Concept



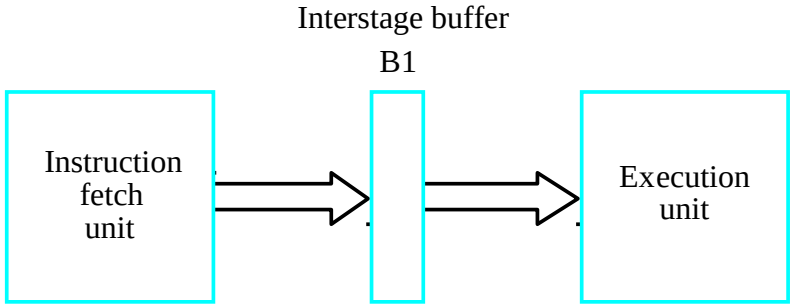

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

# Use the Idea of Pipelining in a Computer

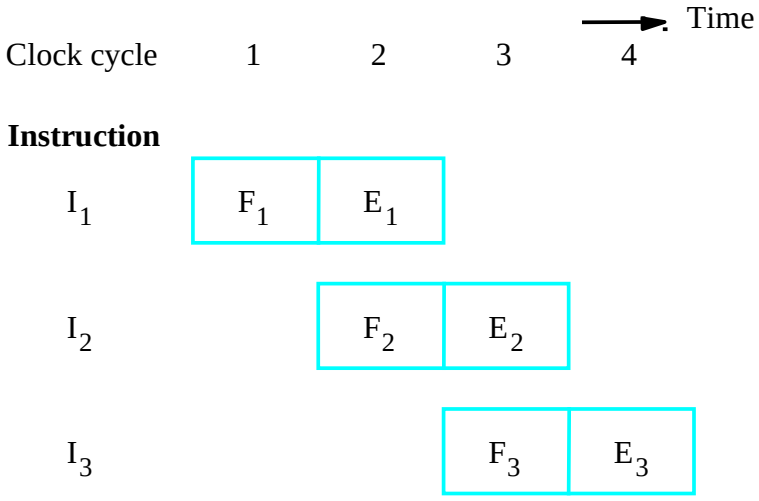
Fetch + Execution




(a) Sequential execution




(b) Hardware organization

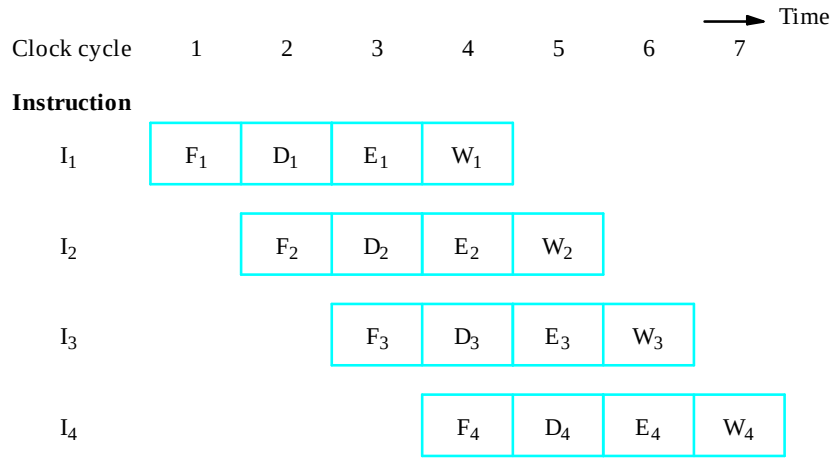



(c) Pipelined execution

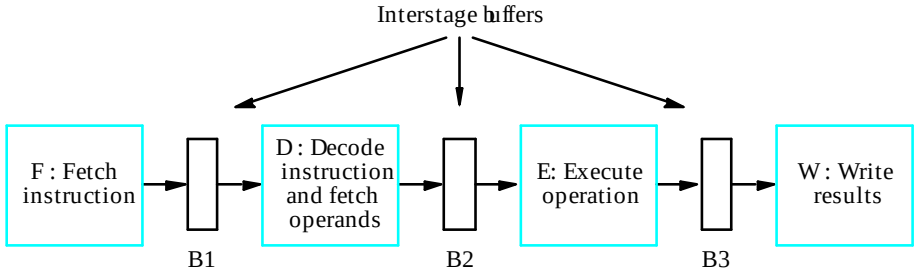
Figure 8.1. Basic idea of instruction pipelining.

# Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write



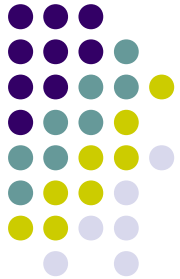

(a) Instruction execution divided into four steps




(b) Hardware organization

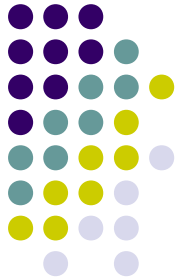
Figure 8.2. A 4-stage pipeline.

# 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 compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, 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

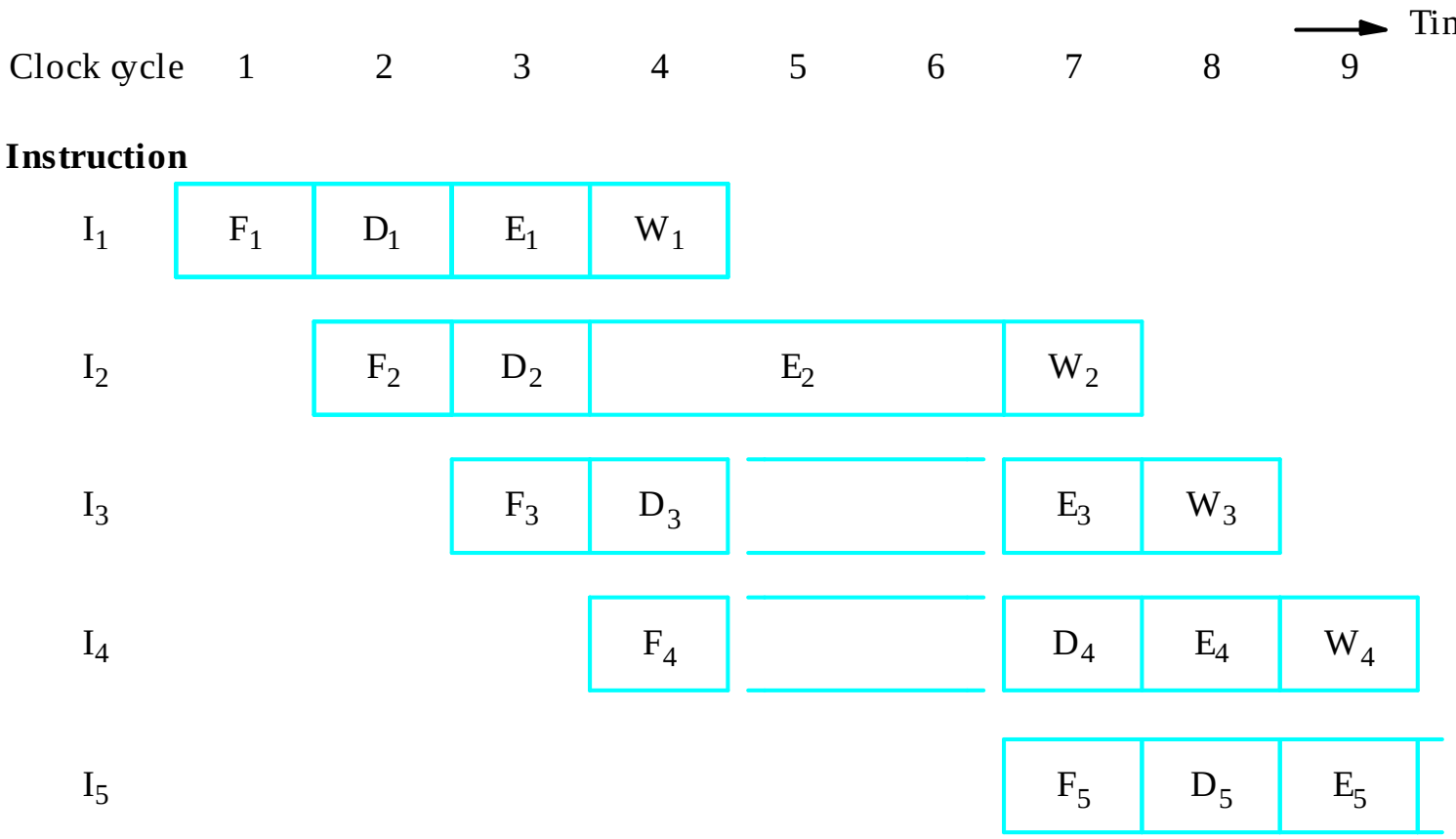
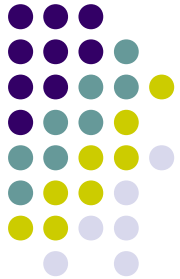



Figure 8.3. Effect of an execution operation taking more than one clock cycle.

# 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.

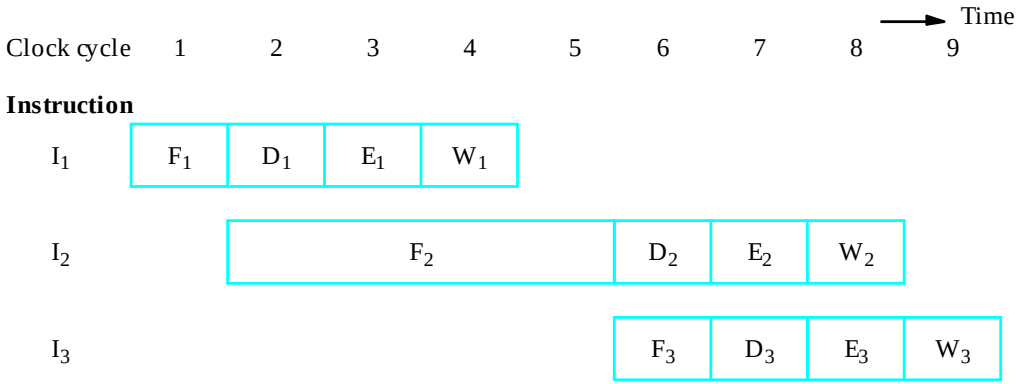
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.

- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.

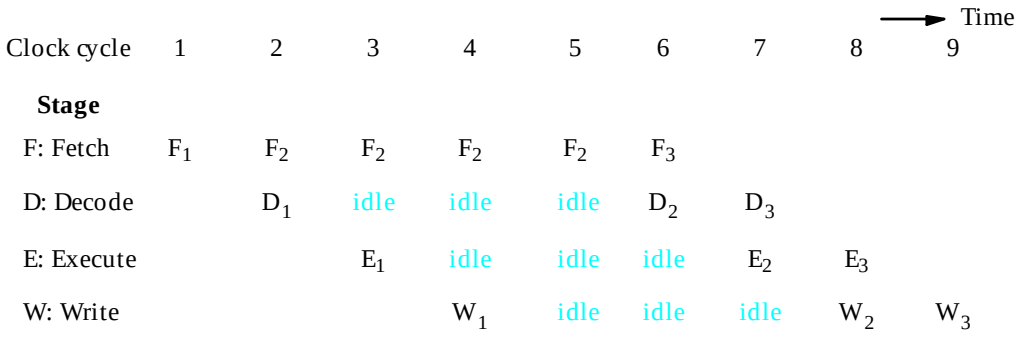
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

# Pipeline Performance

Instruction hazard




(a) Instruction execution steps in successive clock cycles




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

Idle periods – stalls (bubbles)

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

# Pipeline Performance

Structural hazard

Load X(R1), R2

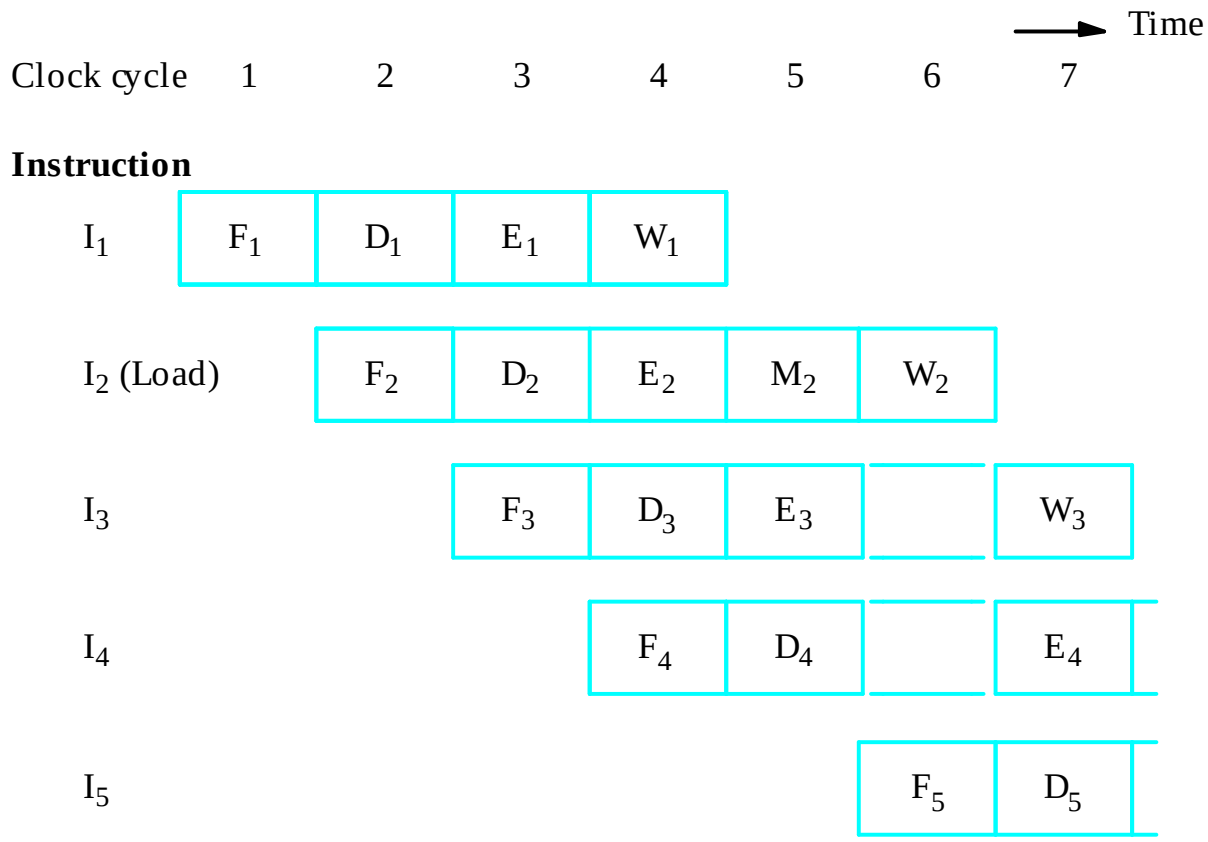
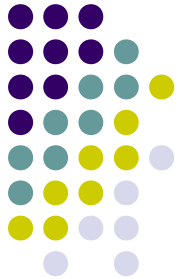



Figure 8.5. Effect of a Load instruction on pipeline timing.

# Pipeline Performance

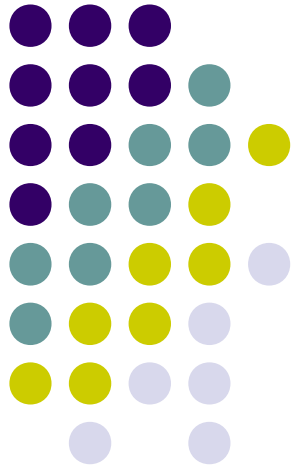


- Pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- Identify all hazards that may cause the pipeline to stall and find ways to minimize their impact.

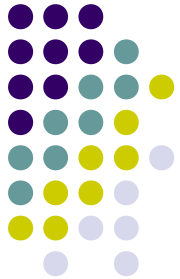
# Quiz

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

# Data Hazards



# 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

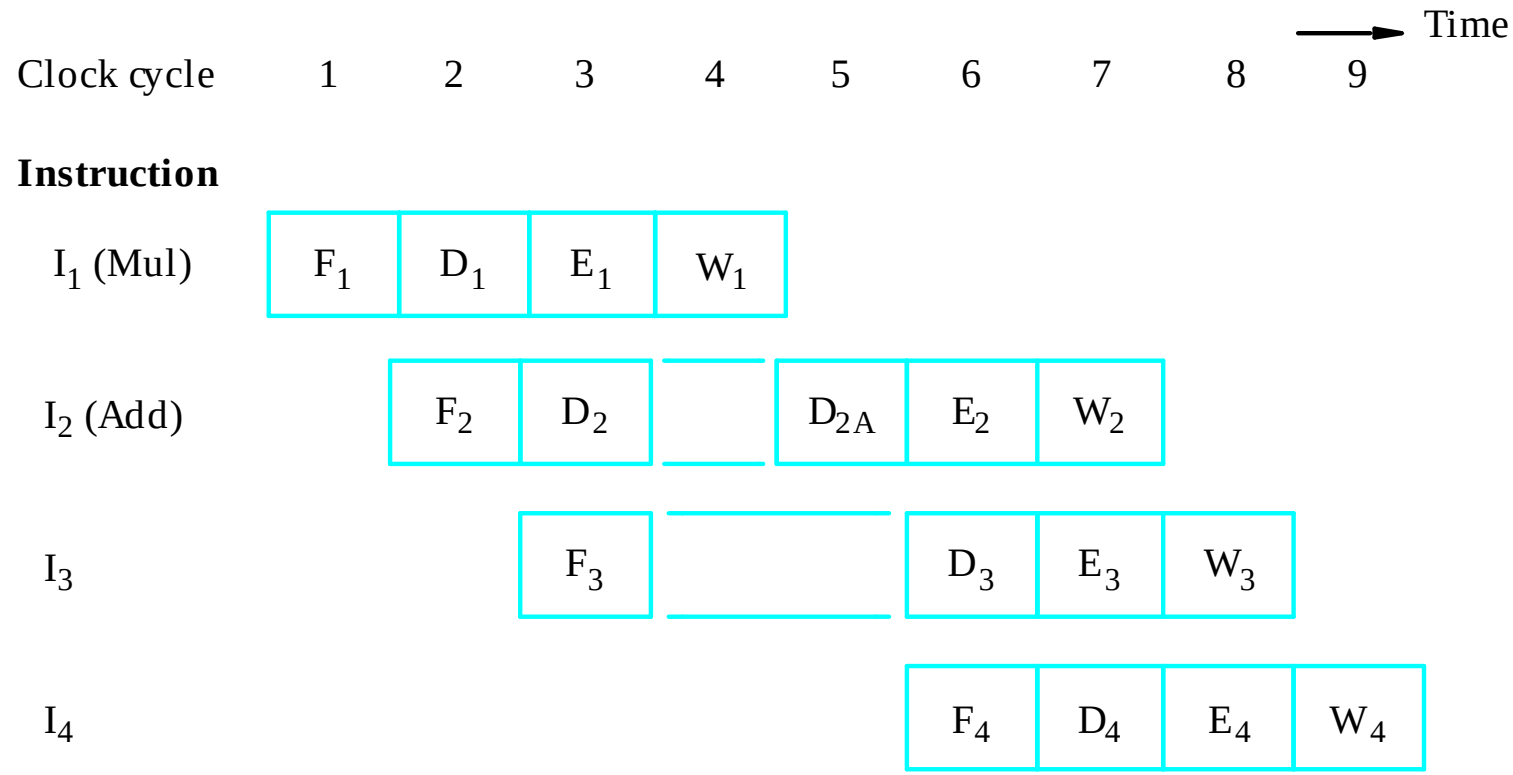
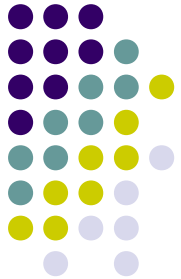
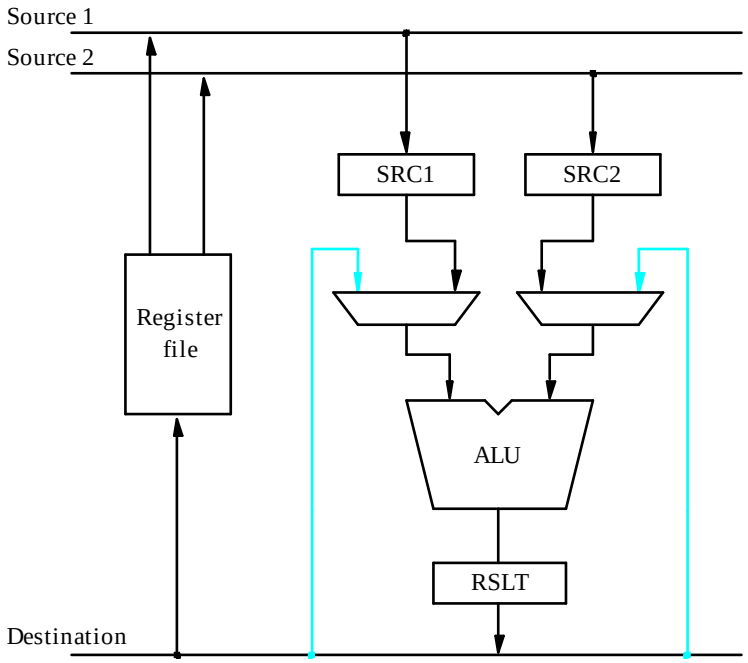



Figure 8.6. Pipeline stalled by data dependency between $D_2$ and $W_1$.

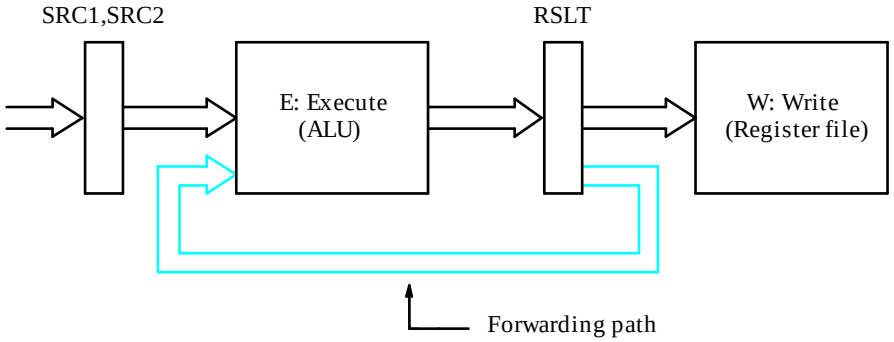
# Operand Forwarding



- 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.



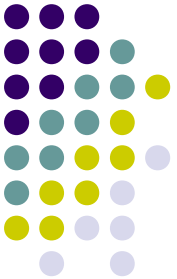

(a) Datapath




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

Figure 8.7. Operand forwarding in a pipelined processor.

# Handling Data Hazards in Software

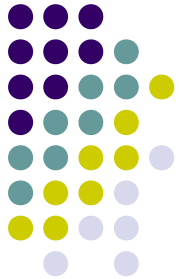


- Let the compiler detect and handle the hazard:

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

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

# Side Effects



- Last example is explicit and easily detected.
- 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?)

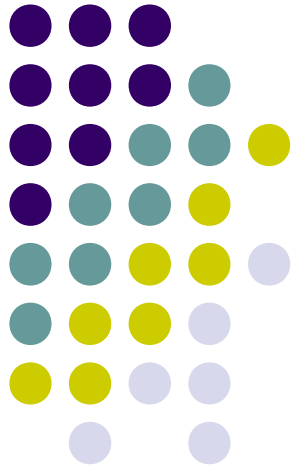
- Example: conditional code flags:

  Add  R1, R3

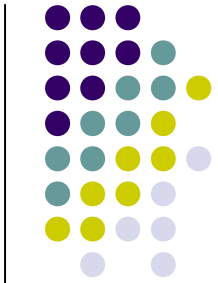
  AddWithCarry  R2, R4

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

# Instruction Hazards



# Overview



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

# Unconditional Branches

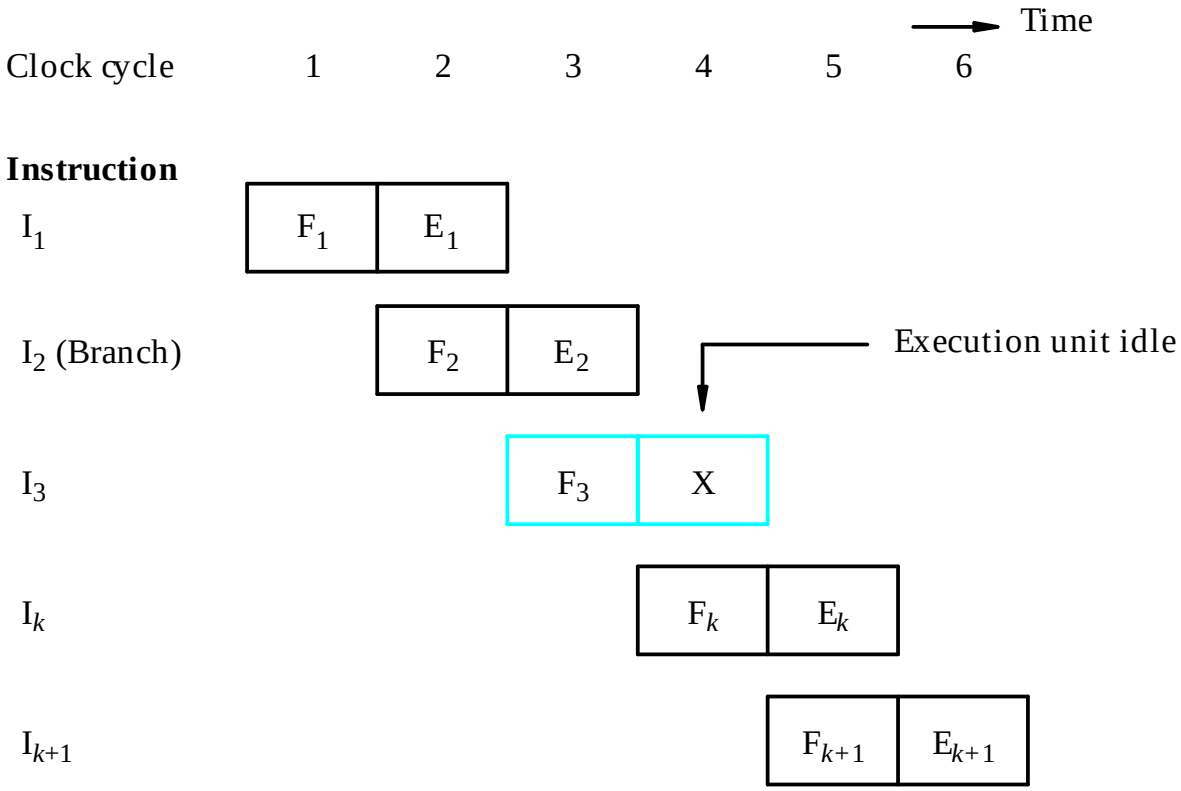
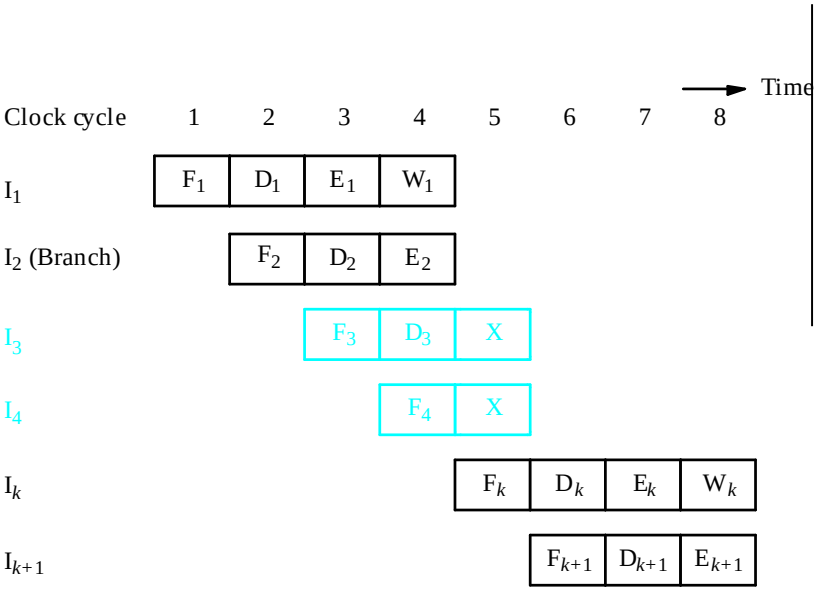



Figure 8.8. An idle cycle caused by a branch instruction.

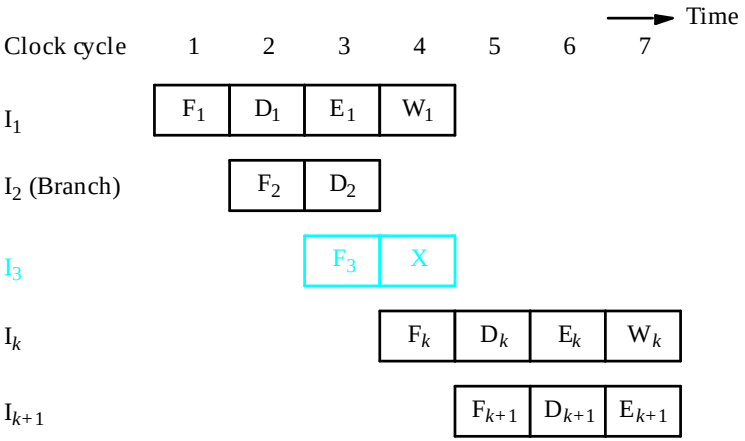
# Branch Timing

- Branch penalty

- Reducing the penalty




(a) Branch address computed in Execute stage




(b) Branch address computed in Decode stage

Figure 8.9. Branch timing.

# Instruction Queue and Prefetching

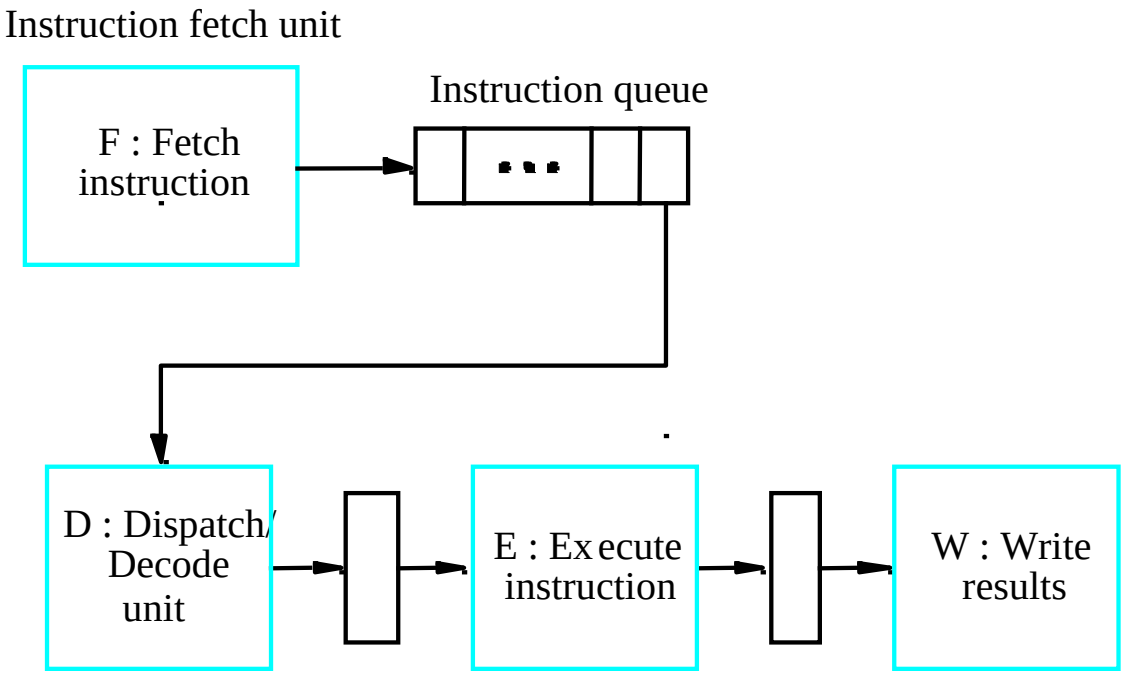
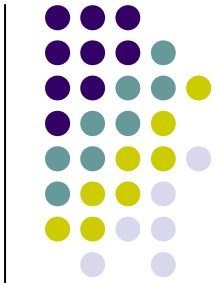


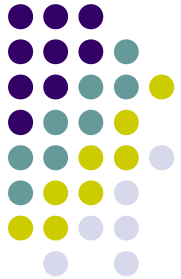

Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2*b*.

# Conditional Braches



- It introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
- The decision to branch cannot be made until the execution of that instruction has been completed.
- Branch instructions represent about 20% of the dynamic instruction count of most programs.

# Delayed Branch



- The instructions in the delay slots are always fetched. Hence, arrange for them to be fully executed whether or not the branch is taken.
- 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

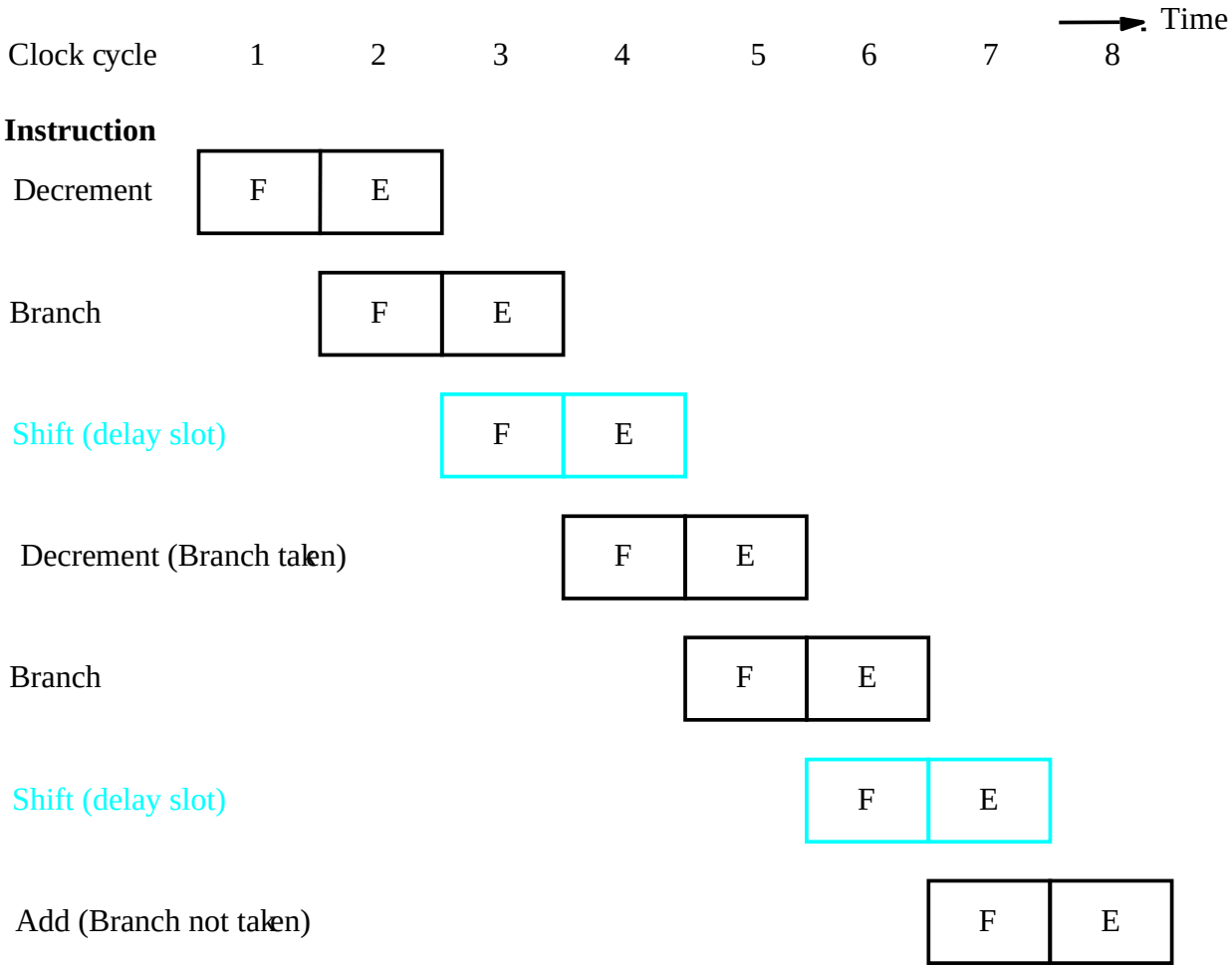
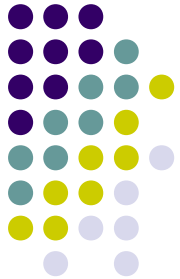



Figure 8.13. Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12.

# 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.
- Need to be careful so that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.

# Incorrectly Predicted Branch

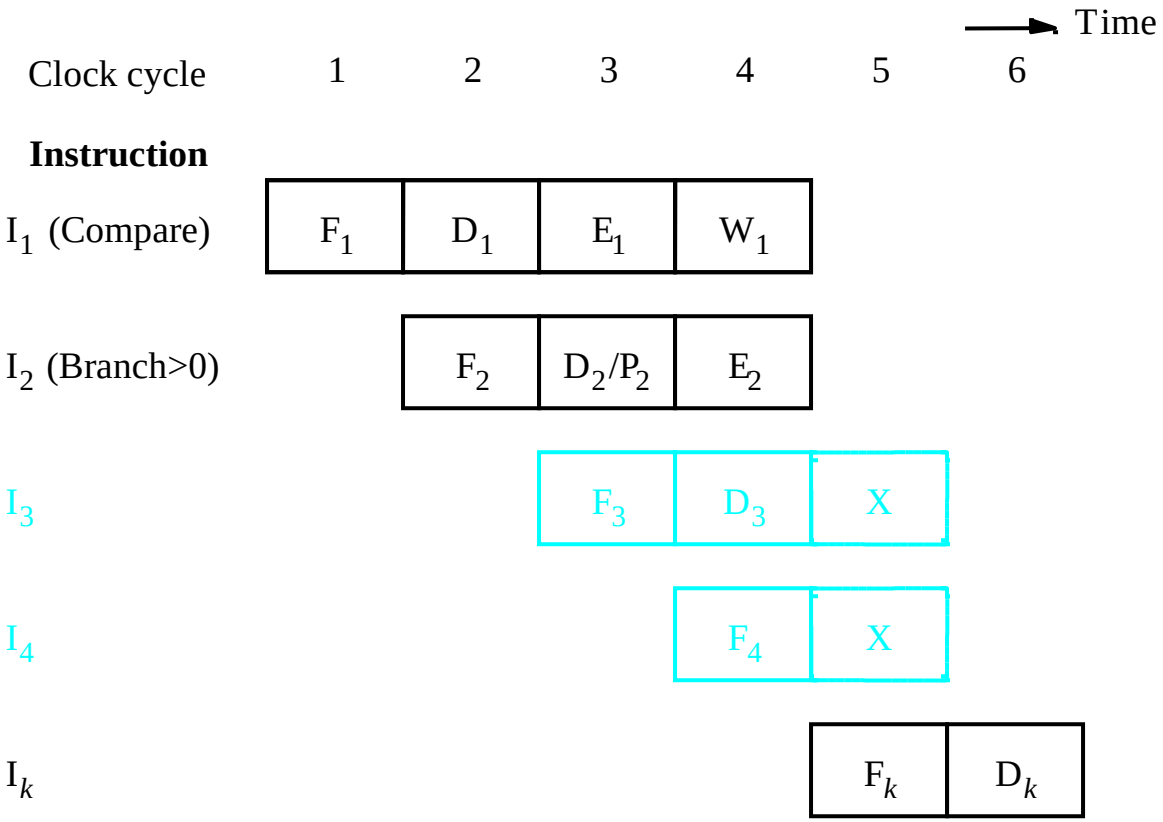
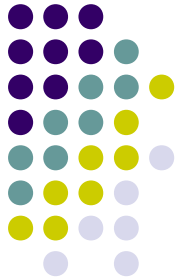



Figure 8.14. Timing when a branch decision has been incorrectly predicted as not taken.

# 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.

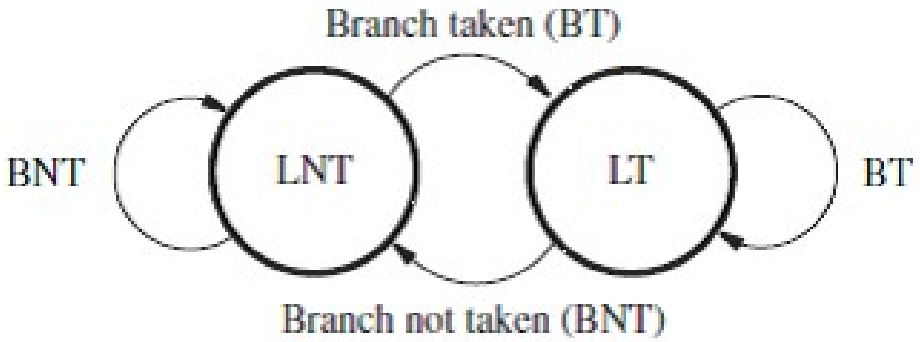
- The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded.

- In dynamic branch prediction schemes, the 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)

- Need one bit of history information for each branch instruction.

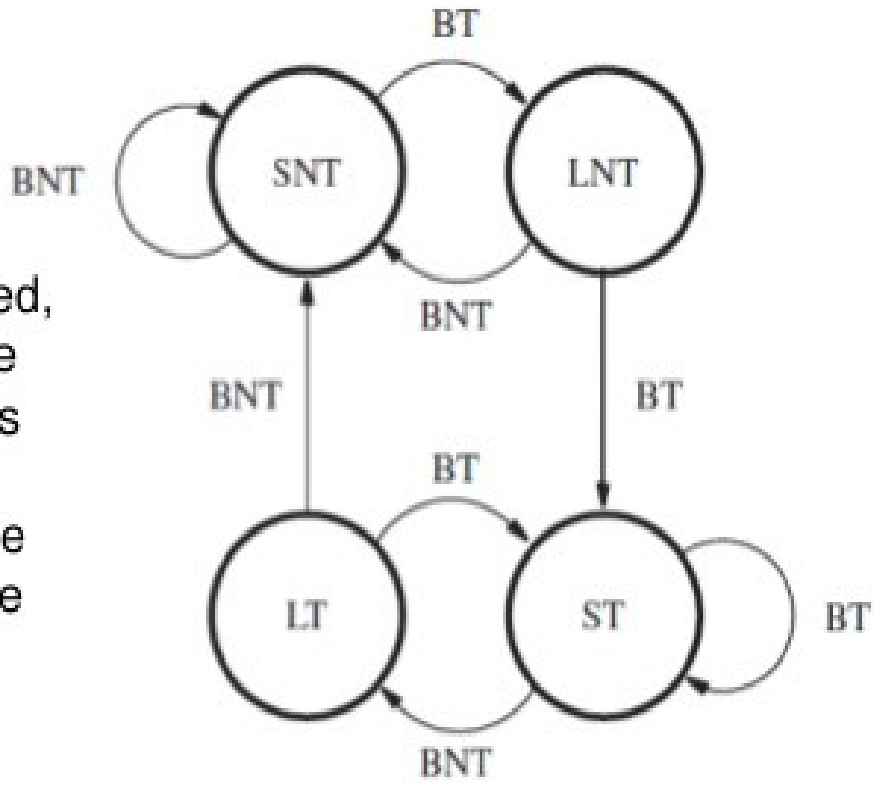
- Will work well for Loop (will be incorrect in the last pass)



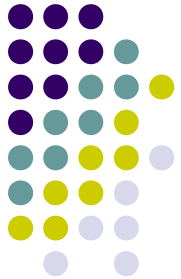

(a) A 2-state algorithm

# A 4-state algorithm

- 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



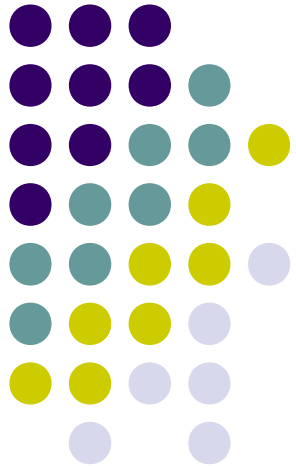
- Better performance can be achieved by keeping more information about execution history.
- Thus requires four-stage machine (2 bits)

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

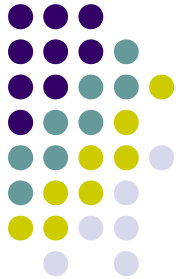
- State information may be kept in a look-up table.

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

# Influence on Instruction Sets

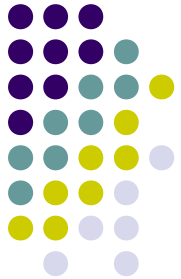


# Overview



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

# Addressing Modes

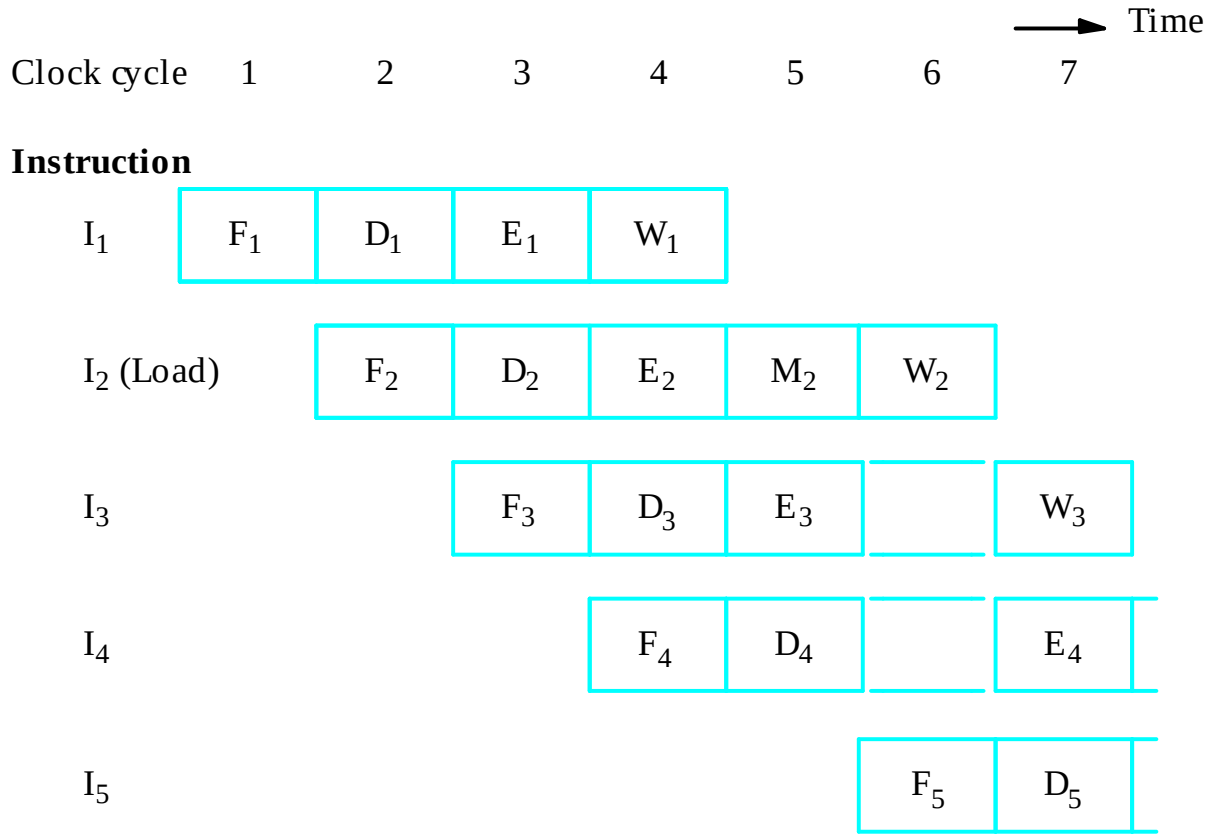


- Addressing modes: simple vs. complex.

- 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

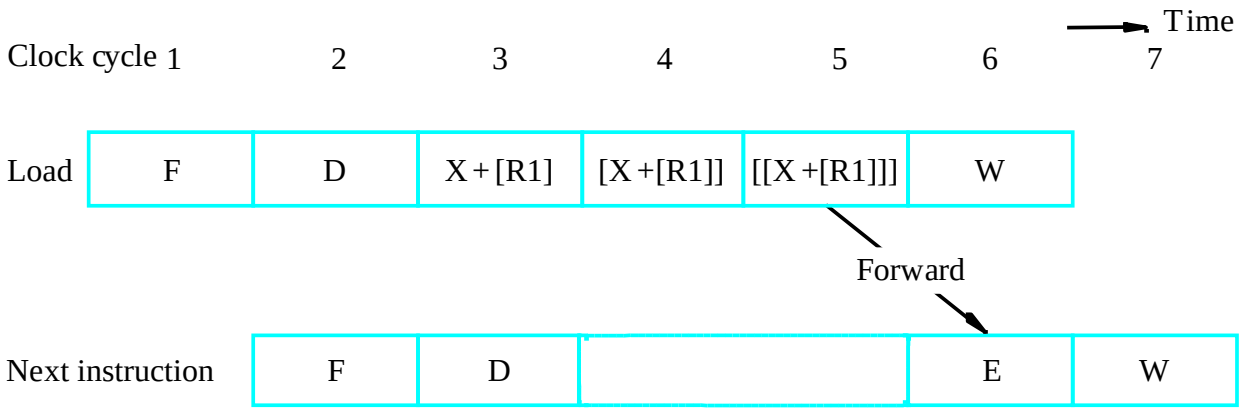



Load (R1), R2

Figure 8.5. Effect of a Load instruction on pipeline timing.

# Complex Addressing Mode

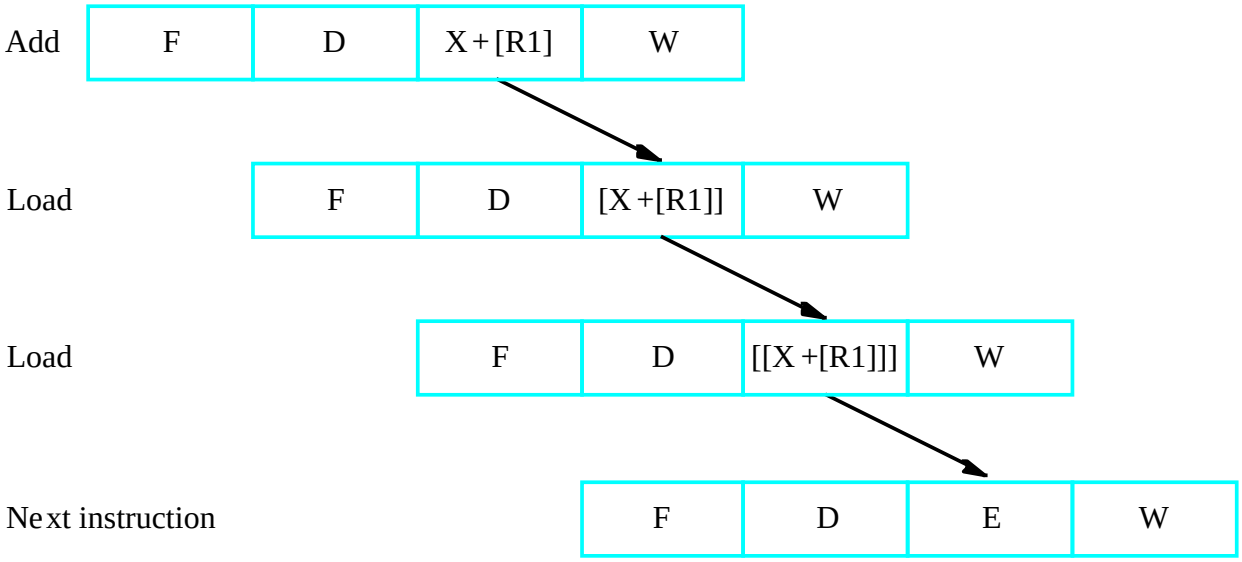
Load (X(R1)), R2




(a) Complex addressing mode

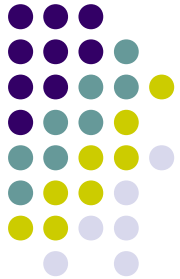
# Simple Addressing Mode

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



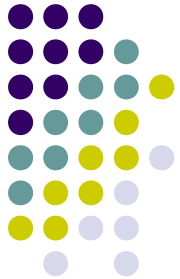

(b) Simple addressing mode

# Addressing Modes



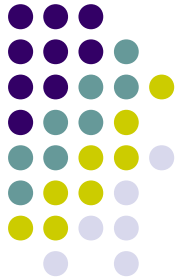
- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- Advantage: reducing the number of instructions / program space
- Disadvantage: cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- Conclusion: complex addressing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

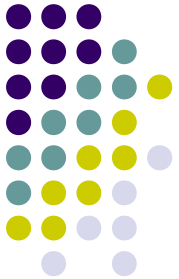
(a) A program fragment

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

(b) Instructions reordered

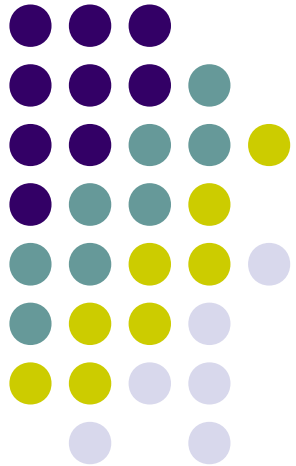
Figure 8.17. Instruction reordering.

# 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

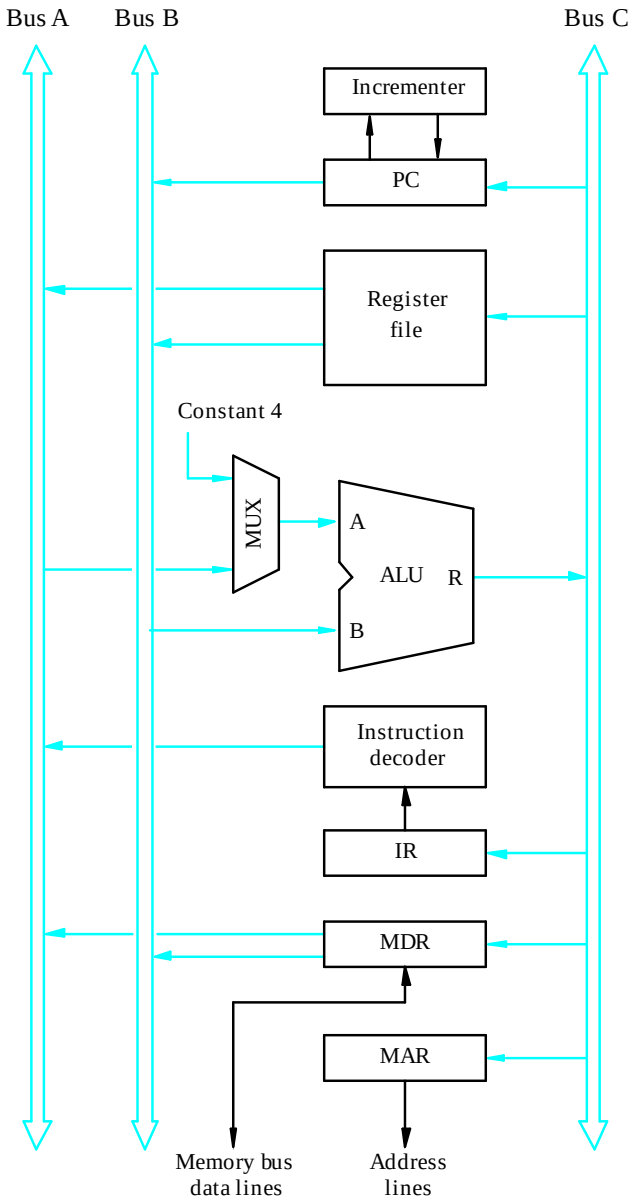
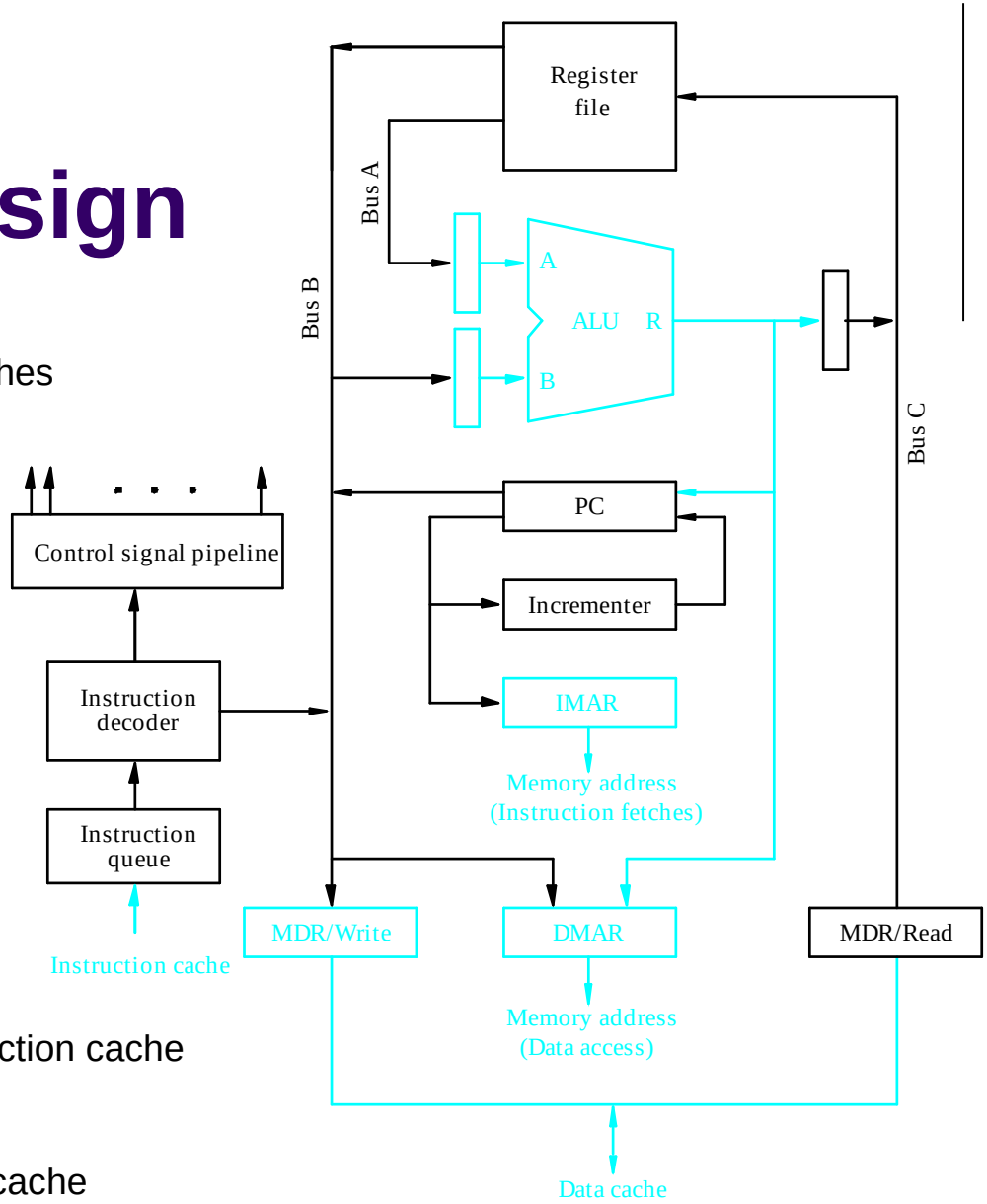



Figure 7.8. Three-bus organization of the datapath.

# Pipelined Design

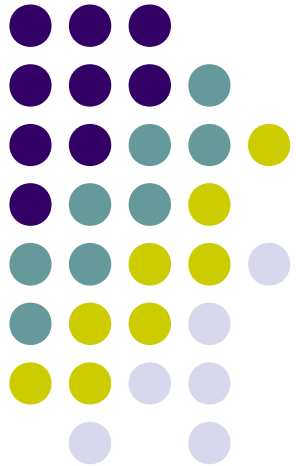
- Separate instruction and data caches
- PC is connected to IMAR
- DMAR
- Separate MDR
- Buffers for ALU
- Instruction queue
- Instruction decoder output



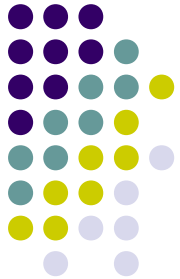

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two regs
- Writing into one register in the reg file
- Performing an ALU operation

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

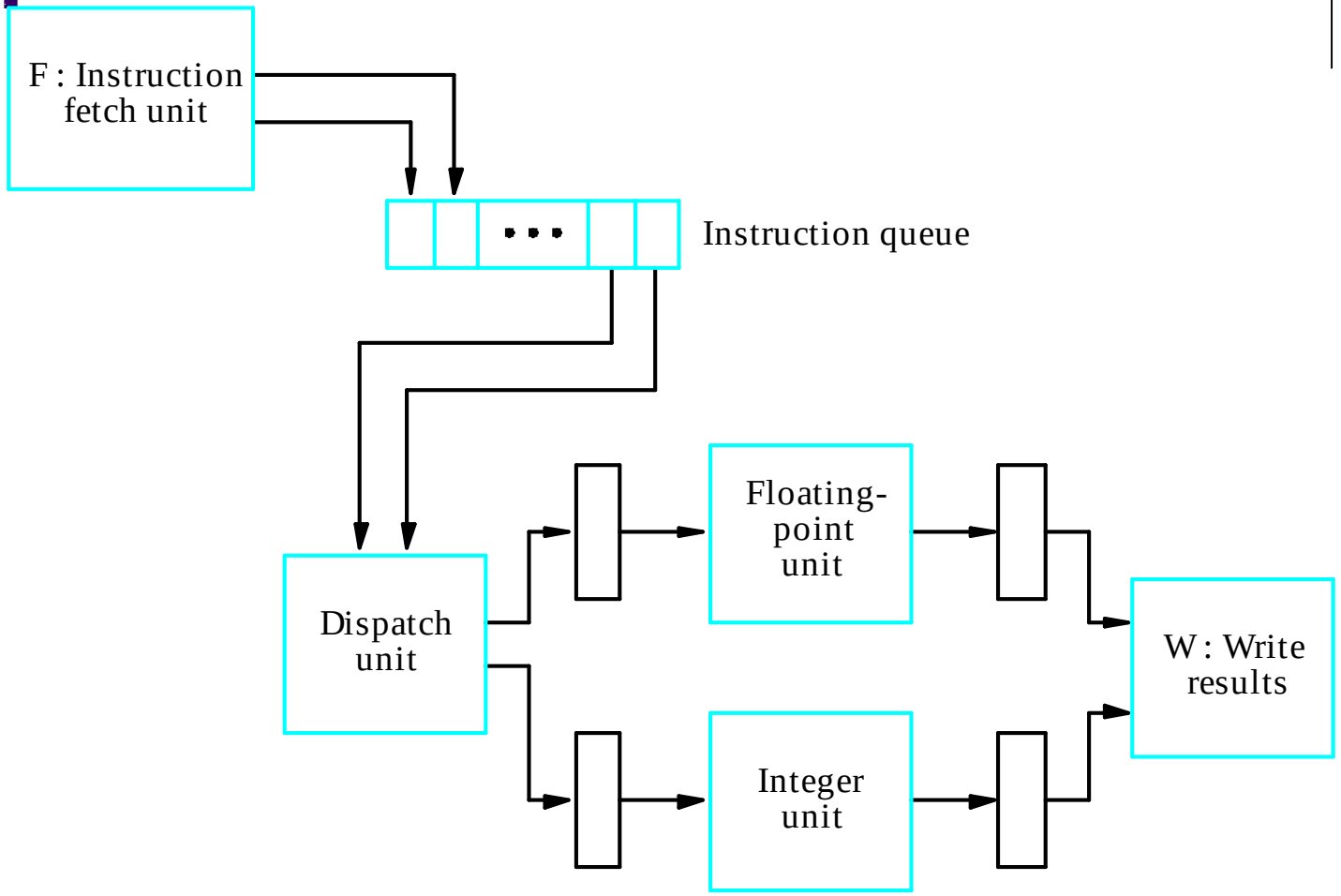



Figure 8.19. A processor with two execution units.

# Timing

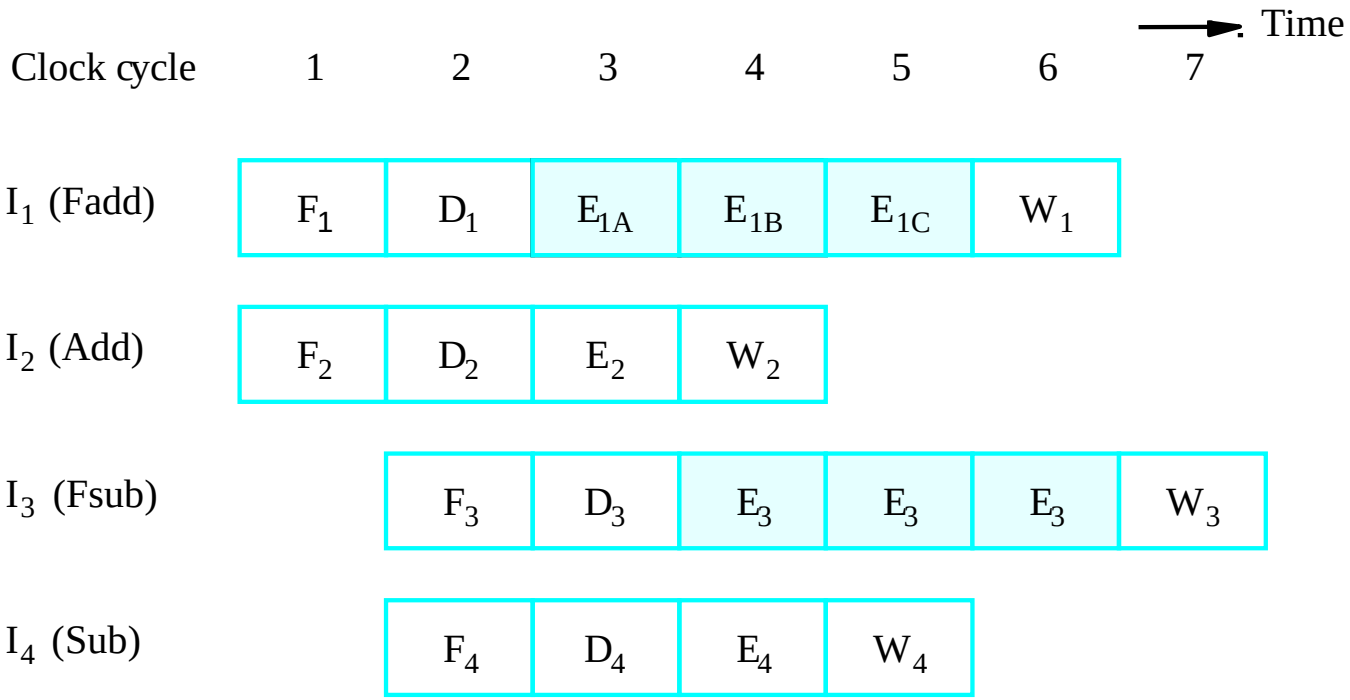
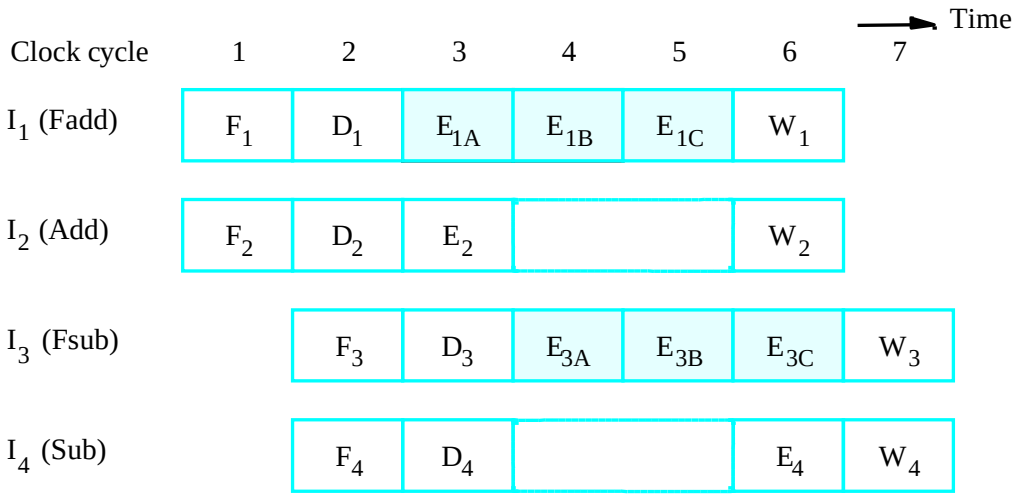


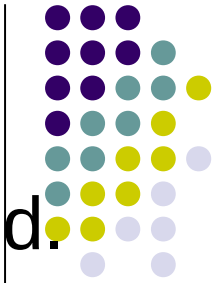

Figure 8.20. An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

# Out-of-Order Execution

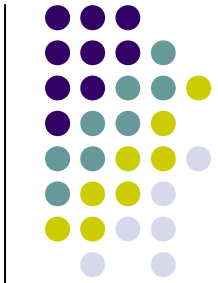
- Hazards
- Exceptions
- Imprecise exceptions
- Precise exceptions




(a) Delayed write

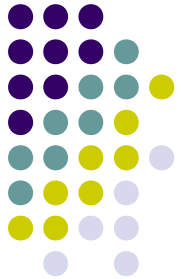


- If the I2 depends on I1, then I2 will be delayed.
- Complication due to exception such as bus error or illegal operation.
- If result of I2 is written back to register and I1 causes an exception: **Inconsistent state.**
- PC may point to instruction for which exception occurred.
- One or more of the 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.



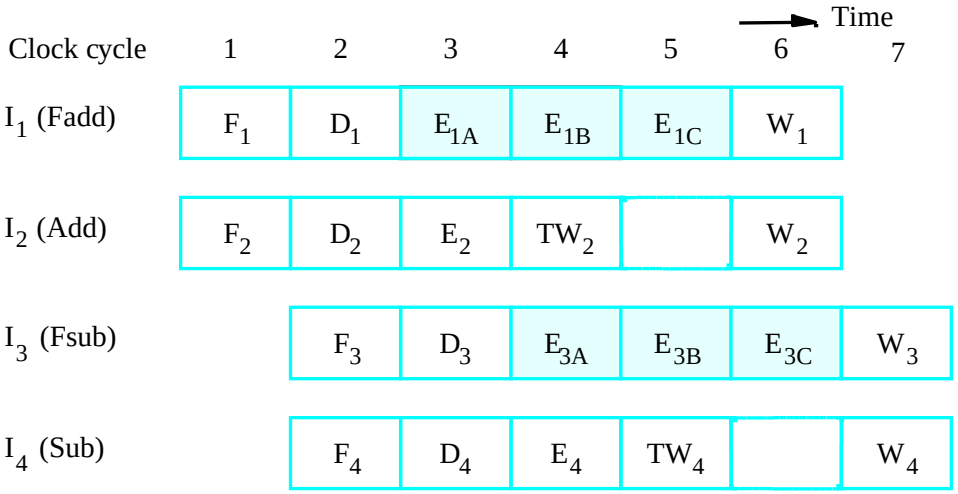
- Results are written into temporary registers, and later transferred to the permanent registers in correct program order. This is called **commitment step** because the effect of the instruction cannot be reversed after that point.

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

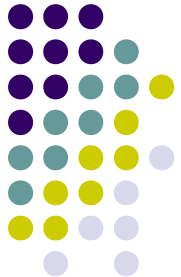
- A temporary register assumes the role of the permanent register whose data it is holding and given the same name. **Register renaming.**

# 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



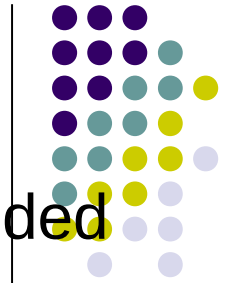

(b) Using temporary registers



- **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.

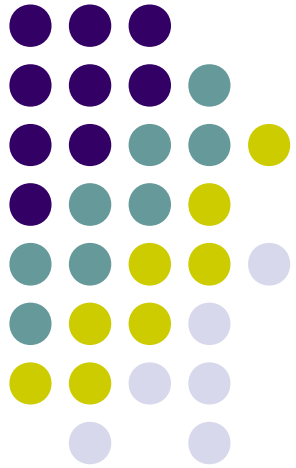
- All the resources that were assigned to the instruction are released. – **Retired**. An instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired.



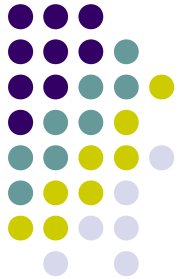
- Dispatch unit must ensure that all the resources needed for the execution of an instruction are available.

- 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:

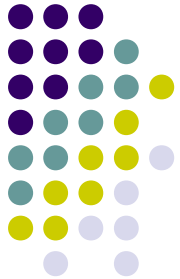
$$T = \frac{N \times S}{R}$$

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.

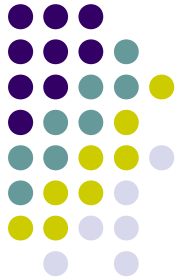
$$P_s = \frac{R}{S}$$

# 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, …)