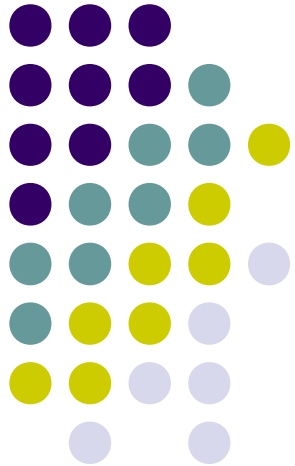
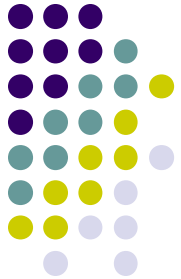
Chapter 8. Pipelining

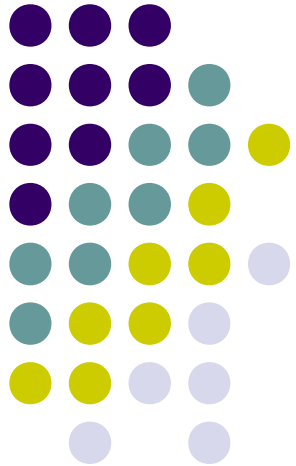


Overview

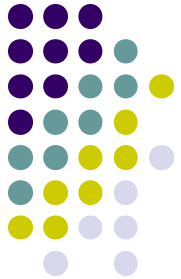


- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.[No of work done at a given time]
- Pipelined organization requires sophisticated compilation techniques.

Basic Concepts

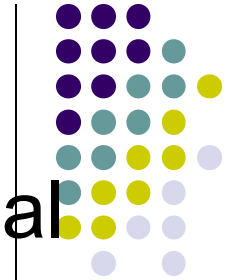


Making the Execution of Programs Faster



- 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.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

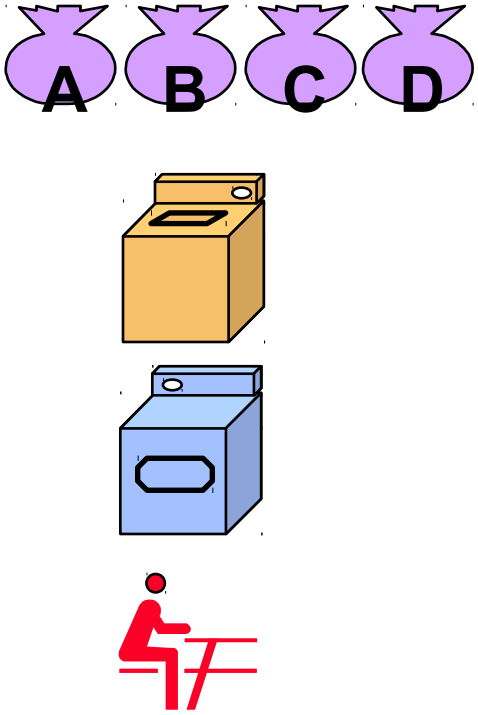
pipeline



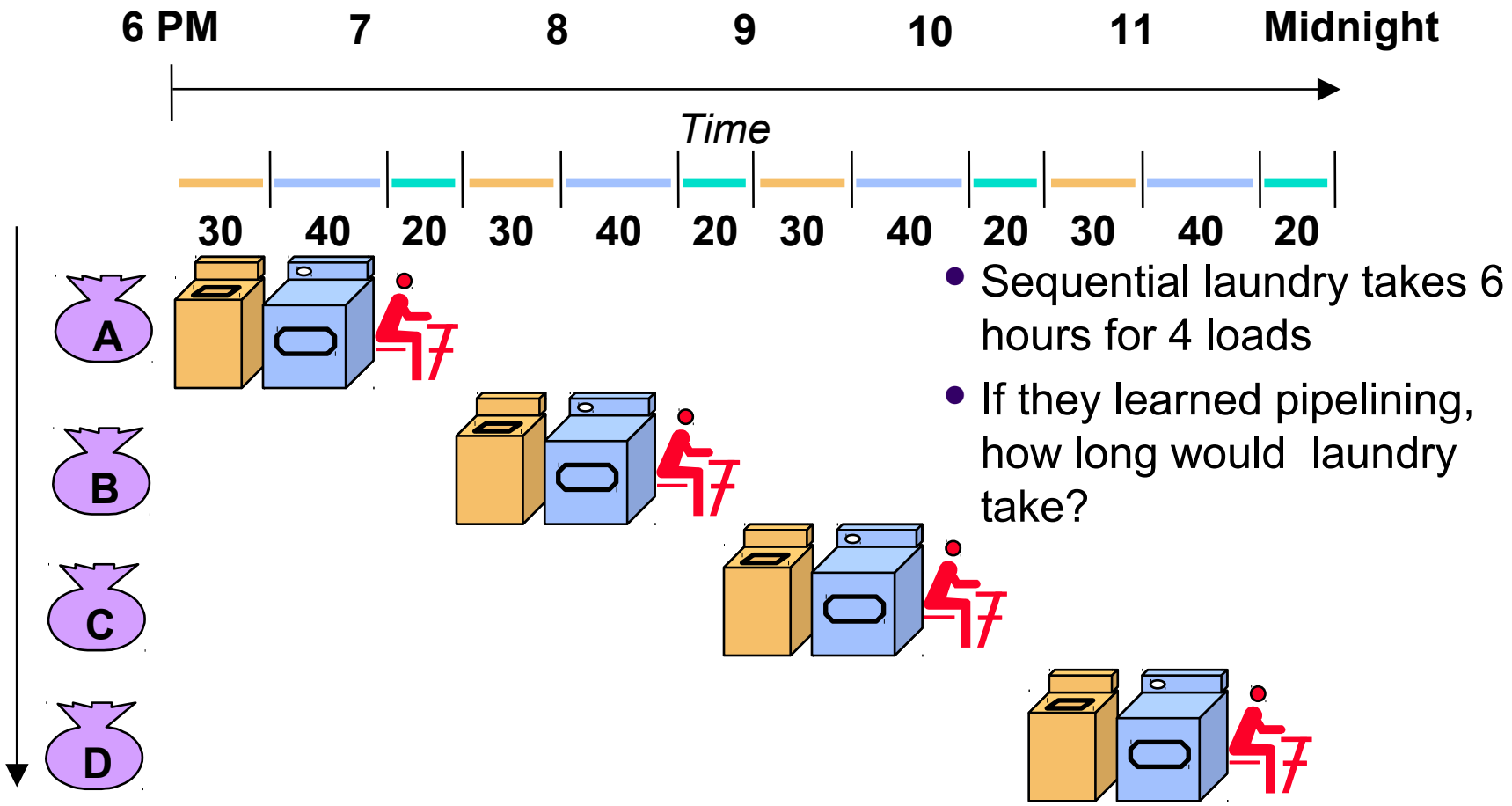
- It is technique of decomposing a sequential process into suboperation, with each suboperation completed in dedicated segment.
- Pipeline is commonly known as an assembly line operation.
- It is similar like assembly line of car manufacturing.
- First station in an assembly line set up a chasis, next station is installing the engine, another group of workers fitting the body.

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

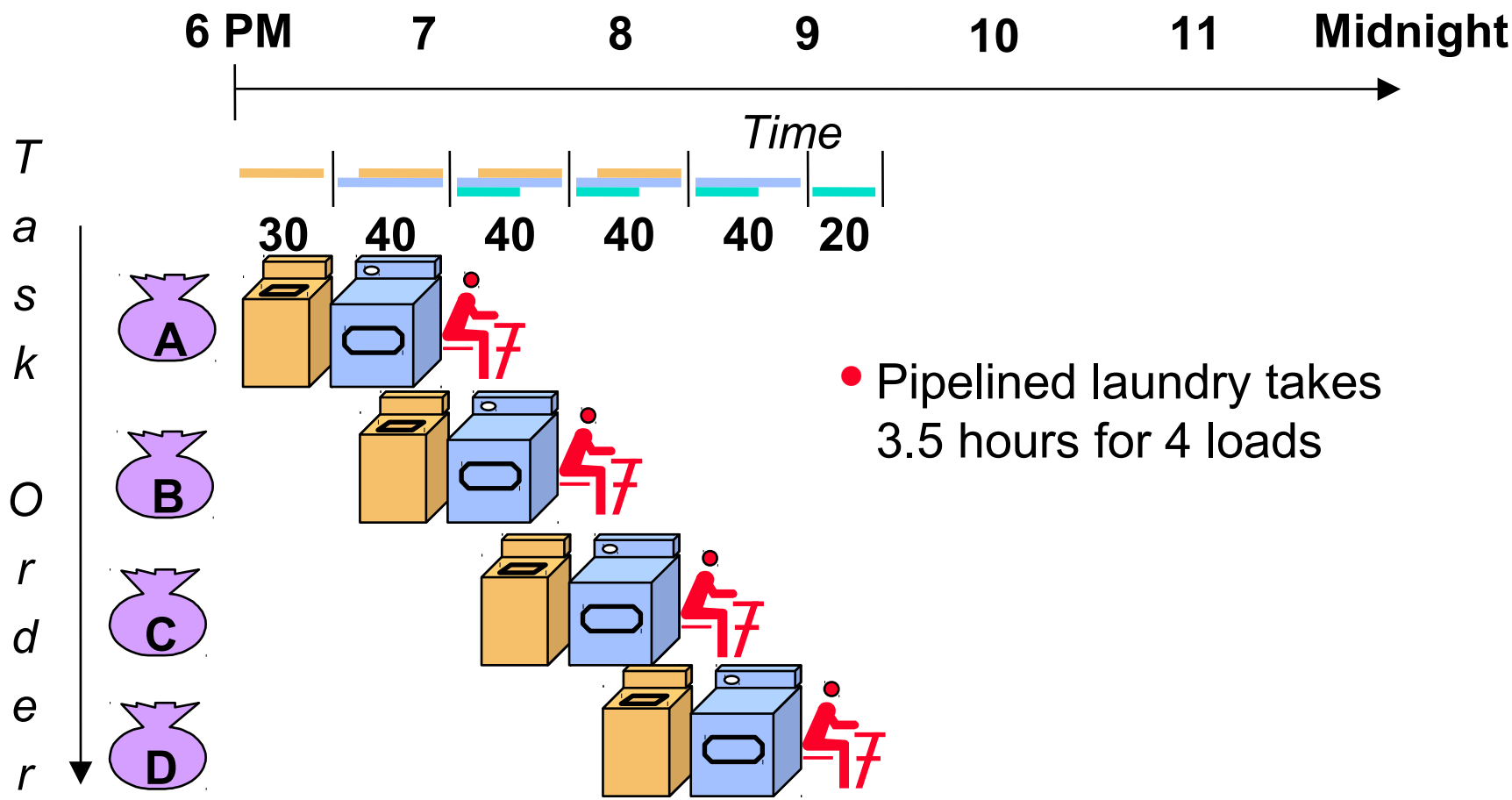


Traditional Pipeline Concept



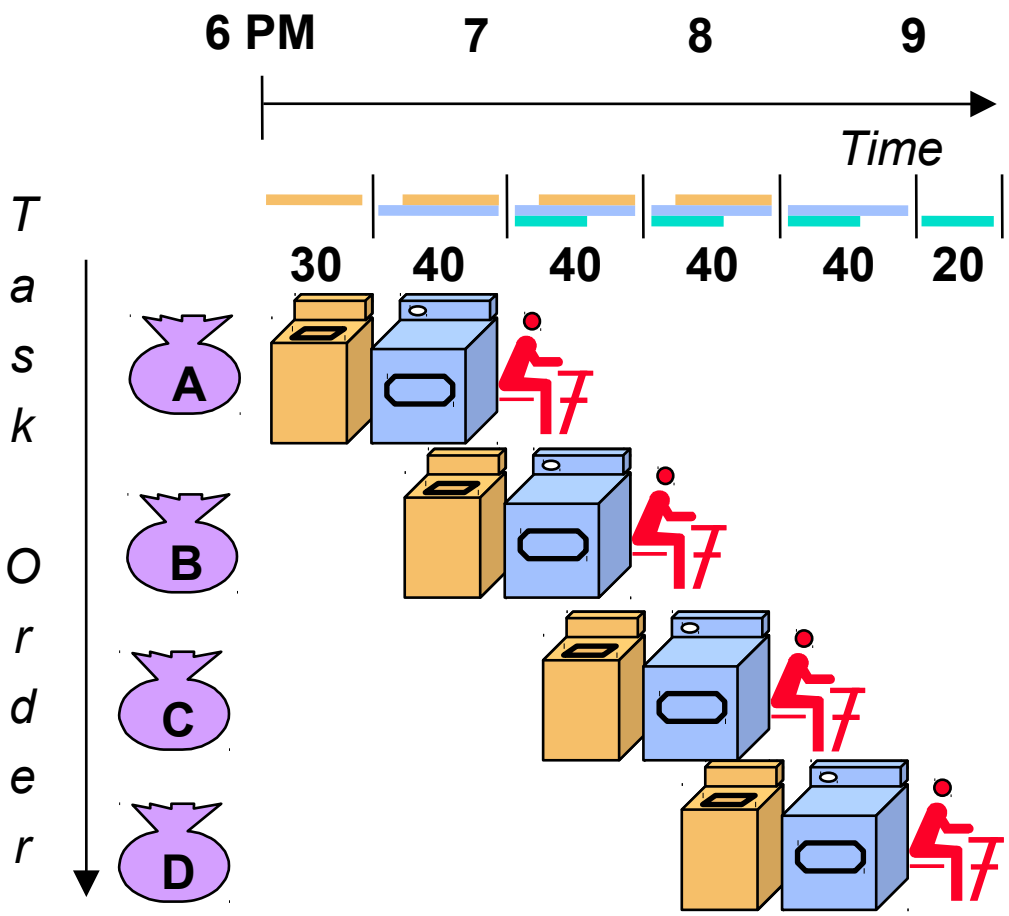
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



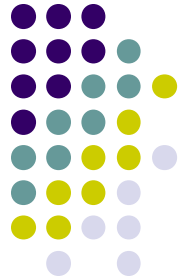
- Pipelined laundry takes 3.5 hours for 4 loads

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

Idea of pipelining in computer

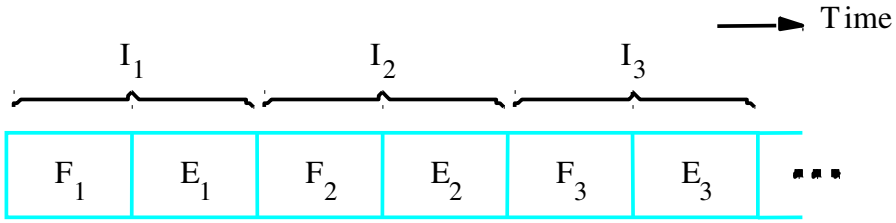


- The processor execute the program by fetching and executing instructions. One after the other.

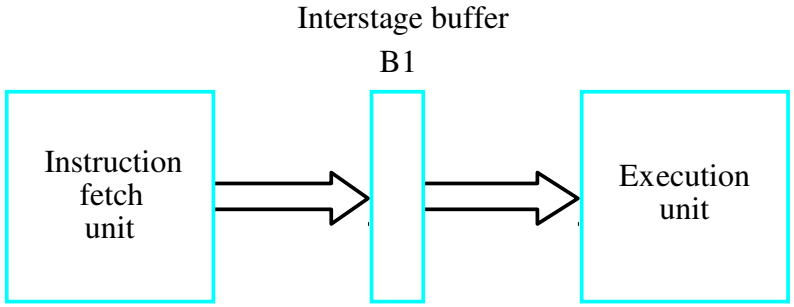
- Let F_i and E_i refer to the fetch and execute steps for instruction I_i

Use the Idea of Pipelining in a Computer

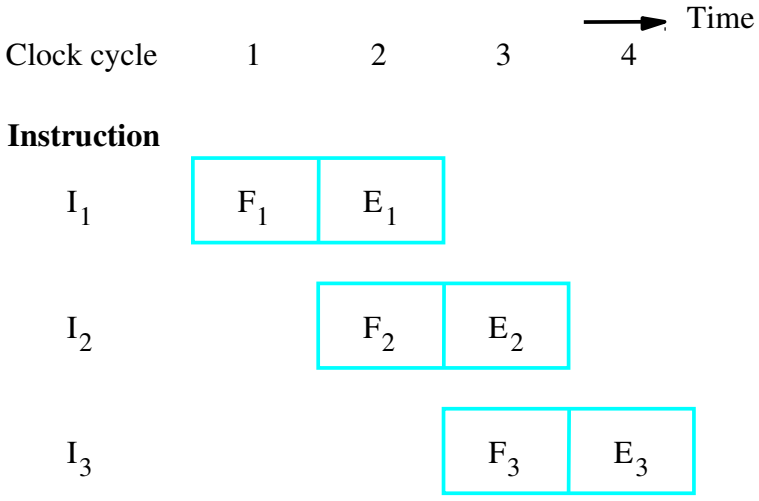
Fetch + Execution



(a) Sequential execution



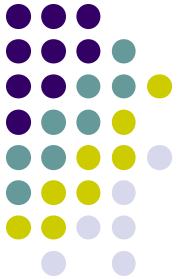
(b) Hardware organization



(c) Pipelined execution

Figure 8.1. Basic idea of instruction pipelining.

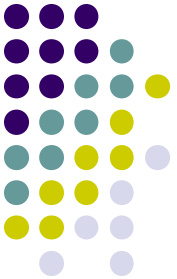
Contd.,



- Computer that has two separate hardware units, one for fetching and another for executing them.
- the instruction fetched by the fetch unit is deposited in an intermediate buffer B1.

- This buffer needed to enable the execution unit while fetch unit fetching the next instruction.

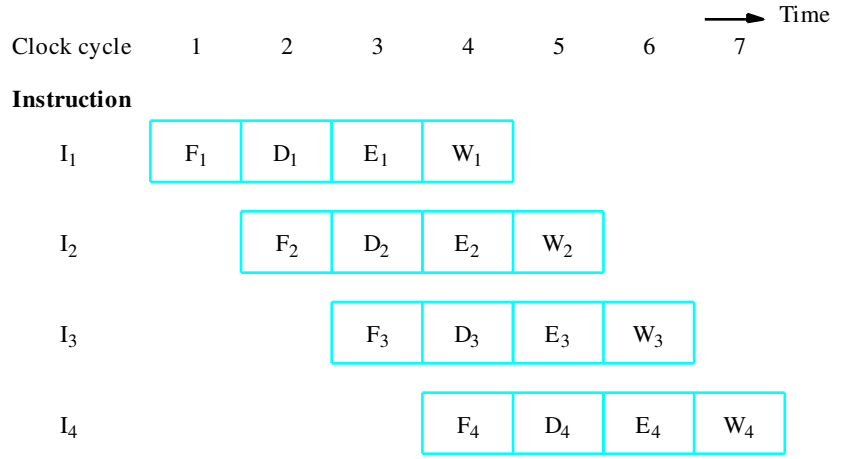
8.1(c)



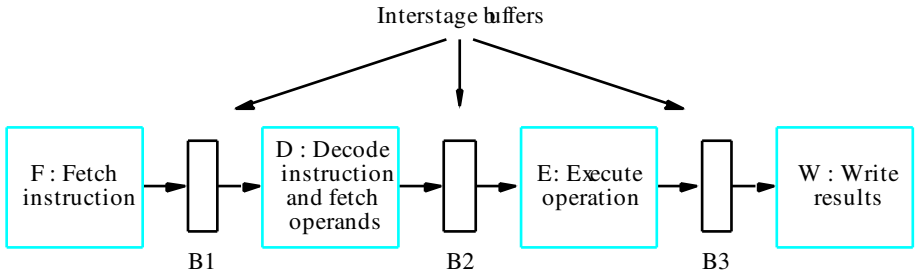
- The computer is controlled by a clock.
- Any instruction fetch and execute steps completed in one clock cycle.

Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write



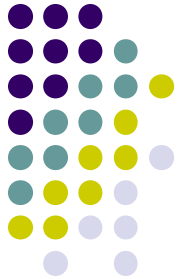
(a) Instruction execution divided into four steps



(b) Hardware organization

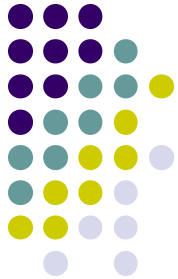
Textbook page: 457

Figure 8.2. A 4-stage pipeline.



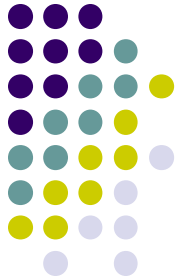
- Fetch(F)- read the instruction from the memory
- Decode(D)- Decode the instruction and fetch the source operand
- Execute(E)- perform the operation specified by the instruction
- Write(W)- store the result in the destination location

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 very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.[ten times greater than the time needed to perform pipeline stage]
- 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.
- Floating point may involve many clock cycle

Pipeline Performance

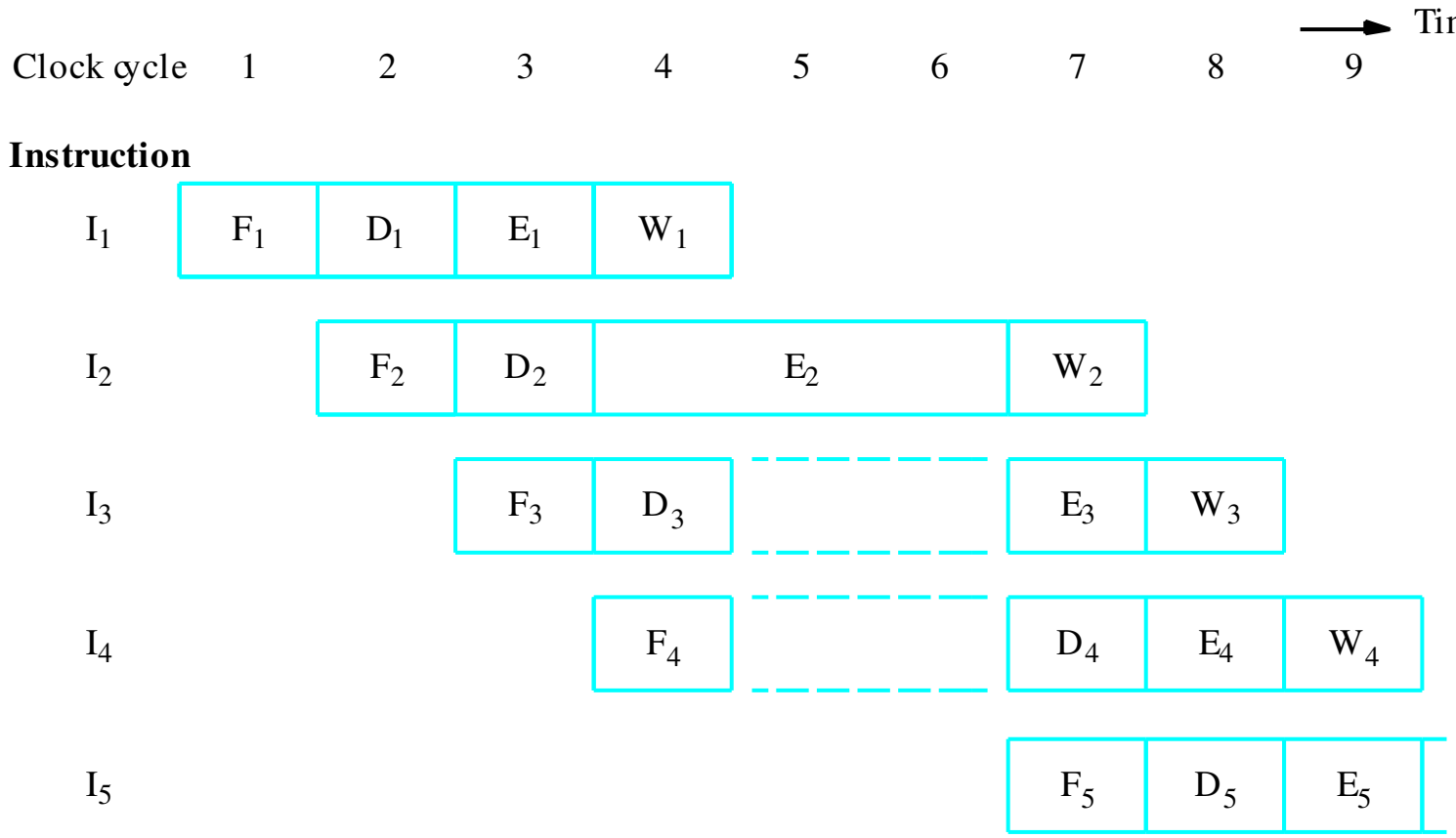
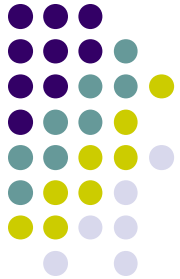


Figure 8.3. Effect of an execution operation taking more than one clock cycle.

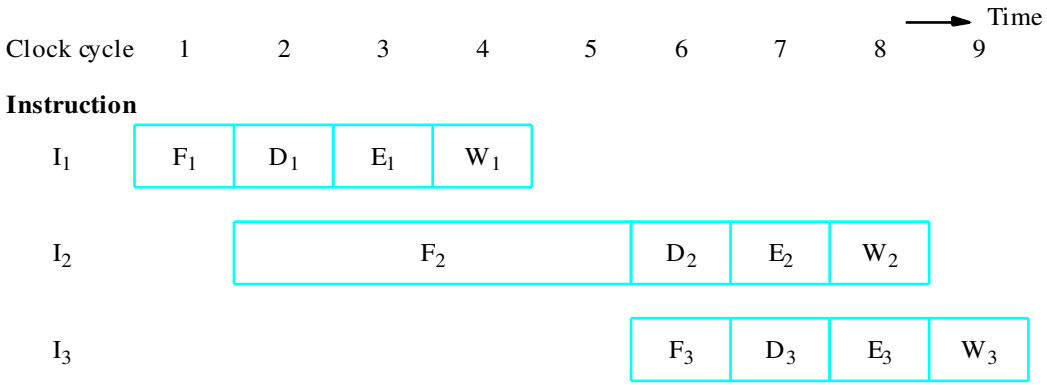
Pipeline Performance



- The previous pipeline 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.[cache miss]
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

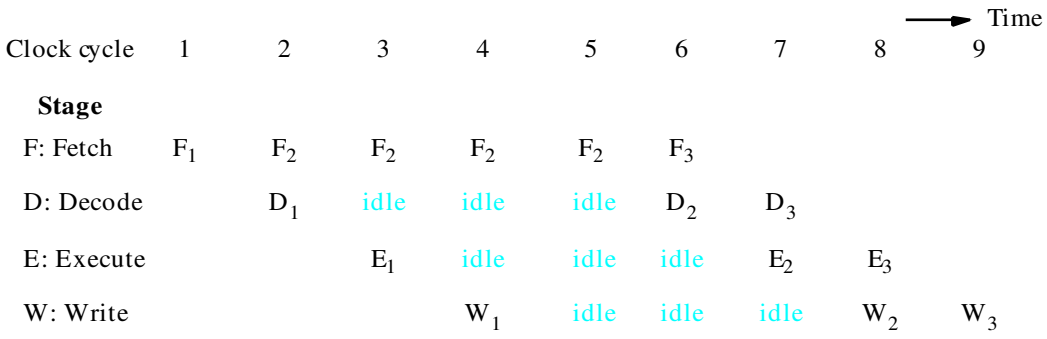
Pipeline Performance

Instruction hazard(Cache miss)



(a) Instruction execution steps in successive clock cycles

Decode unit is idle in cycles 3 through 5, Execute unit idle in cycle 4 through 6 and write unit is idle in cycle 5 through 7 such idle period is called stalls.



(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

The memory address, X+(R1) is computed in step E2in cycle4, then memory access takes place in cycle5. the operand read from memory is written into register R2 in cycle 6[Execution takes 2 cycles] it stalls pipeline to stall for one cycle. Bcox both instruction I2 and I3 require access of register file in cycle 6.

Structural hazard

Load X(R1), R2

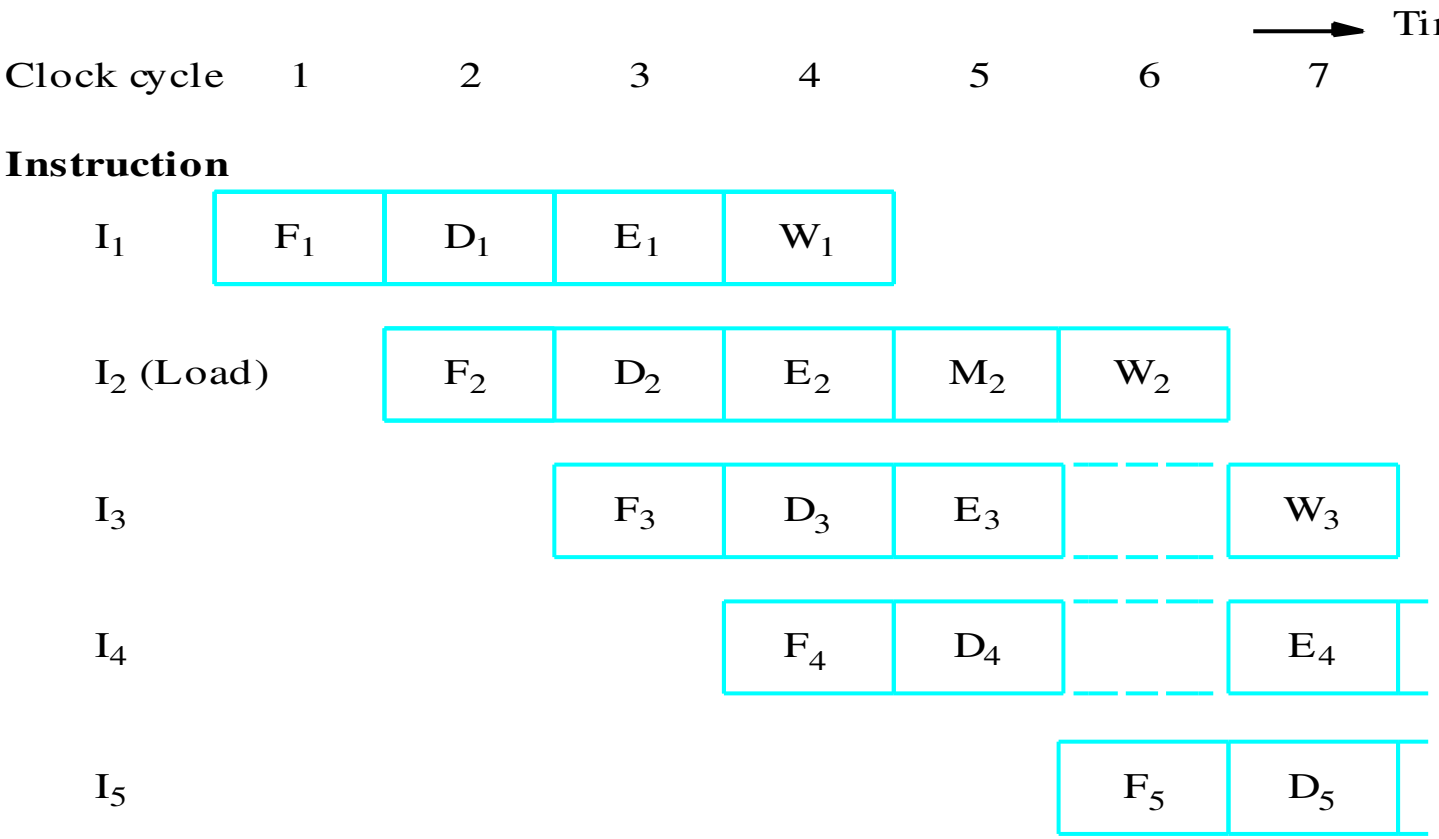
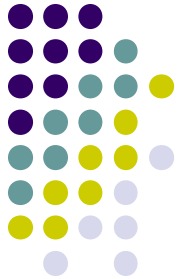


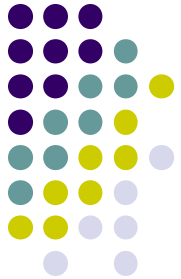
Figure 8.5. Effect of a Load instruction on pipeline timing.

Pipeline Performance



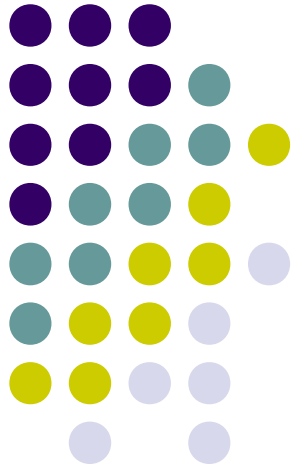
- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards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

Quiz

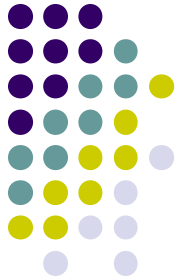


- Four instructions, the I2 takes two clock cycles for execution. Pls draw the figure for 4-stage pipeline, and figure out the total cycles needed for the four instructions to complete.

Data Hazards



Data Hazards



- We 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

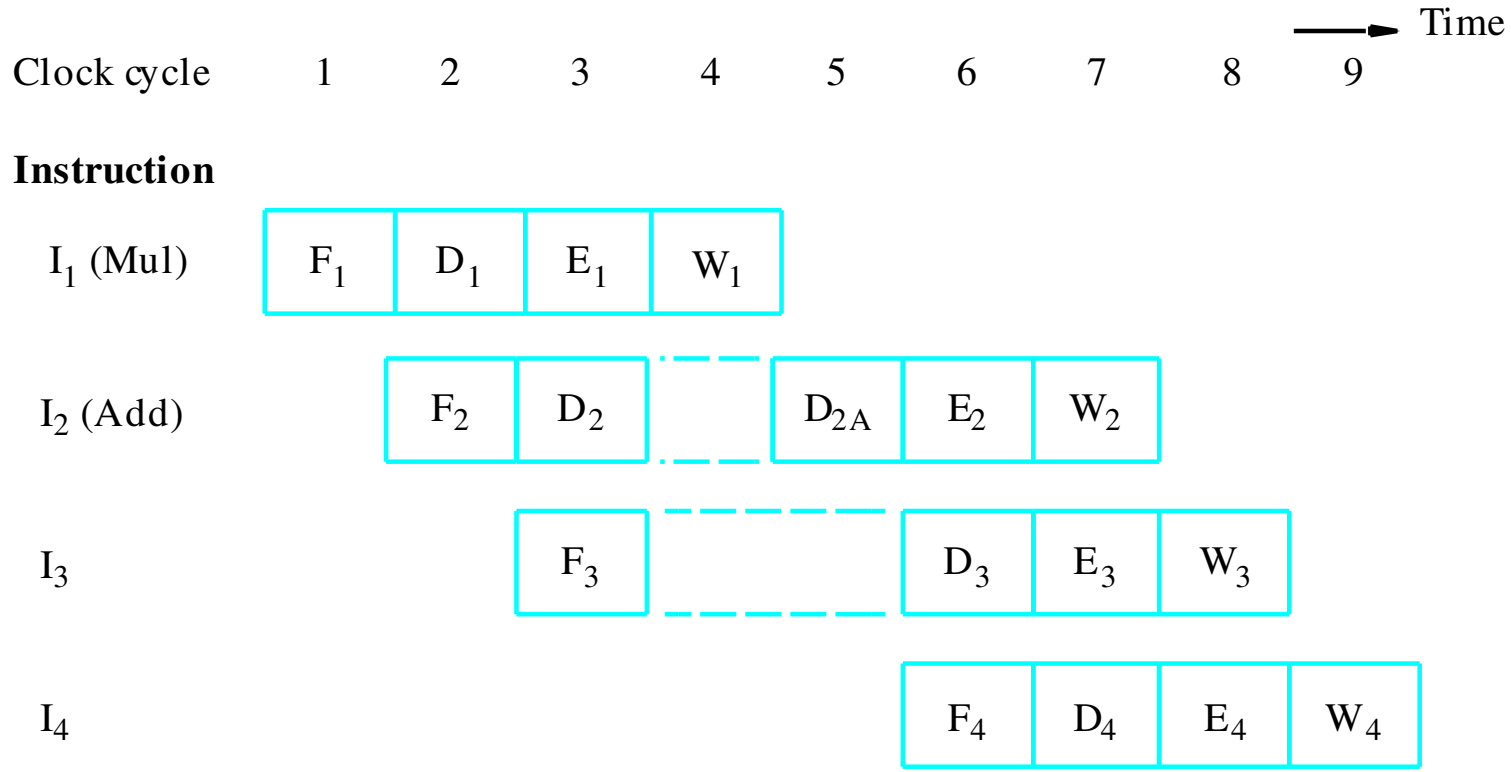
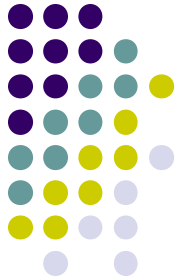
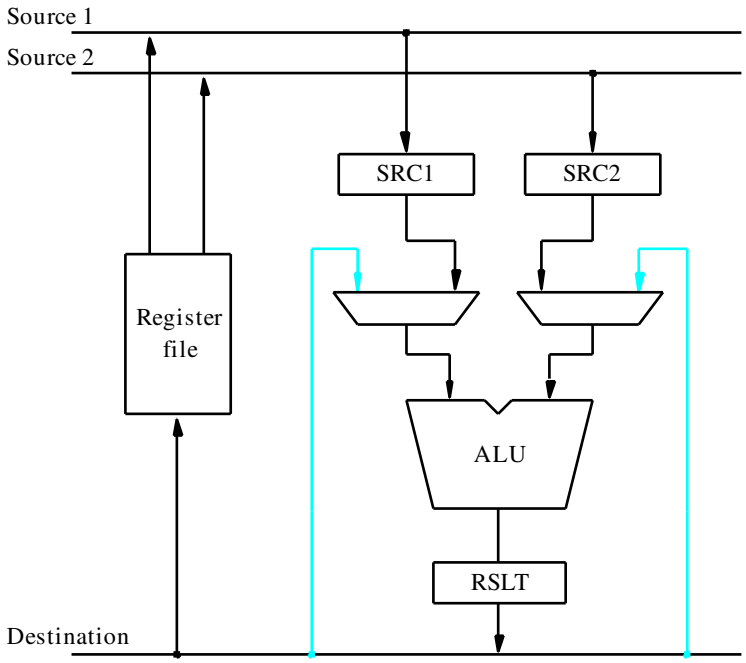


Figure 8.6. Pipeline stalled by data dependency between D_2 and W_1.

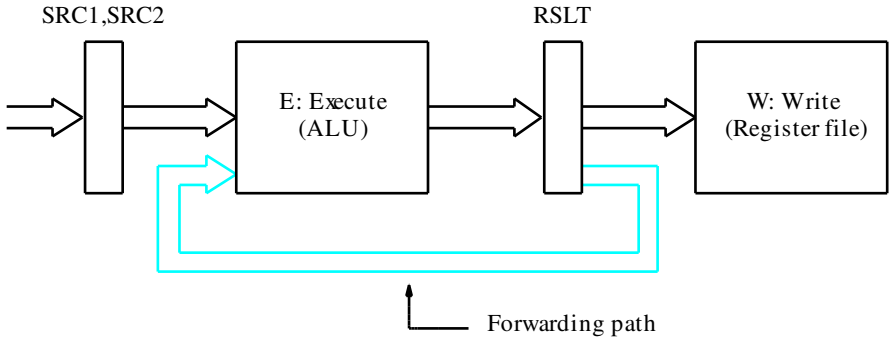
Operand Forwarding



- 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.



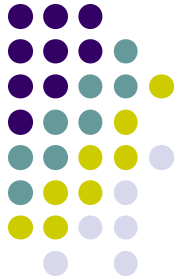
(a) Datapath



(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

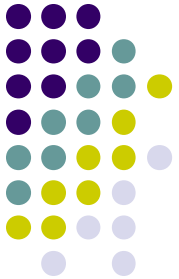


- Let the compiler detect and handle the hazard:

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

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

Side Effects



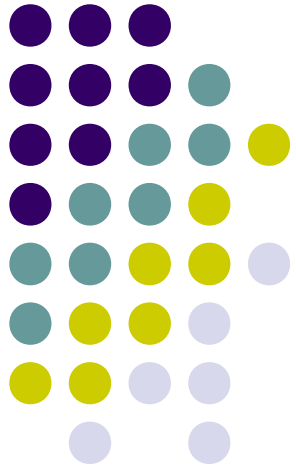
- The previous example is explicit and easily detected.
- Sometimes an instruction changes the contents of a register other than the one named as the destination.
- When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. (Example?)
- Example: conditional code flags:

  Add R1, R3

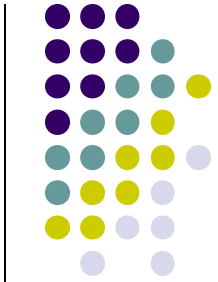
  AddWithCarry R2, R4

- 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

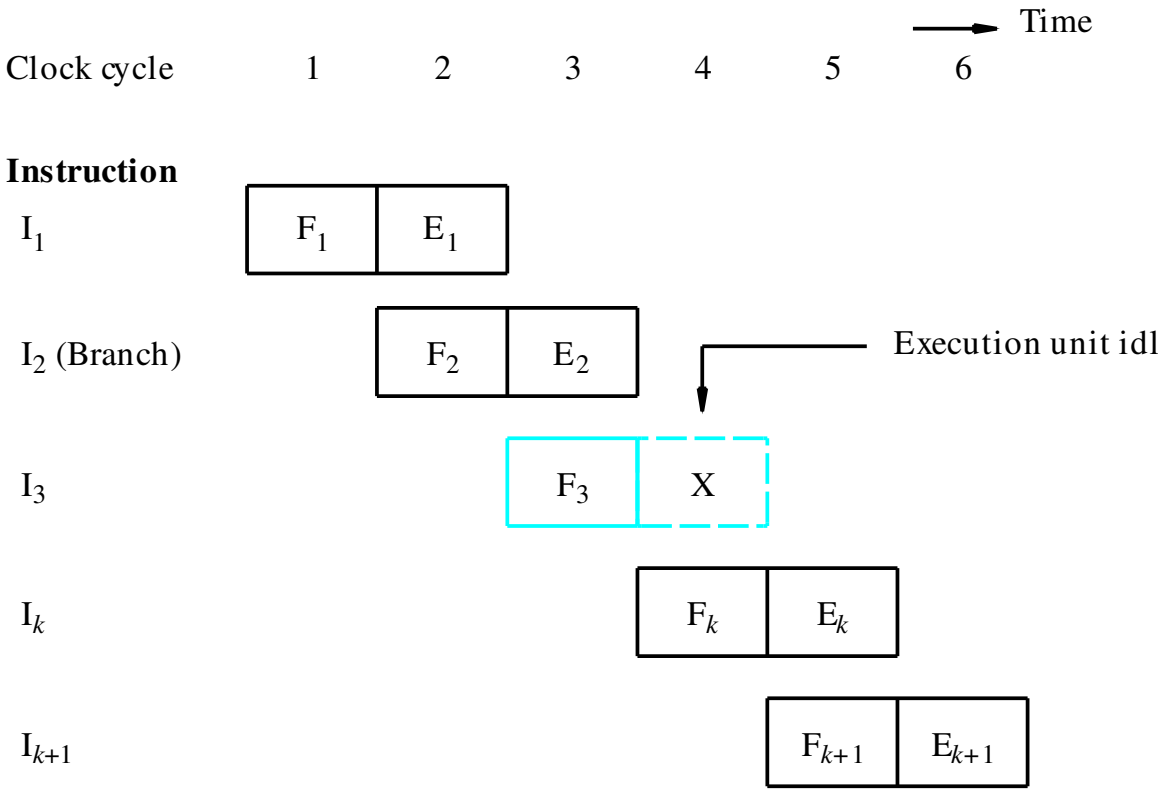
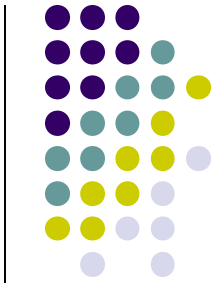


Figure 8.8. An idle cycle caused by a branch instruction.

Unconditional Branches

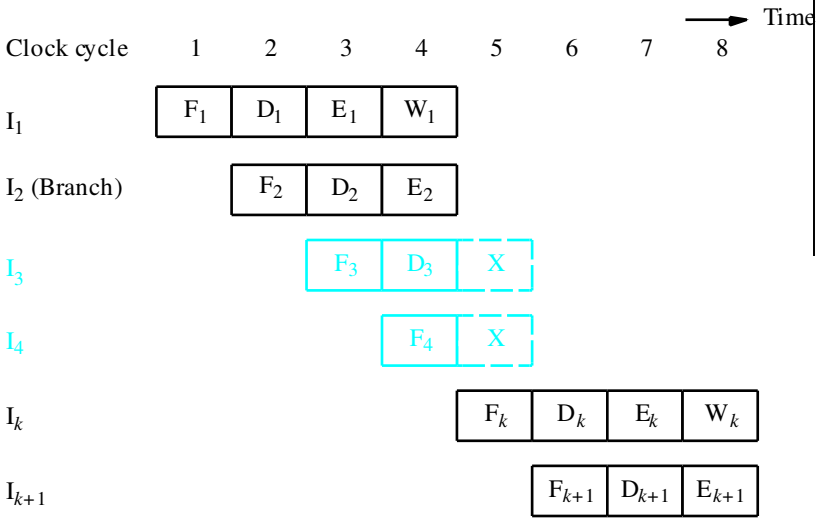


- The time lost as a result of a branch instruction is referred to as the branch penalty.
- The previous example instruction I3 is wrongly fetched and branch target address k will discard the i3.
- Reducing the branch penalty requires the branch address to be computed earlier in the pipeline.
- Typically the Fetch unit has dedicated h/w which will identify the branch target address as quick as possible after an instruction is fetched.

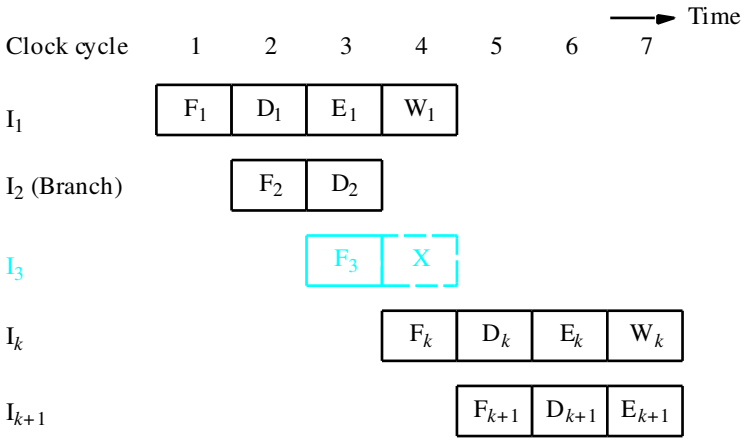
Branch Timing

- Branch penalty

- Reducing the penalty



(a) Branch address computed in Execute stage



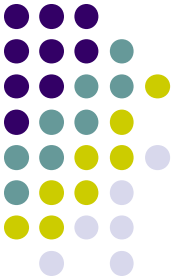
(b) Branch address computed in Decode stage

Figure 8.9. Branch timing.

Instruction Queue and Prefetching

- Either cache (or) branch instruction stalls the pipeline.
- Many processor employs dedicated fetch unit which will fetch the instruction and put them into a queue.
- It can store several instruction at a time.
- A separate unit called dispatch unit, takes instructions from the front of the queue and send them to the execution unit.

Instruction Queue and Prefetching



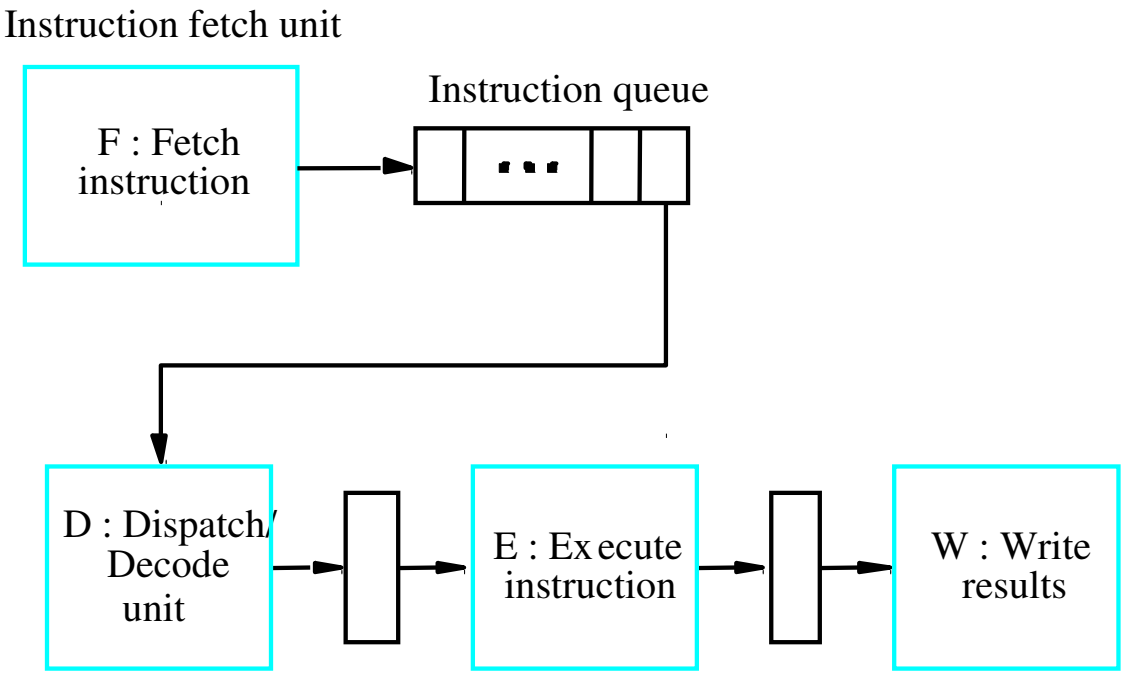
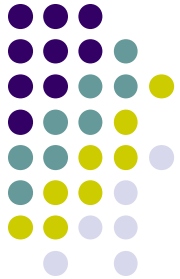


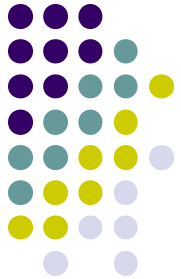
Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2*b*.

Conditional Braches



- A conditional branch instruction 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. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken.
- The objective is to place useful instructions in these slots.
- The 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

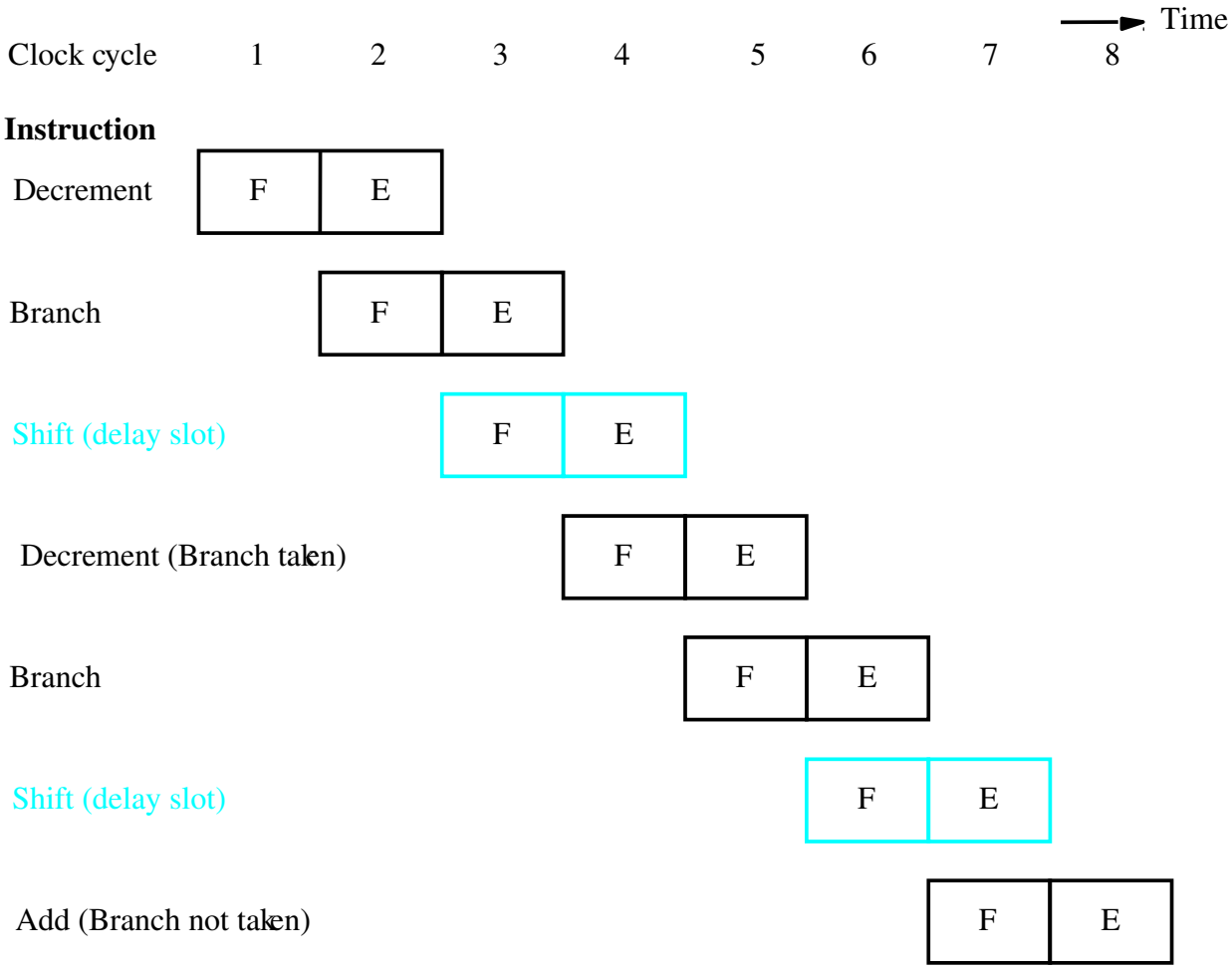
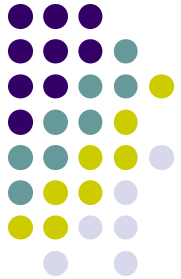


Figure 8.13. Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12.

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

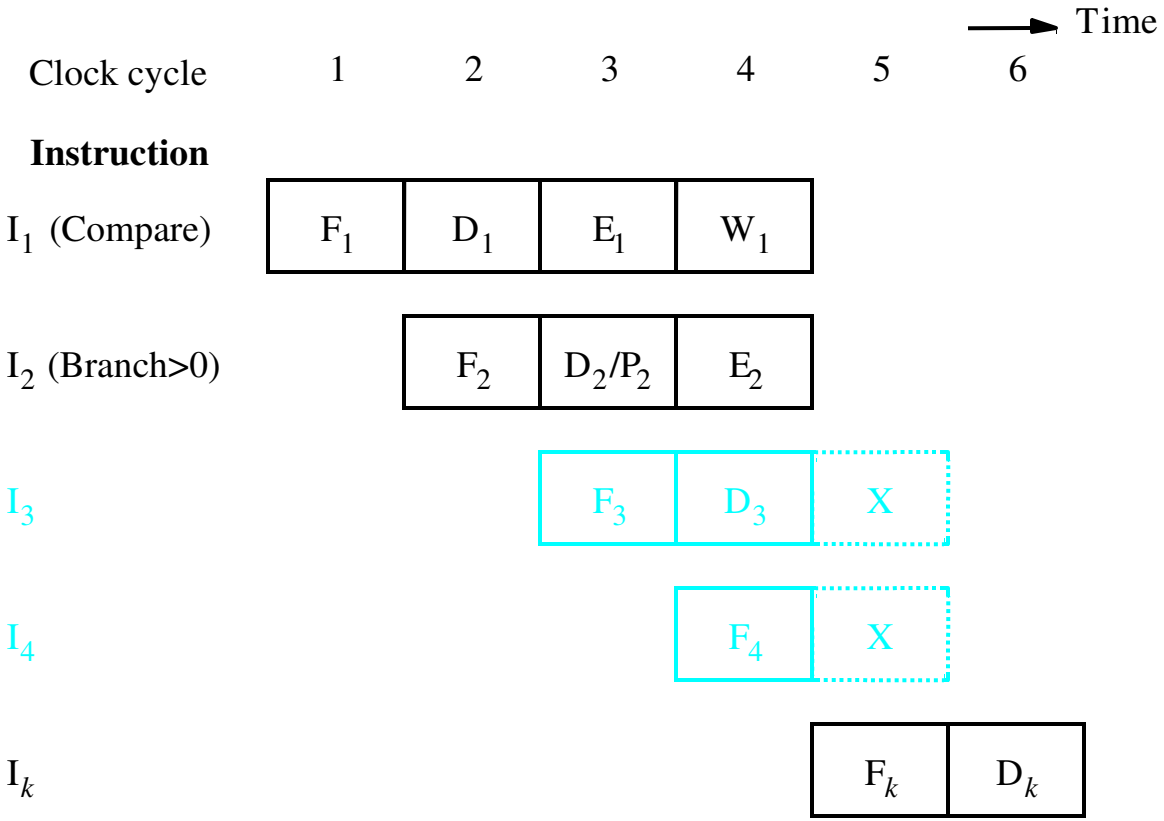
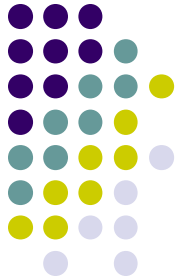


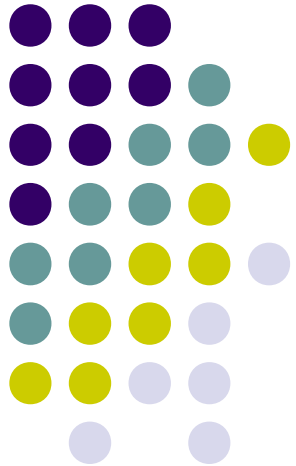
Figure 8.14. Timing when a branch decision has been incorrectly predicted as not taken.

Branch Prediction

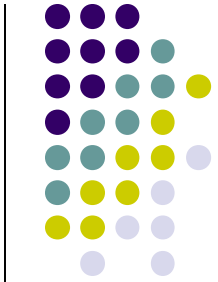


- Better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.
- Use hardware to observe whether the target address is lower or higher than that of the branch instruction.
- Let compiler include a branch prediction bit.
- So far the branch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

Influence on Instruction Sets

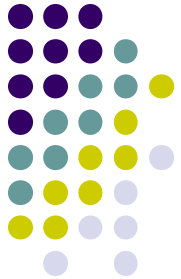


Overview



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

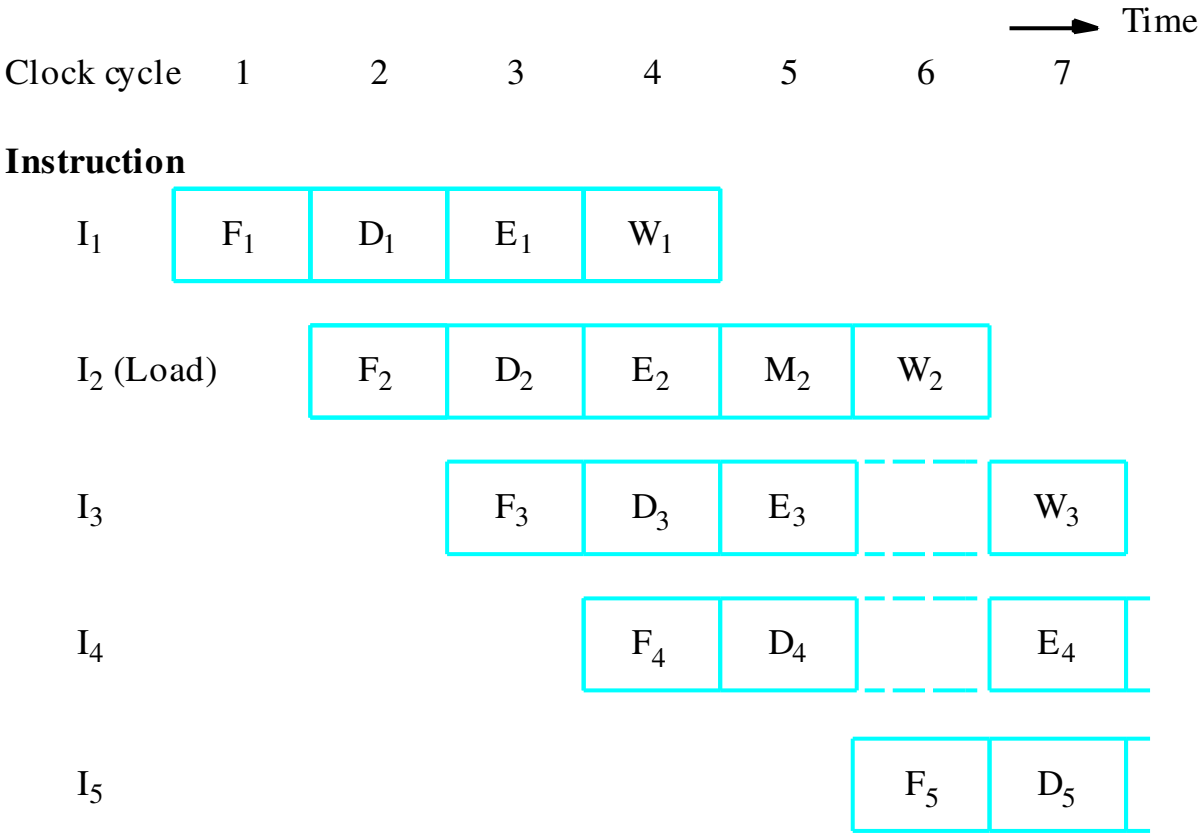
Addressing Modes



- Addressing modes include simple ones and complex ones.
- In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline:
  - Side effects
  - The 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

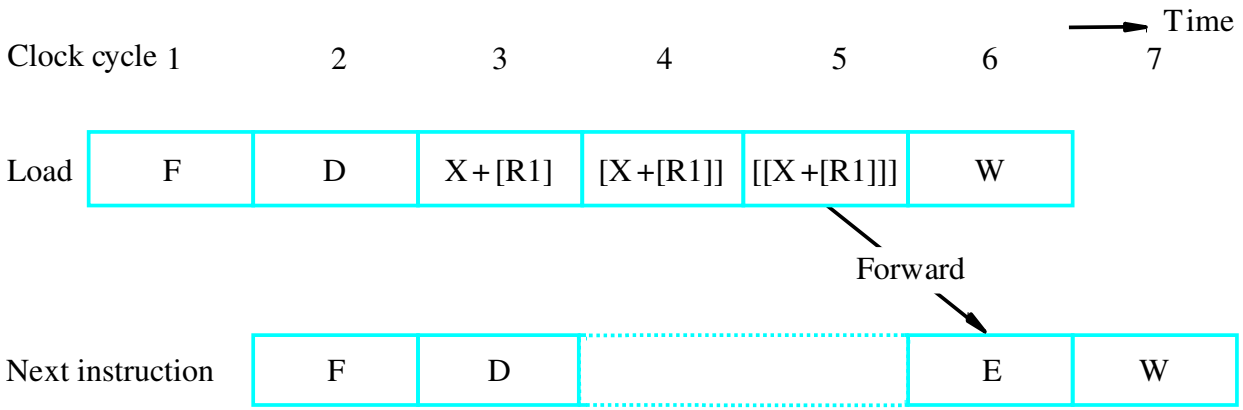


Load (R1), R2

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

Complex Addressing Mode

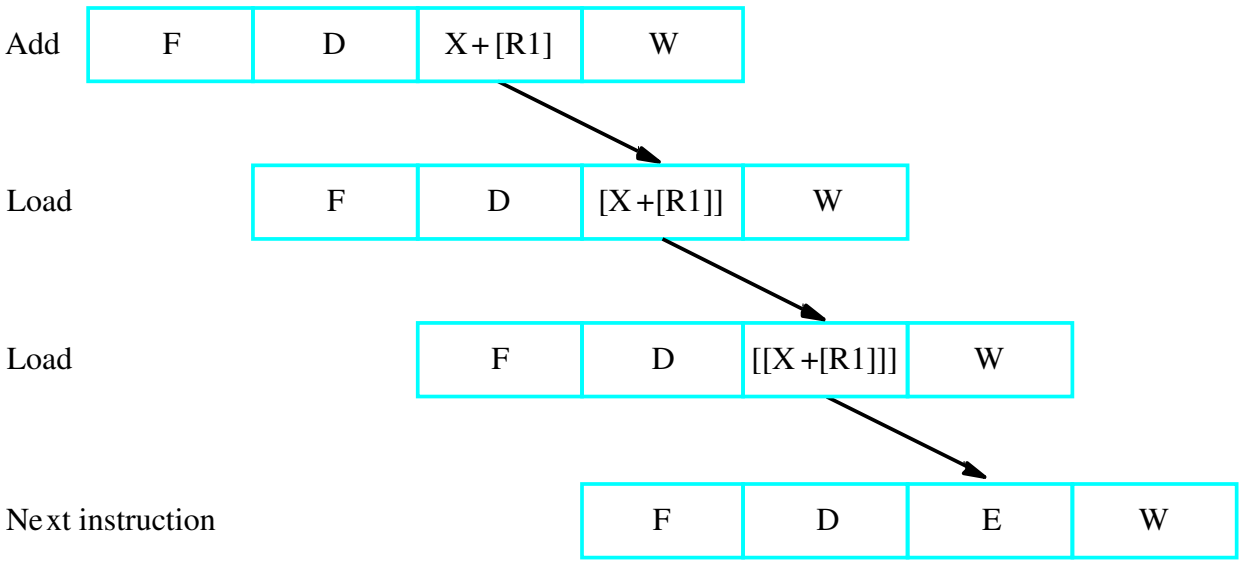
Load (X(R1)), R2



(a) Complex addressing mode

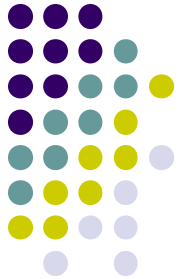
Simple Addressing Mode

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



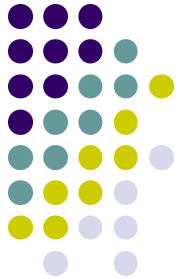
(b) Simple addressing mode

Addressing Modes



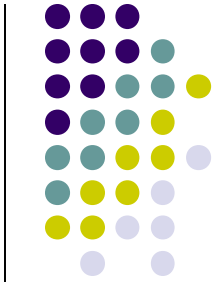
- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- Advantage: reducing the number of instructions / program space
- Disadvantage: cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- Conclusion: complex addressing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

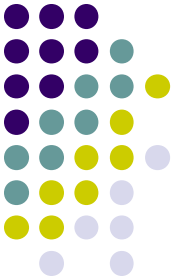
(a) A program fragment

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

(b) Instructions reordered

Figure 8.17. Instruction reordering.

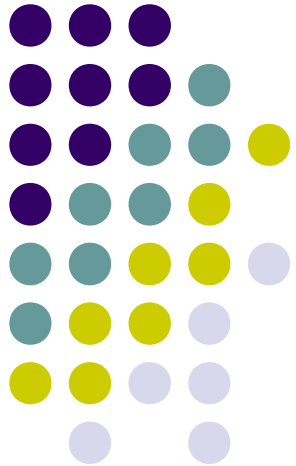
Conditional Codes



- Two conclusion:

- To provide flexibility in reordering instructions, the condition-code flags should be affected by as few instruction as possible.
- The 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

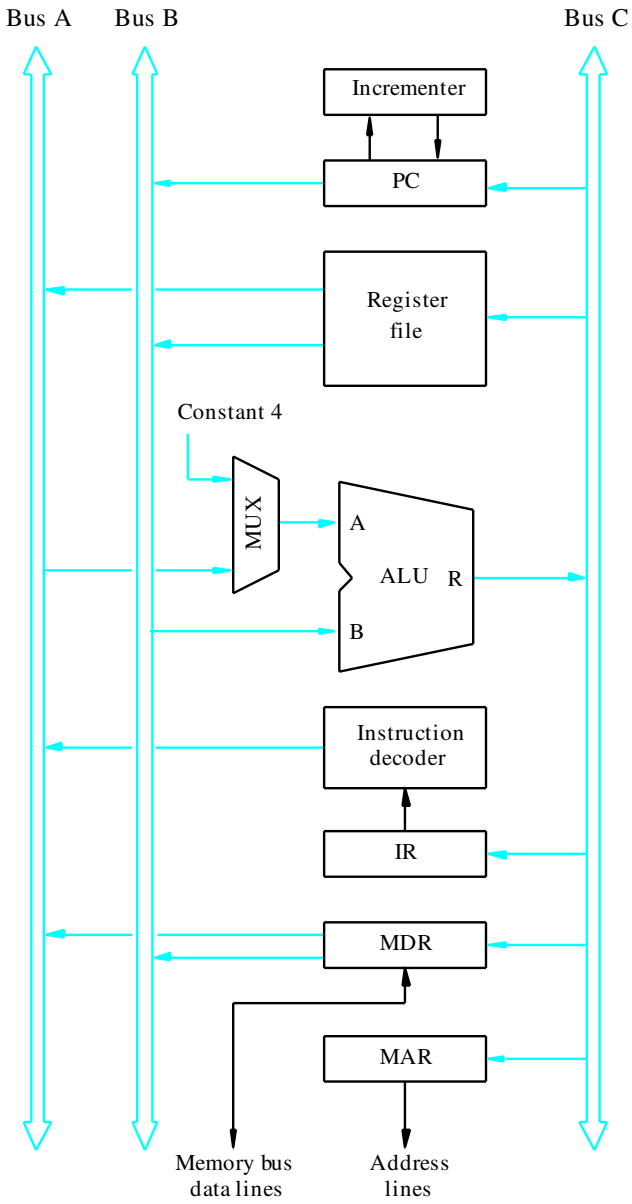
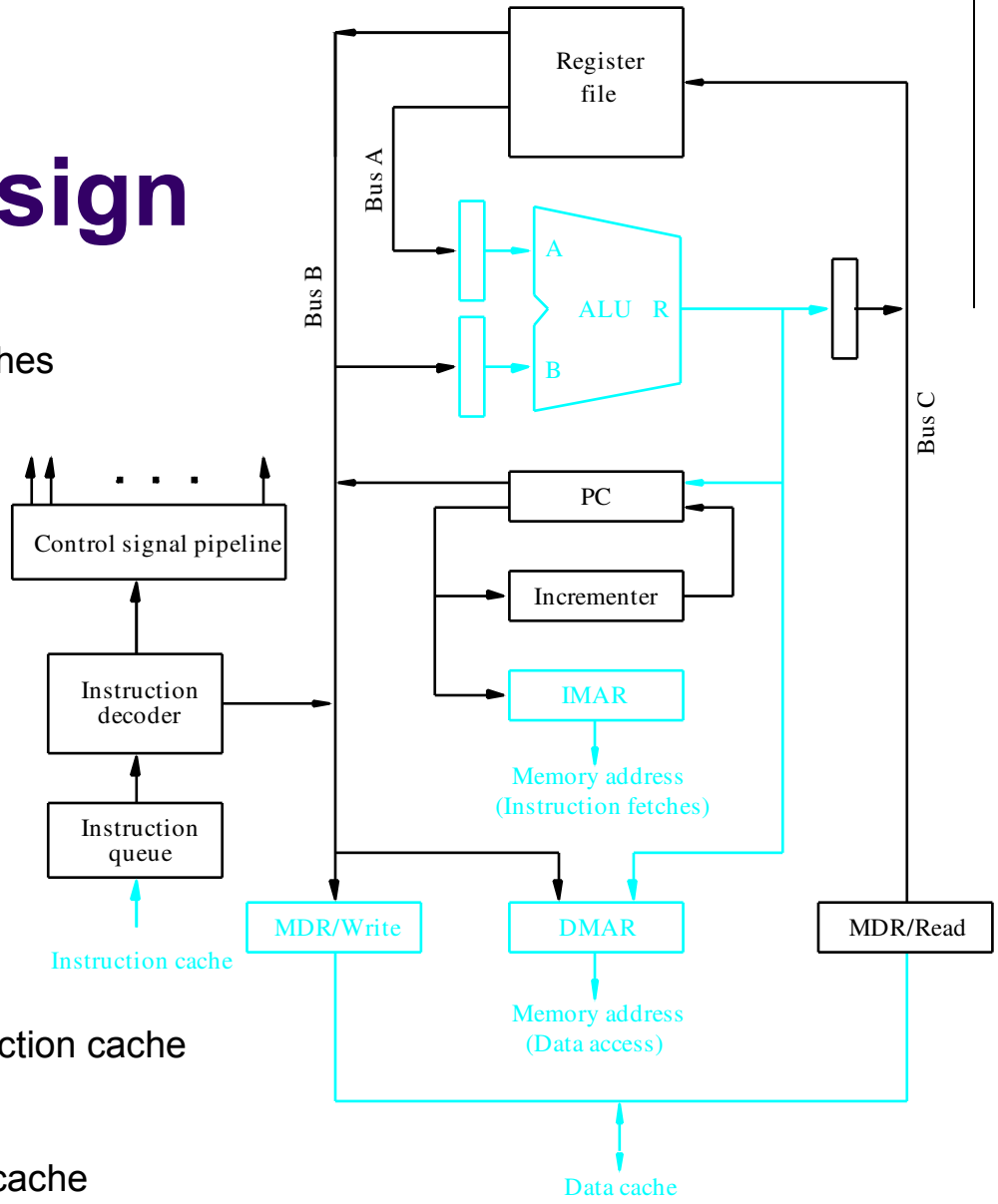


Figure 7.8. Three-b us organization of the datapath.

Pipelined Design

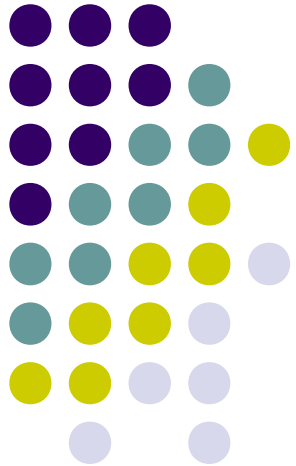
- Separate instruction and data caches
- PC is connected to IMAR
- DMAR
- Separate MDR
- Buffers for ALU
- Instruction queue
- Instruction decoder output

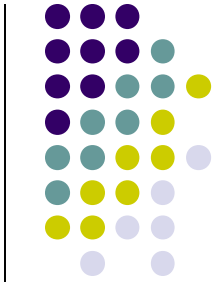


- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two regs
- Writing into one register in the reg file
- Performing an ALU operation

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

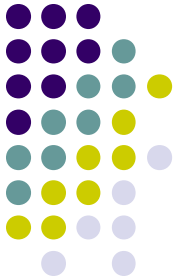
Superscalar Operation



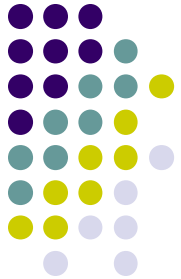


- Pipeline architecture can executes several instruction concurrently.
- Many instructions are present in the pipeline at the same time,but they are in different stages of their execution.
- While instruction being fetched at the same time another instruction being decoded stage (or) execution.
- One instruction completes execution in each clock cycle.

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 to keep the instruction queue to be filled.



- The superscalar operation have multiple execution units.

Superscalar

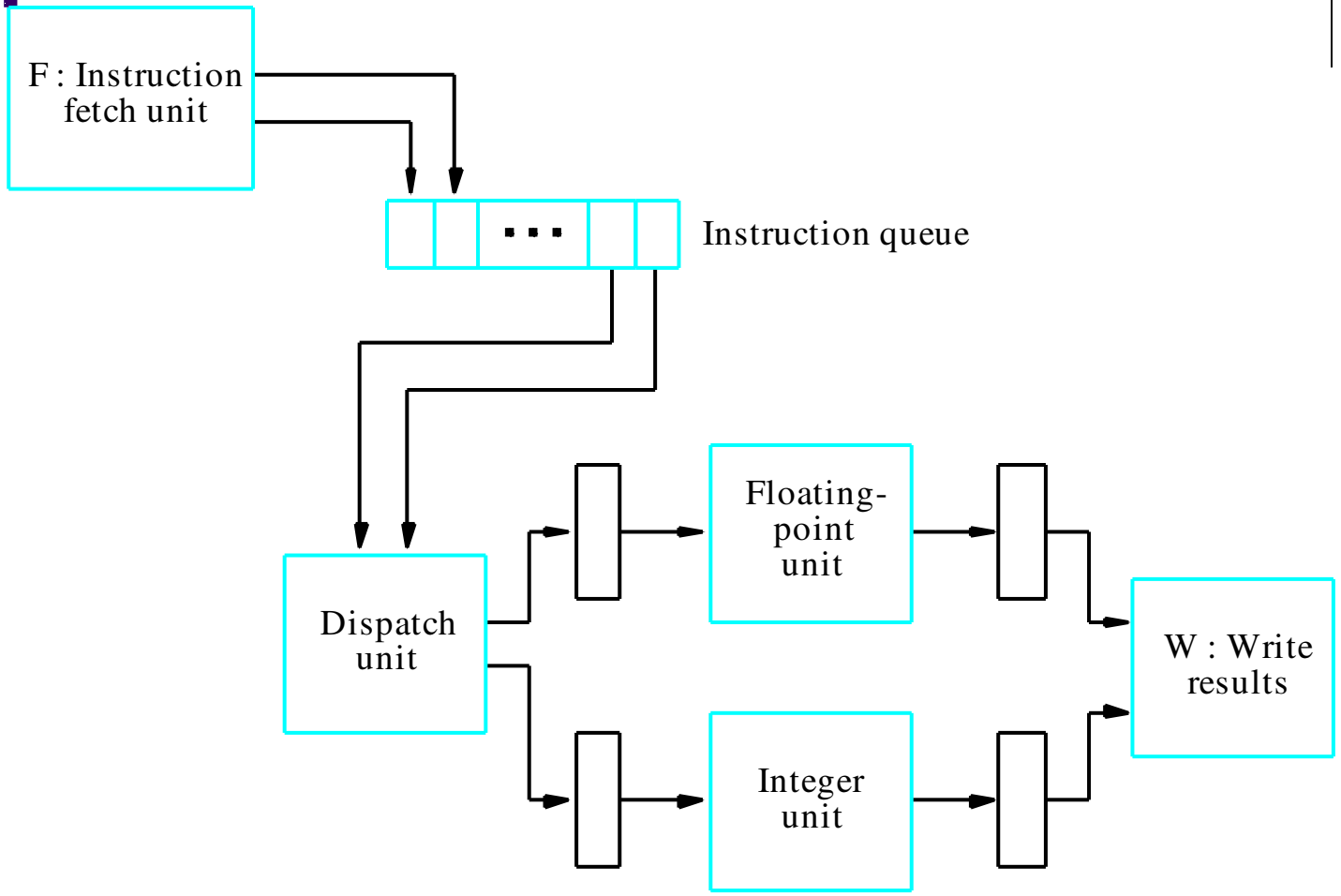


Figure 8.19. A processor with two execution units.

Architecture

- The above fig. shows the superscalar processor with two execution unit.
- The Fetch unit capable of reading two instruction at a time and store it in the queue.
- The dispatch unit decodes upto two instruction from the front of queue(one is integer and another one is floating point) dispatched in the same clock cycle.
- Processor's program control unit – capable of fetching and decoding several instruction concurrently. It can issue multiple instructions simultaneously.

Timing

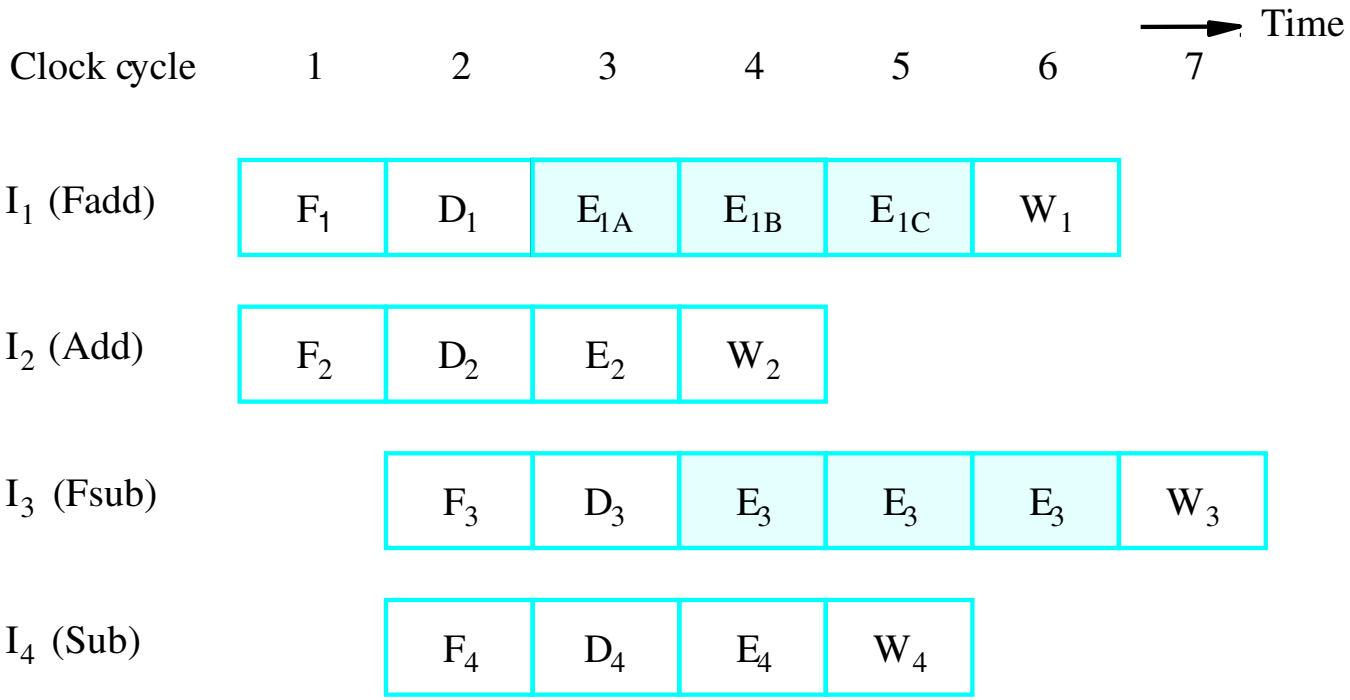


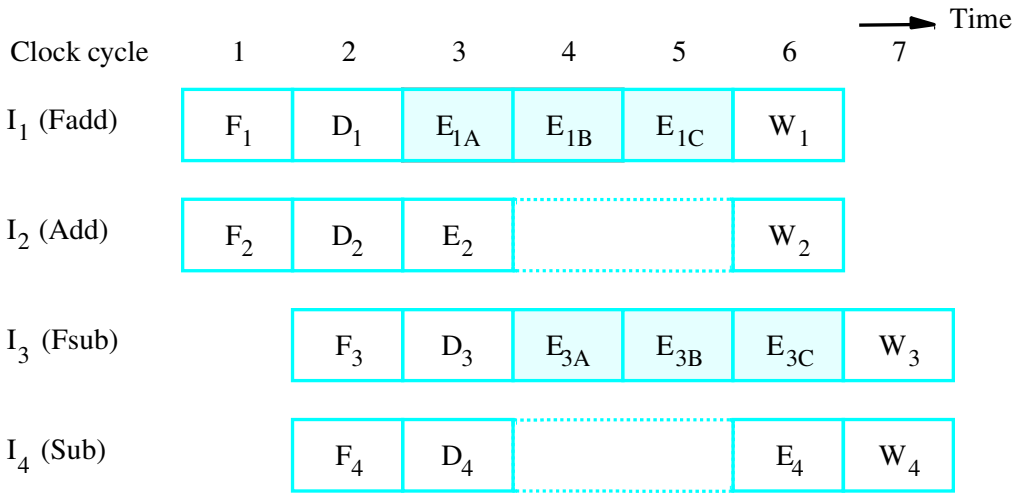
Figure 8.20. An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

Assume Floating point takes 3 clock cycles

5. Assume floating point unit as a 3 stage pipeline. So it can accept a new instruction for execution in each clock cycle. During cycle 4 the execution of I1 in progress but it will enter the different stages inside the execution pipleline, so this unit accept for I3 for execution.
6. The integer unit can accept new instruction for execution because I2 has entered into the write stage.

Out-of-Order Execution

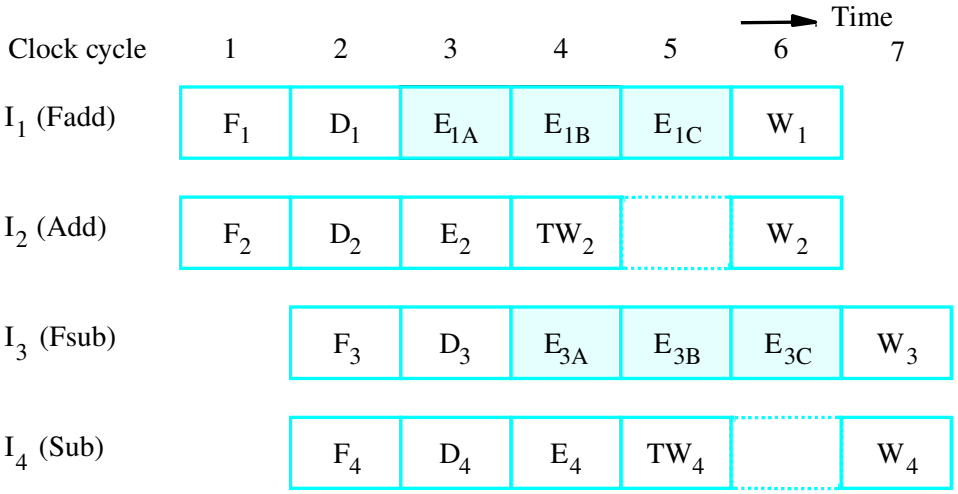
- Hazards
- Exceptions
- Imprecise exceptions
- Precise exceptions



(a) Delayed write

Execution Completion

- It is desirable to used out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible.
- 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

- If I_2 depends on the result of I_1, the execution of I_2 will be delayed. These dependencies are handled correctly,as long as the execution of the instruction will not be delayed.
- one more reason which delays the execution is,
  - Exception

  Two causes:
  - Bus error
  - Illegal operation(divide by zero)

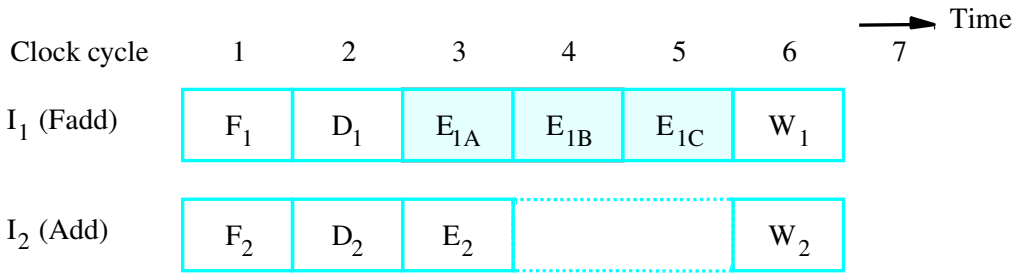
Two types of Exception

* Imprecise * Precise

- Imprecise- the result of I2 written into RF during cycle 4. suppose I1 causes an exception, the processor allows to complete execution of succeeding instruction(I2) is said to have imprecise exception.
- Precise-in imprecise exception, consistent state is not guaranteed, when an exception occurs.(The processor will not allow to write the succeeding instruction)
- In consistent state, the result of the instruction must be written into the program order(ie. The I2 must be allow to written till cycle 6.

- The integer execution unit has to retain the result until cycle 6 and it cannot accept instruction I4 until cycle 6.

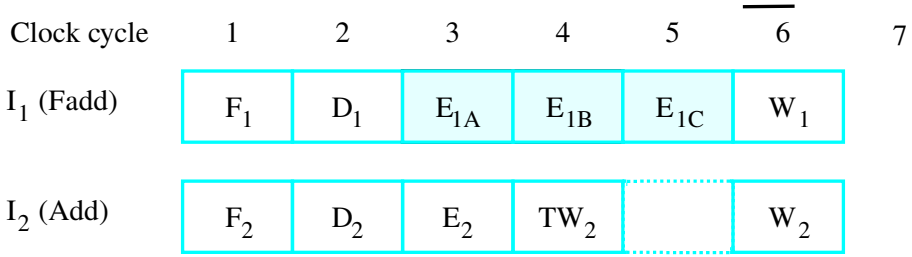
- Thus, in this method the output is partially executed (or) discarded.



- If an external interrupt is received, the dispatch unit stops reading new instructions from instruction queue, and the instruction remaining in the queue are discarded.

Execution Completion

- In precise exception, the results are temporarily stored into the temp register and later they are transferred to the permanent registers in correct program order.
- Thus, two write operations TW and W are carried out.

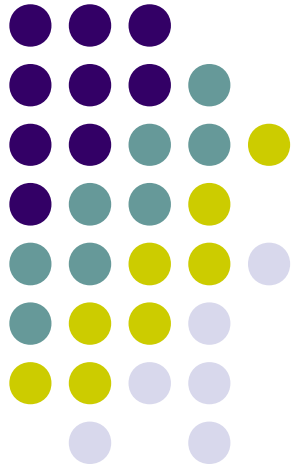


- The step W are called commitment step(temp reg to perm register)
- TW- write into a temp registers
- W- Transfer the contents back to the permanent reg

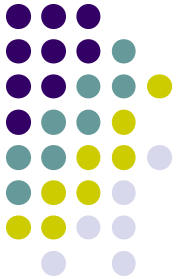
Register renaming

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

Performance Considerations



Overview



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

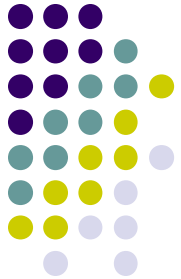
$$T = \frac{N \times S}{R}$$

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.

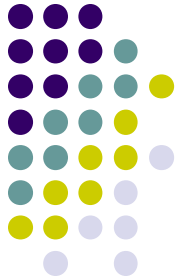
$$P_s = \frac{R}{S}$$

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, …)