Computer Organization & Architecture

Chapter 6 – Pipeline

Zhang Yang 张杨

cszyang@scut.edu.cn

Autumn 2025

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Making the Execution of Programs Faster

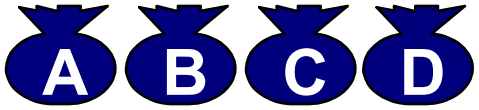
- Two Ways
  - Use faster circuit technology to build the processor and the main memory.
  - Arrange the hardware so that more than one operation can be performed at the same time.
  - In the latter way, the number of operations performed per second is increased even though the elapsed time needed to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

What is Pipelining?

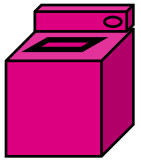
- What is Pipelining?
  - Pipelining is a key implementation technique used to build fast processors. It allows the execution of multiple instructions to overlap in time.
- Key Idea
  - Overlap execution of multiple instructions
- Essence
  - Start executing one instruction before completing the previous one.

Laundry Example (1)

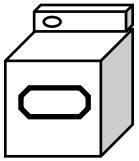
- Ann, Brian, Cathy, David each have one load of clothes to wash, dry, and fold



- Washer takes 30 minutes



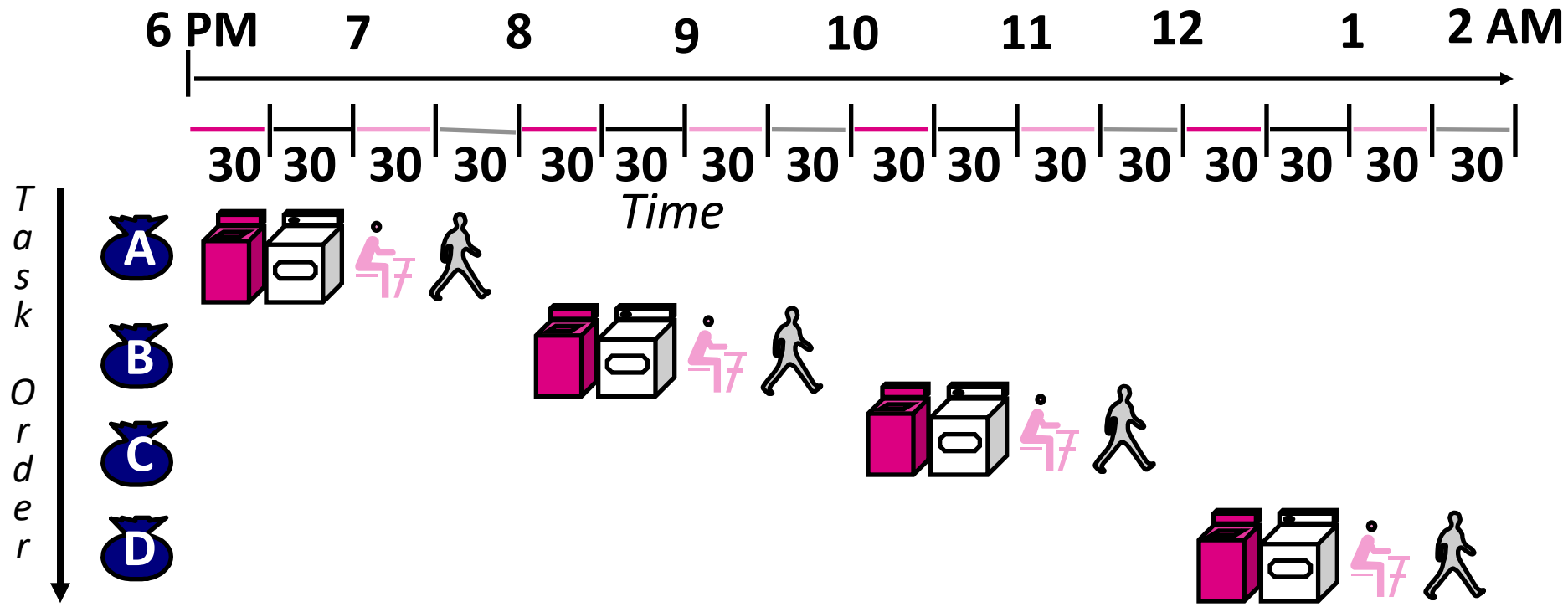
- Dryer takes 30 minutes



- “Folder” takes 30 minutes

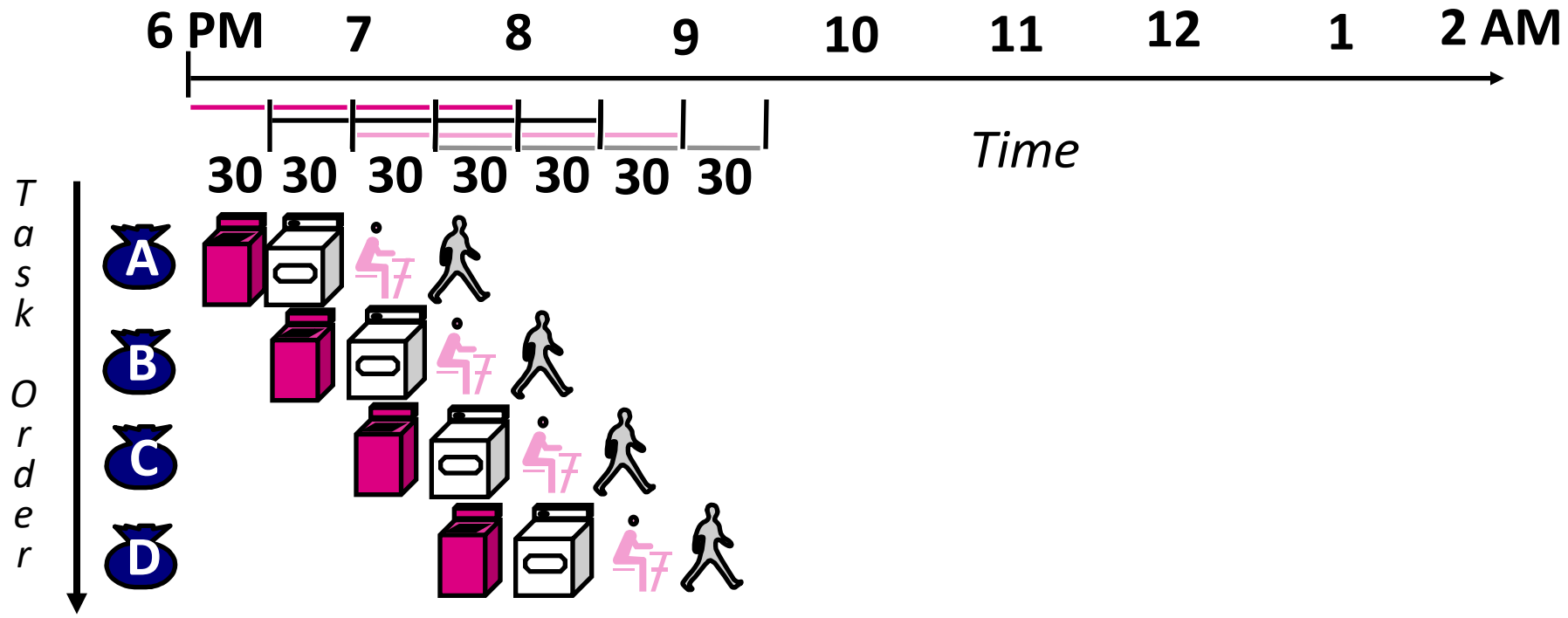
- “Stasher” takes 30 minutes to put clothes into drawers

Laundry Example (2)



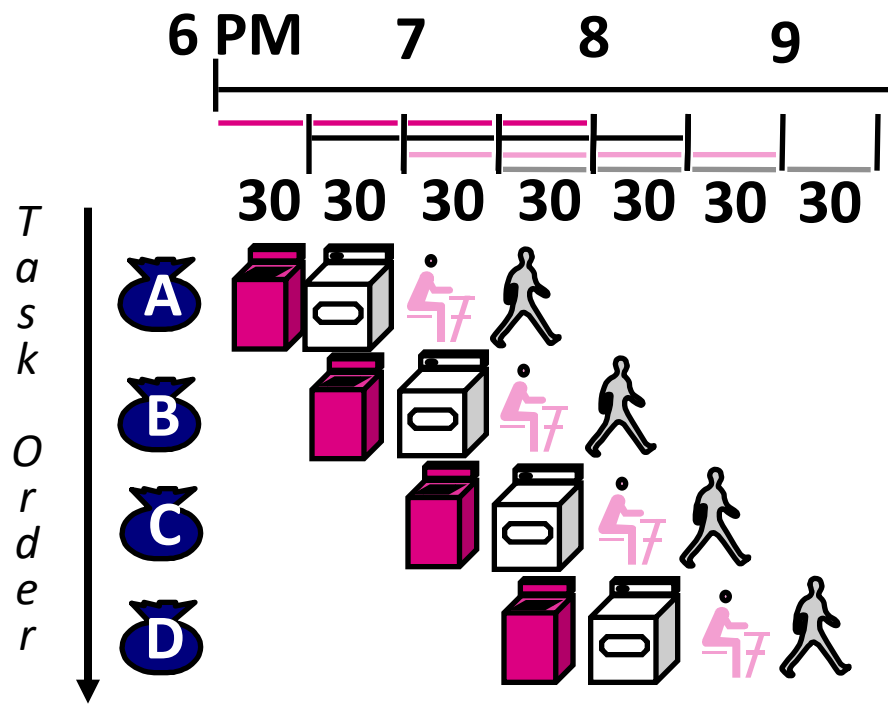
- If we do laundry sequentially
  - Time Required: 8 hours for 4 loads

Laundry Example (3)



- To Pipeline, We Overlap Tasks
  - Time Required: 3.5 Hours for 4 Loads

Laundry Example (4)

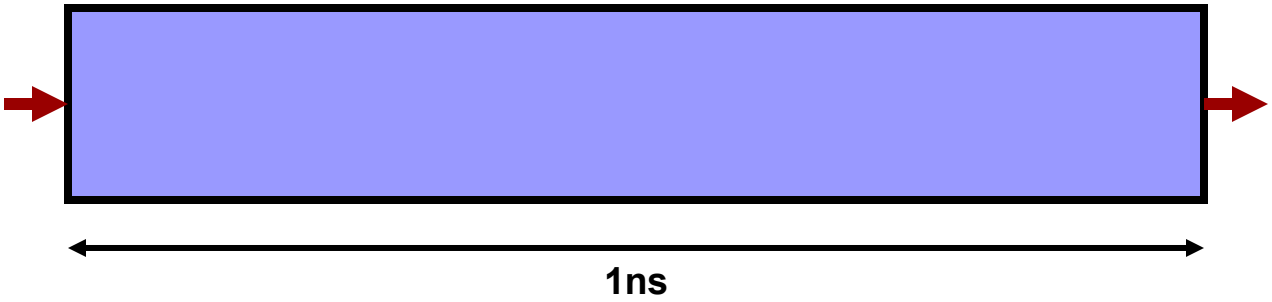


- **Pipelining doesn't help** latency **of single task, it helps** throughput **of entire workload**
- **Pipeline rate limited by** slowest **pipeline stage**
- Multiple **tasks operating simultaneously**
- **Potential speedup =** Number pipe stages
- **Unbalanced lengths of pipe stages reduces speedup**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

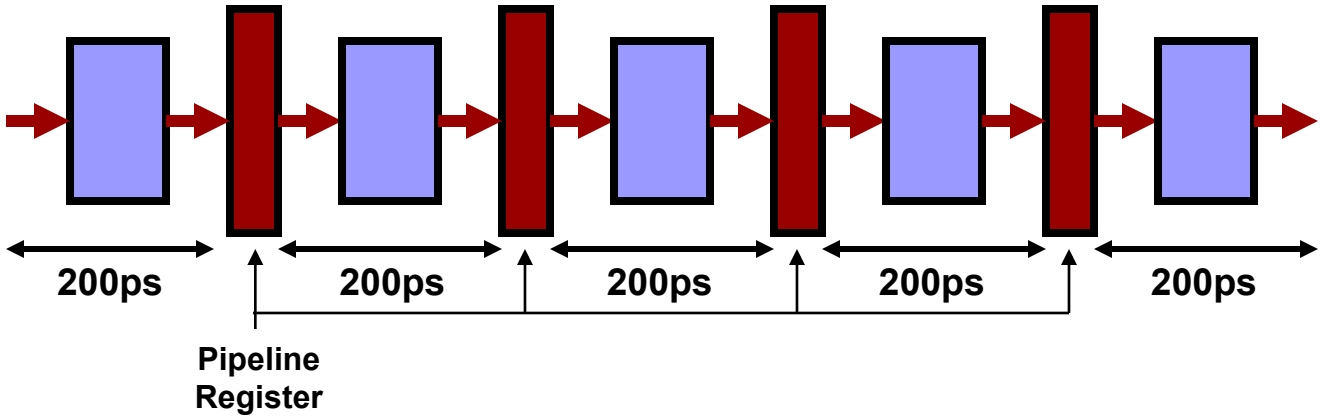
Pipelining a Digital System (1)

- Key idea: break big computation up into pieces

1 nanosecond = 10^-9 second
1 picosecond = 10^-12 second

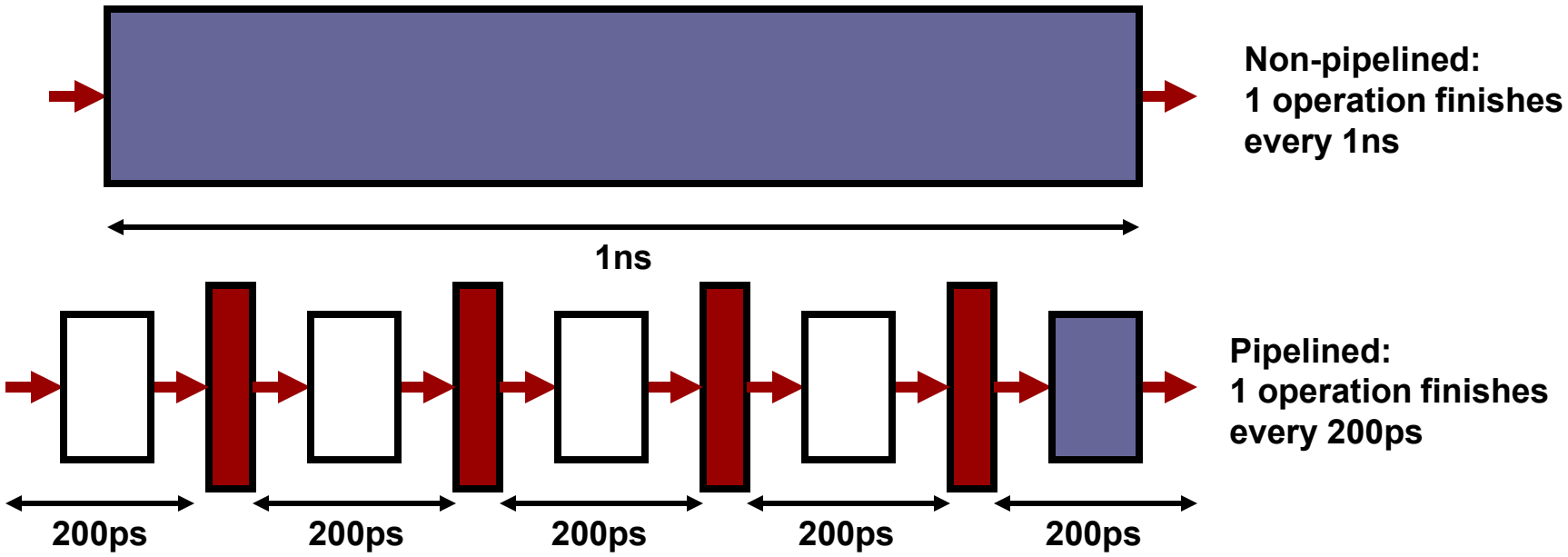


- Separate each piece with a pipeline register



Pipelining a Digital System (2)

- Why do this? Because it's faster for repeated computations



Pipelining a Digital System (3)

- Comments about pipelining
  - Pipelining increases throughput, but not latency
    - Answer available every 200ps, BUT
    - A single computation still takes 1ns
  - Limitations
    - Computations must be divisible into stage size
    - Pipeline registers add overhead

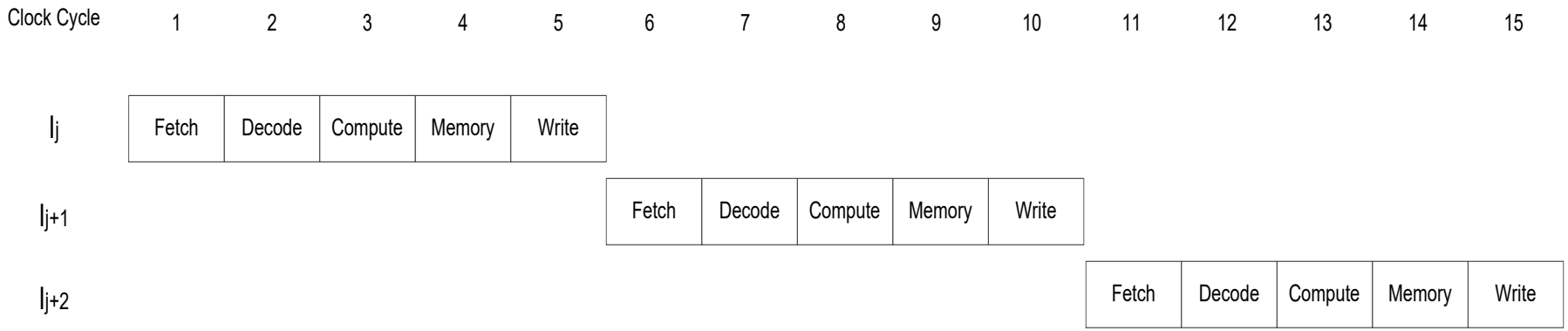
Pipelining a Processor (1)

- Recall the 5 steps in instruction execution: Figure 5.7

1. Instruction Fetch
2. Instruction Decode and Register Read
3. Execution operation or calculate address
4. Memory access
5. Write result into register

Pipelining a Processor (2)

- Unpipelined Execution



Pipelining a Processor (3)

- Pipelined Execution-The Ideal Case
  - Each instruction takes 1 clock cycle for each stage
  - The processor can accept 1 new instruction per clock
  - Instructions are processed in stages as they pass down
  - Multiple instructions in some phase of execution concurrently

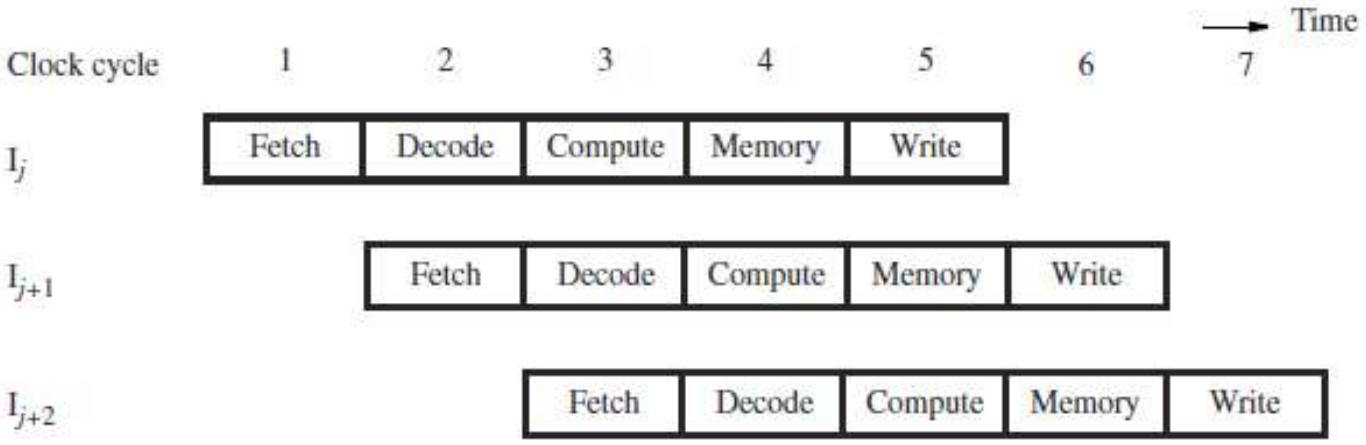


Figure 6.1 Pipelined execution—the ideal case.

Pipelining Terminology

- Pipe Stage / Pipe Segment
  - A step in the pipeline to complete the instruction
- Pipeline Depth
  - Number of stages in a pipeline.
- Pipeline Latency
  - How long does it take to execute a single instruction in a pipeline.
- Pipeline Throughput
  - The number of instructions completed per second.

Pipeline Summary

- Pipelining doesn't improve the latency of instructions (each instruction still requires the same amount of time to complete).
- It reduces the average execution time per instruction.
- It does improve the overall throughput.

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Pipeline Organization (1)

- Use program counter (PC) to fetch instructions
- A new instruction enters pipeline every cycle
- Carry along instruction-specific information as instructions flow through the different stages
  - Use *inter-stage buffers* to hold this information
  - These buffers incorporate RA, RB, RM, RY, RZ, IR, and PC-Temp registers from Chapter 5
  - The buffers also hold control signal settings

Pipeline Organization (2)

- A Five-Stage Pipeline
  - B1 holds a newly-fetched instruction.
  - B2 holds
    - Two operands read from the register file,
    - Source/destination register identifiers
    - Immediate value derived from the instruction,
    - Incremented PC value used as the return address for a subroutine call
    - Settings of control signals determined by the instruction decoder.

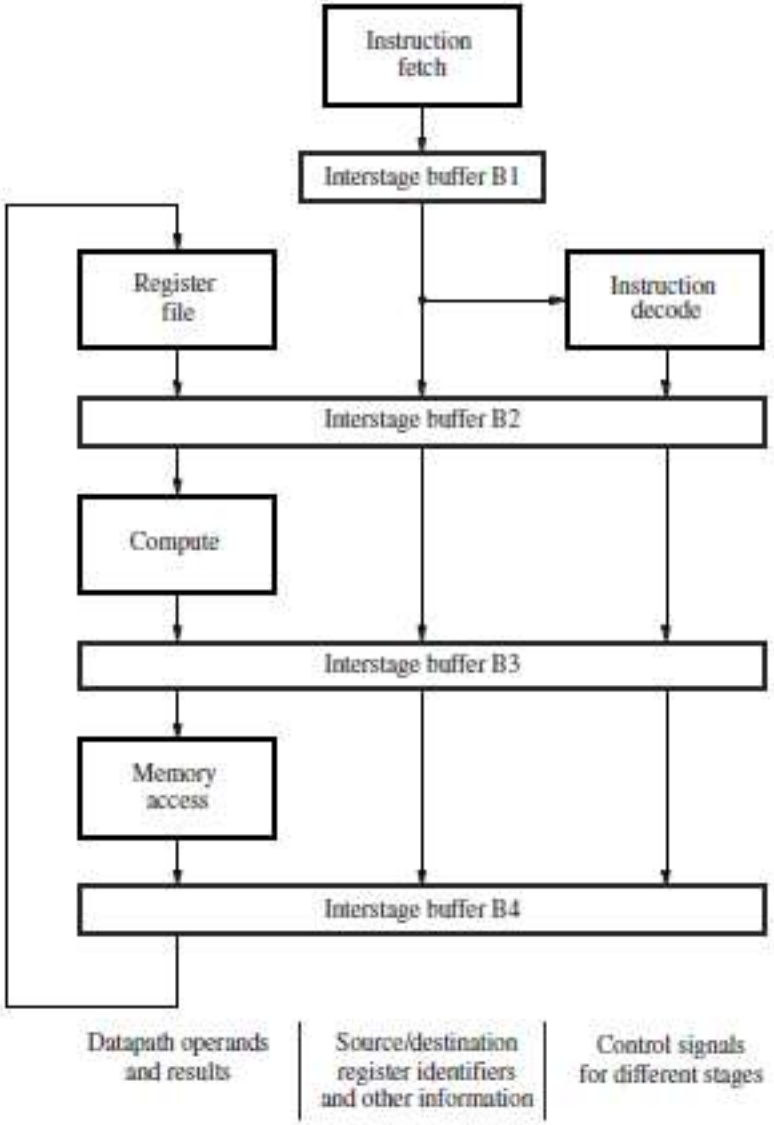


Figure 6.2 A five-stage pipeline.

Pipeline Organization (3)

- A Five-Stage Pipeline (ctd.)
  - B3 holds
    - Result of ALU operation: data or address
    - Incremented PC value used as the return address for a subroutine call
  - B4 holds a value to be written into the register

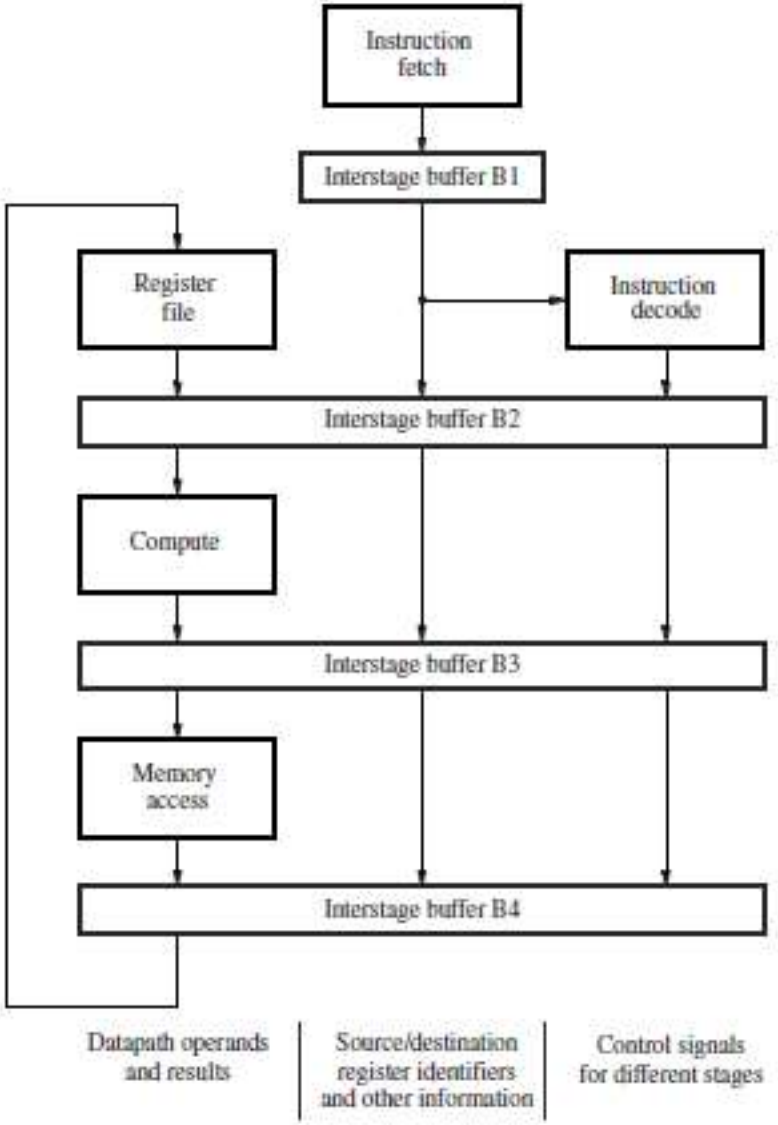


Figure 6.2 A five-stage pipeline.

Summary

- 知识点：Basic Concept of Pipeline
  - What is pipelining?
  - Principle of pipeline
  - Pipeline terminology
    - Pipeline stage
    - Pipeline depth
    - Pipeline latency
    - Pipeline throughput
- 掌握程度
  - 定义
  - 掌握流水线的原理

Exercise (1)

- What is the first stage in a typical five-stage CPU pipeline?
  - Fetch
  - Decode
  - Compute
  - Write

Exercise (2)

- When multiple-instructions are overlapped during execution of program, then function performed is called ( ).
  - Multitasking
  - Multiprogramming
  - Hardwired control
  - Pipelining

Exercise (3)

- Pipelining increases processor performance by decreasing the execution time of an instruction.
  - True
  - False

流水线不会减少单条指令的执行时间，只会减少多条指令的平均执行时间。

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Pipeline Issues (1)

- Any condition that causes a pipeline to stall is called a hazard.
- Three Types of 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 Issues (2)

- An Example of Data Hazard
  - Consider two successive instructions I_j and I_{j+1}
  - Assume that the destination register of I_j matches one of the source registers of I_{j+1}
  - Result of I_j is written to destination in cycle 5
  - But I_{j+1} reads *old* value of register in cycle 3
  - Due to pipelining, I_{j+1} computation is incorrect
  - So *stall* (delay) I_{j+1} until I_j writes the new value

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Data Dependencies (1)

- A Specific Data Hazard Example

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

  - Destination R2 of Add is a source for Subtract
  - There is a *data dependency* between them because R2 carries data from Add to Subtract
  - On *non*-pipelined datapath, result is available in R2 because Add completes before Subtract

Data Dependencies (2)

- Stalling the Pipeline
  - With pipelined execution, old value is still in register R2 when Subtract is in Decode stage
  - So stall Subtract for 3 cycles in Decode stage
  - New value of R2 is then available in cycle 6

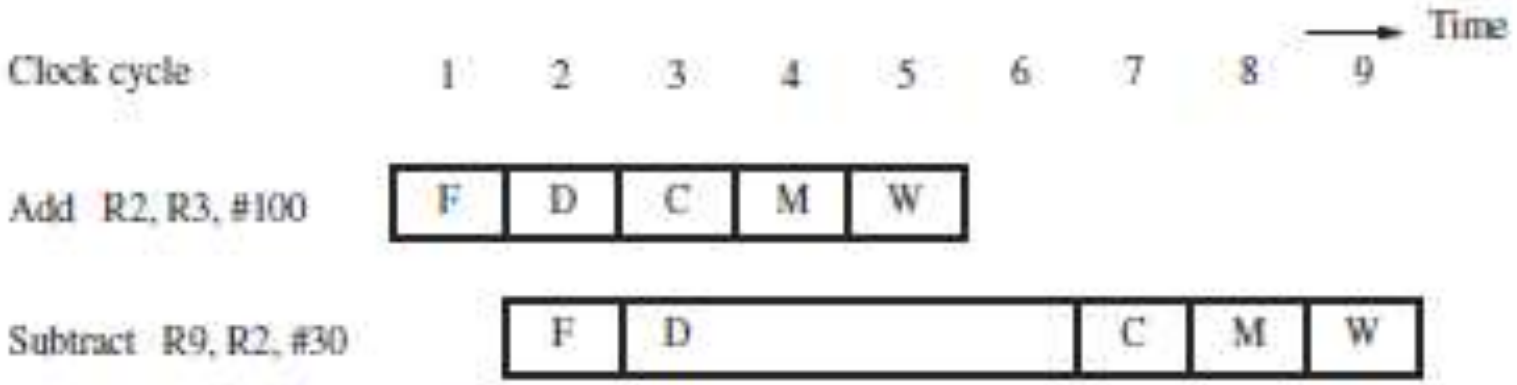


Figure 6.3 Pipeline stall due to data dependency.

Data Dependencies (3)

- Details for Stalling the Pipeline
  - Control circuitry must recognize dependency while Subtract is being decoded in cycle 3
  - Interstage buffers carry register identifiers for source(s) and destination of instructions
  - In cycle 3, compare destination identifier in Compute stage against source(s) in Decode
  - R2 matches, so Subtract kept in Decode while Add allowed to continue normally

Operand Forwarding (1)

- Operand forwarding handles dependencies without the penalty of stalling the pipeline
- For the preceding sequence of instructions, new value for R2 is available at end of cycle 3
- *Forward* value to where it is needed in cycle 4

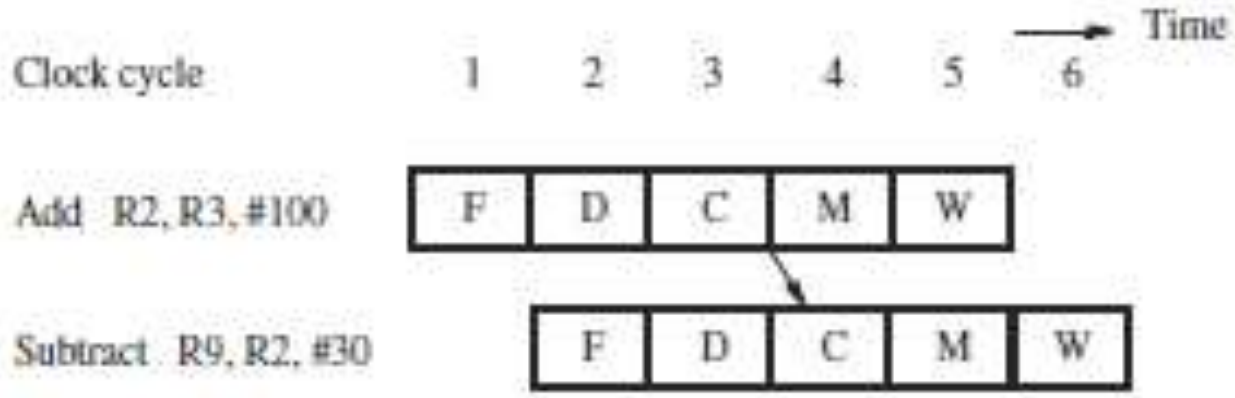


Figure 6.4 Avoiding a stall by using operand forwarding.

Operand Forwarding (2)

- Hardware Details
  - Introduce multiplexers before ALU inputs to use contents of register RZ as forwarded value
  - Control circuitry now recognizes dependency in cycle 4 when Subtract is in Compute stage
  - Interstage buffers still carry register identifiers
  - Compare destination of Add in Memory stage with source(s) of Subtract in Compute stage
  - Set multiplexer control based on comparison

Operand Forwarding (3)

- Hardware Details (ctd.)

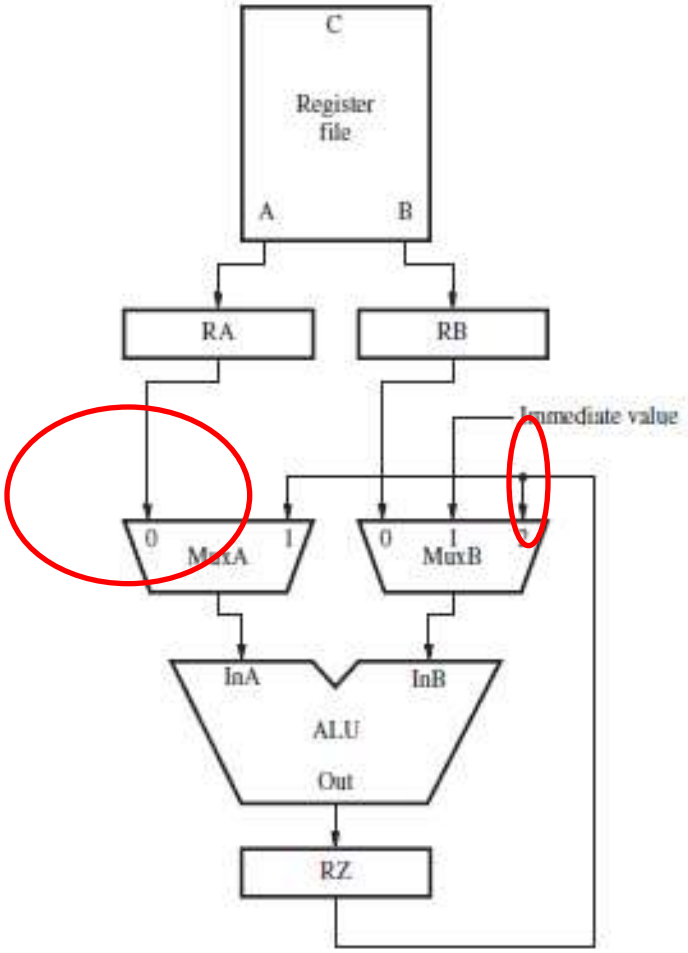


Figure 6.5 Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

Extension of Operand Forwarding (1)

- Forwarding can also be extended to a result in register RY in Figure 5.8.
- This would handle a data dependency such as the one involving register R2 in the following sequence of instructions:

  Add R2, R3, #100

  Or R4, R5, R6

  Subtract R9, R2, #30

Extension of Operand Forwarding (2)

- Hardware Detail

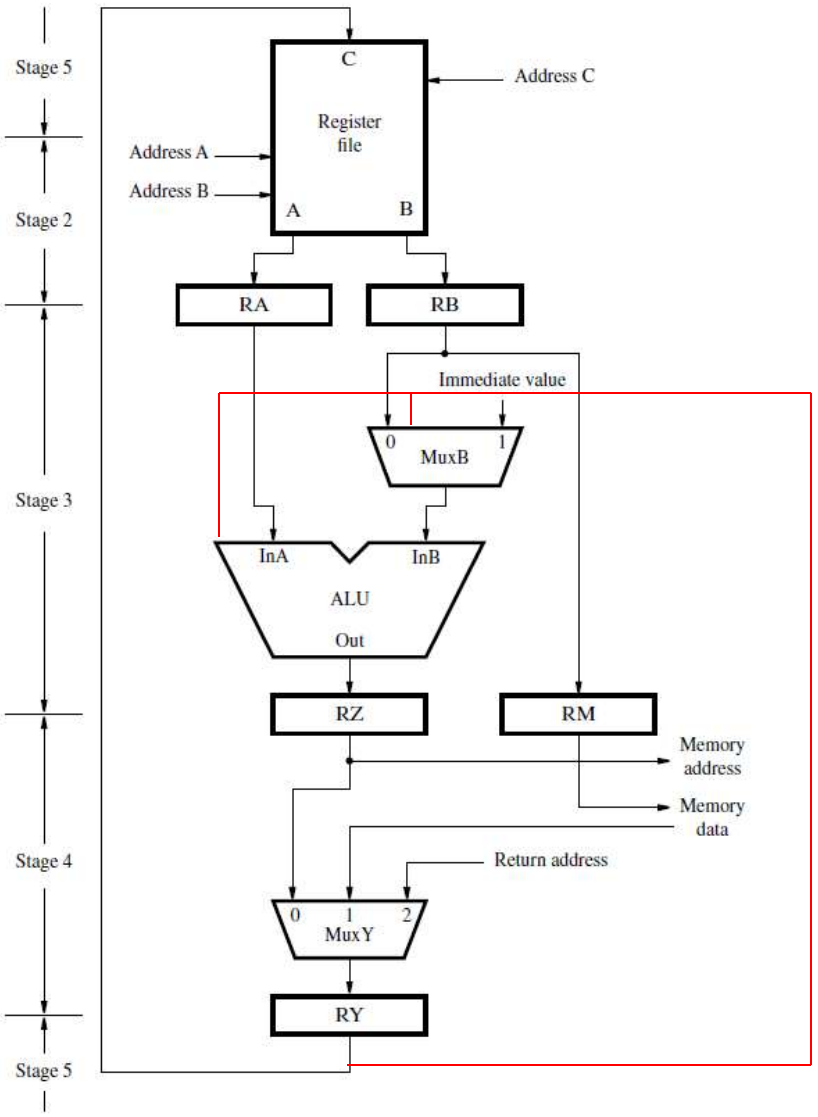


Figure 5.8 Datapath in a processor.

Software Handling of Dependencies (1)

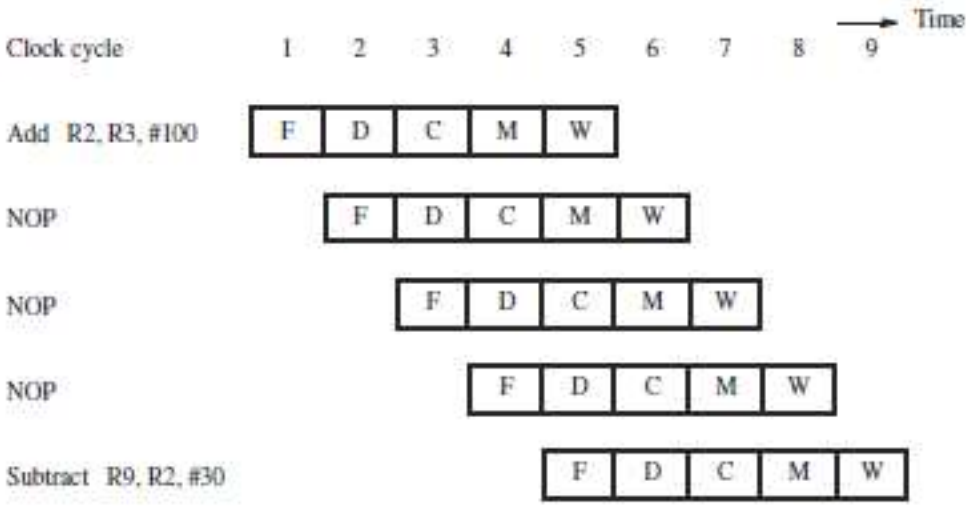
- Compiler can generate & analyze instructions.
- Data dependencies are evident from registers.
- Compiler puts three *explicit* NOP instructions between instructions having a dependency.
- Delay ensures new value available in register but causes total execution time to increase.
- Compiler can *optimize* by moving instructions into NOP slots (if data dependencies permit).

Software Handling of Dependencies (2)

- Example

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

(a) Insertion of NOP instructions for a data dependency



(b) Pipelined execution of instructions

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

Example 6.1 (1)

- Textbook P221 Problem: Consider the pipelined execution of the following sequence of instructions:
  - Add R4, R3, R2
  - Or R7, R6, R5
  - Subtract R8, R7, R4
- Initially, registers R2 and R3 contain 4 and 8, respectively. Registers R5 and R6 contain 128 and 2, respectively. Assume that the pipeline provides forwarding paths to the ALU from registers RY and RZ in Figure 5.8. The first instruction is fetched in cycle 1, and the remaining instructions are fetched in successive cycles. Draw a diagram similar to Figure 6.1 to show the pipelined execution of these instructions assuming that the processor uses operand forwarding. Then, with reference to Figure 5.8, describe the contents of registers RY and RZ during cycles 4 to 7

Example 6.1 (2)

- Solution

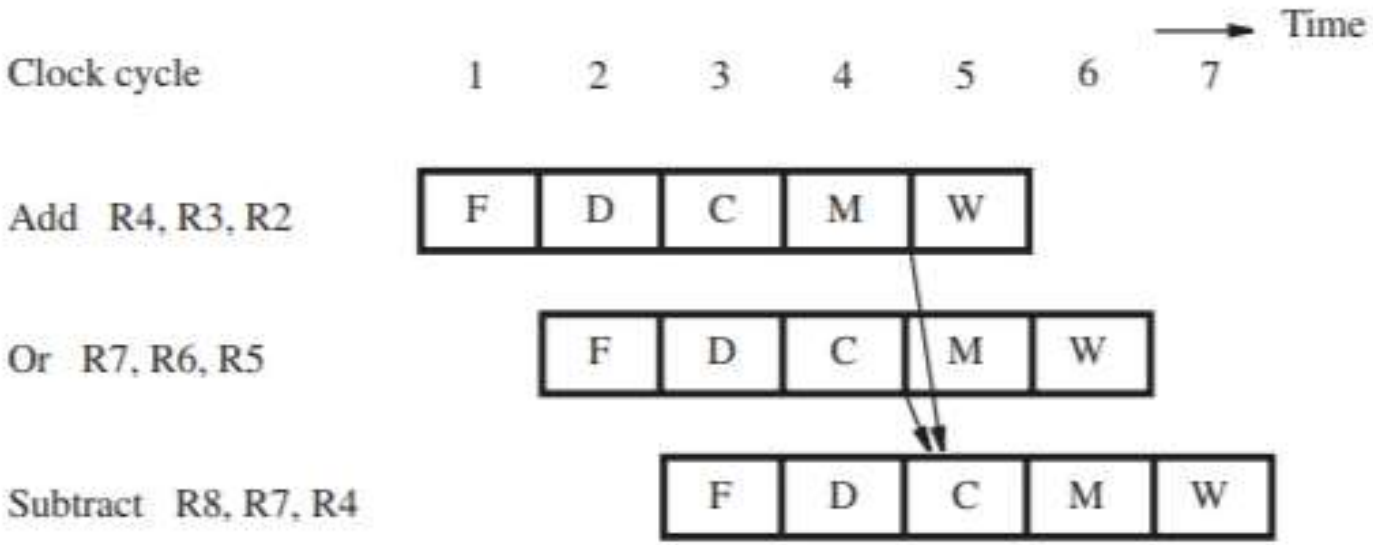


Figure 6.15 Pipelined execution of instructions for Example 6.1.

Summary

- 知识点：Data Dependencies
- 掌握程度
  - 定义
  - 解决方法
    - Operand Forwarding
    - Software

Exercise (1)

- Hazards in pipelined stages are of ( ).
  - Two types
  - Three types
  - Four types
  - Five types

Exercise (2)

- ( ) is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline.
  - Control hazard
  - Data hazard
  - Structural hazard
  - Instruction hazard

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Memory Delays (1)

- Memory delays can also cause pipeline stalls.
- A cache memory holds instructions and data from the main memory, but is faster to access.
- With a cache, typical access time is one cycle.
- But a cache *miss* requires accessing slower main memory with a much longer delay.
- In pipeline, memory delay for one instruction causes subsequent instructions to be delayed.

Memory Delays (2)

- Memory Delay for a Load Instruction

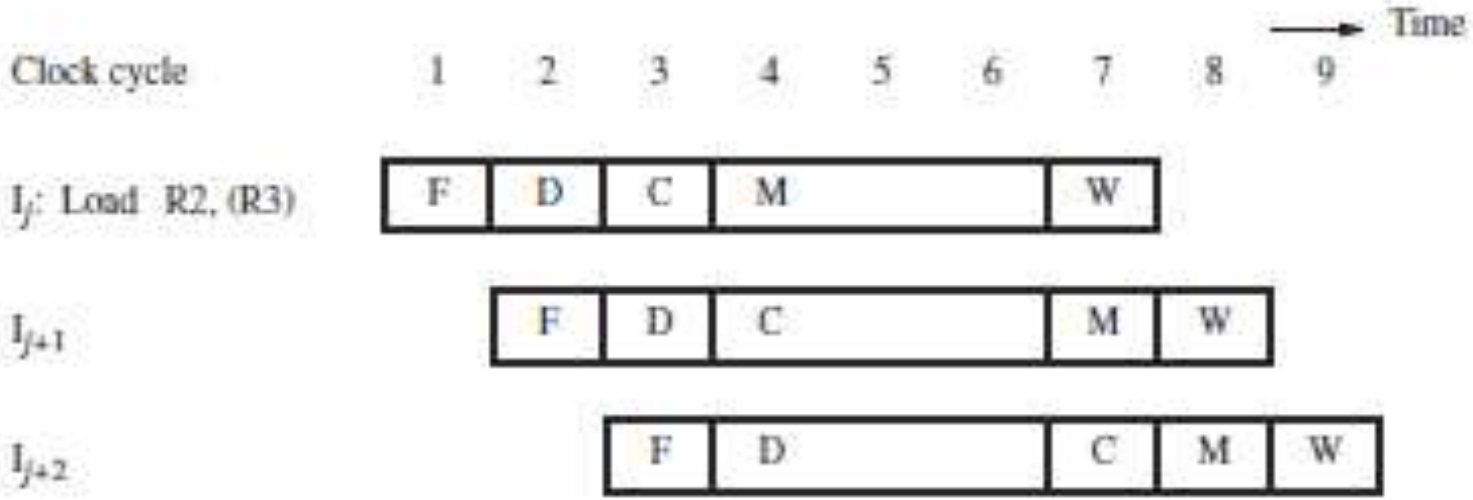


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

Memory Delays (3)

- Even with a cache *hit*, a Load instruction may cause a short delay due to a data dependency
- One-cycle stall required for correct value to be forwarded to instruction needing that value
- The compiler can eliminate the one-cycle stall. Optimize with useful instruction to fill delay

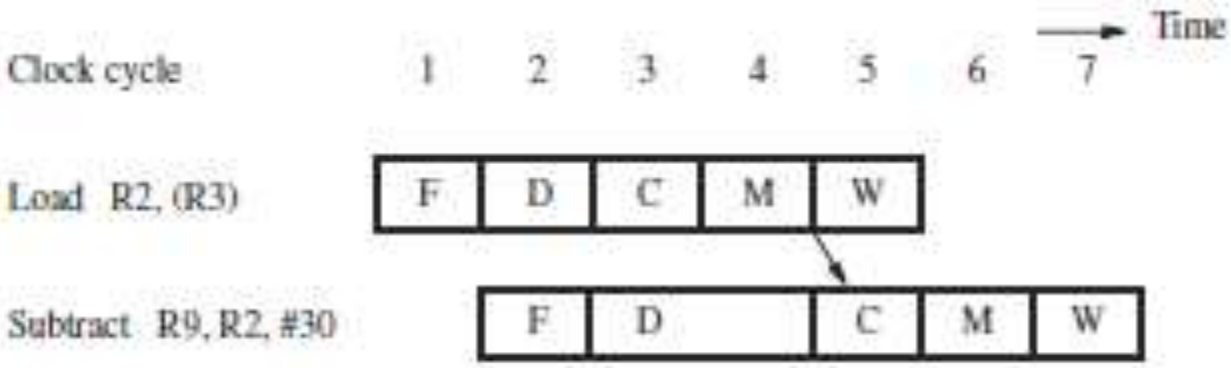


Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

Note

- Not all data dependencies can be resolved by using forwarding.
- Sometimes stall is necessary!

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Branch Delays (1)

- Ideal pipelining: fetch each new instruction while previous instruction is being decoded
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Branch instructions alter execution sequence, but they must be processed to know the effect
- Any delay for determining branch outcome leads to an increase in total execution time
- Techniques to mitigate this effect are desired
- Understand branch behaviour to find solutions

Branch Delays (2)

- Unconditional Branches
  - Consider instructions I_j , I_{j+1} , I_{j+2} in sequence
  - I_j is an unconditional branch with target I_k
  - In Chapter 5, the Compute stage determined the target address using offset and PC+4 value

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	PC ← [PC] + Branch offset
4	No action
5	No action

Figure 5.15 Sequence of actions needed to fetch and execute an unconditional branch instruction.

Branch Delays (3)

- Unconditional Branches (ctd.)
  - In pipeline, target I_k is known for I_j in cycle 4, but instructions I_{j+1} , I_{j+2} fetched in cycles 2 & 3
  - Target I_k should have followed I_j immediately, so discard I_{j+1} , I_{j+2} and incur two-cycle *penalty*

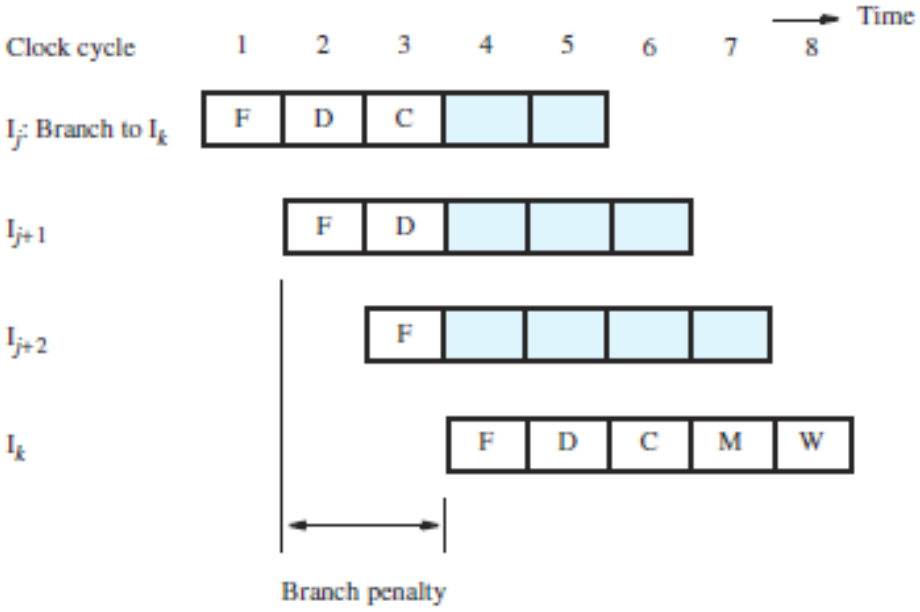


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

Branch Delays (4)

- Reducing the Branch Penalty
  - In pipeline, adder for PC is used every cycle, so it cannot calculate the branch target address
  - So introduce a second adder just for branches
  - Place this second adder in the Decode stage to enable earlier determination of target address
  - For previous example, now only I_{j+1} is fetched
  - Only one instruction needs to be discarded
  - The branch penalty is reduced to one cycle

Branch Delays (5)

- Reducing the Branch Penalty (ctd.)

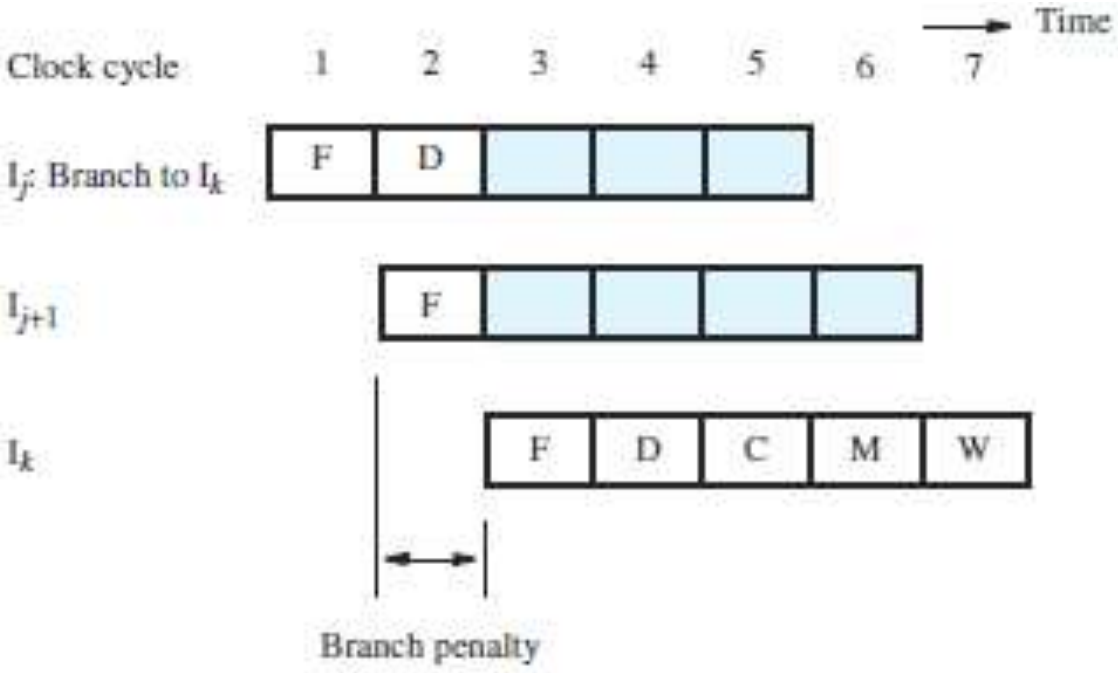


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

Branch Delays (6)

- Conditional Branches
  - Consider a conditional branch instruction:
    Branch_if_[R5]=[R6] LOOP
  - Requires not only target address calculation, but also requires comparison for condition
  - In Chapter 5, ALU performed the comparison

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R5], RB ← [R6]
3	Compare [RA] to [RB], If [RA] = [RB], then PC ← [PC] + Branch offset
4	No action
5	No action

Figure 5.16 Sequence of actions needed to fetch and execute the instruction: Branch_if_[R5]=[R6] LOOP.

Branch Delays (7)

- Conditional Branches (ctd.)
  - To maintain one-cycle penalty, we introduce a comparator just for branches in Decode stage
  - Target address now calculated in Decode stage

Branch Delays (8)

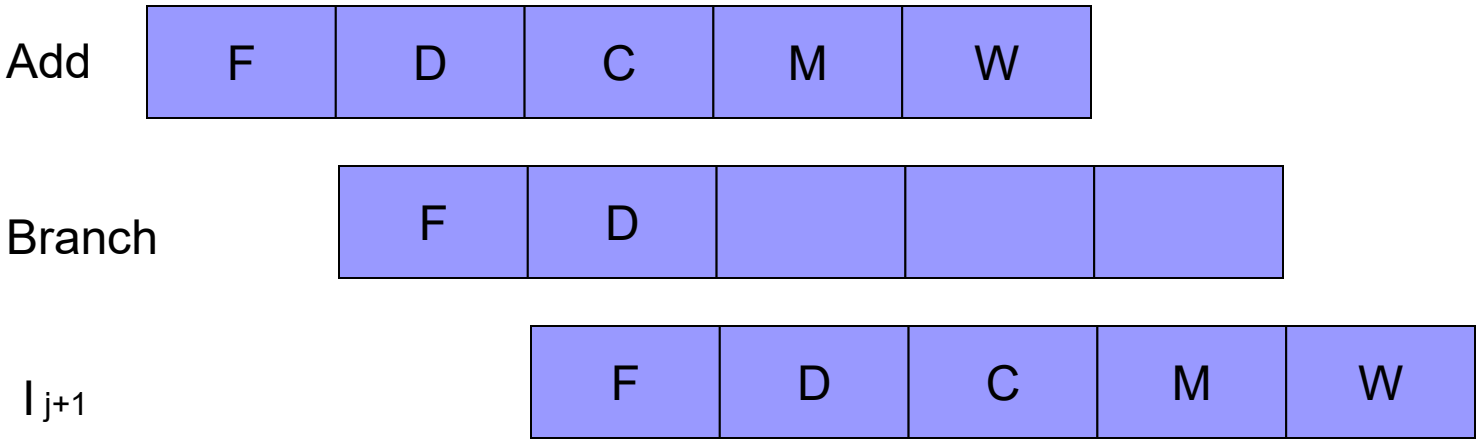
- The Branch Delay Slot
  - Let both branch decision and target address be determined in Decode stage of pipeline.

	Add	R7, R8, R9
	Branch_if_[R3]=0	TARGET
	I_{j+1}	
	⋮	
TARGET:	I_k	

(a) Original sequence of instructions containing a conditional branch instruction

Branch Delays (9)

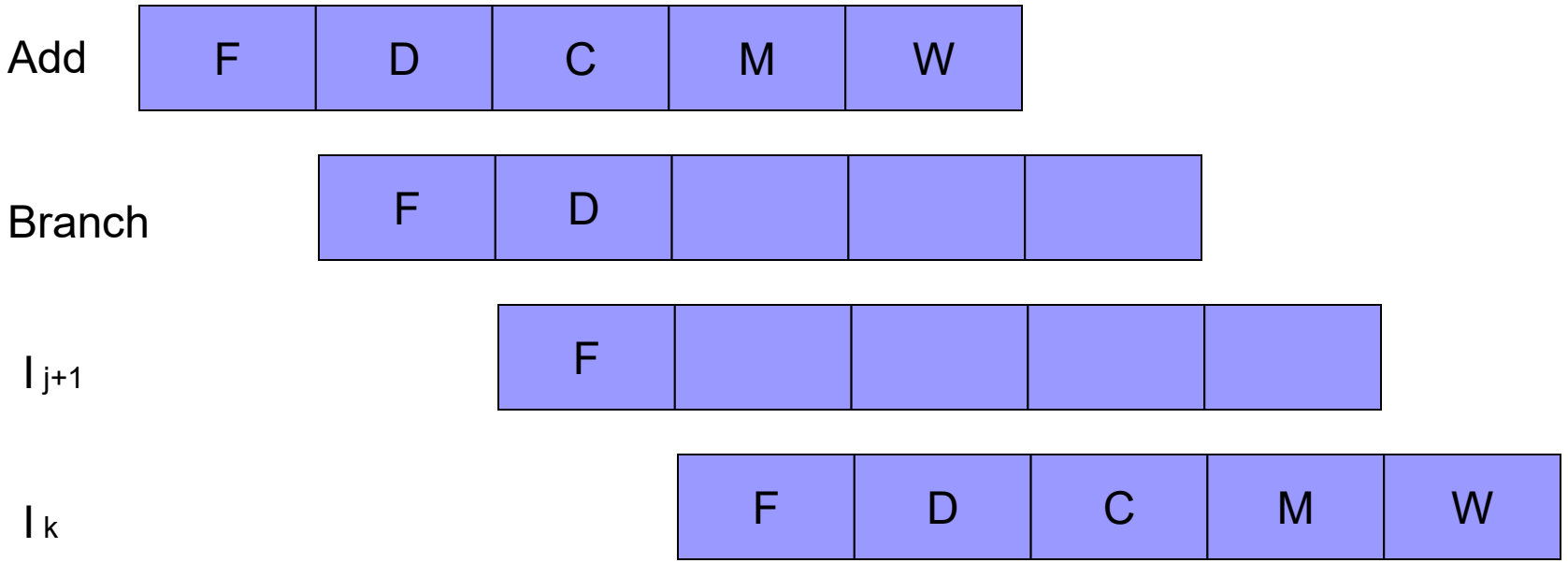
- The Branch Delay Slot (ctd.)



  - Branch not taken, no branch penalty

Branch Delays (10)

- The Branch Delay Slot (ctd.)



  - Branch taken, one cycle of branch penalty

Branch Delays (11)

- The Branch Delay Slot (ctd.)
  - Instruction immediately following a branch is always fetched, regardless of branch decision
  - That next instruction is discarded with penalty, except when conditional branch is not taken
  - The location immediately following the branch is called the *branch delay slot*

Branch Delays (12)

- The Branch Delay Slot (ctd.)
  - Instead of conditionally discarding instruction in delay slot, *always* let it complete execution
  - Let compiler find an instruction *before* branch to move into slot, if data dependencies permit
  - Called *delayed branching* due to reordering
  - If useful instruction put in slot, penalty is *zero*
  - If not possible, insert explicit NOP in delay slot for one-cycle penalty, whether or not taken

Branch Delays (13)

- The Branch Delay Slot (ctd.)

	Branch_if_[R3]=0	TARGET
	Add	R7, R8, R9
	I_{j+1}	
	⋮	
TARGET:	I_k	

(b) Placing the Add instruction in the branch delay slot where it is always executed

Branch Delays (14)

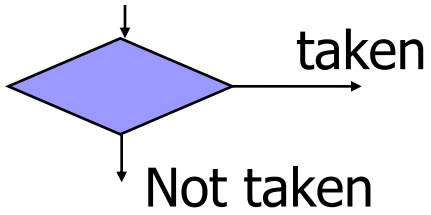
- Limitations of delayed branching
  - 50% of the time the compiler can't fill delay slot with useful instructions while maintaining correctness (has to insert nops instead)
  - High performance pipelines may have >10 delay slots
    - Many cycles for instruction fetch and decode
    - Multiple instructions in each pipeline stage
- Solution: *branch prediction* (later)

Branch Delays (15)

- Branch Prediction
  - A branch is decided in Decode stage (cycle 2) while following instruction is *always* fetched
  - Following instruction may require discarding (or with delayed branching, it may be a NOP)
  - Instead of discarding the *following* instruction, can we anticipate the *actual* next instruction?
  - Two aims: (a) *predict* the branch decision
    (b) use prediction *earlier* in cycle 1

Branch Delays (16)

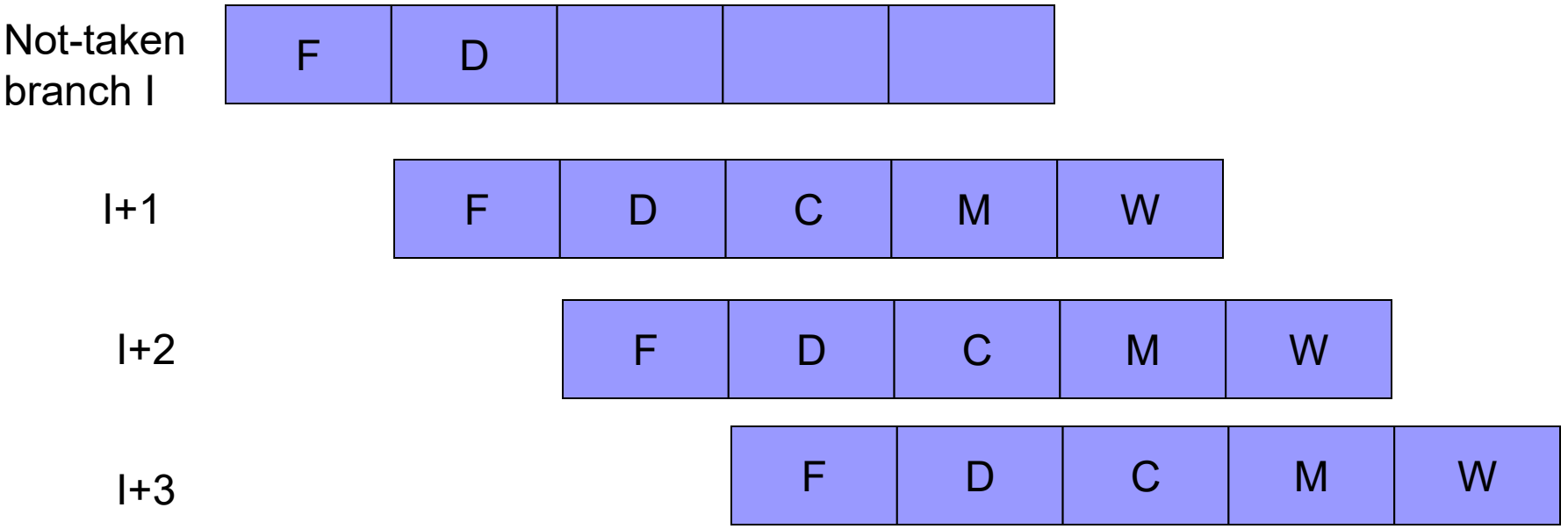
- Predict branch direction: taken or not taken (T/NT)



- Static Branch Prediction: compilers decide the direction
  - Not Taken(30-40% accuracy … not so good)
    - Fetch the next instruction in sequential address order
    - Prediction Correct: the fetched instruction is completed and no penalty
    - Prediction Incorrect: the fetched instruction is discarded and the correct branch target instruction is fetched, full branch penalty

Branch Delays (17)

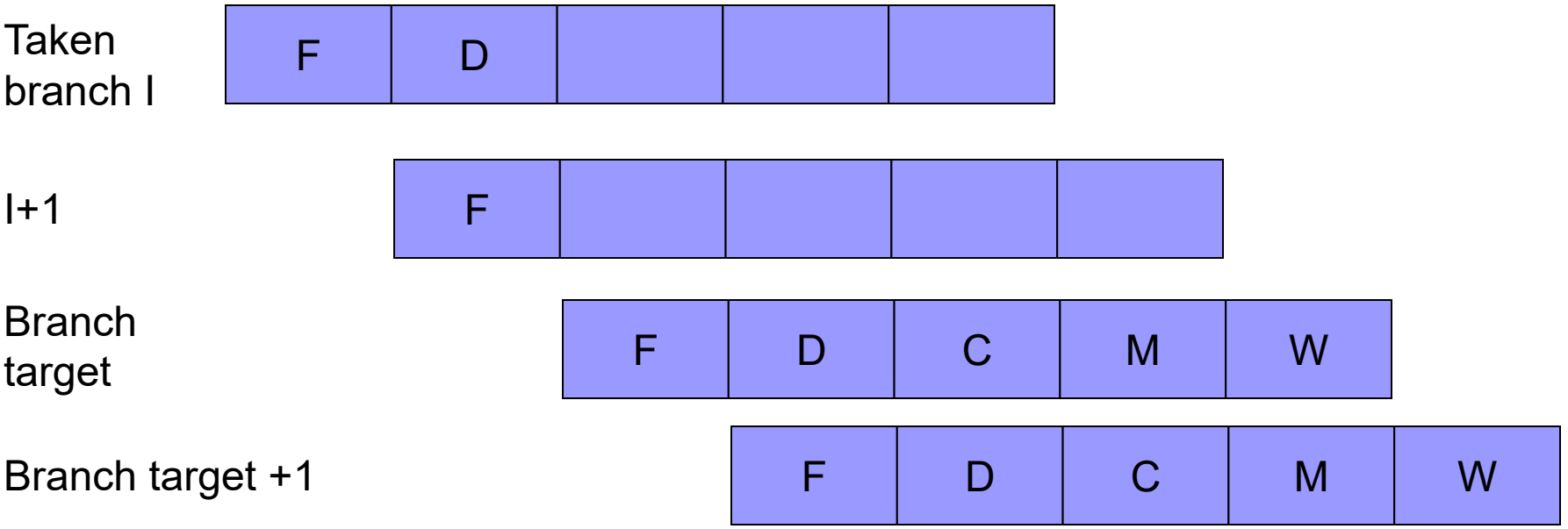
- Static Branch Prediction (ctd.)
  - Not Taken (ctd.)



Prediction Correct

Branch Delays (18)

- Static Branch Prediction (ctd.)
  - Not Taken (ctd.)



Prediction Incorrect

Branch Delays (19)

- Static Branch Prediction (ctd.)
  - Taken (60-70% accuracy)
    - As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target.
  - Example: loop and if-statement branches
    - Predict a backward branch at the end of a loop taken
    - Predict a forward branch at the beginning of a loop not taken
- Dynamic Branch Prediction: hardware decides the direction using dynamic information

Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

Structural Hazards

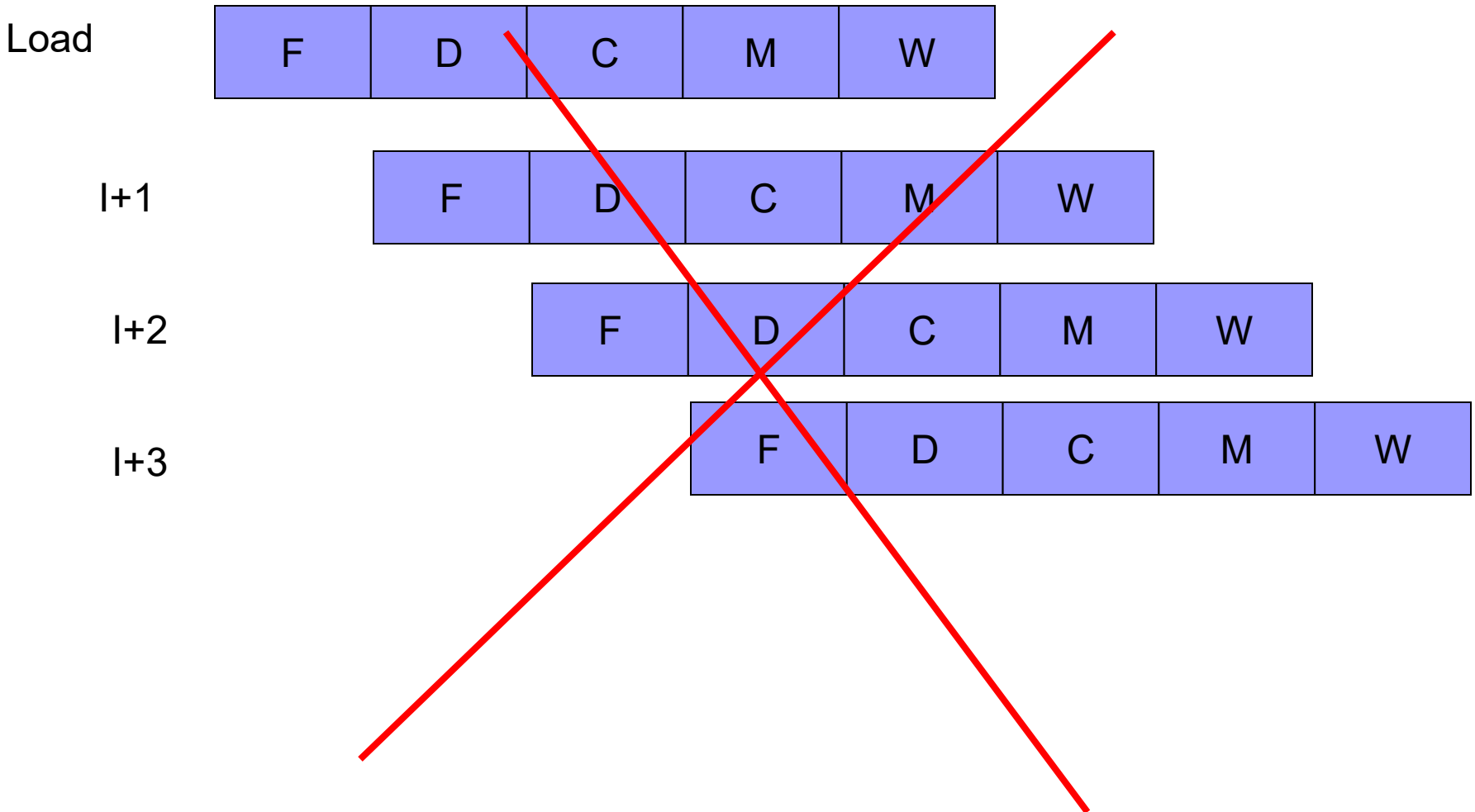
- Resource conflicts when hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
- Can be prevented by providing additional hardware.

Reasons for Structural Hazards

- Reasons for structural hazards
  - Some functional units not fully pipelined
  - Some resources not duplicated enough
- Pipeline stalls when there are insufficient hardware resources to permit all actions to proceed concurrently.

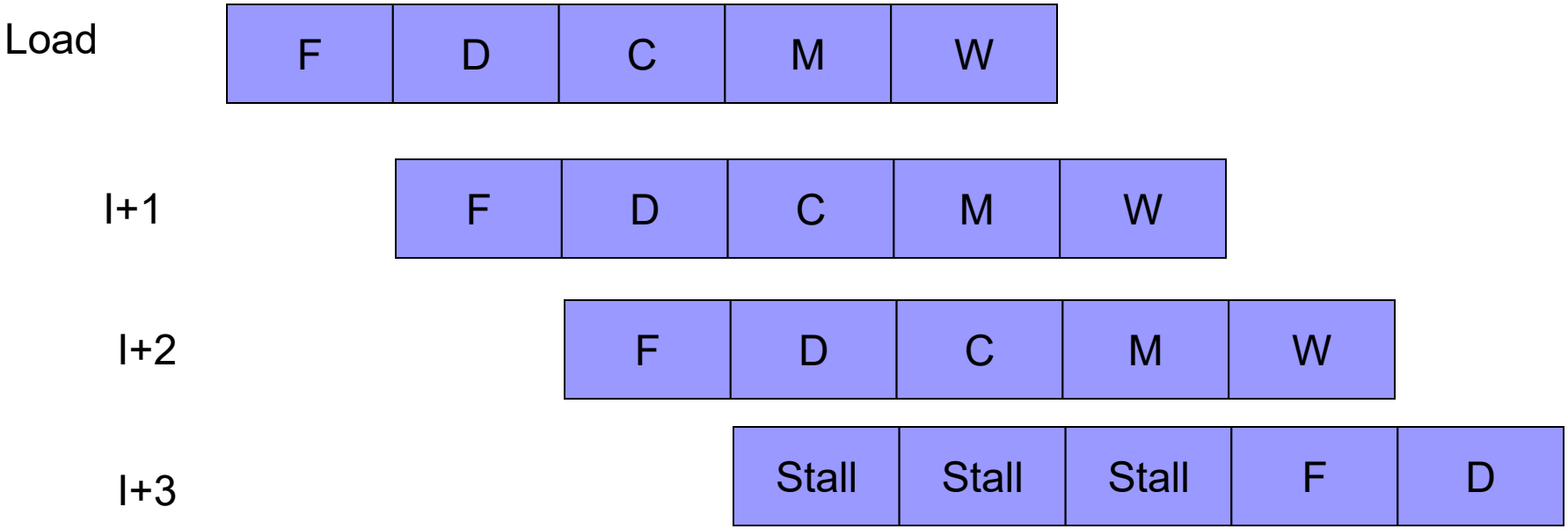
Example (1)

- Assume a processor with only one memory port



Example (2)

- Assume a processor with only one memory port (ctd.)



Content of this lecture

- 6.1 Basic Concept
- 6.2 Pipeline Organization
- 6.3 Pipeline Issues
- 6.4 Data Dependencies
- 6.5 Memory Delays
- 6.6 Branch Delays
- 6.7 Resource Limitations
- 6.9 Superscalar Operation

What is Superscalar?

- A Superscalar machine executes multiple independent instructions in parallel.
- They are pipelined as well.
- “Common” instructions (arithmetic, load/store, conditional branch) can be executed independently.
- Equally applicable to RISC & CISC, but more straightforward in RISC machines.
- The order of execution is usually assisted by the compiler.

A Superscalar Processor (1)

- A Superscalar Processor with two execution units

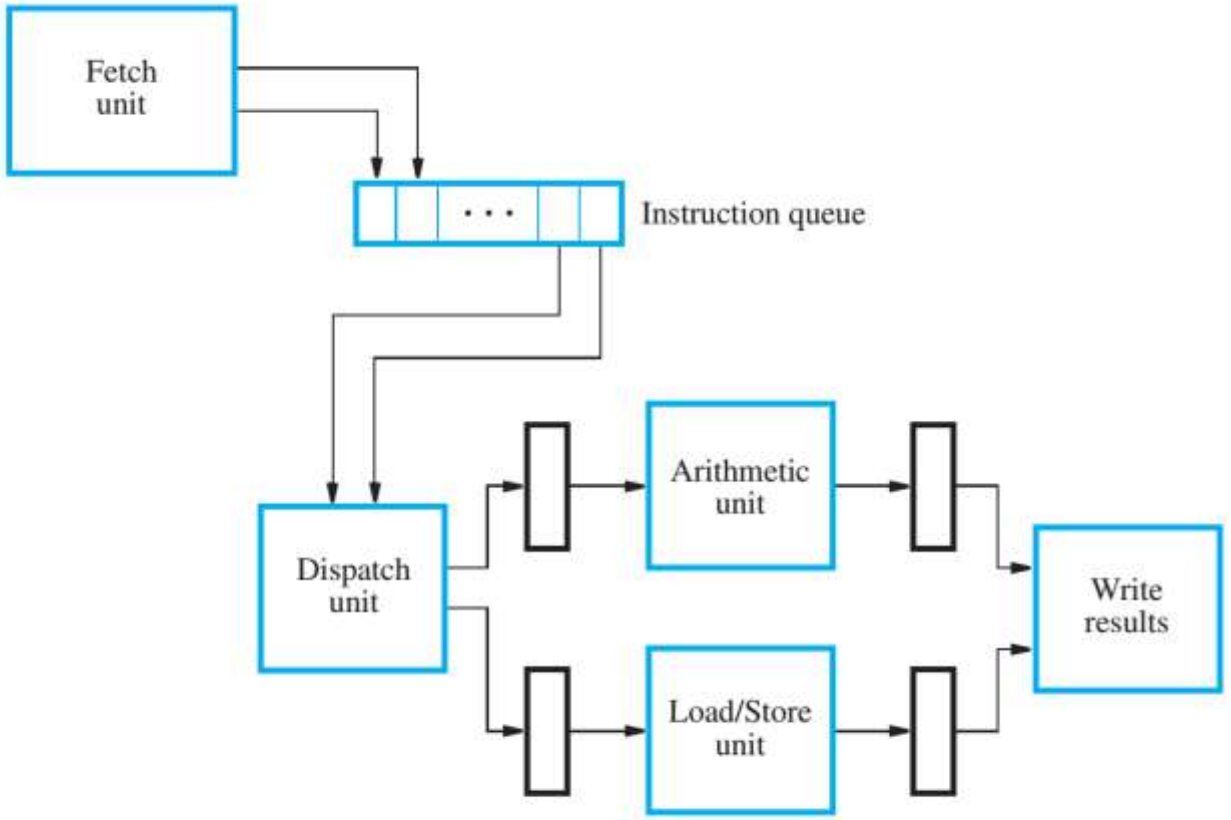


Figure 6.13 A superscalar processor with two execution units.

A Superscalar Processor (2)

- A Superscalar Processor with two execution units (ctd.)
  - The Load/Store unit has a two-stage pipeline.
  - The register file must now have four output ports and two input ports.

Superscalar Execution Example (1)

- Example

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

Superscalar Execution Example (2)

- Example (ctd.)

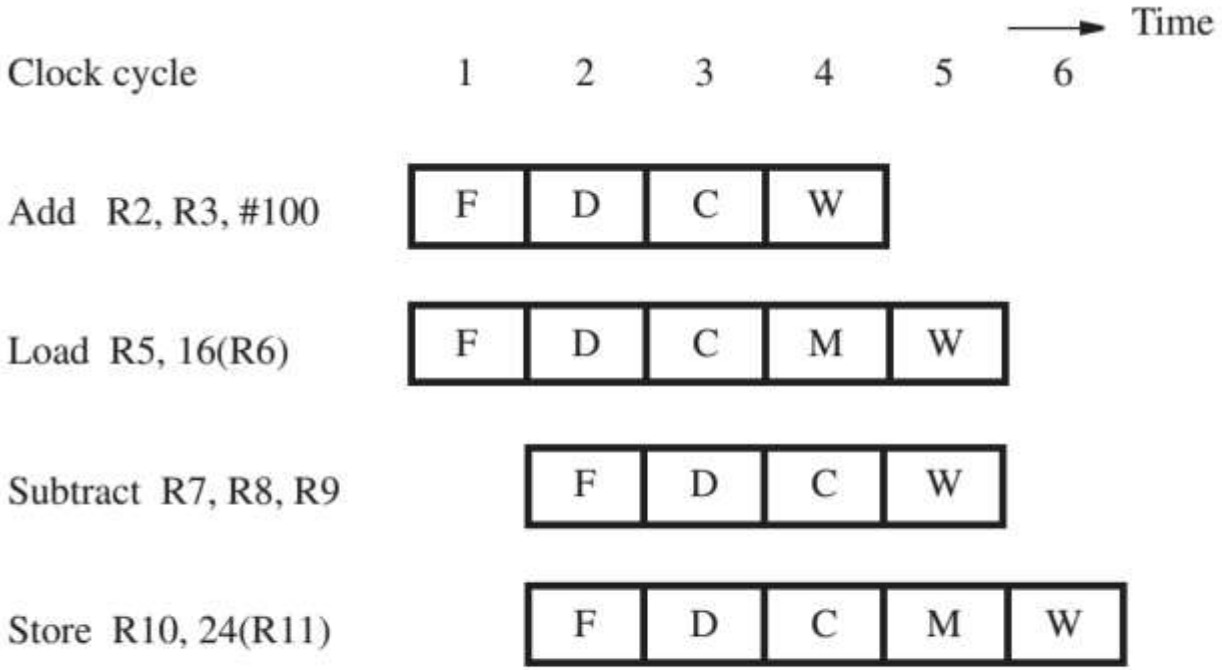


Figure 6.14 An example of instruction flow in the processor of Figure 6.13.

Modern superscalar processors

- Today's superscalar processors attempt to *issue* (initiate the execution of) 4-6 instructions each clock cycle
- Such processors have multiple integer ALUs, integer multipliers, and floating point units that operate in parallel on different instructions
- Because most of these units are pipelined, there is the potential to have 10's of instructions simultaneously executing.

Homework (1)

- P223 6.1, 6.2

其中：(a) Draw a diagram similar to Figure 6.1 that represents the flow of the instructions through the pipeline. 去掉Describe the operation being performed by each pipeline stage during clock cycles 1 through 8.

- 补充题：

Consider the following sequence of instructions being processed on the pipelined 5-stage RISC processor:

```
Load      R4, #100(R2)
Add       R5, R2, R3
Subtract  R6, R4, R5
And       R7, R2, R5
```

Homework (2)

- 补充题：(ctd.)
  - (1)Identify all the data dependencies in the above instruction sequence. For each dependency, indicate the two instructions and the register that causes the dependency.
  - (2)Assume that the pipeline does not use operand forwarding. Also assume that the only sources of pipeline stalls are the data hazards. Draw a diagram that represents instruction flow through the pipeline during each clock cycle.
  - (3)Assume that the pipeline uses operand forwarding. There are separate forwarding paths from the outputs of stage-3 and stage-4 to the input of stage-3. Draw a diagram that represents the flow of instructions through the pipeline during each clock cycle. Indicate operand forwarding by arrows.