the essentials of

Computer Organization and Architecture

Linda Null and Julia Lobur

Chapter 5

A Closer Look at Instruction Set Architectures

Chapter 5 Objectives

- Understand the factors involved in instruction set architecture design.
- Gain familiarity with memory addressing modes.
- Understand the concepts of instruction-level pipelining and its affect upon execution performance.

5.1 Introduction

- This chapter builds upon the ideas in Chapter 4.
- We present a detailed look at different instruction formats, operand types, and memory access methods.
- We will see the interrelation between machine organization and instruction formats.
- This leads to a deeper understanding of computer architecture in general.

5.2 Instruction Formats

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.(0,1,2, 3)
- Operand location.(Register to Register, M to M, R to M)
- Operations(Types of operations, instructions can access memory or cannot).
- Type and size of operands(Operands can be addresses, numbers, charactrers).

5.2 Instruction Formats

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.(decoding necessary to execute an instruction, complexity of the tasks performed)
- Instruction length (in bits).
- Total number of instructions in the instruction set.

5.2 Instruction Formats

In designing an instruction set, consideration is given to:

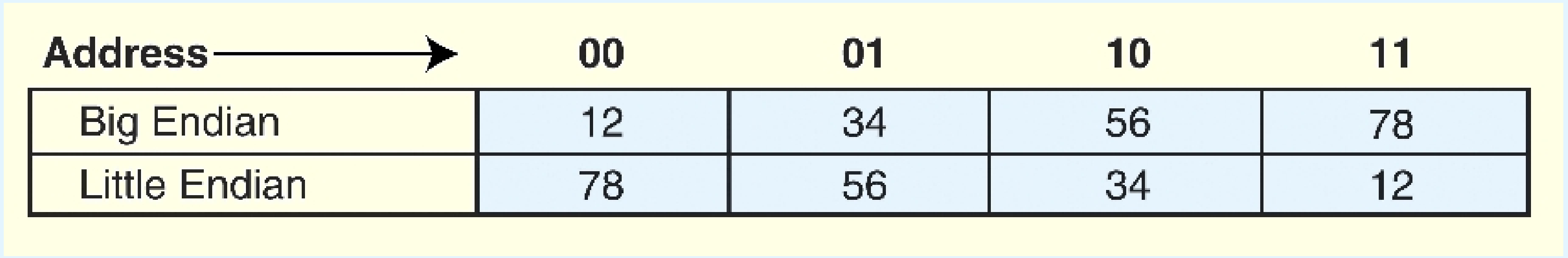
- Instruction length.
  - Whether short, long, or variable.
- Number of operands.
- Number of addressable registers.
- Memory organization.
  - Whether byte- or word addressable.
- Addressing modes.
  - Choose any or all: direct, indirect or indexed.

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - In *little endian* machines, the least significant byte is followed by the most significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 12345678.
- The big endian and small endian arrangements of the bytes are shown below.



Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

5.2 Instruction Formats

- 1. Assume you have a machine that uses 32-bit integers and you are storing the hex value 1234 at address 0.
  - a. Show how this is stored on a big endian machine.
  - b. Show how this is stored on a little endian machine.
- 2.The first two bytes of a 2M x 16 main memory have the following hex values: Byte 0 is FE, Byte 1 is 01,If these bytes hold a 16-bit two's complement integer, what is its actual decimal value if:
  - a. memory is big endian?
  - b. memory is little endian?

5.2 Instruction Formats

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.(little endian where you must know how long the number is)
  - Strings and integers are stored in the same order.
  - Bit mapped graphics is more efficient to access
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - High precision arithmetic on little endian machines is faster and easier.
- Computer networks are big endian

5.2 Instruction Formats

- The next consideration for architecture design concerns how the CPU will store data.
- We have three choices:
  1. A stack architecture
  2. An accumulator architecture
  3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

5.2 Instruction Formats

- In a stack architecture, instructions and operands are implicitly taken from the stack.
  - A stack cannot be accessed randomly.
- In an accumulator architecture, one operand of a binary operation is implicitly in the accumulator.
  - Allow for very short instructions
  - One operand is in memory, creating lots of bus traffic.
- In a general purpose register (GPR) architecture, registers can be used instead of memory.
  - Faster than accumulator architecture.
  - Efficient implementation for compilers.
  - Results in longer instructions.

5.2 Instruction Formats

- Most systems today are GPR systems.
- There are three types of operands:
  - Memory-memory where two or three operands may be in memory.
  - Register-memory where at least one operand must be in a register.(Intel and Motorola)
  - Load-store where no operands may be in memory.(MIPS, ARM, PowerPC…)

5.2 Instruction Formats

- The number of operands and the number of available registers has a direct affect on instruction length.
  - OPCODE only(zero addresses)
  - OPCODE +1 address(usually a memory address)
  - OPCODE +2 address(registers, or one register and one memory address)
  - OPCODE +3 address(registers, combinations of registers and memory

5.2 Instruction Formats

- Stack machines use one - and zero-operand instructions.
- **`Push`** and **`pop`** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- Binary instructions (e.g., **`ADD`**, **`MULT`**) use the top two items on the stack.

5.2 Instruction Formats

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: Z = X + Y.
- Stack arithmetic requires that we use *postfix* notation: Z = XY+.
  - This is also called *reverse Polish notation*, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).

5.2 Instruction Formats

- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

  Z = (X × Y) + (W × U),

  becomes:

  Z = X Y × W U × +

  in postfix notation.

5.2 Instruction Formats

- 1. Convert the following expressions from infix to reverse Polish (postfix) notation.
  - a. X * Y + W * Z + V * U
  - b. W * X + W * (U * V + Z)
  - c. (W * (X + Y * (U * V)))/(U * (X + Y))
- 2. Convert the following expressions from reverse Polish notation to infix notation.
  - a. W X Y Z - + *
  - b. U V W X Y Z + * + * +
  - c. X Y Z + V W - * Z + +

5.2 Instruction Formats

- In a stack ISA, the postfix expression,

  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP Z
```

Note: The result of a binary operation is implicitly stored on the top of the stack!

5.2 Instruction Formats

- Write a program to evaluate 1.c statement above using a stack organized computer with zero-address instructions (so only pop and push can access memory).

5.2 Instruction Formats

- In a one-address ISA, like MARIE, the infix expression,

  $$Z = X \times Y + W \times U$$

  looks like this:

```
LOAD X
MULT Y
STORE TEMP
LOAD W
MULT U
ADD TEMP
STORE Z
```

5.2 Instruction Formats

- In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

$$Z = X \times Y + W \times U$$

might look like this:

```
LOAD  R1,X
MULT  R1,Y
LOAD  R2,W
MULT  R2,U
ADD   R1,R2
STORE Z,R1
```

Note: One-address ISAs usually require one operand to be a register.

5.2 Instruction Formats

- With a three-address ISA, (e.g.,mainframes), the infix expression,

  $$Z = X \times Y + W \times U$$

  might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

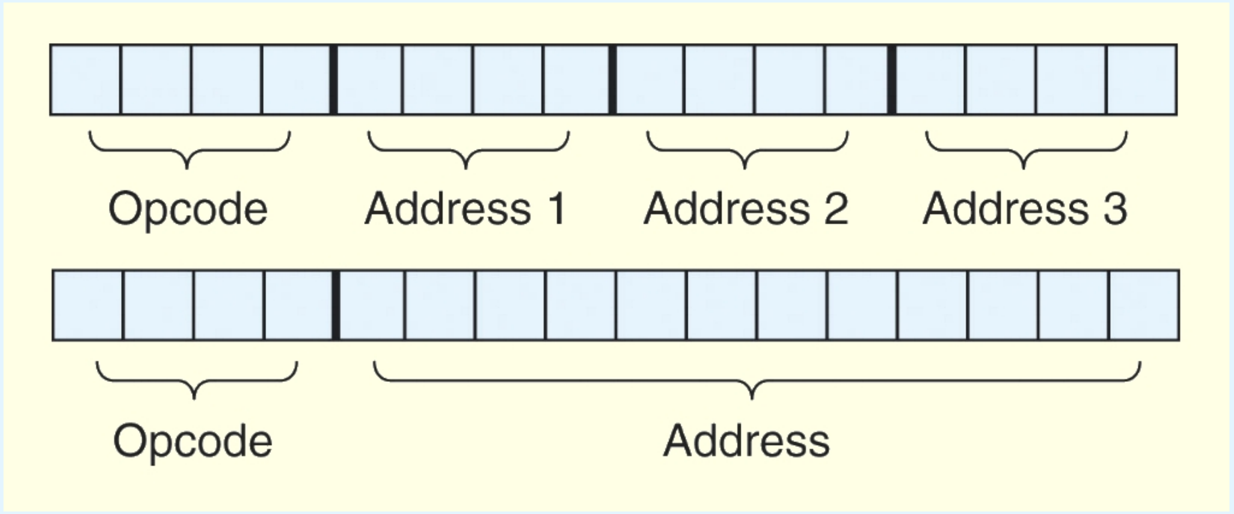
Would this program execute faster than the corresponding (longer) program that we saw in the stack-based ISA?

5.2 Instruction Formats

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as **`HALT`**, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

5.2 Instruction Formats

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



5.2 Instruction Formats

- If we allow the length of the opcode to vary, we could create a very rich instruction set:

```
0000 R1   R2   R3   }
                    }  15 3-address codes
1110 R1   R2   R3   }

1111 0000  R1   R2  }
                    }  14 2-address codes
1111 1101  R1   R2  }

1111 1110 0000  R1  }
                    }  31 1-address codes
1111 1111 1110  R1  }

1111 1111 1111 0000 }
                    }  16 0-address codes
1111 1111 1111 1111 }
```

5.2 Instruction Formats

- This scheme makes the decoding more complex. At each stage, one spare code is used to indicate that we should now look at more bits

```
if (leftmost four bits != 1111 ) {
   Execute appropriate three-address instruction}
else if (leftmost seven bits != 1111 111 ) {
   Execute appropriate two-address instruction}
else if (leftmost twelve bits != 1111 1111 1111 ) {
   Execute appropriate one-address instruction }
else {
   Execute appropriate zero-address instruction
}
```

5.2 Instruction Formats

- In a computer instruction format, the instruction length is 11 bits and the size of an address field is 4 bits. Is it possible to have:
  - 5 2-address instructions
  - 45 1-address instructions
  - 32 0-address instructions

5.3 Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data movement.
- Arithmetic.
- Boolean.
- Bit manipulation.
- I/O.
- Control transfer.
- Special purpose.

5.3 Instruction types

- Data Movement instructions are the most frequently used
  - From memory to register
  - From register to register
  - From register to memory
  - MOVE, MOVER, LOADB LOADDW
- Boolean logic instructions
  - AND, OR, NOT, XOR

5.3 Instruction types

- Bit manipulation
  - Setting and resetting individual bits
  - Arithmetic and logic shift, rotate shift
- Control transfer
  - Branches(conditional or unconditional), skip, procedure calls

5.4 Addressing

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- The actual location of an operand is its *effective address*.
- Certain addressing modes allow us to determine the address of an operand dynamically.

5.4 Addressing

- *Immediate addressing* is where the data is part of the instruction.
- *Direct addressing* is where the address of the data is given in the instruction.
- *Register addressing* is where the data is located in a register.
- *Indirect addressing* gives the address of the address of the data in the instruction.
- *Register indirect addressing* uses a register to store the address of the address of the data.

5.4 Addressing

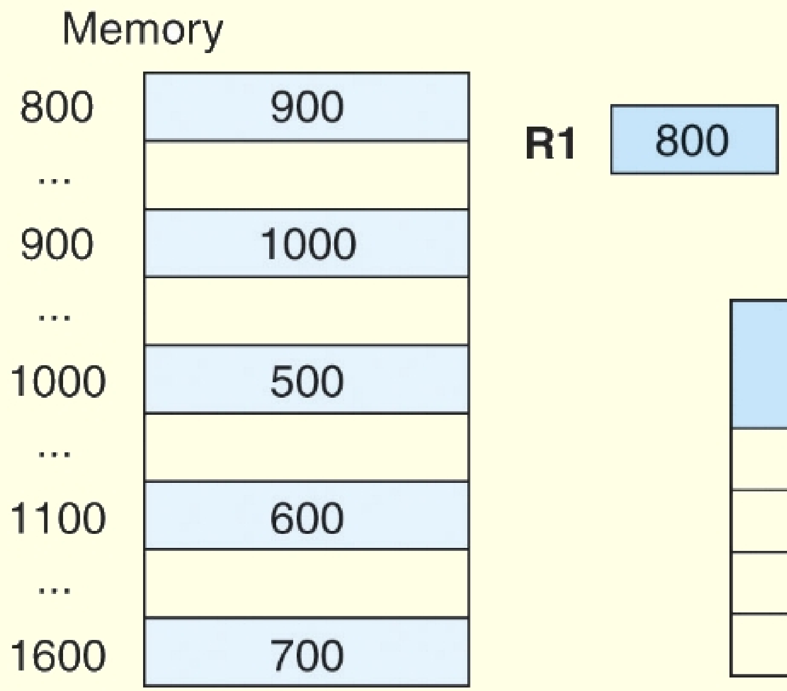
- *Indexed addressing* uses a register (implicitly or explicitly) as an offset, which is added to the address in the operand to determine the effective address of the data.
- *Based addressing* is similar except that a base register is used instead of an index register.
- The difference between these two is that an index register holds an offset relative to the address given in the instruction, a base register holds a base address where the address field represents a displacement from this base.

5.4 Addressing

- In *stack addressing* the operand is assumed to be on top of the stack.
- There are many variations to these addressing modes including:
  - Indirect indexed.
  - Base/offset.
  - Self-relative
  - Auto increment - decrement.

5.4 Addressing

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?

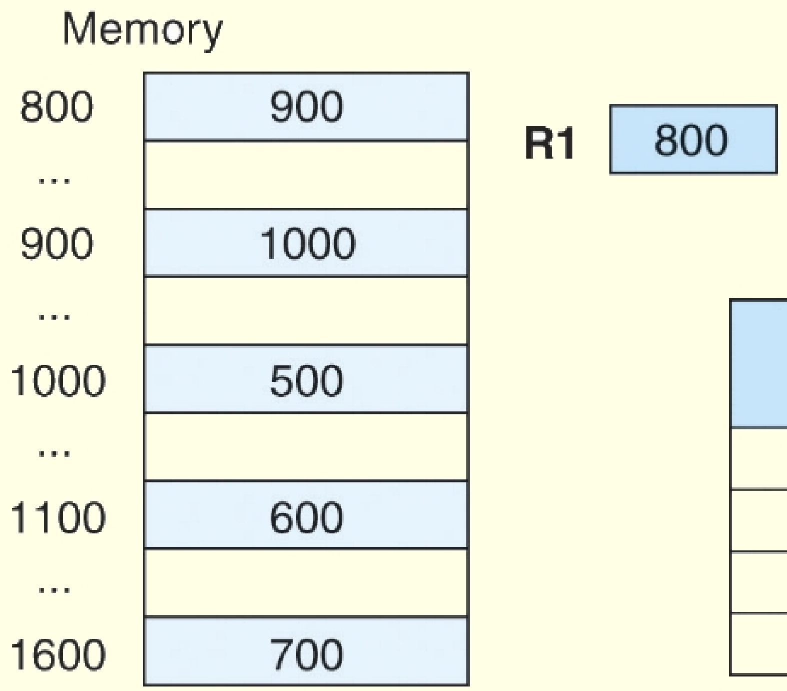


`LOAD 800`

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

5.4 Addressing

- These are the values loaded into the accumulator for each addressing mode.

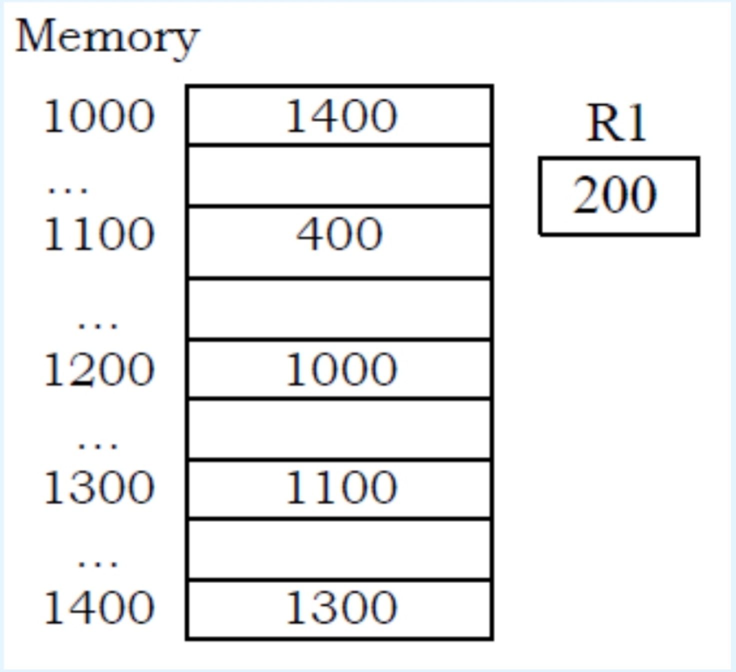


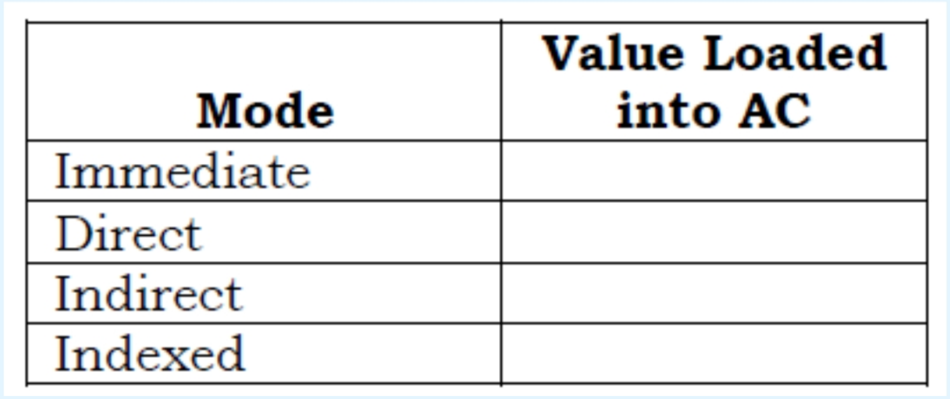
`LOAD 800`

Mode	Value Loaded into AC
Immediate	800
Direct	900
Indirect	1000
Indexed	700

5.4 Addressing

- For the instruction Load1000, what value is loaded into the accumulator for each addressing mode?





Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

5.4 Addressing

- Summary of the addressing mode

Addressing Mode	To Find Operand
Immediate	Operand value present in the instruction
Direct	Effective address of operand in address field
Register	Operand value located in register
Indirect	Address field points to address of the actual operand
Register Indirect	Register contains address of actual operand
Indexed or Based	Effective address of operand generated by adding value in address field to contents of a register
Stack	Operand located on stack

5.5 Instruction-Level Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called *instruction-level pipelining*.
- This term is sometimes abbreviated *ILP* in the literature.

The next slide shows an example of instruction-level pipelining.

5.5 Instruction-Level Pipelining

- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:
  - 1. Fetch instruction.(FI)
  - 2. Decode instruction. (DI)
  - 3. Calculate effective address of operands.(CO)
  - 4. Fetch operands.(FO)
  - 5. Execute instruction.(EI)
  - 6. Store result/Write operand (WO)
- Not all instructions need all these steps
  - LOAD does not write operand

5.5 Instruction-Level Pipelining

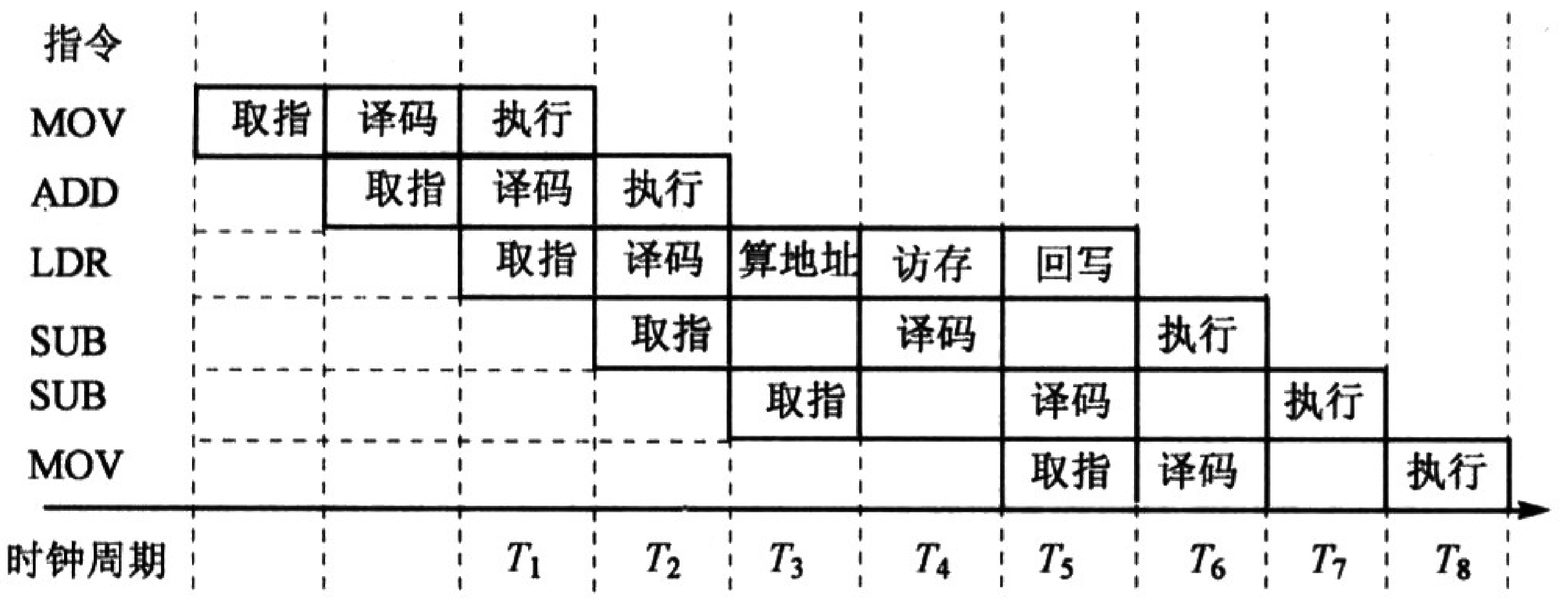


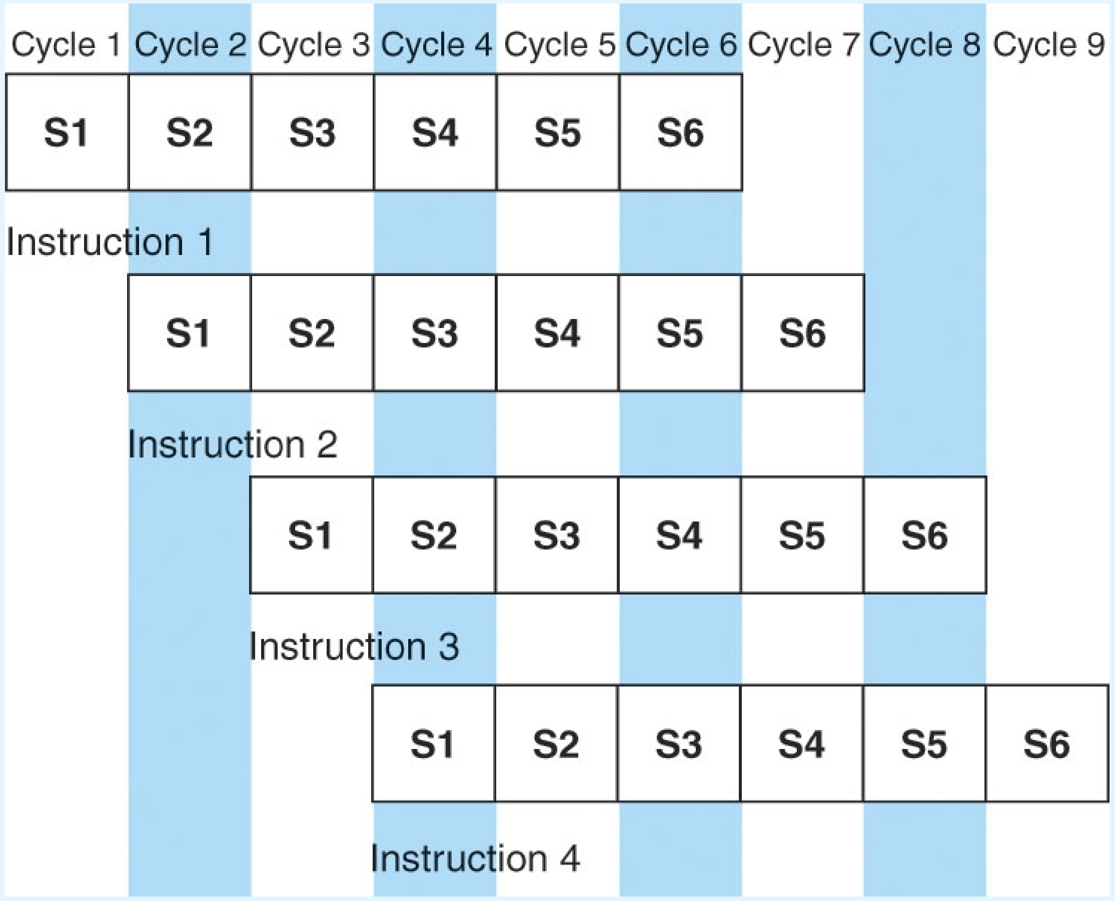
图 2　带有存储器访问指令的流水线

5.5 Instruction-Level Pipelining

- Suppose we have a six-stage pipeline. S1 fetches the instruction, S2 decodes it, S3 determines the address of the operands, S4 fetches them, S5 executes the instruction, and S6 stores the result.

5.5 Instruction-Level Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



5.5 Instruction-Level Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:

  Let t_p be the time per stage. Each instruction represents a task, T, in the pipeline.

  The first task (instruction) requires $k \times t_p$ time to complete in a k-stage pipeline. The remaining $(n - 1)$ tasks emerge from the pipeline one per cycle. So the total time to complete the remaining tasks is $(n - 1)t_p$.

  Thus, to complete n tasks using a k-stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p.$$

5.5 Instruction-Level Pipelining

- If we take the time required to complete *n* tasks without a pipeline and divide it by the time it takes to complete *n* tasks using a pipeline, we find:

$$\text{Speedup } S = \frac{n t_n}{(k + n - 1)\, t_p}$$

- If we take the limit as *n* approaches infinity, (*k* + *n* - 1) approaches *n*, which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

5.5 Instruction-Level Pipelining

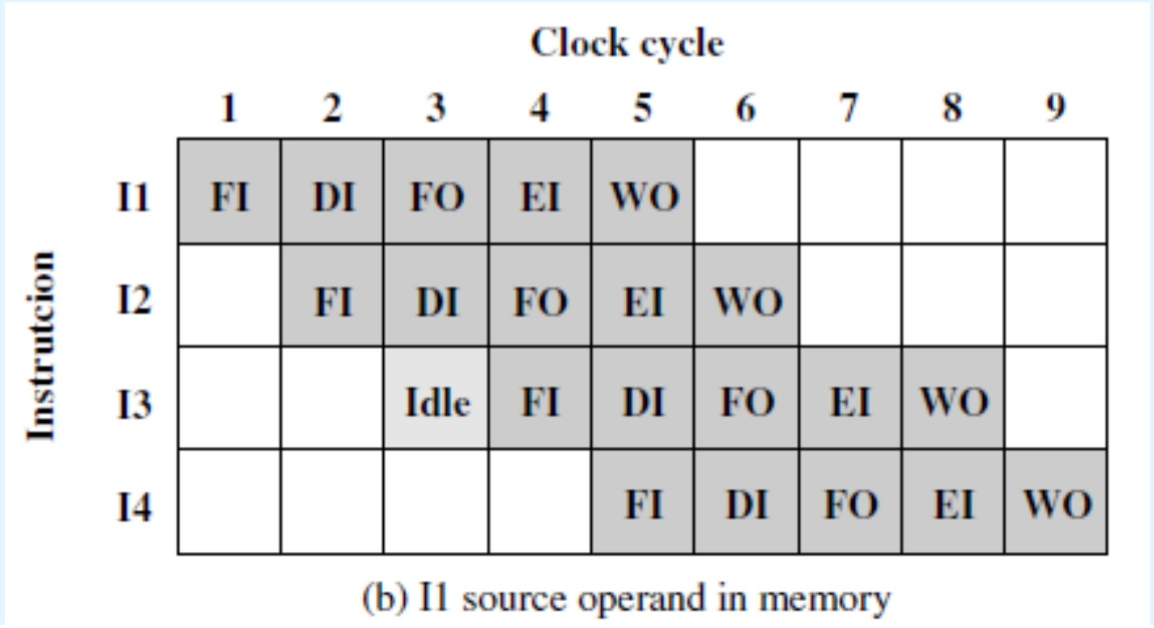
- A nonpipelined system takes 200ns to process a task. The same task can be processed in a 5-segment pipeline with a clock cycle of 40ns. Determine the speedup ratio of the pipeline for 200 tasks. What is the maximum speedup that could be achieved with the pipeline unit over the nonpipelined unit?

5.5 Instruction-Level Pipelining

- Our neat equations take a number of things for granted.
- First, we have to assume that the architecture supports **fetching instructions and data in parallel**.
- Second, we assume that the pipeline can **be kept filled at all times**. This is not always the case. Pipeline *hazards* arise that cause pipeline conflicts and stalls.

5.5 Instruction-Level Pipelining

- An instruction pipeline may stall, or be flushed for any of the following reasons:
  – Resource conflicts.
    •Ex. Storing and reading a memory at the same time by two instructions.
    •Forcing reading to wait
    •Two-way memory



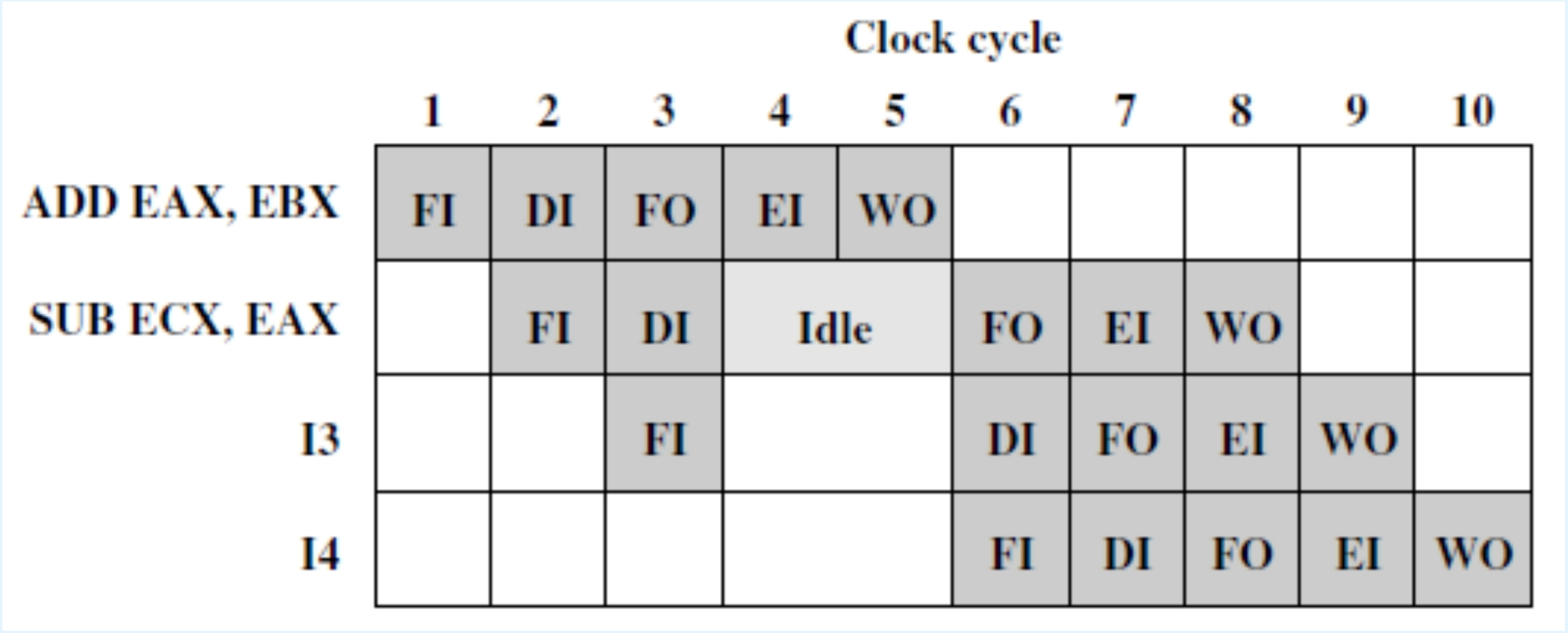
Clock cycle

Instrutcion	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			Idle	FI	DI	FO	EI	WO	
I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

5.5 Instruction-Level Pipelining

– Data dependencies.
- ADD EAX, EBX
- SUB ECX, EAX



Clock cycle

	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

5.5 Instruction-Level Pipelining

– Conditional branching.

- Branch prediction
- Delayed branch

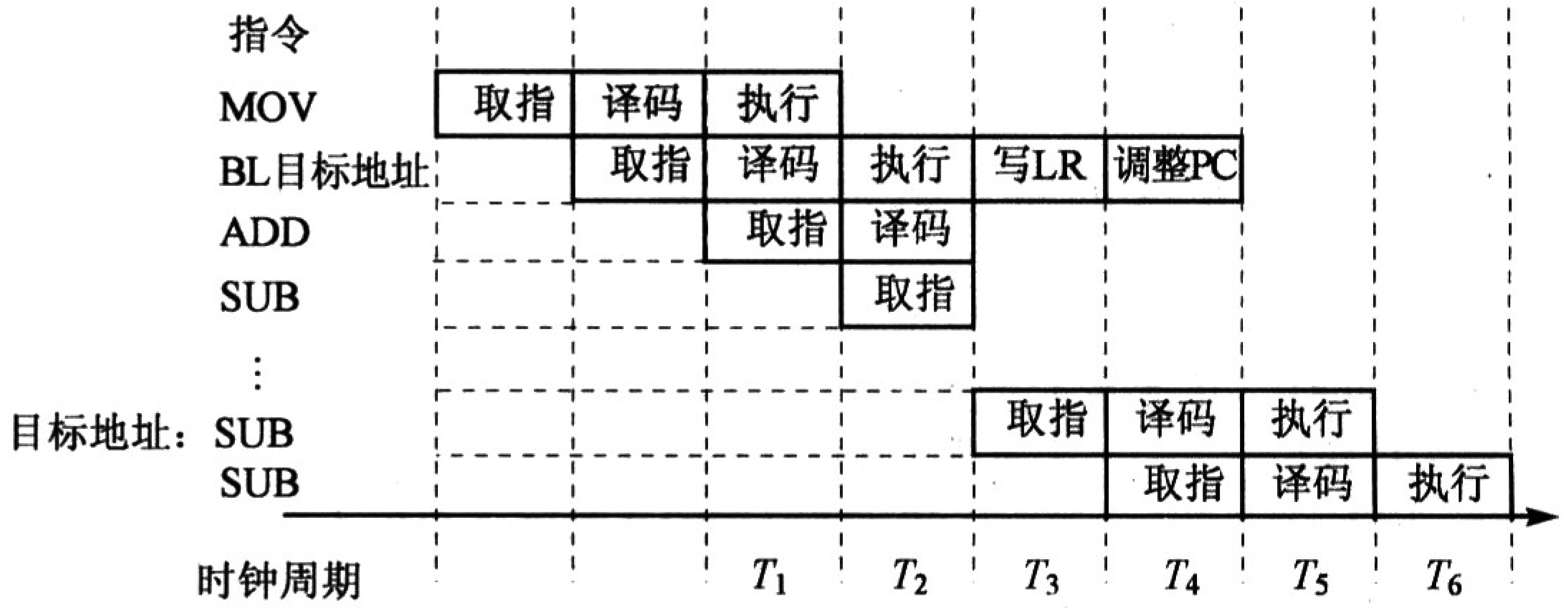


图 3　带有分支指令的流水线

5.5 Instruction-Level Pipelining

- Measures can be taken at the software level as well as at the hardware level to reduce the effects of these hazards, but they cannot be totally eliminated.
- Further reading materials: 指令流水线技术，page283，计算机组成与体系结构-性能设计，

5.6 Real-World Examples of ISAs

- We return briefly to the Intel and MIPS architectures from the last chapter, using some of the ideas introduced in this chapter.
- Little endian
- Two-address architecture
- Variable-length instruction
- Register-memory architecture
- Variable-length of data operation, with length of 1, 2 or 4 bytes

5.6 Real-World Examples of ISAs

- 8086 through 80486 are single-stage pipeline.
- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had **two** five-stage pipelines. Each subsequent Pentium processor had a longer pipeline than its predecessor with the Pentium IV having a 24-stage pipeline.

5.6 Real-World Examples of ISAs

- Intel processors support a wide array of addressing modes.
- The original 8086 provided 17 ways to address memory, most of them variants on the methods presented in this chapter.
- Owing to their need for backward compatibility, the Pentium chips also support these 17 addressing modes.
- The Itanium, having a RISC core, supports only one: register indirect addressing with optional post increment.

5.6 Real-World Examples of ISAs

- MIPS was an acronym for *Microprocessor Without Interlocked Pipeline Stages*.
- The architecture is little endian and word-addressable with three-address, fixed-length instructions.
- Load and Store architecture
- Like Intel, the pipeline size of the MIPS processors has grown: The R2000 and R3000 have five-stage pipelines.; the R4000 and R4400 have 8-stage pipelines.

5.6 Real-World Examples of ISAs

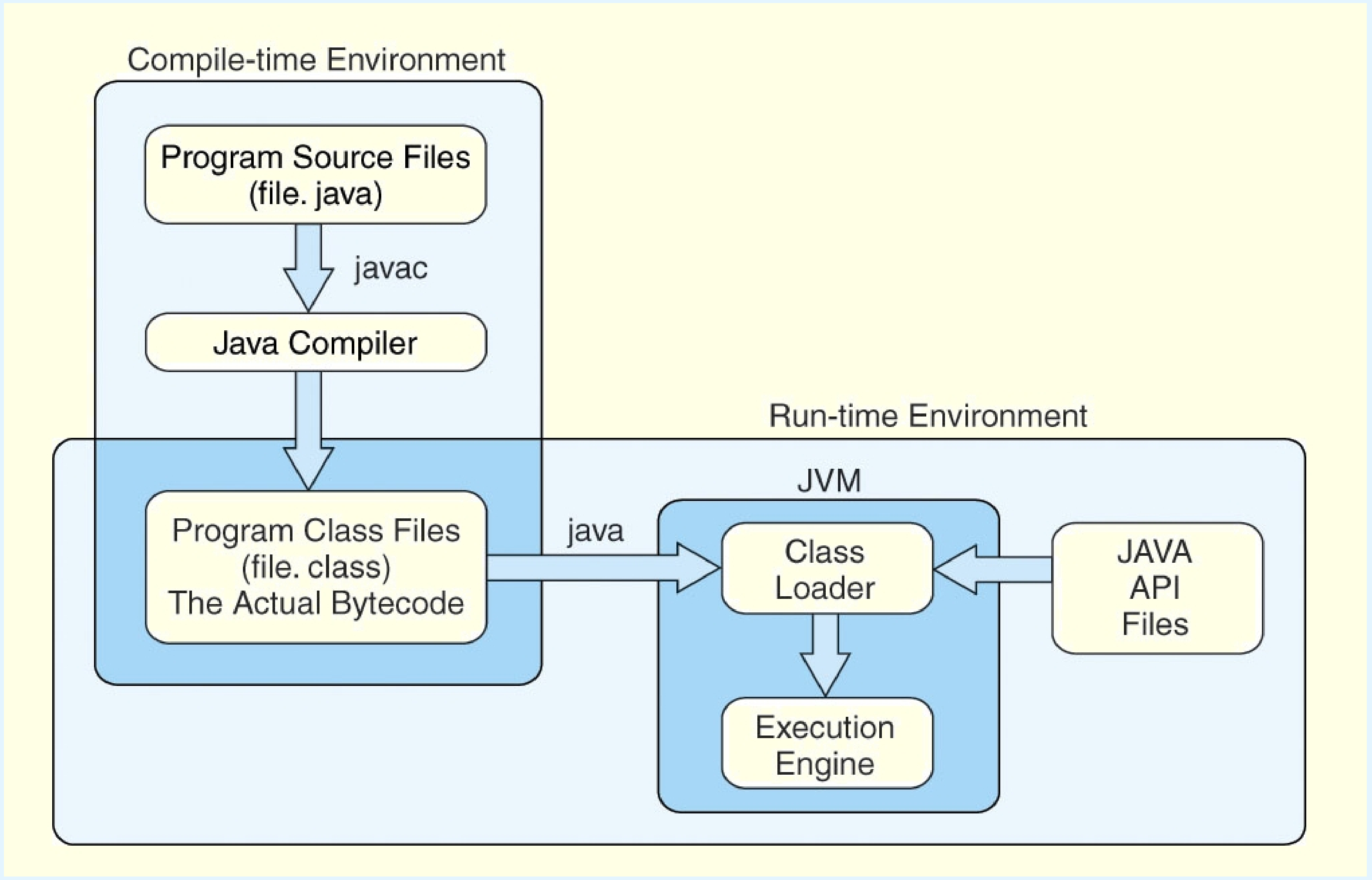
- The R10000 has three pipelines: A five-stage pipeline for integer instructions, a seven-stage pipeline for floating-point instructions, and a six-state pipeline for **`LOAD/STORE`** instructions.
- In all MIPS ISAs, only the **`LOAD`** and **`STORE`** instructions can access memory.
- The ISA uses only base addressing mode.
- The assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.

5.6 Real-World Examples of ISAs

- The Java programming language is an interpreted language that runs in a software machine called the *Java Virtual Machine* (JVM).
- A JVM is written in a native language for a wide array of processors, including MIPS and Intel.
- JVM is platform dependent
- Like a real machine, the JVM has an ISA all of its own, called *bytecode*. This ISA was designed to be compatible with the architecture of any machine on which the JVM is running.

The next slide shows how the pieces fit together.

5.6 Real-World Examples of ISAs



5.6 Real-World Examples of ISAs

- Java bytecode is a stack-based language.
- Most instructions are zero address instructions.
- The JVM has four registers that provide access to five regions of main memory.
- All references to memory are offsets from these registers. Java uses no pointers or absolute memory references.
- Java was designed for **platform interoperability, not performance!**

Chapter 5 Conclusion

- ISAs are distinguished according to their bits per instruction, number of operands per instruction, operand location and types and sizes of operands.
- Endianness as another major architectural consideration.
- CPU can store store data based on
  1. A stack architecture
  2. An accumulator architecture
  3. A general purpose register architecture.

Chapter 5 Conclusion

- Instructions can be fixed length or variable length.
- To enrich the instruction set for a fixed length instruction set, expanding opcodes can be used.
- The addressing mode of an ISA is also another important factor. We looked at:

– Immediate	– Direct
– Register	– Register Indirect
– Indirect	– Indexed
– Based	– Stack

Chapter 5 Conclusion

- A *k*-stage pipeline can theoretically produce execution speedup of *k* as compared to a non-pipelined machine.
- Pipeline hazards such as resource conflicts and conditional branching prevents this speedup from being achieved in practice.
- The Intel, MIPS, and JVM architectures provide good examples of the concepts presented in this chapter.

Chapter 5 Conclusion

- Homework
- P232. 21. Pick an architecture (other than those covered in this chapter). Do research to find out how your architecture approaches the concepts introduced in this chapter, as was done for Intel, MIPS, and Java.