CHAPTER 5

A Closer Look at Instruction Set Architectures

5.1 Introduction 199

- In this chapter, we expand on the topics presented in the last chapter, the objective being to provide you with a more **detailed** look at machine **instruction sets**.
- Employers frequently prefer to hire people with **assembly language** backgrounds not because they need an assembly language programmer, but because they need someone who can understand computer architecture to write **more efficient and more effective programs**.
- We look at different i**nstruction formats** and **operand types**, and how instructios access **data** in memory.
- We will see that the variations in instruction sets are integral in different computer architectures.
- Understanding how instruction sets are designed and how they function can help you understand the more intricate details of the architecture of the machine itself.

5.2 Instruction Formats 199

- MARIE had an instruction length of 16 bits and could have, at most 1 operand.
- Instruction sets are differentiated by the following:
    - Operand **storage** in the CPU (data can be stored in a stack structure or in register)
    - **Number** of explicit operands per instruction (zero, one, two, three being the most common)
    - Operand **location** (instructions can be classified as register-to-register, register-to-memory or memory-to-memory, which simply referto the combination of oeprands allowed per instruction)
    - Types of **operations** (including not only types of operations but also which instructions can access memory and which cannot)
    - **Type and size** of operands (operands can be addresses, numbers, or even characters)

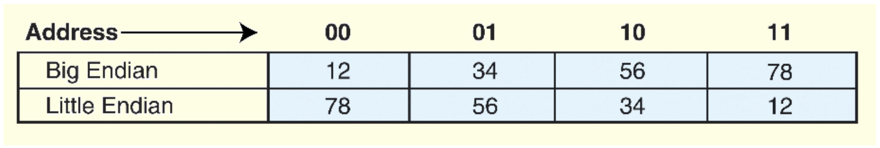
5.2.1 Design Decisions for Instruction Sets 200

- Instruction set architectures are measured according to:
    - The amount of **space** a program requires
    - The **complexity** of the instruction set, in terms of the amount of decoding necessary to execute an instruction, and the complexity of the tasks performed by the instructions
    - The **length** of the instructions
    - The **total number** of instructions
- In designing an instruction set, consideration is given to:
    - **Short** instructions are typically **better** because they take up less space in memory and can be fetched quickly. Howeer, this limits the number of instructions, because there must be **enough bits** in the instruction to specify the number of instructions we need.

- Instructions of a **fixed length** are **easier to decode but waste space**.
- Memory organization affects instruction format. **Byte-addressable** memory means every byte has a unique address even though words are longer then 1 byte.
- A fixed length instruction **does not** necessarily imply a fixed number of operands.
- There are many different types of **addressing modes**.
- If words consist of multiplee bytes, in **what order** should these bytes be stored on a byte-addreable machine? (Little edian versus Big edian)
- **How many registers** should the architecture contain and how should these register be organized?

5.2.2 Little versus Big Endian 201

- The term endian refers to a computer architecture's "**byte order**," or the way the computer stores the bytes of a multiple-byte data element.
- Most **UNIX** machines are big endian, whereas most **PCs** are little endian machines. Most newer **RISC** architectures are also big edian.
- 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).



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

FIGURE 5.1 The Hex Value 12345678 Stored in Both Big and Little Endian Format

- Big endian:
    - Is more natural.
    - The sign of the number can be determined by looking at the byte at address **offset 0**.
    - Strings and integers are stored in the same order.
- Little endian:
    - Makes it easier to place values on **non-word** boundaries.
    - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.
- **Computer networks** are big endian, which means that when little endian computers are going to pass integers over the network, they need to convert them to network byte order.
- Any program that writes data to or reads data from a file must be aware of the byte ordering on the particular machine.
    - Windows BMP graphics: little endian
    - Adobe Photoshop: big endian

- GIF: little endian
- JPEG: big endian
- MacPaint: Big endian
- PC Paintbrush: little endian
- RTF by Microsoft: little endian
- Sun raster files: big endian
- MS WAV, AVI, TIFF, XWD (X windows Dup): support both, typically by encoding an **identifier** into the file

5.2.3 Internal Storage in the CPU: Stacks versus Registers 203

- 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 (**GPR**) architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.
- In a **stack** architecture, instructions and operands are implicitly taken from the stack.
  - A stack cannot be accessed randomly, which makes it difficult to generate efficient code.
- In an **accumulator** architecture such as MARIE, one operand is **implicitly** in the accumulator, minimize the internal complexity of the machine and 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, causing longer fetch and decode times.
- Most systems **today** are GPR systems.
- The general-purpose architecture can be broken into three classifications, depending on where the operands are located:
  - **Memory-memory** where two or three operands may be in memory.
  - **Register-memor**y where at least one operand must be in a register.
  - **Load-store** where no operands may be in memory.
- **Intel** and **Motolrola** are examples of register-memory architectures
- Digital Equipment's **VAX** architecture allows memory-memory operations.
- **SPARC, MIPS, ALPHA, and PowerPC** are all load-store machines.

5.2.4 Number of Operands and Instruction Length 204

- The **number of operands** and the number of available registers has a direct affect on instruction length.

- The traditional method for describing a computer architecture is to specify the maximum number of operands, or addresses, contained in each instruction.
- MARIE uses a fixed length instruction with a 4-bit opcode and a 12-bit operand.
- Instructions can be formatted in two ways:
    - Fixed length: **Wastes space** but is **fast** and results in better performance when instruction-level pipelining is used.
    - Variable length: More **complex** to decode but **saves** storage space.
- The most **common** instruction formats include zero, one, two, or three operands.
- Arithmetic and logic operations typically have **two** operands, but can be executed with one operand (as we saw in MARIE), if the accumulator is implicit.
- In MARIE, the maximum was **one**, although some instructions had no operands (Halt and Skipcond).
- Machine instructions that have no operands must use a stack.
- Stack machines use one - and zero-operand instructions.
- In architectures based on stacks, most instructions consist of opecode only.
- Stack architecture need a push instruction and a pop instruction, each of which is allowed **one** operand (Push X, and Pop X). PUSH and POP operations involve only the stack's top element.
- LOAD and STORE 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.
- 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.
- The principal advantage of postfix notation is that **parentheses** are **not used**.
- EXAMPLE 5.1 Suppose we wish to evaluate the following expression (Page 206):
  Z = (X * Y) + (W * U)
    - Typically, when **three** operands are allowed one operand must be a register and the first operand is normally the destination. The infix expression:
      Z = (X * Y) + (W * U)

      might look like this:

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

    - In a **two**-address ISA, normally one address specifies a register. The other operand could be either a register or a memory address. The infix expression:
      Z = (X * Y) + (W * U)

      might look like this:

```
LOAD  R1,  X
MULT  R1,  Y
LOAD  R2,  W
MULT  R2,  U
```

```
ADD    R1,  R2
STORE  Z,   R1
```

- In a **one**-address ISA (like **MARIE**), we must assume a register (normally the **accumulator**) is implied as the destination. The infix expression,

  Z = (X * Y) + (W * U)

  looks like this:

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

- In a **stack** ISA, the infix expression,

  Z = (X * Y) + (W * U),

  becomes in postfix notation.

  Z = X Y * W U * +

  might look like this:

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

- 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.

5.2.5 Expanding Opcodes 208

- Expanding opcodes represent a **compromise** between the need for a rich set opcode and desire to have short opcode.
- One way to recover some of this space is to use expanding opcodes.
- 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:

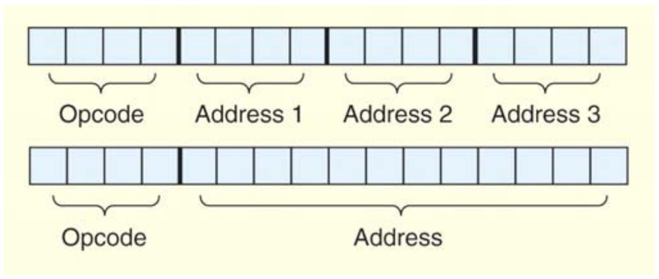
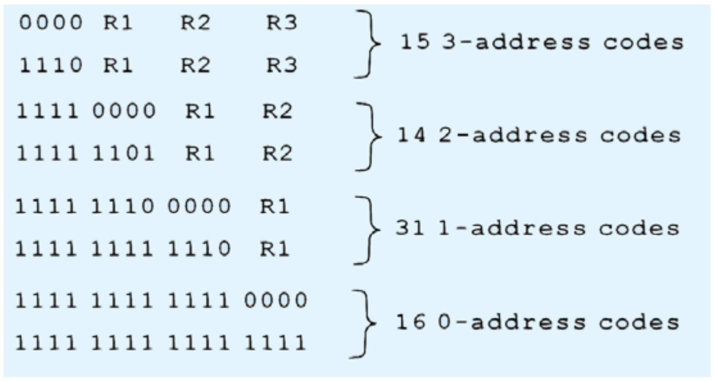


FIGURE 5.2 Two Possibilities for a 16-Bit Instruction Format

- If we allow the length of the opcode to vary, we could create a very rich instruction set.
- Suppose we wish to encode the following instructions:
    - o 15 instructions with 3 addresses
    - o 14 instructions with 2 addresses
    - o 31 instructions with 1 address
    - o 16 instructions with 0 addresses
- Can we encode this instruction set in 16 bits? The answer is yes, as long as we use expanding opcodes. The encoding is as follows:



```
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 }
```

5.3 Instruction Types 210

- Instructions fall into several broad categories that you should be familiar with:
  - Data movement
  - Arithmetic
  - Boolean
  - Bit manipulation (shift and rotate)
  - I/O
  - Control transfer (ex. branch, skip, procedure call)
  - Special purpose (ex. String processing, protection, flag control, cache management)

5.4 Addressing 211

- We now present the two most important of these addressing issues:
    - Types of data that can be addressed and
    - The various addressing modes.

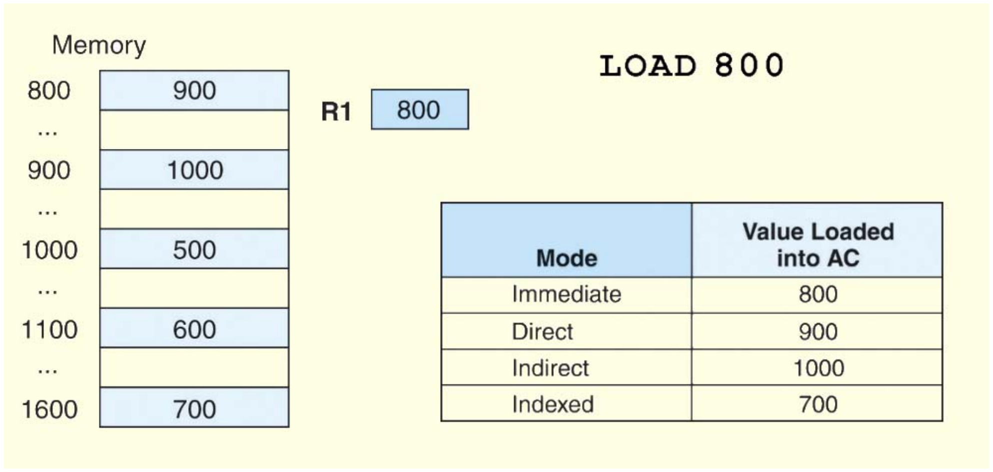
5.4.1 Data Types 211

- Numeric data consists of integers and floating point values.
- Nonnumeric data types consist of strings, Booleans, and pointers.
    - String instructions typically include operations such as copy, move, search, or modify.
    - Boolean operations include AND, OR, XOR, and NOT.
    - Pointers are actually addresses in memory.

5.4.2 Address Modes 212

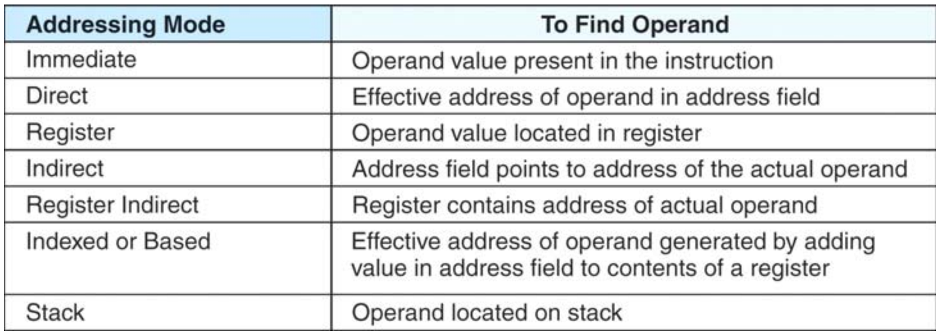
- Addressing modes allow us to specify where the instruction **operands are located**.
- An addressing mode can specify a **constant**, a **register**, or a **memory location**.
- The actual location of an operand is its **effective address**.
- **Immediate addressing** is where the **data** to be operated on is part of the instruction.
- **Direct addressing** is where the **address** of the data is directly 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.
- **Indexed addressing** uses a register (implicitly or explicitly) as an **offset (or displacement)**, 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.
- 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 addressing: use both indirect and indexed addressing at same time
    - Base/offset addressing: add an offset to a **specific base registe**r and then add this to the specified operand, resulting in the effective address of the actual operand to be use in the instruction
    - Auto-increment and auto-decrement mode: automatically increment or decrement the register used, thus reducing the code size, which can be extremely important in applications such as **embedded systems**
    - Self-relative addressing: compute the address of the operand as an **offset from the current instruction**

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



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

FIGURE 5.3 Contents of Memory When Load 800 Is Executed



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

TABLE 5.2 A Summary of the Basic Addressing Mode

5.5 Instruction-Level Pipelining 214

- Some CPUs divide the fetch-decode-execute cycle into smaller steps, where some of these smaller steps can often be executed in **parallel** to increase throughput.
- This **overlapping** speed up execution. The method, used by all current CPUs, is known as **pipelining**. Such parallel execution is called instruction-level pipelining (ILP).
- Suppose a fetch-decode-execute cycle were broken into the following smaller steps:
  1. Fetch instruction
  2. Decode opcode
  3. Calculate effective address of operands
  4. Fetch operands
  5. Execute instruction
  6. Store result

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

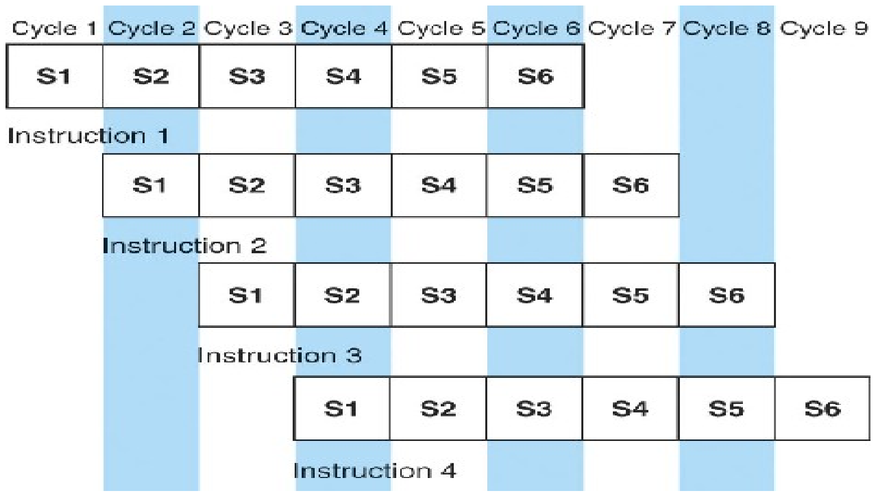


FIGURE 5.4 Four Instructions Going through a 6-stage Pipeline

- The theoretical speedup offered by a pipeline can be determined as follows:
  - Assume the clock cycle time is *tp*, it be the time per stage. Each instruction represents a task, *T*, in the pipeline.
  - The first task (instruction) requires ***k* × *tp*** 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)*tp***.
  - Thus, to complete *n* tasks using a *k*-stage pipeline requires:
    $(k \times tp) + (n - 1)tp = (k + n - 1)tp$.
  - Without a pipeline, the time required is nt_n cycles, where $\mathbf{t_n = k \times tp}$

- 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:

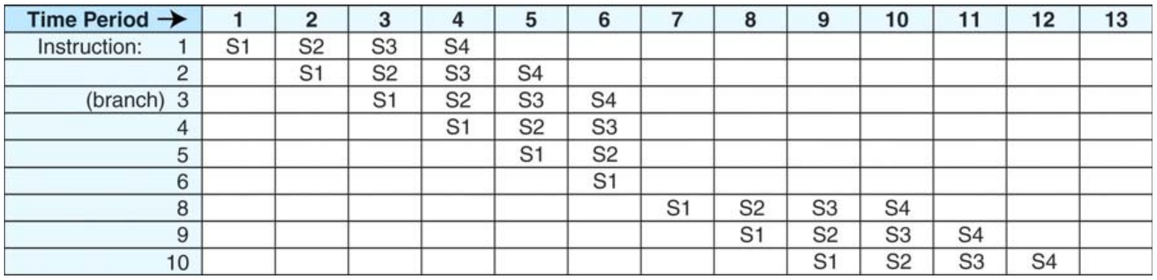
$$\text{Speedup } S = \frac{nt_n}{(k + n - 1)\, t_p}$$

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

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

- The more stages that exist in the pipeline, the faster everything will run.

- Suppose we have a 4-stage pipeline:
    - S1 = fetch instruction
    - S2 = decode and calculate effective address
    - S3 = fetch operand
    - S4 = execute instruction and store results
- Pipeline hazards arise that cause **pipeline conflicts and stalls**.
- An instruction pipeline may stall, or be flushed for any of the following reasons:
    - Resource conflicts
    - Data dependencies.
    - Conditional branching.
- 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.



Time Period →	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
8							S1	S2	S3	S4			
9								S1	S2	S3	S4		
10									S1	S2	S3	S4	

FIGURE 5.5 Example Instruction Pipeline with Conditional Branch (4 stages, I3 a branch instruction from I3 to I8)

5.6 Real-World Examples of ISAs 219

- We return briefly to the Intel and MIPS architectures from the last chapter, using some of the ideas introduced in this chapter.

5.6.1 Intel 220

- Intel uses a **little** endian, two-address architecture, with **variable-length** instructions.
- Intel introduced pipelining to their processor line with its Pentium chip.
- The first Pentium had two parallel five-stage pipelines, called U pipe and the V pipe, to execute instruction. Stages for these pipelines include Prefetch, Instruction Decode, Address Generation, Execute, and Write Back.
- The Pentium II increased the number of stages to 12. The Pentium III increased the stages to 14, and the Pentium IV to 24.
- The Itanium (IA-64) has only a 10-stage pipeline.
- 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 more complex addressing modes require specialized hardware.

5.6.2 MIPS 220

- MIPS architecure (which originallly stood for "*Microprocessor Without Interlocked Pipeline Stages*"*)* is **little** endian, **word**-addressable, **three**-address, **fixed**-length ISA.
- 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.
- 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 assembler accommodates programmers who need to use immediate, register, direct, indirect register, base, or indexed addressing modes.
- Essentially **three** instruction formats are available: the **I** type (immediate), the **R** type (register), and the **J** type (jump).

5.6.3 Java Virtual Machine 221

- 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.
- 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.

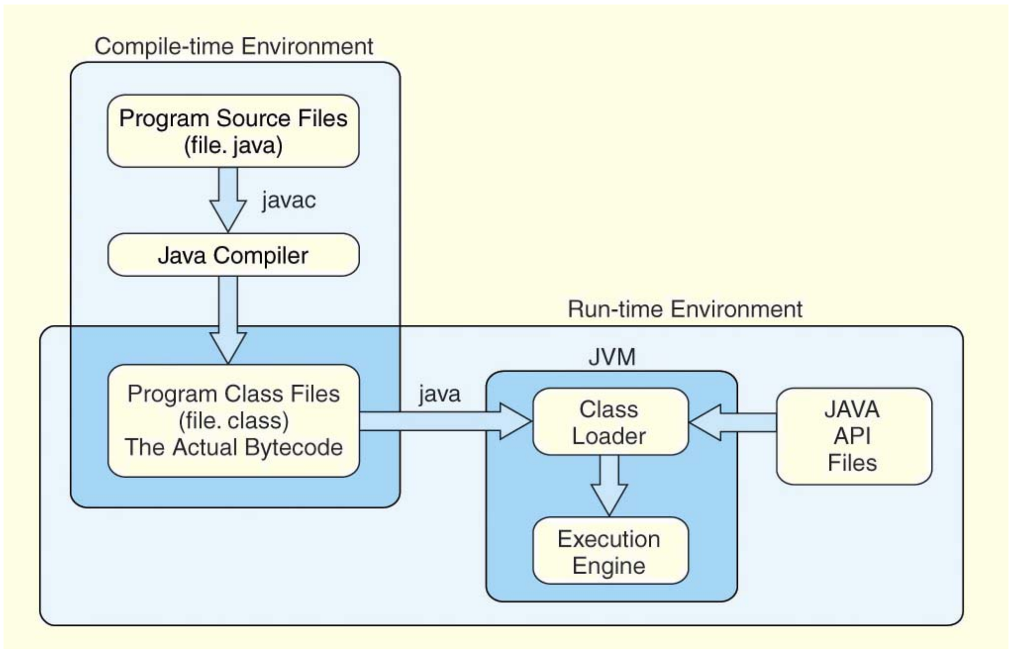
- 

FIGURE 5.6 The Java Programming Environment

- 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: pointers or absolute memory addresses are **never** used.
- Because the JVM is a stack machine, no general registers are provided.
- The lack of general registers is detrimental to performance, as **more memory** references are generated. We are trading **performance** for **portability**.
- EXAMPLE 5.2 (Page 223)
    - After we compile this program (using javac), we can disassemble it to examine the bytecode, by issuing the following command:
      javap –c ClassName

Chapter Summary 225

- 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.
- 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

- A *k*-stage pipeline can theoretically produce execution speedup of ***k*** as compared to a non-pipelined machine.
- **Pipeline hazards** such as resource conflicts, data dependencies, 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.