CHAPTER 5
A Closer Look at Instruction Set Architectures

5.1 Introduction 299
5.2 Instruction Formats 299
  5.2.1 Design Decisions for Instruction Sets 300
  5.2.2 Little Versus Big Endian 301
  5.2.3 Internal Storage in the CPU: Stacks Versus Registers 304
  5.2.4 Number of Operands and Instruction Length 305
  5.2.5 Expanding Opcodes 311
5.3 Instruction Types 316
  5.3.1 Data Movement 316
  5.3.2 Arithmetic Operations 316
  5.3.3 Boolean Logic Instructions 317
  5.3.4 Bit Manipulation Instructions 317
  5.3.5 Input/Output Instructions 317
  5.3.6 Instructions for Transfer of Control 318
  5.3.7 Special-Purpose Instructions 318
  5.3.8 Instruction Set Orthogonality 318
5.4 Addressing 319
  5.4.1 Data Types 319
  5.4.2 Address Modes 319
5.5 Instruction Pipelining 322
5.6 Real-World Examples of ISAs 328
  5.6.1 Intel 328
  5.6.2 MIPS 329
  5.6.3 Java Virtual Machine 333
  5.6.4 ARM 337
Chapter Summary 340
5.1 Introduction 299

- 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 instruction formats and operand types, and how instructions 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 299

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

5.2.1 Design Decisions for Instruction Sets 300

- Instruction set architectures are measured according to:
  o The amount of space a program requires
  o 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
  o The length of the instructions
  o The total number of instructions
- In designing an instruction set, consideration is given to:
  o Short instructions are typically better because they take up less space in memory and can be fetched quickly. However, this limits the number of instructions, because there must be enough bits in the instruction to specify the number of instructions we need.
  o Instructions of a fixed length are easier to decode but waste space.
o Memory organization affects instruction format. **Byte-addressable** memory means every byte has a unique address even though words are longer than 1 byte.

o A fixed length instruction **does not** necessarily imply a fixed number of operands.

o There are many different types of **addressing modes**.

o If words consist of multiple bytes, in **what order** should these bytes be stored on a byte-addressable machine? (Little Endian versus Big Endian)

o **How many registers** should the architecture contain and how should these registers be organized?

### 5.2.2 Little Versus Big Endian 301

- 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 endian.

- 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.
  
  o In little endian machines, the least significant byte is followed by the most significant byte.

  o Big endian machines store the most significant byte first (at the lower address).

![Figure 5.1 The Hex Value 12345678 Stored in Both Big and Little Endian Format](image)

- **Example 5.1**: Assume that a computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as here.
• Big endian:
  o Is more natural.
  o The sign of the number can be determined by looking at the byte at address offset 0.
  o Strings and integers are stored in the same order.

• Little endian:
  o Makes it easier to place values on non-word boundaries.
  o 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.
  o Windows BMP graphics: little endian
  o Adobe Photoshop: big endian
  o GIF: little endian
  o JPEG: big endian
  o MacPaint: Big endian
  o PC Paintbrush: little endian
  o RTF by Microsoft: little endian
  o Sun raster files: big endian
  o Microsoft 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 304

- 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-memory where at least one operand must be in a register.
  - Load-store where no operands may be in memory (require data to be moved into registers before any operations on those data are performed).
- Digital Equipment’s VAX architecture allows memory-memory operations.
- Intel and Motorola are examples of register-memory architectures
- SPARC, MIPS, ALPHA, and PowerPC are all load-store machines.
5.2.4 Number of Operands and Instruction Length 305

- 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 opcodes 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.
- For example, the infix expression,
  \[ Z = (X \times Y) + (W \times U) \]
becomes:
  \[ Z = X Y \times W U \times + \]
in postfix notation.
- **EXAMPLE 5.3** Consider the following infix expression \((2 + 3) - 6 / 3\). We convert this to postfix as follows:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 + - 6 / 3</td>
<td>The sum 2 + 3 is in parentheses and takes precedence; we replace it with 2 3 +.</td>
</tr>
<tr>
<td>2 3 + - 6 3 /</td>
<td>The division operator takes precedence, so we replace 6/3 with 6 3 /.</td>
</tr>
<tr>
<td>2 3 + 6 3 / -</td>
<td>We wish to subtract the quotient of 6/3 from the sum of 2 + 3, so we move the − operator to the end.</td>
</tr>
</tbody>
</table>

  o Therefore, the postfix expression **2 3 + 6 3 / −** is equivalent to the infix expression \((2 + 3) - 6 / 3\).

- **EXAMPLE 5.4** Consider the RPN expression **10 2 3 + /**. Using a stack to evaluate the expression and scanning left to right, we would first push 10 onto the stack, followed by 2, and then 3, to get:

  o The “+” operator is next, which pops 3 and 2 from the stack, performs the operation \((2 + 3)\), and pushes 5 onto the stack, resulting in:

  ![Stack top](image)

  o The “/” operator then causes 5 and 10 to be popped from the stack; 10 is divided by 5, and then the result **2** is pushed onto the stack. (Note: For noncommutative operations such as subtraction and division, the top stack element is always the second operand.)

- **EXAMPLE 5.7** Suppose we wish to evaluate the following expression:
  
  \[ Z = (X \times Y) + (W \times U) \]

  o Typically, when **three** operands are allowed one operand must be a register and the first operand is normally the destination. The infix expression:
  
  \[ Z = (X \times Y) + (W \times U) \]

  might look like this:

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

\[ Z = (X \times Y) + (W \times U) \]

might look like this:

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

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

\[ Z = (X \times Y) + (W \times U) \]

looks like this:

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

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

\[ Z = (X \times Y) + (W \times U), \]

becomes in postfix notation.

\[ Z = X Y * W U * + \]

might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP 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 311

- 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:

![Diagram](image)

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.
- EXAMPLE 5.8 Suppose we wish to encode the following instructions:
  - 15 instructions with 3 addresses
  - 14 instructions with 2 addresses
  - 31 instructions with 1 address
  - 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 three-address codes
...
1110 R1 R2 R3
1111 - escape opcode
1111 0000 R1 R2  \{ 14 two-address codes
...
1111 1101 R1 R2
1111 1110 - escape opcode
1111 1110 0000 R1  \{ 31 one-address codes
...
1111 1111 1110 R1
1111 1111 1111 - escape opcode
1111 1111 1111 0000
...
1111 1111 1111 1111  \{ 16 zero-address codes
```
EXAMPLE 5.11 Given 8-bit instructions, is it possible to allow the following to be encoded?

- 3 instructions with two 3-bit operands.
- 2 instructions with one 4-bit operand.
- 4 instructions with one 3-bit operand.

First, we must determine if the encoding is possible.

- $3 \times 2^3 \times 2^3 = 192$ bit patterns for the 3-bit operands
- $2 \times 2^4 = 32$ bit patterns for the 4-bit operands
- $4 \times 2^3 = 32$ bit patterns for the 3-bit operands.

If we sum the required number of bit patterns, we get $192 + 32 + 32 = 256$. 8 bits in the instruction means a total of $2^8 = 256$ bit patterns, so we have an exact match (which means the encoding is possible, but every bit pattern will be used in creating it.)

The encoding we can use is as follows:

```
00  xxx  xxx  }  3 instructions with two 3-bit operands
01  xxx  xxx
10  xxx  xxx
11 - escape opcode
1100  xxxx  }  2 instructions with one 4-bit operand
1101  xxxx
1110 - escape opcode
1111 - escape opcode
11100  xxx
11101  xxx
11110  xxx  }  4 instructions with one 3-bit operand
11111  xxx
```
5.3 Instruction Types 316

- Instructions fall into several broad categories that you should be familiar with:
  - Data movement (ex. move, load, store, push, pop, exchange)
  - Arithmetic (ex. add, subtract, multiply, divide)
  - Boolean (ex. and, or, not, xor)
  - Bit manipulation (ex. shift and rotate)
  - I/O
  - Control transfer (ex. branches, skips, procedure calls, returns, and program termination)
  - Special purpose (ex. string processing, protection, flag control, cache management)
5.4 Addressing 319

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

5.4.1 Data Types 319

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

5.4.2 Address Modes 319

• 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.
• 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 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:
  o Indirect indexed addressing: use both indirect and indexed addressing at same time
  o Base/offset addressing: add an offset to a specific base register and then add this to the specified operand, resulting in the effective address of the actual operand to be use in the instruction
  o 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
  o 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.

![Diagram showing memory contents and addressing modes](image)

**FIGURE 5.4** Contents of Memory When **Load 800** Is Executed

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

**TABLE 5.2** A Summary of the Basic Addressing Mode
5.5 Instruction Pipelining 322

- 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.

- 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 \( k \times tp \) 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 \( t_n = k \times tp \).
o 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:

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

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

\[
\text{Speedup } S = \frac{k t_p}{t_p} = k
\]

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

- **EXAMPLE 5.12** 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.

<table>
<thead>
<tr>
<th>Time Period</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction:</td>
<td>1</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(branch)</td>
<td>3</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 5.6** Example Instruction Pipeline with Conditional Branch (4 stages, I3 a branch instruction from I3 to I8)
5.6 Real-World Examples of ISAs 328

- 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 328

- 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 329

- MIPS architecture (which originally 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 333

- 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.

![Java Programming Environment Diagram](image)

**FIGURE 5.10 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 5.13**
  - After we compile this program (using javac), we can **disassemble** it to examine the bytecode, by issuing the following command:
    
    `javap –c ClassName`
5.6.4 ARM 337

- ARM is a family of RISC-like (Reduced Instruction Set Computer) processor cores found in many portable devices today. It is the most widely used 32-bit instruction architecture:
  - 95%+ of smartphones,
  - 80%+ of digital cameras
  - 40%+ of all digital television sets
- Founded in 1990, ARM (Advanced RISC Machine) was originally funded by Apple, Acorn, and VLSI and is now licensed by ARM Holdings in British.
- ARM Holdings does not manufacture these processors; it sells licenses to manufacture.
- ARM is a load/store architecture: all data processing must be performed on values in registers, not in memory.
- It uses fixed-length, three-operand instructions and simple addressing modes
- ARM processors have a minimum of a three-stage pipeline (consisting of fetch, decode, and execute). Newer ARM processors have deeper pipelines (more stages). For example, the very popular ARM9 typically has a five-stage pipeline (similar to MIPS); certain implementations of the ARM8 have 13-stage integer pipelines
- ARM has 37 total registers, but their visibility depends on the processor mode.
- It can simultaneously load or store any subset of the 16 general-purpose registers from/to sequential memory addresses.
- Control flow instructions include unconditional and conditional branching and procedure calls
- Most ARM instructions execute in a single cycle, provided there are no pipeline hazards or memory accesses.
Chapter Summary 340

- 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 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, JVM, and ARM architectures provide good examples of the concepts presented in this chapter.