5.1 Introduction

Instruction sets are differentiated by the following:

- Number of bits per instruction.
- Stack-based or register-based.
- Number of explicit operands per instruction.
- Operand location.
- Types of operations.
- Type and size of operands.

5.2.1 Design Decisions for Instructions Sets

Instruction set architectures are measured according to:

- Main memory space occupied by a program.
- Instruction complexity.
- Instruction length (in bits).
- Total number of instructions in the instruction set.

- **Instruction length:** 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. Shorter instructions also have tighter limits on the size and number of operands.

  Comments:
  - The goal is to go fast.
  - When trading space for speed, the rule is you should almost always favor speed over space.
  - How many instructions are needed? A NAND gate is functionally complete. MARIE has less than 16 instructions. What programs cannot be implemented on MARIE?

- **Fixed versus varying length instructions:** Instructions of a fixed length are easier to decode but waste space.

  Comments:
  - What space are we discussing? Are we discussing the space in memory or are re we discussing the space on the integrated circuit to implement the more complicated logic required to decode variable length instructions?

- **Memory organization** affects instruction format. If memory has, for example, 36- (Univac 1100) or 64-bit (CDC 6600, Cray) words and is not byte addressable, it is difficult to access a single character.

  Comments:
  - This difference and floating-point instructions are the two differences that distinguish a business data processing architecture from a scientific data processing architecture.

- **Number of operands:** A fixed-length instruction does not necessarily imply a fixed number of operands.

- **Addressing modes:** MARIE used two addressing modes: direct and indirect; but there are many more.

- **Byte ordering:** Should the least significant byte be stored at the highest or lowest byte address? Intel elected little endian where the least significant byte has the smallest address. IBM elected the reverse, the least significant byte has the largest address.

- **Register organization:** How many registers should the architecture contain? How should these registers be organized? How should operands be stored in the CPU?

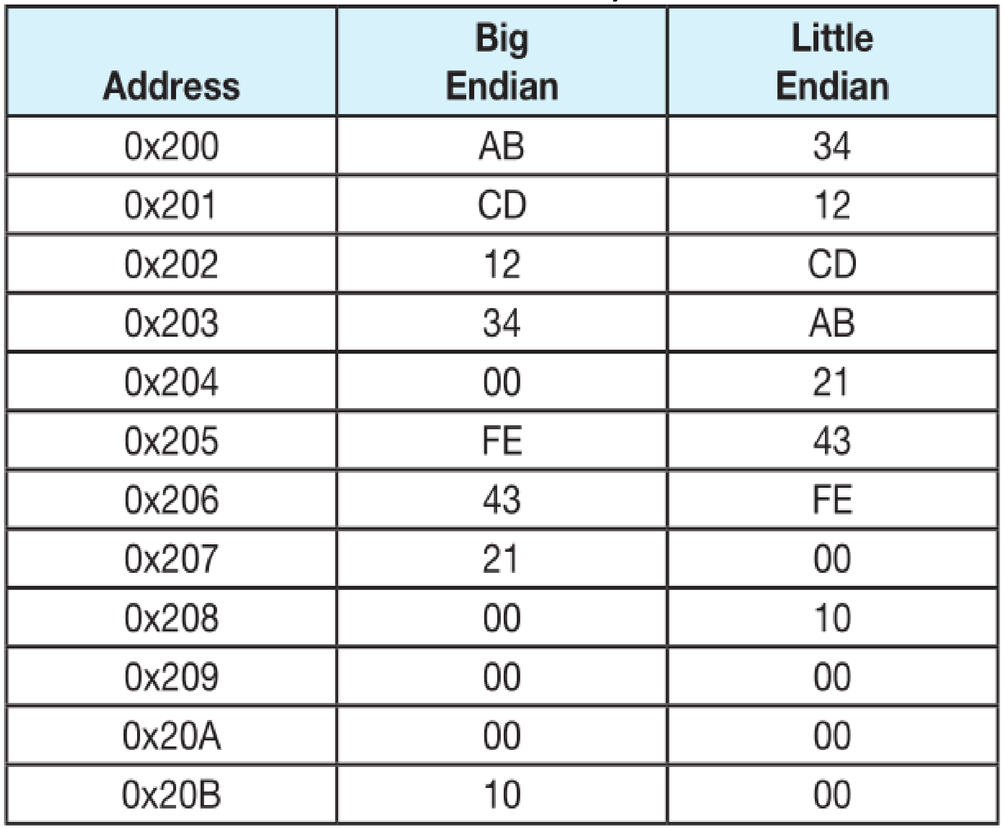
5.2.2 Little Versus Big Endian

- As an example, suppose we have the hexadecimal number 0x12345678.
- 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

Figure 5.1 The Hex Value 12345678 Stored in Both Big and Little Endian Formats

- A larger example: A computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.



	Byte Order	
Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

Example 5.2

- Bytes need to be reordered when transferring data from a big endian machine to a little endian machine.

5.2.3 Internal Storage in the CPU: Stacks Versus Registers

- 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.

Stack Architectures:

- Operation:
  - In a stack architecture, instructions and operands are implicitly taken from the stack.
  - A stack cannot be accessed randomly.

- Advantages:
  - Compilers are more easily implemented on a machine that supports a stack architecture.

- Disadvantages:
  - Because a stack cannot be accessed randomly, partially complete computations cannot be accessed, thus, disabling one of the fundamental characteristic that enables optimization.

Single Accumulator Architectures:

- Operation:
  - In a single accumulator architecture, every computation is performed by the single accumulator.
  - For binary operations, one operand is in the accumulator and the other is in memory. The result is stored in the accumulator.
  - MARIE is an example of a single accumulator architecture.

- Advantages:
  - Simple and easy to implement.

- Disadvantages:
  - When compared to other architectures, a single accumulator architecture is slow.
  - Contributions to the inefficiency of the architecture is the frequent use of the bus.

General Purpose Register (GPR) Architectures:

- Operation:
  - Each register can be used as a single accumulator.
  - Instructions can have both register and memory operands.

- Advantages:
  - Enables optimization in compilers resulting in faster execution.

- Disadvantages:
  - Longer instructions.
  - Complex implementation.

5.2.4 Number of Operands and Instruction Length

- Most systems today are GPR systems.
- There are three types:
  - 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.
- The number of operands and the number of available registers has a direct affect on instruction length.

- Stack machines use one - and zero-operand instructions.
- LOAD and STORE instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- PUSH and POP operations involve only the stack's top element.
- Binary instructions (e.g., ADD, MULT) use the top two items on the stack.

Common instruction formats:

- **OPCODE only** (zero addresses)
- **OPCODE + 1 Address** (usually a memory address)
- **OPCODE + 2 Addresses** (usually registers, or one register and one memory address)
- **OPCODE + 3 Addresses** (usually registers, or combinations of registers and memory)

Stack Architectures:

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

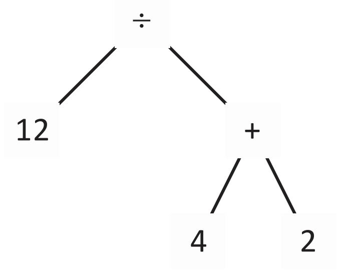
  Z = (X × Y) + (W × U),

  becomes:

  Z = X Y × W U × +

  in postfix notation.

Example 5.2. Convert the infix expression $12 \div (4 + 2)$ to postfix notation.



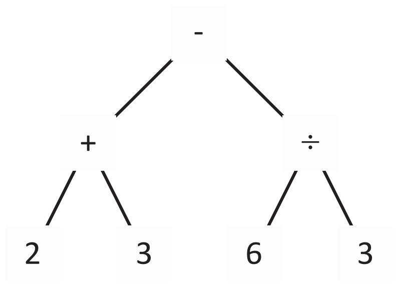
Step 1. Convert the infix expression to an expression tree.

$$12\ 4\ 2 + \div$$

Step 2. Perform a post-order traversal of the expression tree.

Stack					
			2		
		4	4	6	
	12	12	12	12	2
Postfix Expression	12	4	2	+	÷

Example 5.3. Convert the infix expression $(2+3)-6 \div 3$ to postfix notation.



Step 1. Convert the infix expression to an expression tree.

$$2\,3 + 6\,3\ \div\ -$$

Step 2. Perform a post-order traversal of the expression tree.

					3		
Stack		3		6	6	2	
	2	2	5	5	5	5	3
	2	3	+	6	3	÷	-
Postfix Expression	2	3	+	6	3	÷	-

Three-address ISA

- Let's see how to evaluate an infix expression using different instruction formats.
- With a three-address ISA, (e.g.,mainframes), the infix expression,

  Z = X × Y + W × U

  might look like this:

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

Two-address ISA

- In a two-address ISA, (e.g.,Intel, Motorola), 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,R
```

One-address ISA

- In a one-address ISA, like MARIE, 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

Zero-address ISA (Stack Architecture)

- 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
  PUSH Z

5.2.5 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:

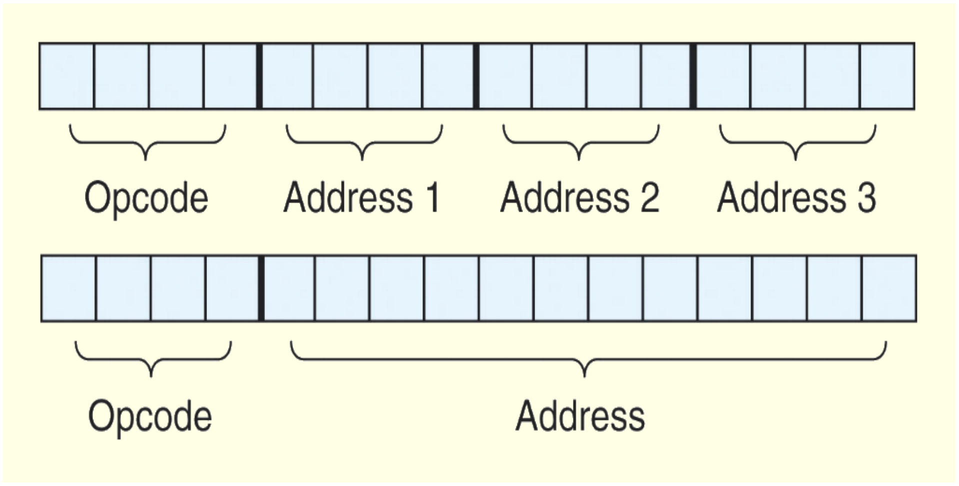
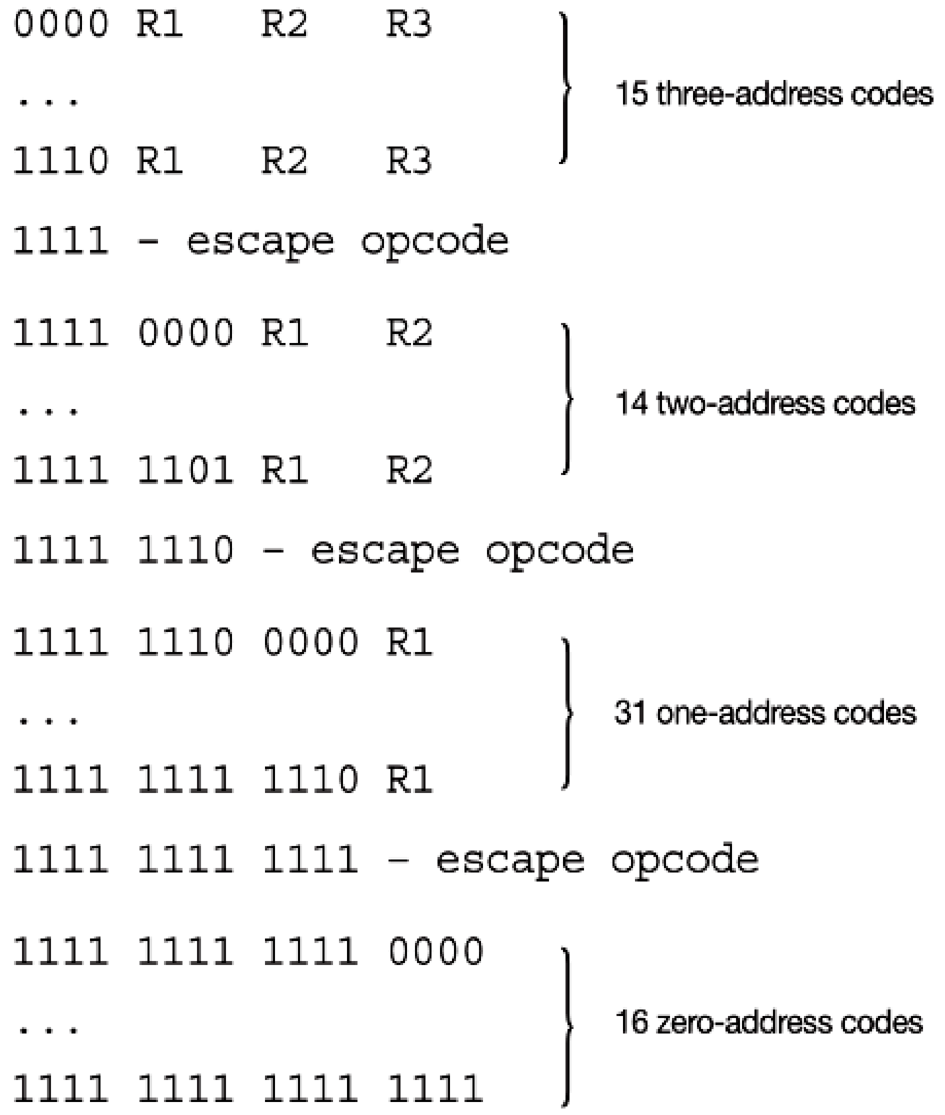


Figure 5.2 Two Possibilities for a 16-bit Instruction Format

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



```
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 }
...                 } 16 zero-address codes
1111 1111 1111 1111 }
```

Example 5.8

- 15 Instructions with three addresses
- 14 Instructions with two addresses
- 31 Instructions with one address
- 16 Instructions with zero addresses

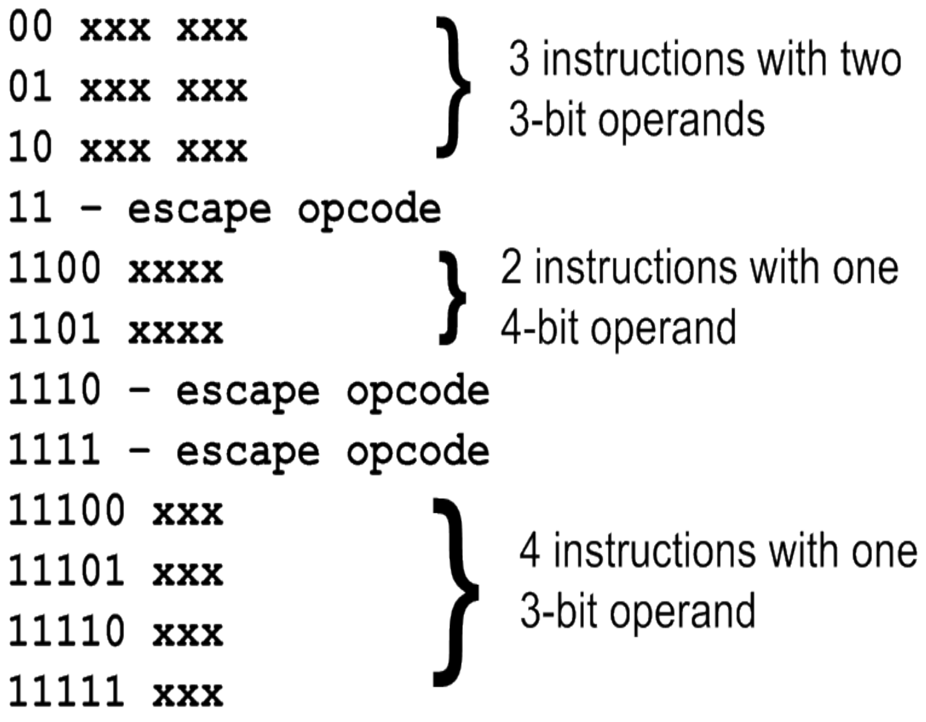
- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
  - 3 instructions with two 3-bit operands.
  - 2 instructions with one 4-bit operand.

– 4 instructions with one 3-bit operand.

We need:

3×2^3 = 192 bits for the 3-bit operands
2×2^4 = 32 bits for the 4-bit operands
4×2^3 = 32 bits for the 3-bit operands.

Total: 256 bits



```
00 xxx xxx
01 xxx xxx
10 xxx xxx
11 - escape opcode
1100 xxxx
1101 xxxx
1110 - escape opcode
1111 - escape opcode
11100 xxx
11101 xxx
11110 xxx
11111 xxx
```

3 instructions with two 3-bit operands

2 instructions with one 4-bit operand

4 instructions with one 3-bit operand