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 effect 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.
- Operand location.
- Types of operations.
- Type and size of operands.

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.
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, 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).

As an example, suppose we have the hexadecimal number 12345678.

The big endian and small endian arrangements of the bytes are shown below.

<table>
<thead>
<tr>
<th>Address</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big Endian</td>
<td>12</td>
<td>34</td>
<td>96</td>
<td>78</td>
</tr>
<tr>
<td>Little Endian</td>
<td>78</td>
<td>56</td>
<td>34</td>
<td>12</td>
</tr>
</tbody>
</table>

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.
- 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.

Standard...What Standard?

- Pentium (80x86), VAX are little-endian
- IBM 370, Motorola 680x0 (Mac), and most RISC are big-endian
- Internet is big-endian
  - Makes writing Internet programs on PC more awkward!
  - WinSock provides htonl and ntohl (Host to Internet & Internet to Host) functions to convert

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.
  - 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:
  - 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.

5.2 Instruction Formats

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

5.2 Instruction Formats

- 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:
  \[ Z = X \times Y \times W \times U \times + \]
  in postfix notation.
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
```

- 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.

- 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 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:

```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address 1</th>
<th>Address 2</th>
<th>Address 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
0000 R1 R2 R3
1110 R1 R2 R3
1111 0000 R1 R2
1111 1101 R1 R2
1111 1110 0000 R1
1111 1111 1110 R1
1111 1111 1111 0000
1111 1111 1111 1111
```

15 3-address codes
14 2-address codes
31 1-address codes
16 0-address codes