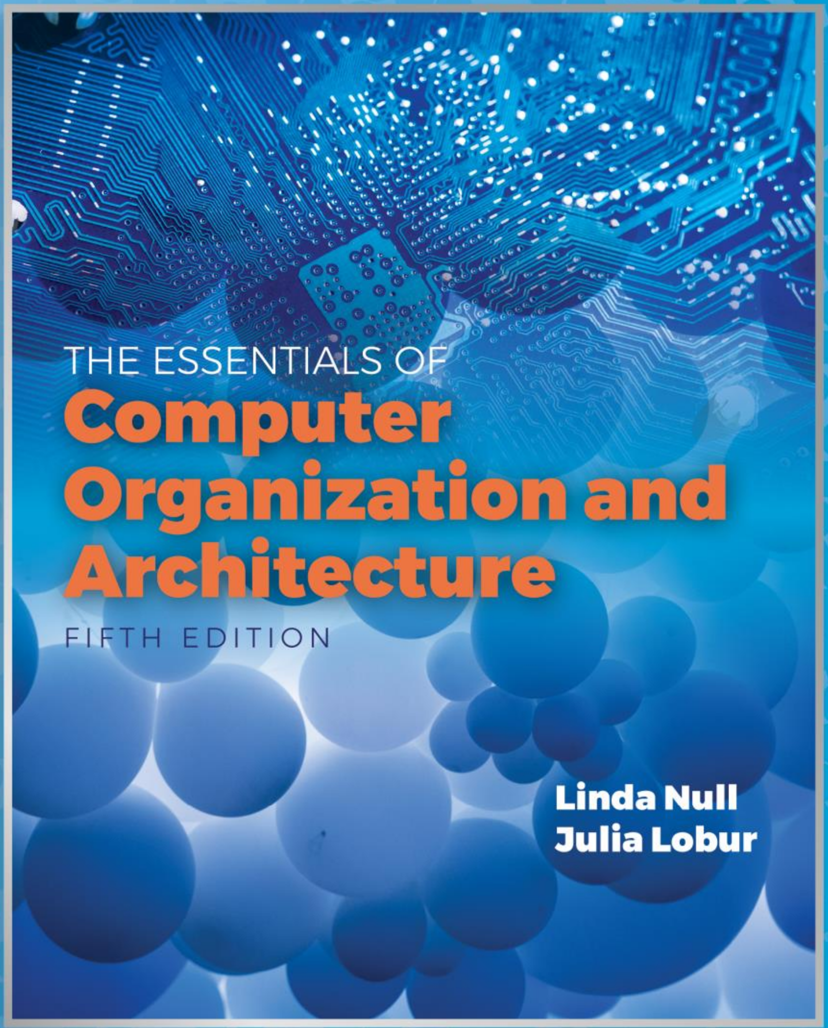
This is the first lecture of Chapter 5

Chapter 5

A Closer Look at Instruction Set Architectures



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 (1 of 31)

- 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 Instruction Formats (2 of 31)

- 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 (3 of 31)

- 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 (4 of 31)

- 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 (5 of 31)

- 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

5.2 Instruction Formats (6 of 31)

- 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 here.

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

Examples of Sign Extension

- Given the following two 16-bit integers in 2's complement
  - 0x2345
  - 0xA345

→

- Extend them to 32-bit integers in 2's complement
  - 0x00002345
  - 0xFFFFA345

Little endian

Base+0	45
Base+1	23

→

Base+0	45
Base+1	23
Base+2	00
Base+3	00

Big endian

Base+0	23
Base+1	45

→

Base+0	00
Base+1	00
Base+2	23
Base+3	45

5.2 Instruction Formats (7 of 31)

- 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 to a 32-bit integer does not require any arithmetic.

5.2 Instruction Formats (8 of 31)

- 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 (9 of 31)

- 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 (10 of 31)

- 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 (11 of 31)

- 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.

5.2 Instruction Formats (12 of 31)

- 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 (13 of 31)

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

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

- becomes:

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

- in postfix notation.

5.2 Instruction Formats (14 of 31)

- Example: Convert the infix expression (2+3) – 6/3 to postfix:

2 3+ - 6/3	The sum 2 + 3 in parentheses takes precedence; we replace the term with 2 3 +.

5.2 Instruction Formats (15 of 31)

- Example: Convert the infix expression (2+3) – 6/3 to postfix:

2 3+ - 6 3/	The division operator takes next precedence; we replace 6/3 with 6 3 /.

5.2 Instruction Formats (16 of 31)

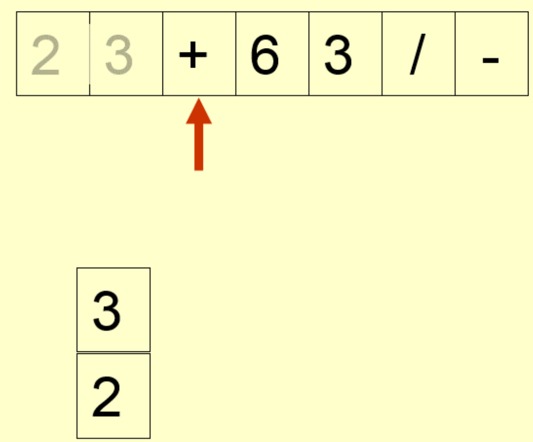
- Example: Convert the infix expression (2+3) – 6/3 to postfix:

2 3+ 6 3/ -	The quotient 6/3 is subtracted from the sum of 2 + 3, so we move the - operator to the end.

5.2 Instruction Formats (17 of 31)

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

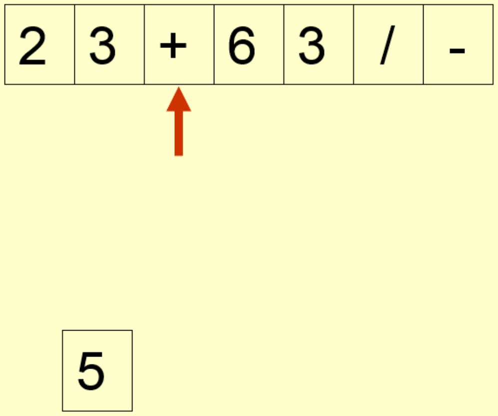
Scanning the expression from left to right, push operands onto the stack, until an operator is found



5.2 Instruction Formats (18 of 31)

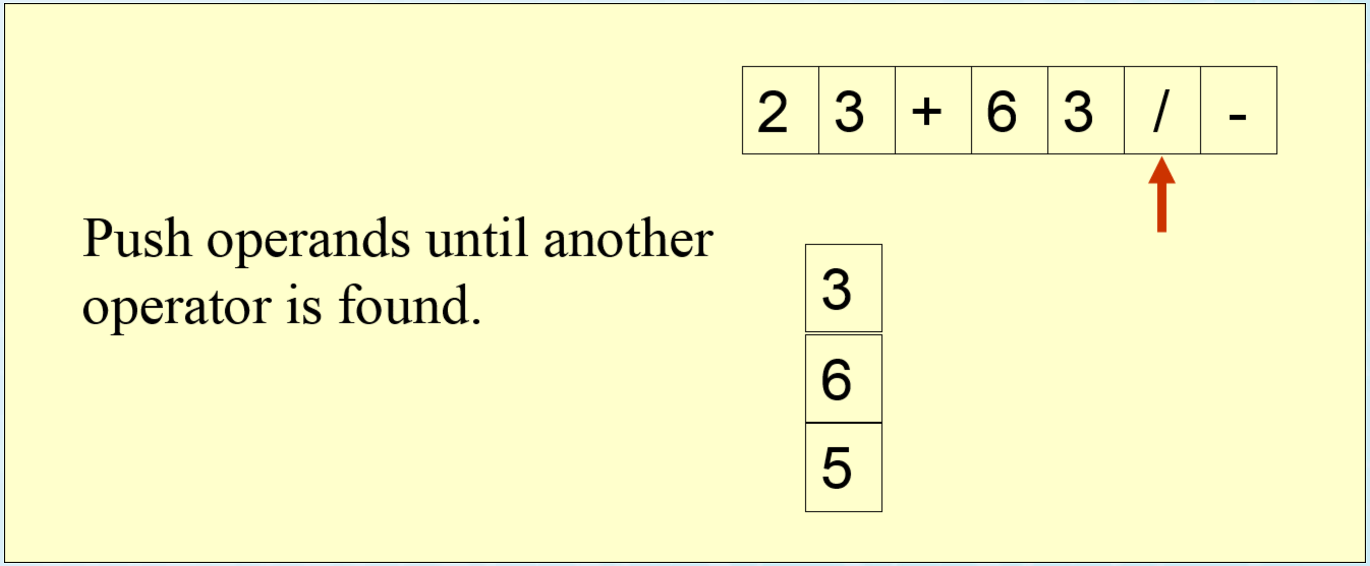
- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Pop the two operands and carry out the operation indicated by the operator. Push the result back on the stack.



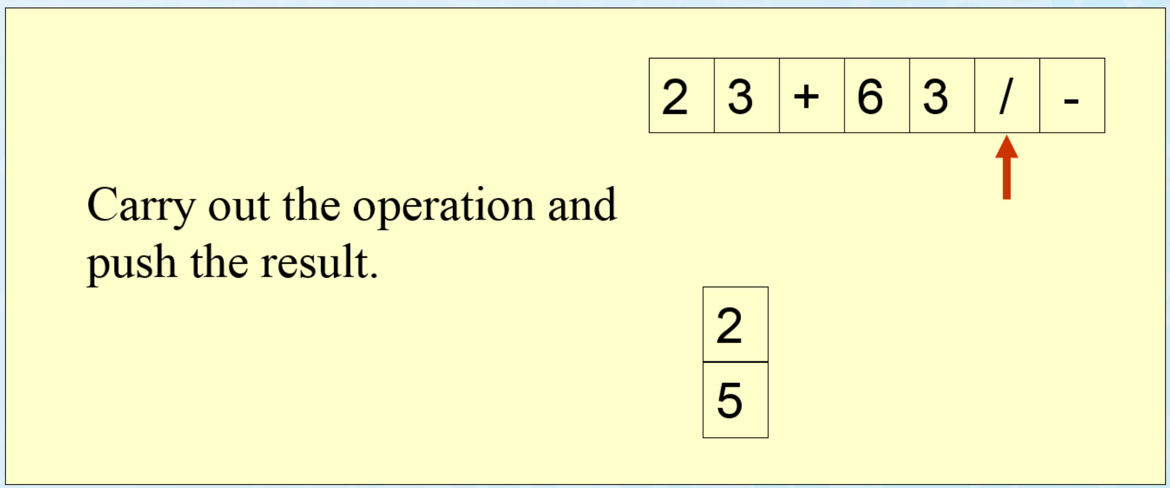
5.2 Instruction Formats (19 of 31)

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :



5.2 Instruction Formats (20 of 31)

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :



5.2 Instruction Formats (21 of 31)

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Finding another operator, carry out the operation and push the result.

The answer is at the top of the stack.