# Lecture-5

Chapter-3.3

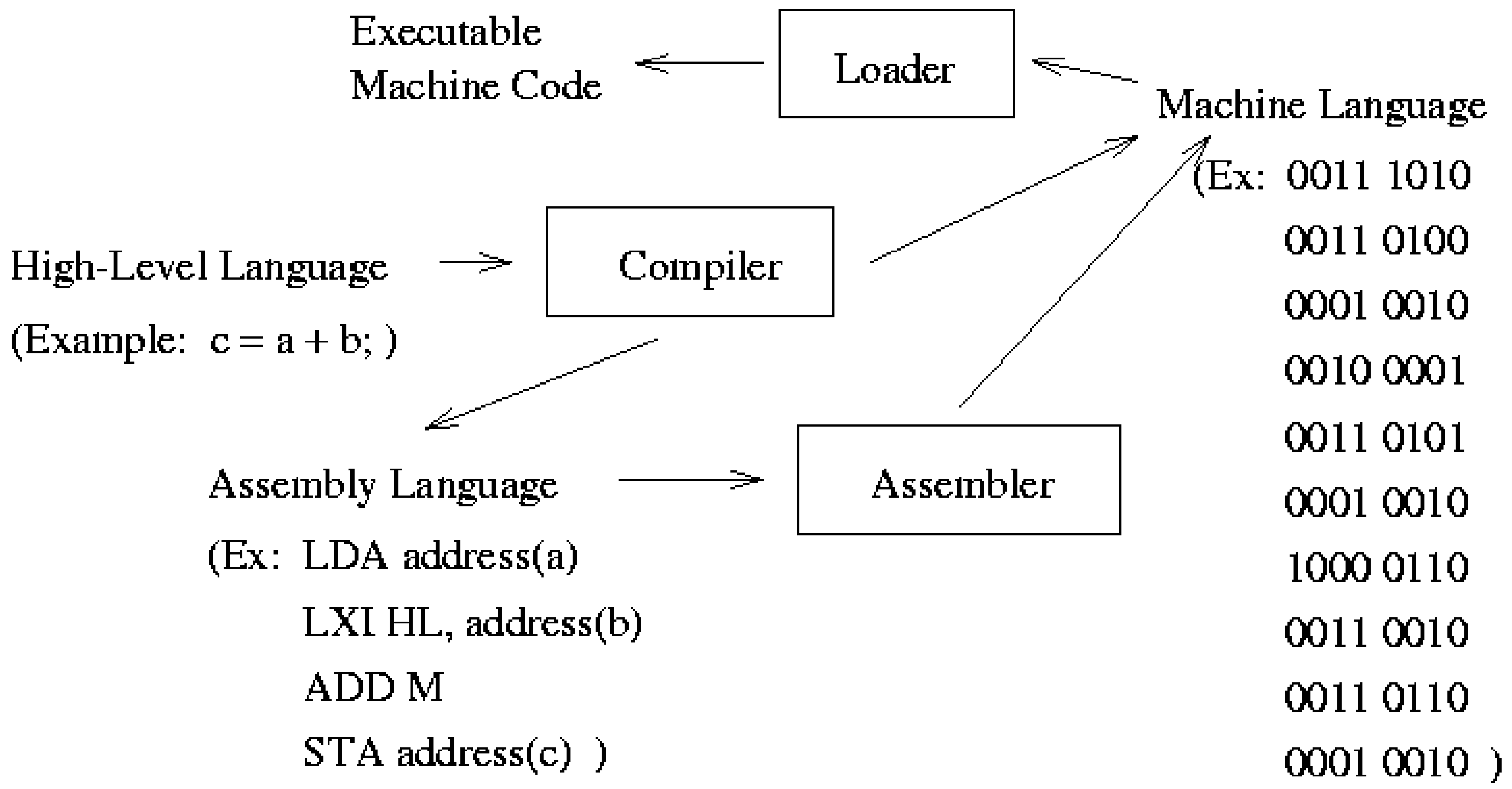
Computer Architecture and Organization-

Jhon P. Hayes

## **Processor Basics**

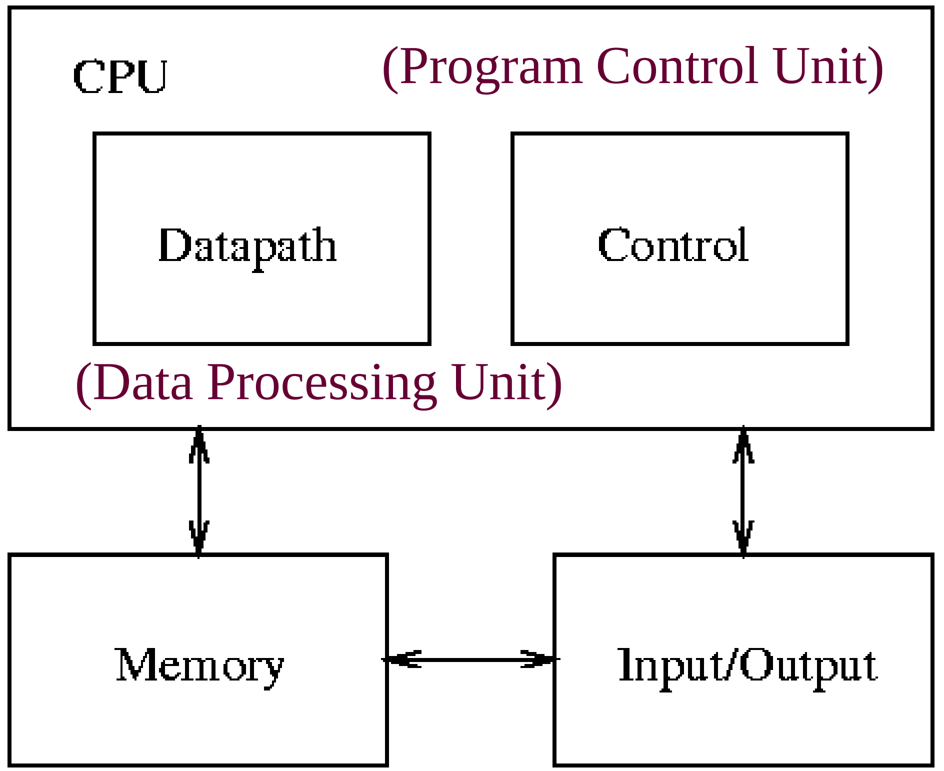
**(Pipelining, Instruction Set Design)**

# Computer Program Execution

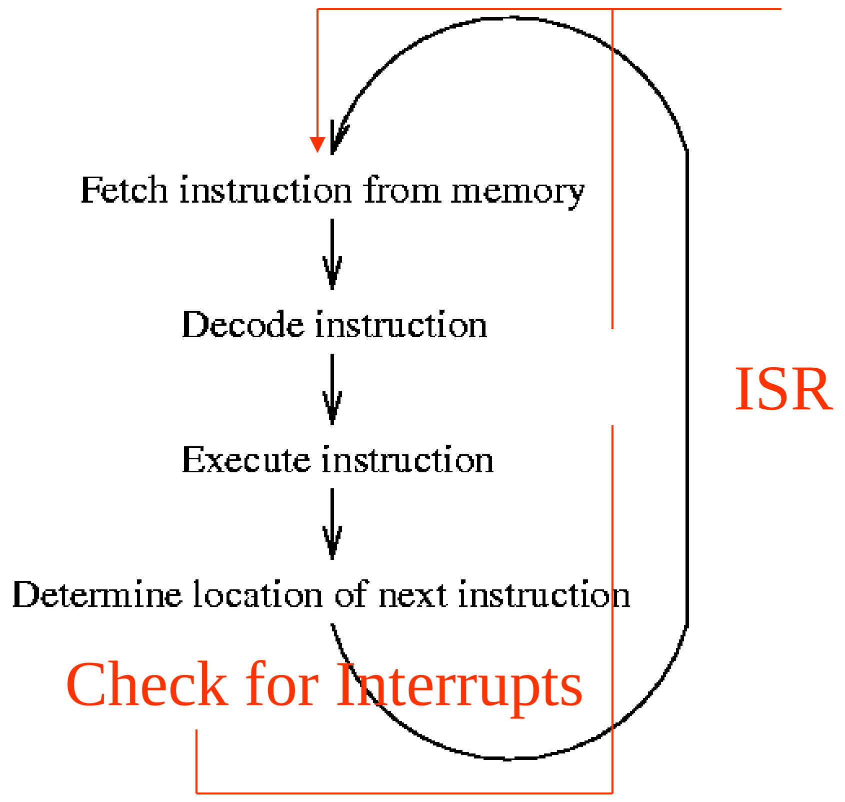


# Block Diagram and Basic Operation

## Block Diagram



## Basic Operation


Fetch instruction from memory
Decode instruction
Execute instruction
Determine location of next instruction
Check for Interrupts
ISR

# Instruction Set Design

## Fixed vs Variable-Length Instruction Formats:

**Fixed-length instruction** formats and **variable-length instruction** formats are two way s of encoding instructions in computer architecture.

- Fixed-length instruction format
  - use same number of bits to represent all instructions
  - leads to simpler (and faster) instruction decoding
  - facilitates fast and effective prefetching of instructions
- Variable-length instruction format
  - use different numbers of bits for different instructions
  - can accommodate short and long instructions without a waste of memory
  - prefetching and decoding of instructions is more difficult

# Instruction Set Design

Example of **fixed-length instruction formats** used in the MIPS (Millions of Instruction Per Second) microprocessor

R-type Instruction Format

| opcode | source reg-1 | source reg-2 | destin. register | shift amount | function variant |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

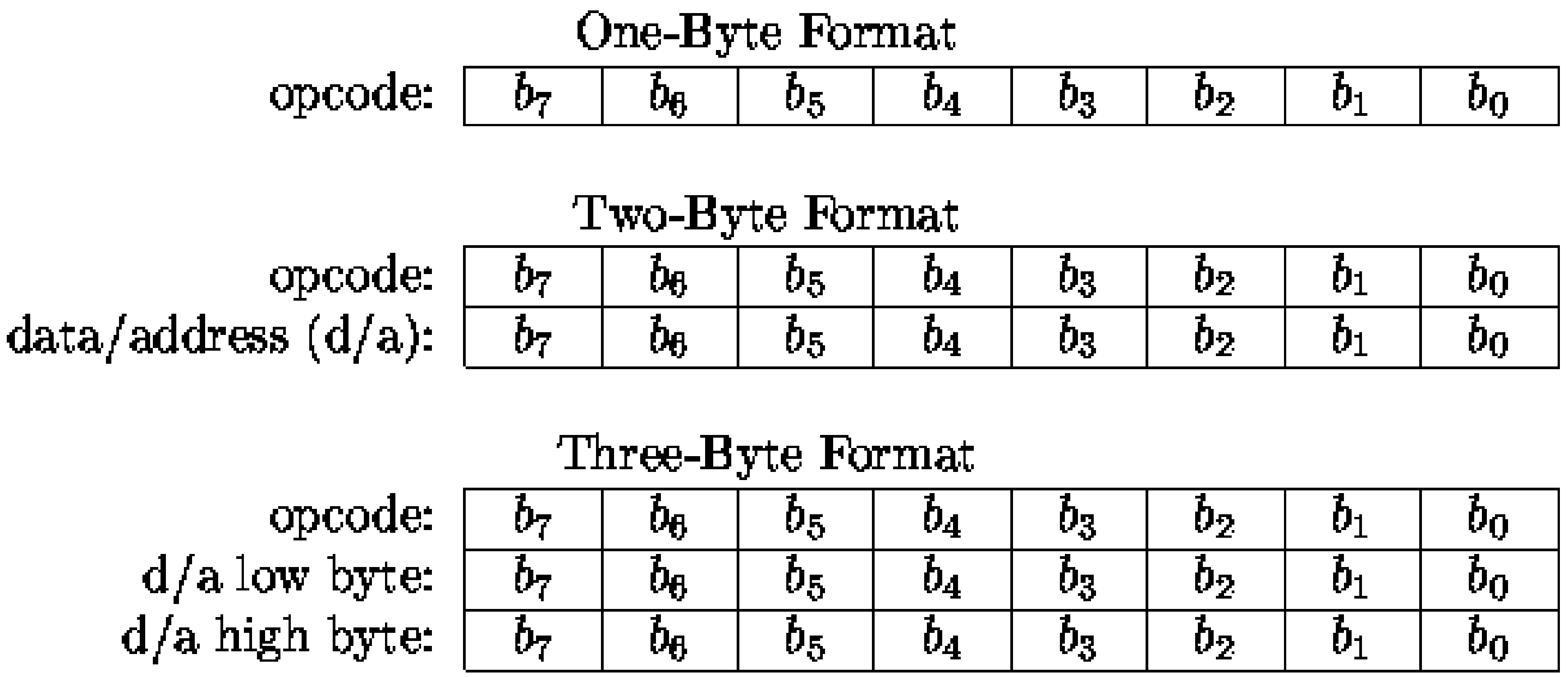
I-type Instruction Format

| opcode | source reg-1 | source reg-2 | address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

Fig: Two instruction formats used in the MIPS architecture

# Instruction Set Design

**Variable-length instruction formats** used in the Intel 8080 architecture



One-Byte Format

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| opcode: | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |

Two-Byte Format

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| opcode: | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| data/address (d/a): | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |

Three-Byte Format

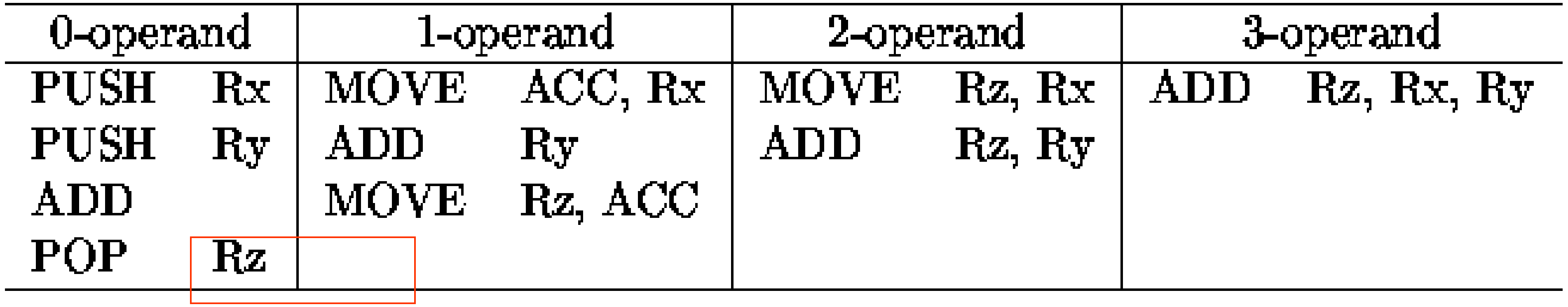
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| opcode: | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| d/a low byte: | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| d/a high byte: | $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ | $b_0$ |

**Figure 6.5.** Instruction formats for the Intel 8080 architecture.

## Number of Data Operands

- **Zero-operand instructions**
  - data is accessed from the “stack” (a LIFO (last-in-first-out) queue)
- One-operand instructions
  - the accumulator (ACC) register is used as the default second data input and destination
- Two-operand instructions
  - one of the data inputs is the default destination
- Three-operand instructions

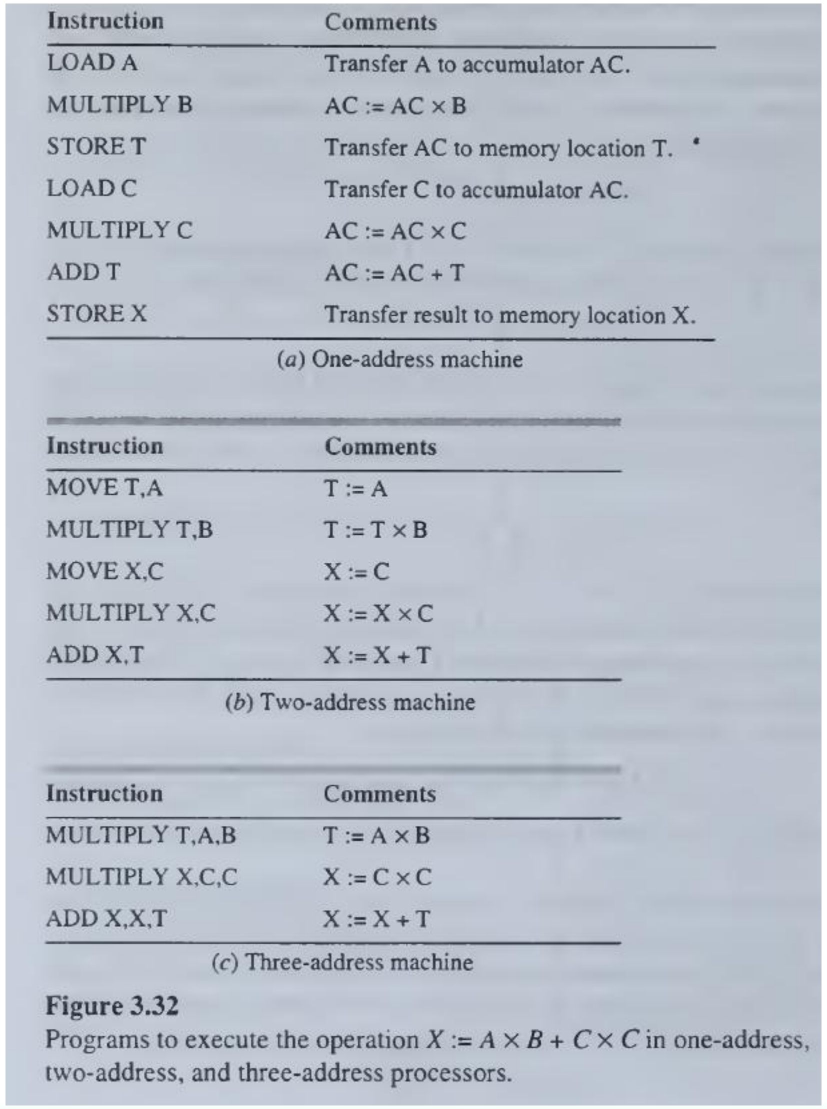
# Number of Data Operands



| 0-operand | 1-operand | 2-operand | 3-operand |
|---|---|---|---|
| PUSH Rx | MOVE ACC, Rx | MOVE Rz, Rx | ADD Rz, Rx, Ry |
| PUSH Ry | ADD Ry | ADD Rz, Ry | |
| ADD | MOVE Rz, ACC | | |
| POP Rz | | | |

**Figure 6.6.** Examples of 0, 1, 2, and 3-operand instruction sequences for the operation $\mathbf{z} \leftarrow \mathbf{x} + \mathbf{y}$ assuming all data are in registers.

# Example



| Instruction | Comments |
|---|---|
| LOAD A | Transfer A to accumulator AC. |
| MULTIPLY B | AC := AC × B |
| STORE T | Transfer AC to memory location T. |
| LOAD C | Transfer C to accumulator AC. |
| MULTIPLY C | AC := AC × C |
| ADD T | AC := AC + T |
| STORE X | Transfer result to memory location X. |

(*a*) One-address machine

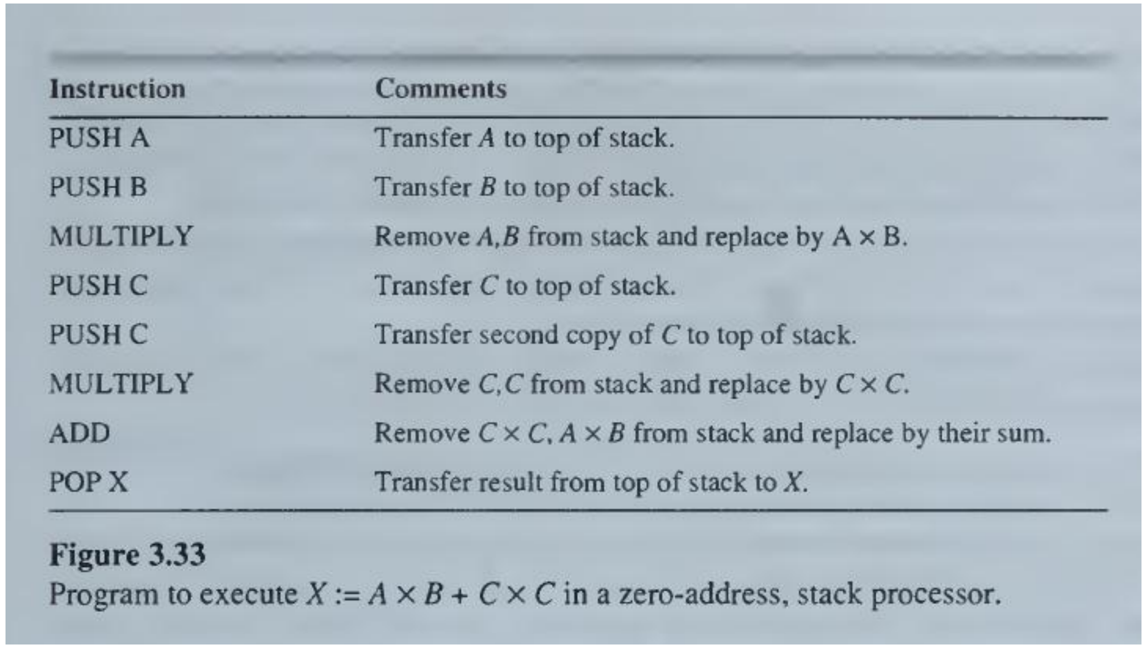
| Instruction | Comments |
|---|---|
| MOVE T,A | T := A |
| MULTIPLY T,B | T := T × B |
| MOVE X,C | X := C |
| MULTIPLY X,C | X := X × C |
| ADD X,T | X := X + T |

(*b*) Two-address machine

| Instruction | Comments |
|---|---|
| MULTIPLY T,A,B | T := A × B |
| MULTIPLY X,C,C | X := C × C |
| ADD X,X,T | X := X + T |

(*c*) Three-address machine

**Figure 3.32**
Programs to execute the operation $X := A \times B + C \times C$ in one-address, two-address, and three-address processors.



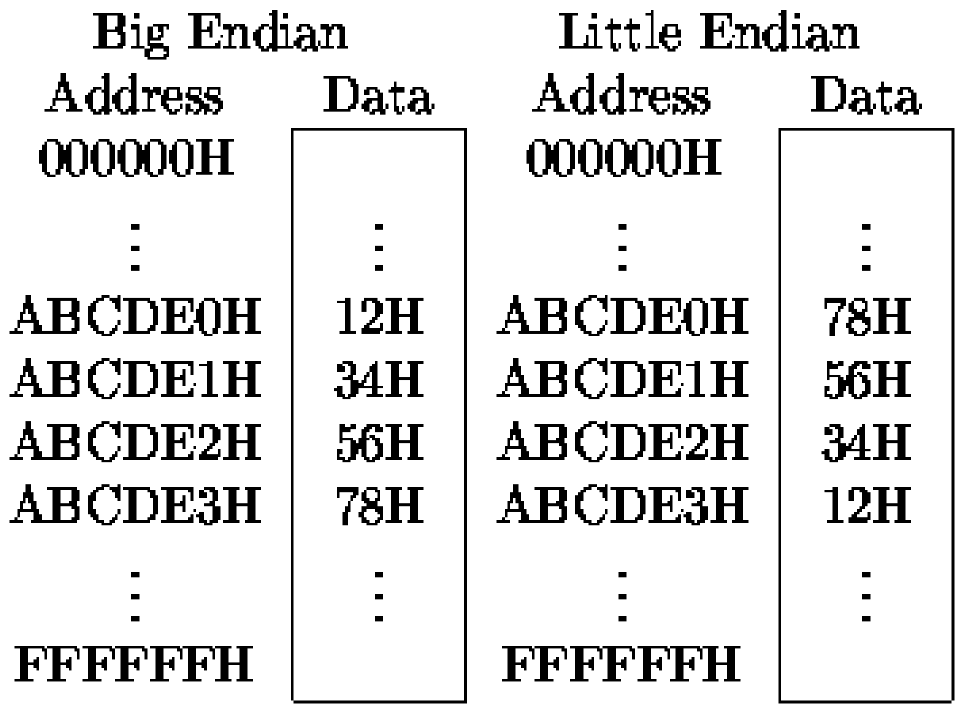
| Instruction | Comments |
|---|---|
| PUSH A | Transfer *A* to top of stack. |
| PUSH B | Transfer *B* to top of stack. |
| MULTIPLY | Remove *A*,*B* from stack and replace by A × B. |
| PUSH C | Transfer *C* to top of stack. |
| PUSH C | Transfer second copy of *C* to top of stack. |
| MULTIPLY | Remove *C*,*C* from stack and replace by *C* × *C*. |
| ADD | Remove *C* × *C*, *A* × *B* from stack and replace by their sum. |
| POP X | Transfer result from top of stack to *X*. |

**Figure 3.33**
Program to execute $X := A \times B + C \times C$ in a zero-address, stack processor.

# Endian Mode

Big-**endian** and little-**endian** are terms that describe the order in which a s equence of bytes are stored in computer memory.

Example Instruction: Store 12345678H, ABCDE0H



| Big Endian | | Little Endian | |
|---|---|---|---|
| Address | Data | Address | Data |
| 000000H | | 000000H | |
| ⋮ | ⋮ | ⋮ | ⋮ |
| ABCDE0H | 12H | ABCDE0H | 78H |
| ABCDE1H | 34H | ABCDE1H | 56H |
| ABCDE2H | 56H | ABCDE2H | 34H |
| ABCDE3H | 78H | ABCDE3H | 12H |
| ⋮ | ⋮ | ⋮ | ⋮ |
| FFFFFFH | | FFFFFFH | |

**Figure 6.7.** An example of big and little endian addressing modes.

# Pipeline

- Modern CPUs have a variety of speedup techniques, including cache memories, and several forms of instruction level parallelism.
- Such parallelism may be present in the internal organization of DPU or in the overlapping of the operations carried out by the DPU and PCU.
- Overlapping of instruction fetching and execution is an example of instruction pipelining which is an important speedup feature of RISC processor.

# Pipeline

- Each instruction can be thought of as passing through two consecutive stages of processing: a fetch stage implemented mainly by PCU and an execution stage implemented mainly by DPU.
- Hence two instructions can be processed simultaneously in every CPU clock cycle, with one completing its fetch phase and other completing it's execute phase. This Simultaneous process is called two stage instruction pipelines. [Fig 3.8]

# Instruction Pipelining

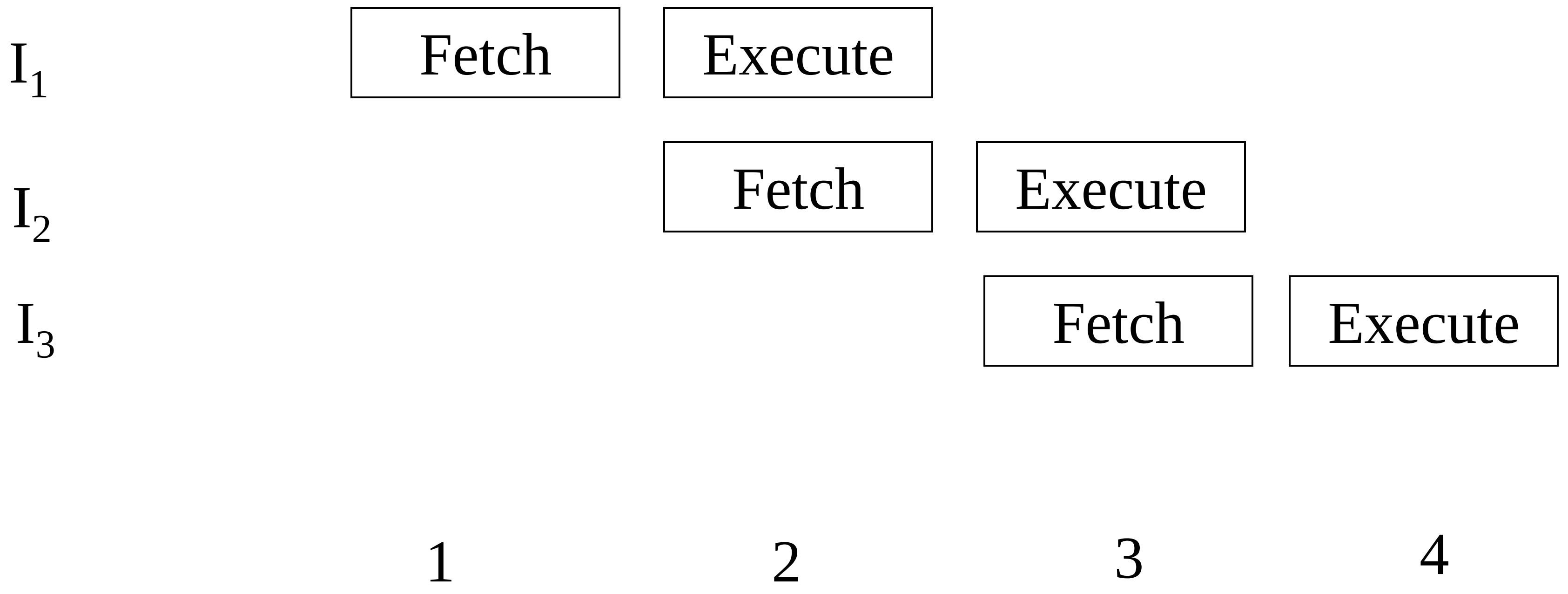


**Fig 3.8**: Overlapping instruction in a two stage instruction pipeline

# That’s All
# Thank You