Chapter 4

PROCESSOR INSTRUCTION SET - I

Introduction

- The processor operates by repeatedly cycling through two steps which define the execution of a single instruction.
  - *Fetch* and *Execute*
- This combination of *Fetch* and *Execute* steps runs as long as the processor is powered up and not explicitly halted.
- Algorithmic notation for a typical processor

```
repeat forever
  {
       fetch the next instruction from main memory
       execute – i.e. carry out – the instruction
         fetched
  }
```

- Important independent and complementary aspects of the working of a processor:

  (a) how a processor is programmed

  (b) how a processor is designed to one by one execute instructions which make up a program

- The *Fetch cycle* does not vary much from instruction to instruction.

- The *Execute cycle* must vary from instruction to instruction, since different instructions operate on data in different ways.

Programmable Registers

- Operands stored in general-purpose programmable registers within the processor can be accessed much faster than those stored in main memory

- Frequently used operands within a program can be *loaded* from main memory into processor registers

- If an operand is no longer to be used frequently, it can be *stored* back into main memory

- Word Size

  A processor is said to have a word size of n bits, when it is capable of performing a single instruction -- an arithmetic operation, logical operation, or memory operation -- on n-bit quantities.

A Typical Instruction

- Processor instructions are also referred to as *machine instructions*. Structures of a typical instruction:

  < opcode >, or **< opcode, operand >** or **< opcode, operand-1, operand-2 >**

- *Opcode* - Abbreviation for *operation code*, indicating which operation the processor is to perform.
  - Ex: addition of two integers is indicated by opcode ADD.

- The processor performs operations on operands. An instruction may have none, one or two operands.

- Thus *operand*, *operand-1* and *operand-2* here represent the operands of the operation indicated by *opcode*.

- Suffixes '*-1*' and '*-2*' distinguish the two operands of an instruction which has two-operands.

A Simple Program

```
//
// Program to add three 32-bit integers
//
   LOAD        2000, R1      // Load first operand into R1
   LOAD        2004, R2      // ... second operand into R2
   LOAD        2008, R3      // ... third operand into R3
   ADD         R2, R1        // Add second operand into the
                             // first
   ADD         R3, R1        // Add third operand into the
                             // sum
   STORE       2000, R1      // Store result back in place of
                             // the first operand
   HALT                      // Halt the program
```

Instruction-wise execution of the program

Instruction	Contents					
	m:2000	m:2004	m:2008	R1	R2	R3
(initial condition)	**200**	**300**	**500**	?	?	?
LOAD 2000, R1	200	300	500	**200**	?	?
LOAD 2004, R2	200	300	500	200	**300**	?
LOAD 2008, R3	200	300	500	200	300	**500**
ADD R2, R1	200	300	500	**500**	300	500
ADD R3, R1	200	300	500	**1000**	300	500
STORE 2000, R1	**1000**	300	500	1000	300	500
HALT	1000	300	500	1000	300	500

Conditional Execution

- Real-life computer programs perform computations which cannot be expressed as simple sequences

- Assume the following requirement in the program
  - If the sum of the three integers is less than 1000, then change the sum to 1000

```
//
// Program to add three 32-bit integers
// If the sum is less than 1000, it is changed to 1000.
//
        LOAD    2000, R1        // Load first operand into R1
        LOAD    2004, R2        // ... second operand into R2
        LOAD    2008, R3        // ... third operand into R3
        ADD     R2, R1          // Add second operand into the first
        ADD     R3, R1          // Add third operand into the sum
        COMP    R1, #1000       // Compare contents of R1 to the
                                // constant 1000
        JGE     DONE            // If greater or equal, jump to DONE
        MOV     #1000, R1       // Change contents of R1 to 1000
DONE:   STORE   2000, R1        // Store result back in place of
                                // the first operand
        HALT                    // Halt the program
```

- Symbol ‘#’ indicates that the operand is the indicated constant itself, rather than the content of the indicated memory location.

- In this case, the instruction COMP *compares* the contents of register R1 with the specified constant 1000.

- The instruction JGE specifies that if the result of the last comparison is ‘Greater or Equal’, then the next instruction to be fetched and executed is the one which has *label* DONE.

- In this case, MOV causes the constant 1000 to be *moved* into register R1.

Instruction Classes and Instruction Set

- The instructions seen so far may be grouped into several functional categories or classes:
  - ADD is an *arithmetic* instruction,
  - LOAD, STORE and MOV are *data transfer* instructions,
  - COMP is a *comparison* instruction,
  - JGE is a *transfer of control* instruction, and
  - HALT may be considered a *processor control* instruction

- The set of all instructions available on a processor is known as its *instruction set*.

- Due to electronic design and fabrication technology, fairly complex instruction sets are provided on modern processors.

- Many processor designers have switched to *RISC* (*reduced instruction set computer)* designs.

- A simple instruction set processor is able to get more work done per second than a processor with a more complex instruction set.

- Complex instructions can be programmed from simpler ones

- Processors based on *CISC* (*complex instruction set computer)* provide compatibility with earlier generations of commercially successful processors.

Iteration

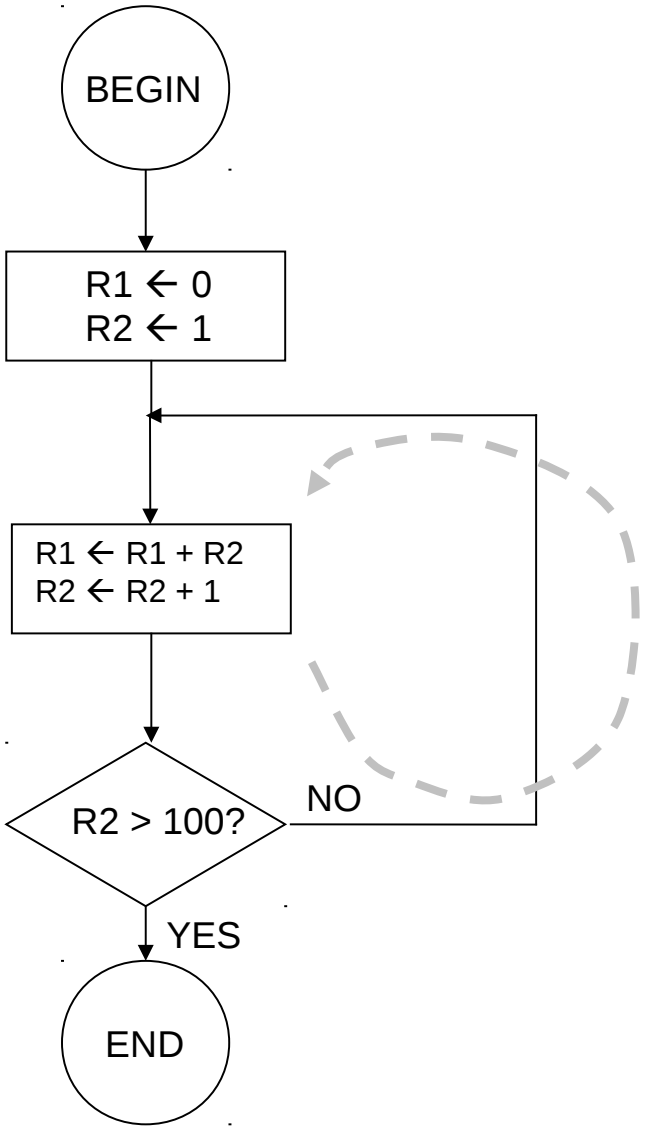
- *Iteration* – repeated execution of a set of operations
- Programs of any practical significance employ iteration in one form or another, for example as under:

```
//
// Program to find the sum of integers from 1 to 100
//
   MOV         #0, R1         // Initialize R1 to 0
   MOV         #1, R2         // Initialize R2 to 1
//
LOOP:  ADD     R2, R1         // Add R2 into R1
   ADD         #1, R2         // Increment R2 by 1
   COMP        R2, #100       // Compare contents of R2 to 100
   JLE         LOOP           // If less than or equal, jump
   to
                              // instruction with label LOOP
HALT                          // Halt the program
```

Flow chart is a graphical form of depicting program logic.

- Rectangular boxes - operations to be performed
- Diamond-shaped boxes - decisions to be made
- Arrows - how execution moves from one box to another
- Two circles containing the words BEGIN and END - the beginning and end of the program represented
- The thick dashed line marks out the *program loop* – or simply *loop* (which is executed 100 times)

Flowchart for Program to find the sum of integers from 1 to 100

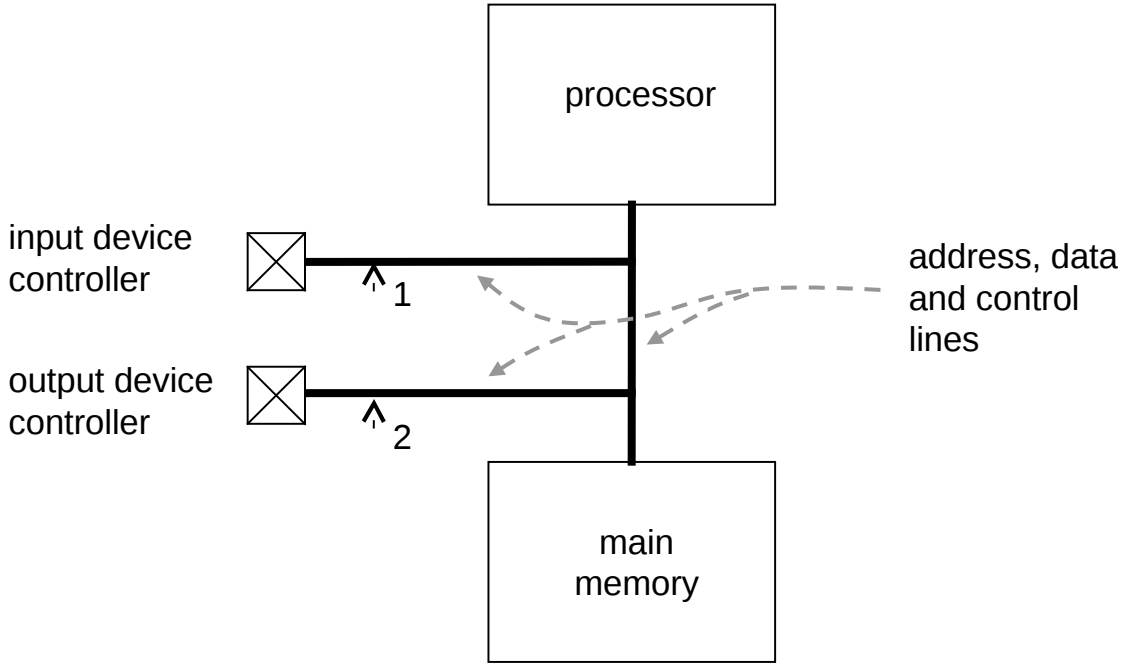


Basic Input and Output

- Instructions which can
  - read data *in* from an input device
  - write data *out* to an output device

```
//
// Program to read 100 bytes from an input device and
// write them to an output device
//
        MOV             #1, R2  // Initialize R2 to 1
//
   LOOP:IN              10, R1  // Read a byte from device at
                                // address 10
        OUT             20, R1  // Write that byte out to device
                                // at address 20
        ADD             #1, R2  // Increment R2 by 1
        COMP            R2, #100// Compare contents of R2 to 100
        JLE             LOOP    // If less or equal, jump to LOOP
                                //
        HALT                    // Halt the program
```

Computer system with the processor, main memory and two device controllers



- Same address and data lines connect the processor, the memory, and the device controllers.

- Special signal line designated as *memory-or-I/O*

  - If this line carries the value ‘1’, any address on the address lines is intended for the main memory, and is decoded accordingly.

  - If this line carries the value ‘0’, any address on the address lines is intended for an input or output device

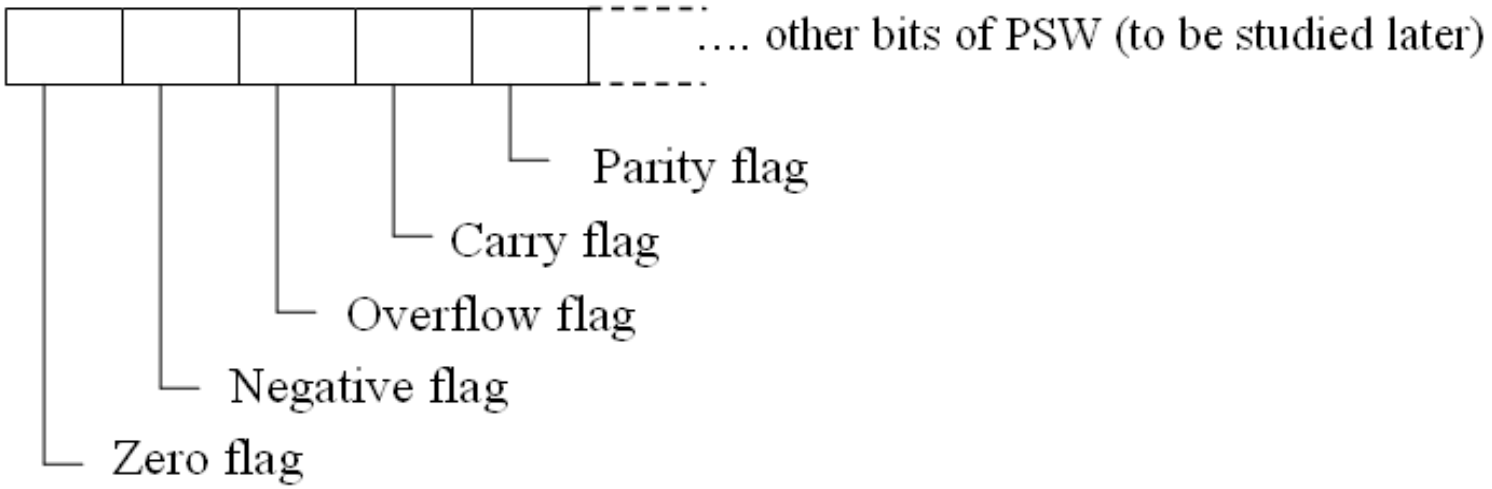
Program Status Word (PSW)

- PSW is a register provided within the processor which records the relevant bits of information from the result of execution of one instruction.

- These bits may be useful for conditional operations of subsequent instructions.

Flag	Abbreviation	Condition which causes flag to be set
Zero	Z	Result of previous operation is zero
Negative	N	Result of previous operation is negative
Overflow	O	Previous operation causes overflow in result
Carry	C	Previous operation causes carry to be generated from leftmost bit
Parity	P	Result of previous operation has odd number of 1's

Functions of some of the bits in Program Status Word (PSW)



Some of the flag bits in PSW

- PSW of a processor also contains other information related to I/O operations and memory protection, which is required when multiple programs share main memory

Summary

- Typical machine instruction formats; use of programmable registers; *load & store* architecture.
- Use of assembly language
- A few programs involving *sequential*, *conditional*, and *iterative* execution of instructions.
- Programs illustrating input & output on hypothetical devices.
- Program logic - A combination of the basic *sequential*, *conditional*, and *iterative* control flow structures.
- Classes of machine instructions - *data transfer* instructions, *arithmetic & logic* instructions, *comparison* instructions, *transfer of control* instructions, and *processor control* instructions.
- Instruction set design approaches - CISC and RISC.
- Condition flags in the program status word (PSW).