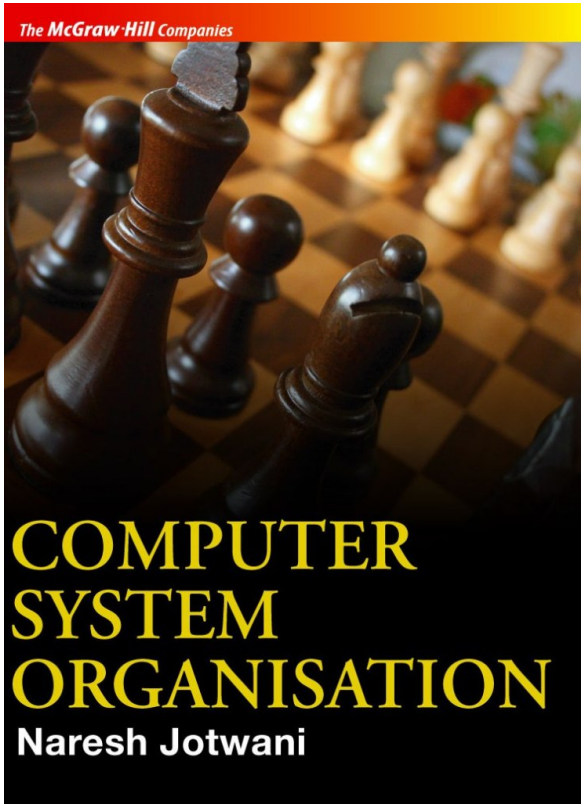
Mc Graw Hill Education

COMPUTER SYSTEM ORGANISATION

Naresh Jotwani

PowerPoint Slides



CHAPTER 5

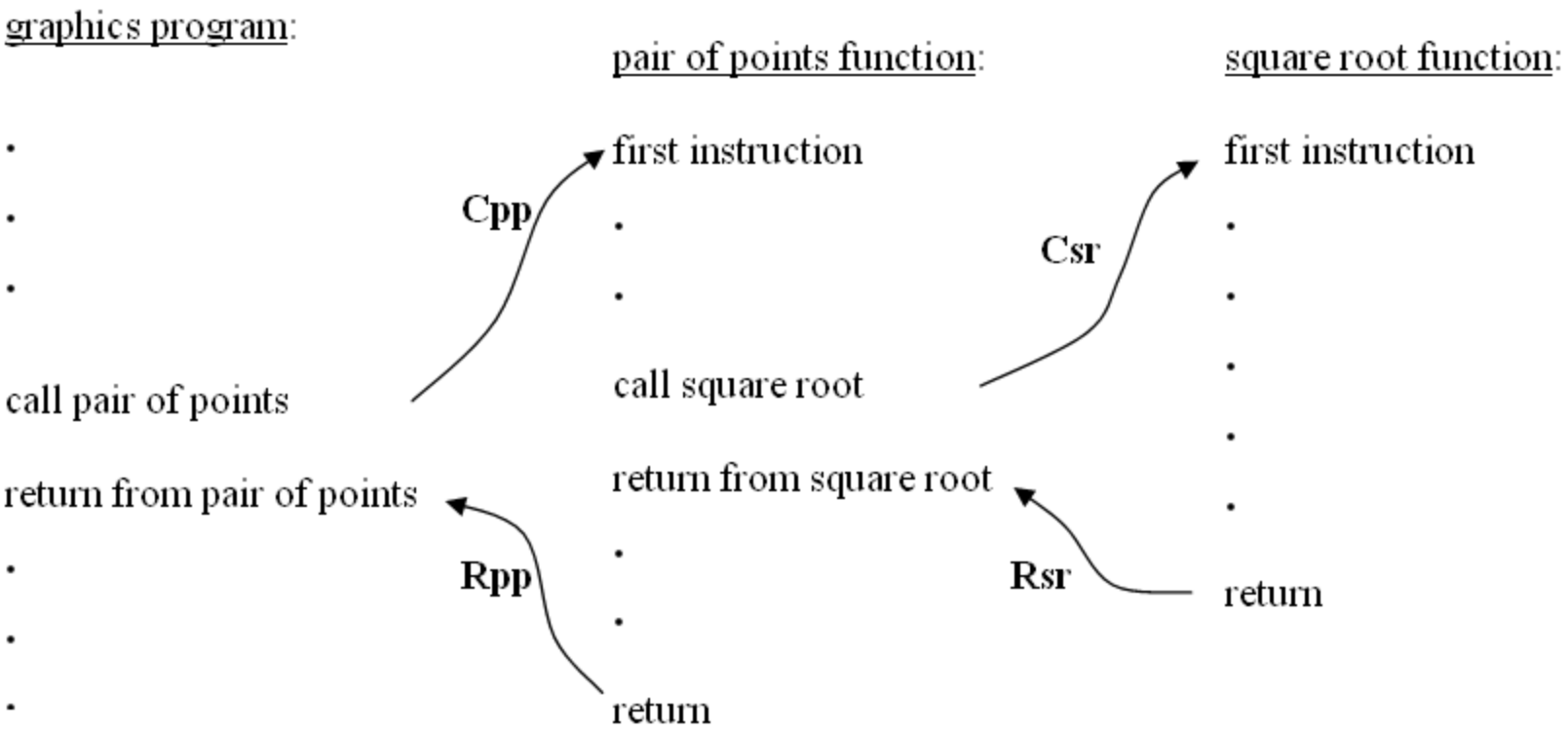
PROCESSOR INSTRUCTION SET - II

Addressing modes

- Register addressing
- Direct addressing
- Immediate addressing
- Indirect addressing
- Indexed addressing
- Implicit addressing

Functions and Function Calls

- Standard technique for structuring large programs - organize large programs as a set of *functions* of manageable size.
- Each function in a program performs a clearly specified task. Libraries of useful functions can also be made available to the programmer.
- Examples: mathematical functions sine, square root, *etc*; input and output functions.
- Consider below the schematic diagram of function *call* and *return*. We see a main program, a ‘pair of points’ function, and a ‘square root’ function.


graphics program:
pair of points function:
square root function:
first instruction
first instruction
Cpp
Csr
call pair of points
call square root
return from pair of points
return from square root
Rpp
Rsr
return
return

- We can see the logical relationship between the main program and the ‘pair of points’ function.
- The next instruction to be executed in the main program is the one immediately after the call to the pair of points function.
- After *jump* (conditional or unconditional) there is no mechanism for returning to the next instruction.
- A *function call* is also a transfer of control - similar to *jump*. But, in this case, the address of return instruction is saved by the processor.
- *Nested* function call and return are seen in the above diagram.

- In *nested* function calls, the last function called is the first one from which return is made.
- Therefore *return addresses* must be saved, and used in a *last-in first-out* manner.
- CALL uses direct addressing. *Return address* – *i.e.* address of the next instruction after CALL - is saved.
- RET has no explicit operand; causes the program to continue execution from the last return address saved.
- Combination of CALL & RET ensures that the last function called is the first one from which return is made.

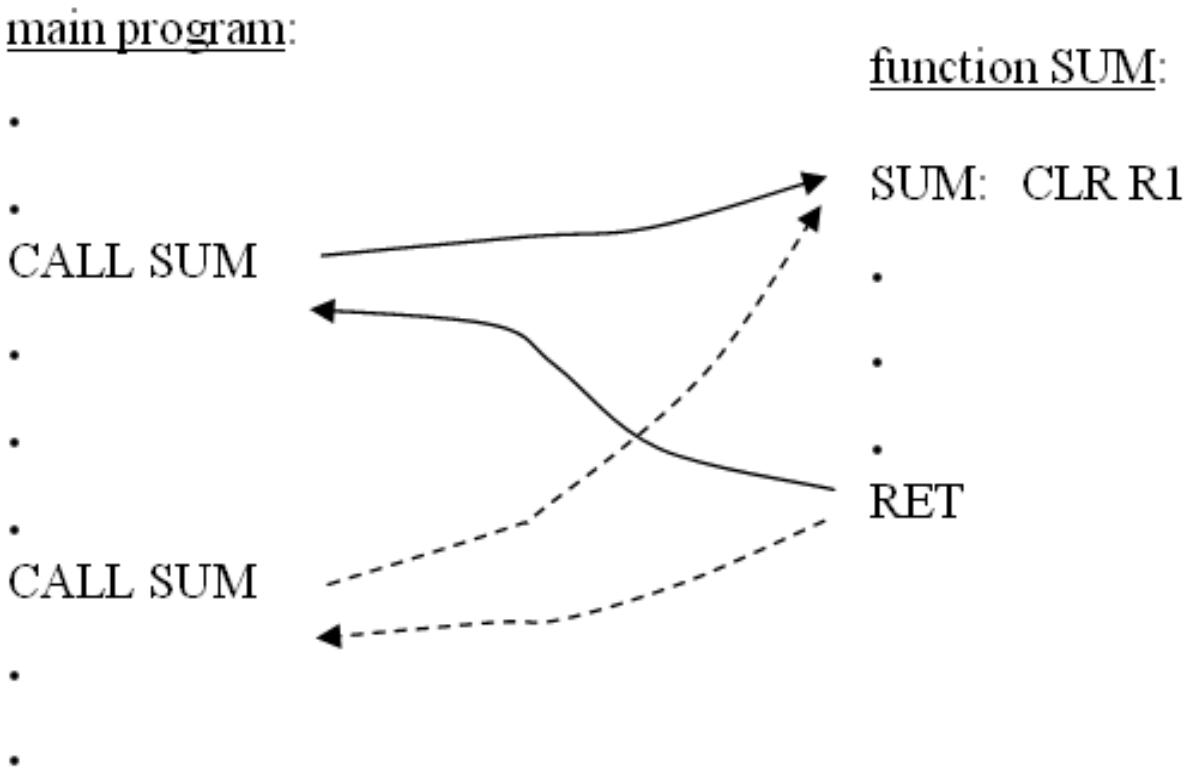
Function to find the sum of a hundred integers:

```
//
// Function to find the sum of a hundred integers.
// Input: Memory address of first integer is specified in R0.
// Output: The final sum is returned in R1.
// Registers used: R0, R1, R2, R3
//
SUM:  MOV         #0, R1      // Initialize R1 to 0
      MOV         #1, R2      // Initialize R2 to 1
//
LOOP: LOAD        [R0], R3    // Load operand from memory to R3
      ADD         R3, R1      // Add it into R1
      ADD         #4, R0      // Add 4 to R0 so that it points
                              // to the next integer to be added
      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
      RET                     // Return to calling program
```

- To find the sum of a hundred integers stored sequentially in memory, we can invoke the function SUM.

- SUM may be called from any point in the main program – as shown in the next slide.

Function SUM called from two different points in the main program:



- Only a single copy of a function is stored in main memory, which may be called from one or more points in the main program.

- This arrangement has two important benefits:

  (i) Multiple copies of the same piece of code do not occupy memory space, and

  (ii) only one copy of code needs to be maintained.

- Most software today is written in higher level languages.

- Large programs are structured using functions.

- Processor *NICE* has a *load & store* type of instruction set, which means that:

  (i) Memory operands must be loaded from main memory into programmable registers before any operation can be performed on them.

  (ii) Results must be explicitly stored back from processor registers to main memory.

  (iii) Instructions LOAD and STORE are provided on the processor for memory operands.

  (iv) Arithmetic & logic operations can be performed on operands only after they are brought into programmable registers.

Instruction set of the processor *NICE**:

- DATA TRANSFER instructions
- ARITHMETIC & LOGIC instructions
- COMPARISON instruction
- TRANFER OF CONTROL instructions
- MISCELLANEOUS instruction

*See Table 5.1 in the text

NICE instruction formats:

- Instruction modifier indicates the variant of the processor instruction being used.
- **Byte / Word -** indicates whether operands are one byte each or one word each. Letter B indicates *byte*, while letter W (or no letter) indicates *word*.
- e.g. LOAD B is *load byte*, and LOAD or LOAD W is *load word*.
- SHIFT instruction has two other modifiers – *shift left* or *right*, *shift with carry* or *without carry*.
- Columns *First Operand* and *Second Operand* in Table 5.1 indicate the addressing modes available with the respective operand.

Data Transfer instructions:

- The first group of instructions in Table 5.1. They include LOAD & STORE, IN & OUT, and MOV instructions.
- LOAD and STORE perform data transfers between main memory and a register.
- MOV performs data transfers between registers.
- IN & OUT are for input and output of data.

Arithmetic, Logic & Comparison instructions:

- Operations are performed either between two register operands, or between a register operand and an immediate operand.
- But one of these instructions – NOT, bitwise complement – operates on a single register operand.
- Operations may be specified in byte or in word mode.
- See Table 5.1 for *NICE* set of instructions.

- If result of ADD cannot be accommodated in 32 bits (word mode) or 8 bits (byte mode), this *carry* condition is recorded in the C flag of PSW. [Recall the discussion in Chapter 2].

- The corresponding condition after SUB is known as *borrow*, which arises when the result of subtraction cannot be accommodated in 32 bits (word mode) or 8 bits (byte mode).

- Comparison instruction COMP works in exactly the same way as SUB, except that the result is not stored back in the destination register.

- This is the difference between subtraction and comparison; in both cases, the flags – Z, N, O, C and P – are set accordingly in the same way.

- AND, OR and XOR are bit-wise logical instructions, which correspond respectively to bit-wise *and*, *or* and *exclusive or* operations in the registers specified.

- Each of the 8 or 32 bits of the destination register is operated bit-wise with the corresponding bit of the source operand, and the result stored in the destination register.

- *Exclusive or* function is useful in many contexts; it is defined as shown below.

Table of *exclusive or* function*

Value of input X_1	Value of input X_2	Value of output Y
0	0	0
0	1	1
1	0	1
1	1	0

Note: This differs from *or* function only in the last row.

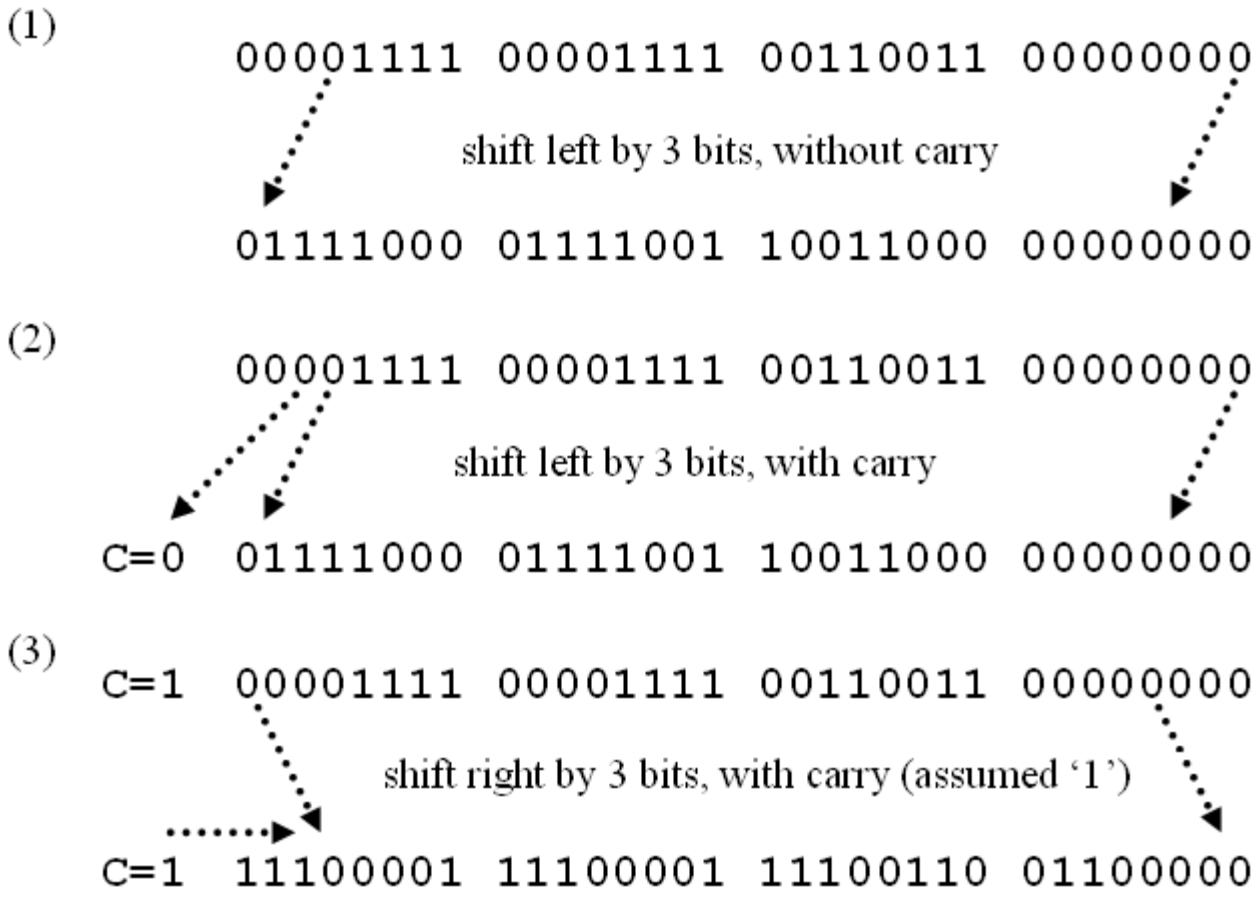
- Variants of *shift* operation prove to be useful in many different contexts – e.g. in programming floating point operations.

- The first operand of SHIFT instruction is the register whose contents are to be shifted, and the second operand is the number of bits to be shifted.

- SHIFT instruction on *NICE* has three modifiers: *byte/word*, *left/right* and *with/without carry*.

  - modifier *left/right* indicates the direction of shift
  - modifier *with/without carry* indicates whether the C flag is included with the register bits in the shift operation
  - see examples below

Three variants of 3 bit shift on a 32 bit quantity:



- Part (1) of the figure: 32 bit contents of a register before and after a *left* shift by 3 bits, without carry.

- In such a left or right shift without carry, an equal number of ‘0’ bits are inserted from the other end. Carry flag C takes no part in the operation.

- Shifts with carry are shown parts (2) and (3) of the figure.

- In SHIFT *with carry*, carry flag C is logically placed to the left of the operand, and the resulting 1+8 or 1+32 bits are shifted left or right by the number of bits specified.

- Part (2) of the figure: Left shift with carry, causes '0' bits to be inserted from the right.

- Part (3) of the figure: Right shift with carry, causes C bit value to be inserted from the left, without any change in the C bit itself.

- This asymmetry between left and right shift with carry is useful in programming arithmetic operations.

Transfer of Control instructions:

- After an arithmetic, logic, or comparison instruction, PSW records resulting condition, by setting or resetting condition codes – i.e. 'flags' – Z, N, O, C, P.

- Based on these, three types of conditional jump instructions are possible:

  (i) Jump on a flag bit being set, i.e. '1',

  (ii) Jump a flag bit being reset, i.e. '0', or

  (iii) Jump on a useful combination of flag bits being set and/or reset (JEQ, JNE, JLT, JLE, JGT, JGE).

- In Table 5.1, the row with **J***flag* in the first column corresponds to (i) above. A jump takes place if the indicated *flag* bit is set.

  Mnemonics of instructions are, respectively, JZ, JN, JO, JC and JP.

- The row with **JN***flag* in the 'Instruction' column corresponds to (ii) above. A jump takes place if the indicated *flag* bit is reset.

  Mnemonics of instructions are, respectively, JNZ, JNN, JNO, JNC and JNP.

- The next row of Table 5.1 has **J***condition* in the first column.
- In the last column of this row, we see values of *condition* in mnemonic form: GT, GE, LT, LE, EQ and NE.
  - These stand for, respectively, the six *arithmetic* conditions: *greater than*, *greater than or equal*, *less than*, *less than or equal*, *equal*, and *not equal*.

- Note: Mnemonics JEQ, JNE generate the same machine instructions as the pair of mnemonics JZ, JNZ, respectively.

- Of these, the first four conditions – GT, GE, LT, LE – are normally tested after a subtraction or comparison operation (SUB or COMP); they refer to arithmetic rather than logical conditions.

- Within the processor, these conditions are generated as appropriate combinations of the flag bits Z, N, O and C.

- Why do we need so many variants of the conditional jump instruction?
  - F*low of control* within a program is the nett result of conditional and unconditional jumps, as well as nested function calls and returns.
  - The choice of conditional jump instructions enables programmers to generate relatively efficient machine language code for any required flow of control.

Character and string operations:

- Information processing also involves processing of *character strings*. Recall that all text is stored in computer systems as sequences of characters.

- Word processors operate on text, and search engines on the *world wide web* operate on text stored in many millions of web pages around the world.

- For the program on the next slide, assume that:
  - Each character occupies one byte.
  - Length of the given sequence of characters is known.
  - Memory locations of the source and destination strings are non-overlapping.

```
//
// Function to copy a string of characters.
// Input: R0 - Memory address of source string
//        R1 - Memory address of destination string
//        R2 - Length of source string
//
// Assumes non-overlapping source and destination locations.
//
// Registers used: R0, R1, R2, R3
//
COPY: LOADB      [R0], R3    // Load source byte into R3
      STOREB     [R1], R3    // Store it at destination address
      ADD        #1, R0      // Add 1 to source address
      ADD        #1, R1      // Add 1 to destination address
      SUB        #1, R2      // Decrement count by 1
      JNZ        COPY        // If not zero, iterate
      RET                    // Return to calling program
```

- Other possible operations on character data:
  - Searching within data for a specified string,
  - Lexicographic comparison of two strings,
  - Concatenating two strings,
  - Finding the length of a variable length string,
  - Changing the attributes of a character string, and so on.

- 'Null termination' is a common method of defining variable length character strings.
  - The starting location of the character string in memory is known. The string extends towards higher memory addresses until the first '*null* byte' is encountered.

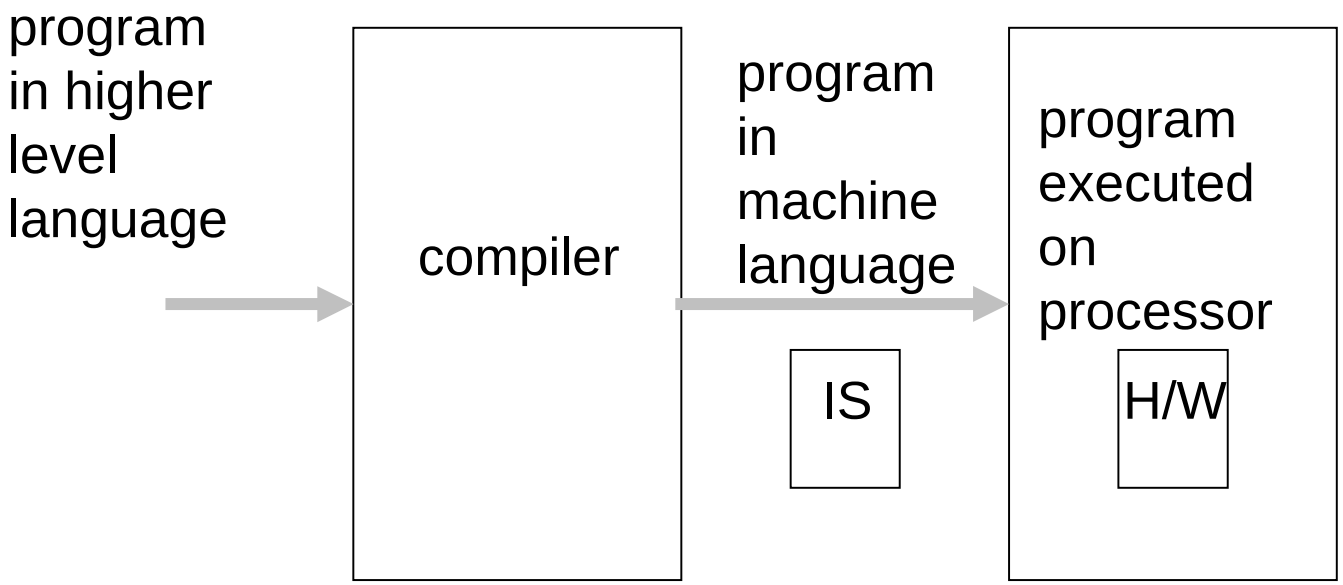
Role of instruction set:

- Whether instruction set of a processor is complex or 'reduced' – has impact *processor performance*.

- Instruction set - convenient way of packaging commonly performed operations
  - Instructions finally executed in hardware using primitive operations such as *and*, *or* and *not*.

- The next figure shows the stages of
  (i) writing a program in a higher level language
  (ii) compiling the program
  (iii) executing – i.e. running - the machine language program on the processor

Stages in writing, compiling and executing a program



- The instruction set of the processor plays the key role in the intermediate stage, marked ‘IS’ in the figure.

- Increased complexity of instruction set (‘IS’) may not necessarily lead to greater processor performance − if this increased complexity causes the last stage (‘H/W’ i.e. hardware) to run slower.

- On *NICE*, we have a fairly simple instruction set.

- For example, *NICE* does not have:
  (a) integer multiply and divide instructions
  (b) floating point instructions
  (c) instructions which operate on blocks of data.

- Such operations can be programmed using the machine instructions available.
- An operation implemented in software will be slower than in hardware.
- A low-cost or low-power version of a processor may not have extra capability, while a ‘higher end’ processor in the same *processor family* may have it.
- Based on requirements of the application, a user can choose the right processor from within a processor family. All of members of the processor family provide the same ‘core’ instruction set.

Summary

- *Addressing modes* – register, direct, immediate, indirect, indexed, implicit
- *Functions* – nested function calls and returns
- Groups of processor instructions –

  data transfer instructions,

  arithmetic, logic, & comparison instructions,

  transfer of control instructions,

  Other miscellaneous instructions.
- Operations on blocks of data
- Role of instruction set