Computer Organization and Architecture

Memory operations

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Load* (or *Read* or *Fetch*) and *Store* (or *Write*).

The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Load operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

Instruction and instruction sequencing

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation:

We need to describe the transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify a location by a symbolic name standing for its hardware binary address. For example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor register names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression

$$R1 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

Assembly language Notation:

We need another type of notation to represent machine instructions and programs. For this, we use an *assembly language* format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

```
Move  LOC,R1
```

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

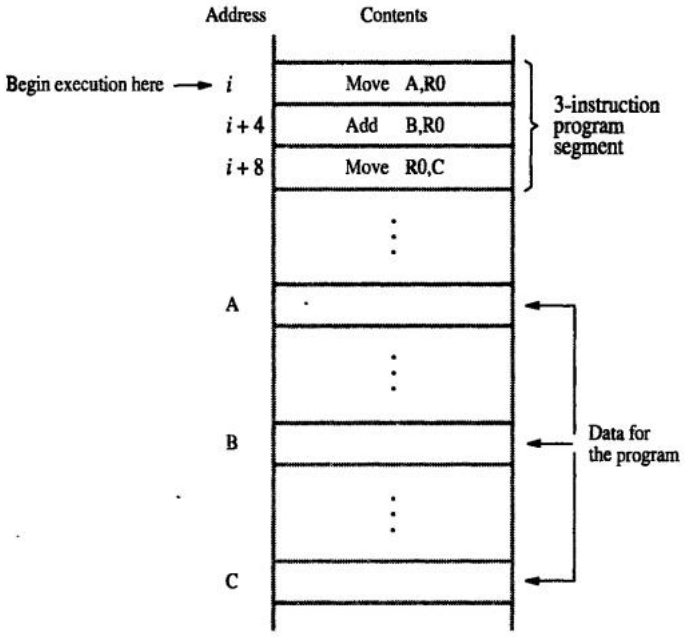
The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

```
Add  R1,R2,R3
```

Instructions:

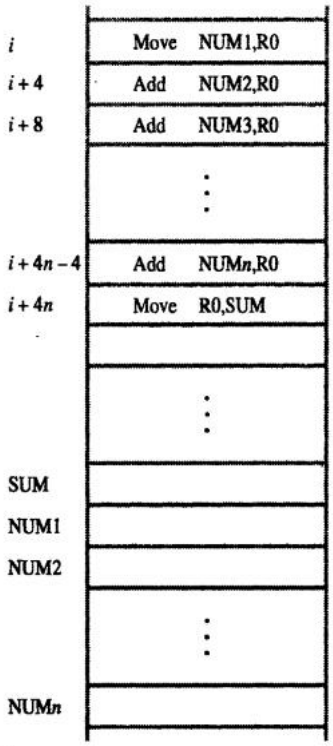
```
Move  A,Ri
Add   B,Ri
Move  Ri,C
```

Instruction Execution and Straight-line Sequencing:

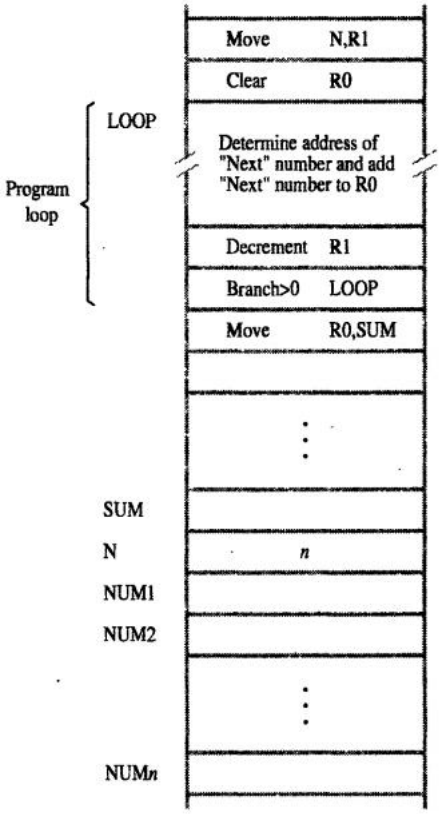


A program for C ← [A] + [B].

Branching:



A straight-line program for adding *n* numbers.



Using a loop to add *n* numbers.

- In the above table N is the count number which is loaded into register R1.

- The instruction Decrement R1 reduces the content of R1 by 1 each time through the loop.
- Execution of the loop is repeated as long as the result of the decrement operation is 0. For this a conditional branch instruction is used.
- A conditional branch instruction causes a branch only if a specified condition is satisfied.
- If the condition is not satisfied the PC is incremented in the normal way.
- Branch > 0 LOOP This means that the LOOP is repeated as long as there are entries in the list that are yet to be added to R0.
- The MOVE instruction moves the final result from R0 into the memory location SUM.

CPU Organization

- **Single Accumulator**
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- **General Register**
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- **Stack**
  - Operands and result are always in the stack

Instruction Formats

- **Three-Address Instructions**
  - **ADD R1, R2, R3** **R1 ← R2 + R3**
- **Two-Address Instructions**
  - **ADD R1, R2** **R1 ← R1 + R2**
- **One-Address Instructions**
  - **ADD M** **AC ← AC + M[AR]**
- **Zero-Address Instructions**
  - **ADD** **TOS ← TOS + (TOS – 1)**
- **RISC Instructions**
  - **Lots of registers. Memory is restricted to Load & Store**

Instruction: Opcode , Operand(s) or Address(es)

Example: Evaluate (A+B) ∗ (C+D)

- **Three-Address**

1. ADD R1, A, B ; R1 ← M[A] + M[B]
2. ADD R2, C, D ; R2 ← M[C] + M[D]
3. MUL X, R1, R2 ; M[X] ← R1 ∗ R2

- **Two-Address**

1. MOV R1, A ; R1 ← M[A]
2. ADD R1, B ; R1 ← R1 + M[B]
3. MOV R2, C ; R2 ← M[C]
4. ADD R2, D ; R2 ← R2 + M[D]
5. MUL R1, R2 ; R1 ← R1 ∗ R2
6. MOV X, R1 ; M[X] ← R1

- **One-Address**

1. LOAD A ; AC ← M[A]
2. ADD B ; AC ← AC + M[B]
3. STORE T ; M[T] ← AC
4. LOAD C ; AC ← M[C]
5. ADD D ; AC ← AC + M[D]
6. MUL T ; AC ← AC ∗ M[T]
7. STORE X ; M[X] ← AC

- **Zero-Address**

1. PUSH A ; TOS ← A
2. PUSH B ; TOS ← B
3. ADD ; TOS ← (A + B)
4. PUSH C ; TOS ← C
5. PUSH D ; TOS ← D
6. ADD ; TOS ← (C + D)
7. MUL ; TOS ← (C+D)∗(A+B)
8. POP X ; M[X] ← TOS

- **RISC**

1. LOAD R1, A ; R1 ← M[A]
2. LOAD R2, B ; R2 ← M[B]
3. LOAD R3, C ; R3 ← M[C]
4. LOAD R4, D ; R4 ← M[D]
5. ADD R1, R1, R2 ; R1 ← R1 + R2
6. ADD R3, R3, R4 ; R3 ← R3 + R4
7. MUL R1, R1, R3 ; R1 ← R1 ∗ R3
8. STOREX, R1 ; M[X] ← R1