**3. Instructions and Instruction Sequencing**

An *instruction* specifies an operation to be performed and the operands involved.

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

**3.1 Register Transfer Notation**

We need to describe the transfer of information from one location in a 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 such locations symbolically with convenient names. For example, name that represents the address of memory locations may be LOC. Predefined names for the processor registers may be R2. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the below expression means that the contents of memory location LOC are transferred into processor register R2.

**R2 ← [LOC]**

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

**R4 ← [R2] + [R3]**

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.

In computer jargon, the words “transfer” and “move” are commonly used to mean “copy.” Transferring data from a *source* location A to a *destination* location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

**3.2 Assembly-Language Notation**

*Assembly language* is another type of notations to represent machine instructions and programs. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

**Load R2, LOC**

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register. The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

**Add R4, R2, R3**

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

**3.3 RISC and CISC Instruction Sets**

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in “pipelined” fashion to overlap activity and reduce total execution time of a program. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called *Reduced Instruction Set Computers* (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

**3.4 Instruction Execution and Straight-Line Sequencing**

For the execution of the task C = A + B, implemented as C←[A] + [B]. Figure 1 shows a possible program segment for this task as it appears in the memory of a computer. We assume that the word length is 32 bits and the memory is *byte addressable*. The four instructions of the program are in successive word locations, starting at location *i*. Since each instruction is 4 bytes long, the second, third and fourth instructions are at addresses *i* + 4, *i* + 8, and *i* + 12.

Let us consider how this program is executed.

The processor contains a register called the *program counter* (PC), which holds the address of the next instruction to be executed.

To begin executing a program, the address of its first instruction (*i* in our example) must be placed into the PC.

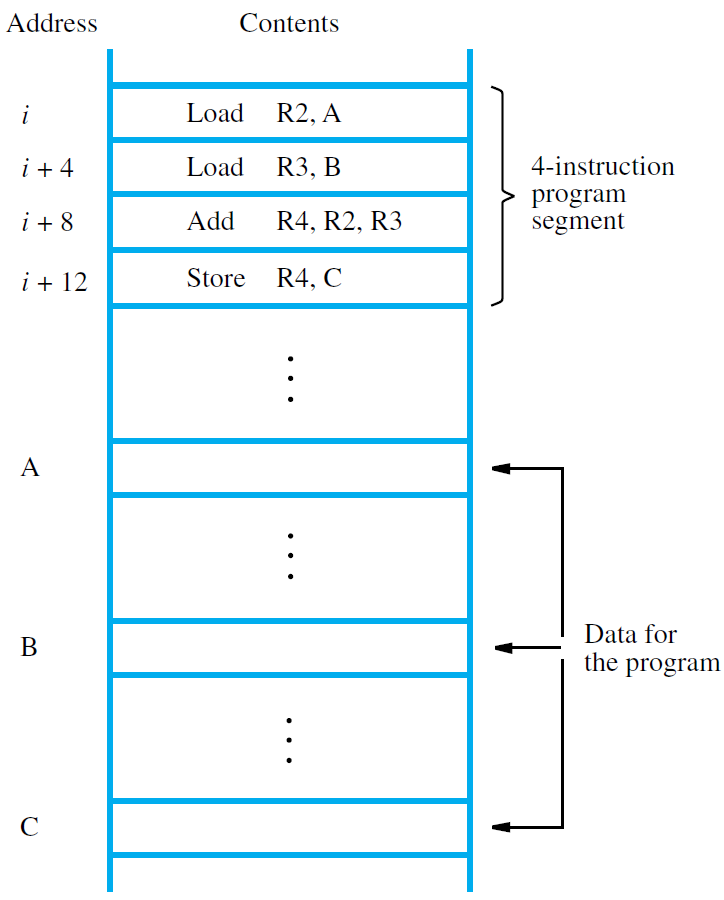


Figure 1: A program for C ← [A] + [B]

Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*.

During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location *i* + 12 is executed, the PC contains the value *i* + 16, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

**3.5 Branching**

Consider the task of adding a list of *n* numbers. The program outlined in Figure 2 is a generalization of the program in Figure 1. The addresses of the memory locations containing the *n* numbers are symbolically given as NUM1, NUM2*, . . . ,* NUM*n*, and separate Load and Add instructions are used to add each number to the contents of register R2. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Load and Add instructions, as in Figure 2, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. Figure 3 shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch\_if\_[R2]*>*0. During each pass through this loop, the address of the next list entry is determined, and that entry is loaded into R5 and added to R3. The address of an operand can be specified in various ways. For now, we concentrate on how to create and control a program loop. Assume that the number of entries in the list, *n*, is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction

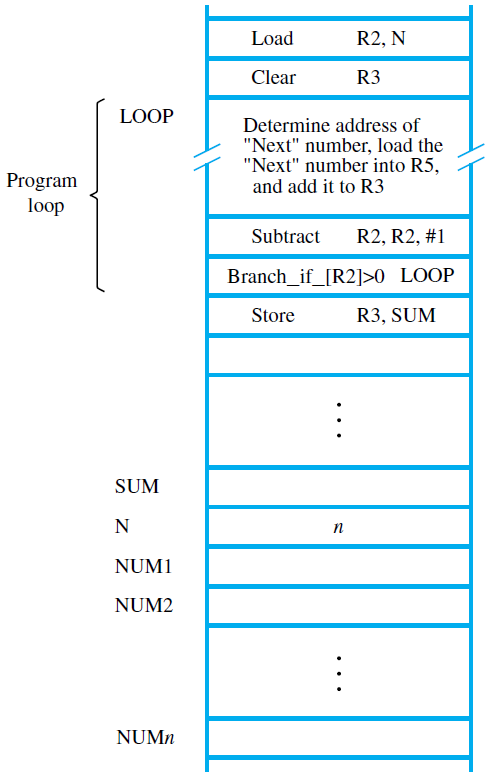


Figure 3: Using a loop to add *n* numbers

**Subtract R2, R2, #1**

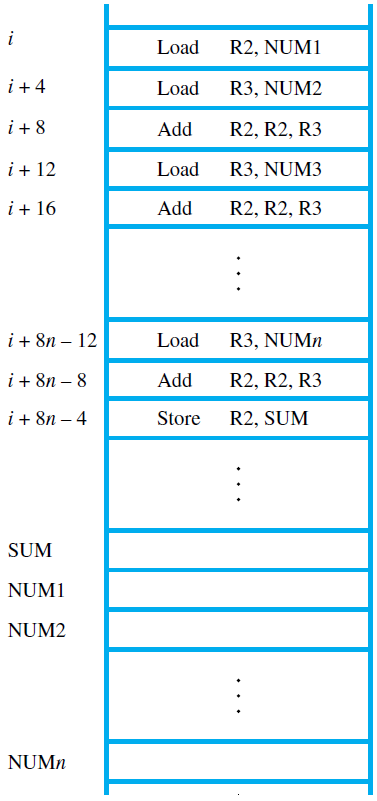


Figure 2: A program for adding *n* numbers

reduces the contents of R2 by 1 each time through the loop. Execution of the loop is repeated as long as the contents of R2 are greater than zero. We now introduce *branch* instructions. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. 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, and the next instruction in sequential address order is fetched and executed.

In the program in Figure 3, the instruction

**Branch\_if\_[R2]*>*0 LOOP**

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R3. At the end of the *n*th pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R3 into memory location SUM.

The capability to test conditions and subsequently choose one of a set of alternative ways to continue computation has many more applications than just loop control. Such a capability is found in the instruction sets of all computers and is fundamental to the programming of most nontrivial tasks.

**3.6 Generating Memory Addresses**

Let us return to Figure 3. The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop. Hence, the Load instruction in that block must refer to a different address during each pass. How are the addresses specified? The memory operand address cannot be given directly in a single Load instruction in the loop. Otherwise, it would need to be modified on each pass through the loop. As one possibility, suppose that a processor register, R*i*, is used to hold the memory address of an operand. If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop, it can provide the needed capability.

This situation gives rise to the need for flexible ways to specify the address of an operand. The instruction set of a computer typically provides a number of such methods, called *addressing modes*. While the details differ from one computer to another, the underlying concepts are the same.