SNS COLLEGE OF TECHNOLOGY

Coimbatore-35.

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME : 19ITT202 – COMPUTER ORGANIZATION AND ARCHITECTURE

II YEAR/ III SEMESTER

UNIT – I Basic Structure of Computers

Topic: Addressing Modes

Ms.Narmada C
Assistant Professor
Department of Computer Science and Engineering

Addressing Modes

- The data used in computations are organized in the form of lists, linked lists, arrays, queues, tables, etc.,.

- The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*.

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

Implementation of Variables & Constants

- Variables and Constants are the simplest data types found in almost every computer program.

Variable –

- A variable is represented by allocating a register or a memory location to hold its value.
- Thus, the value can be changed as needed using appropriate instruction.
- It can be specified by the name of the register or the address of the memory location where the operand is located.

2 Addressing modes to access variable

Register mode — The operand is the contents of a processor register; the name (address) of the register is given in the instruction.

- Eg- Move LOC,R2

Absolute mode — The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called *Direct*.)

- Eg- Add A,B

Constants –

- In constants the values are fixed integers.
- Address and data constants can be represented in assembly language using the Immediate mode.

Immediate mode — The operand is given explicitly in the instruction.

- A (#) sharp sign is used in front of the value to indicate that this value is to be used as an immediate operand. Syntax- #value
- Eg- Move #200,R0

Example –

$$A = B + 6$$

Instruction -

```
Move  B,R1
Add   #6,R1
Move  R1,A
```

Indirection and Pointers

In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the *effective address* (EA) of the operand.

Indirect mode — The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

We denote indirection by placing the name of the register or the memory address given in the instruction in parentheses

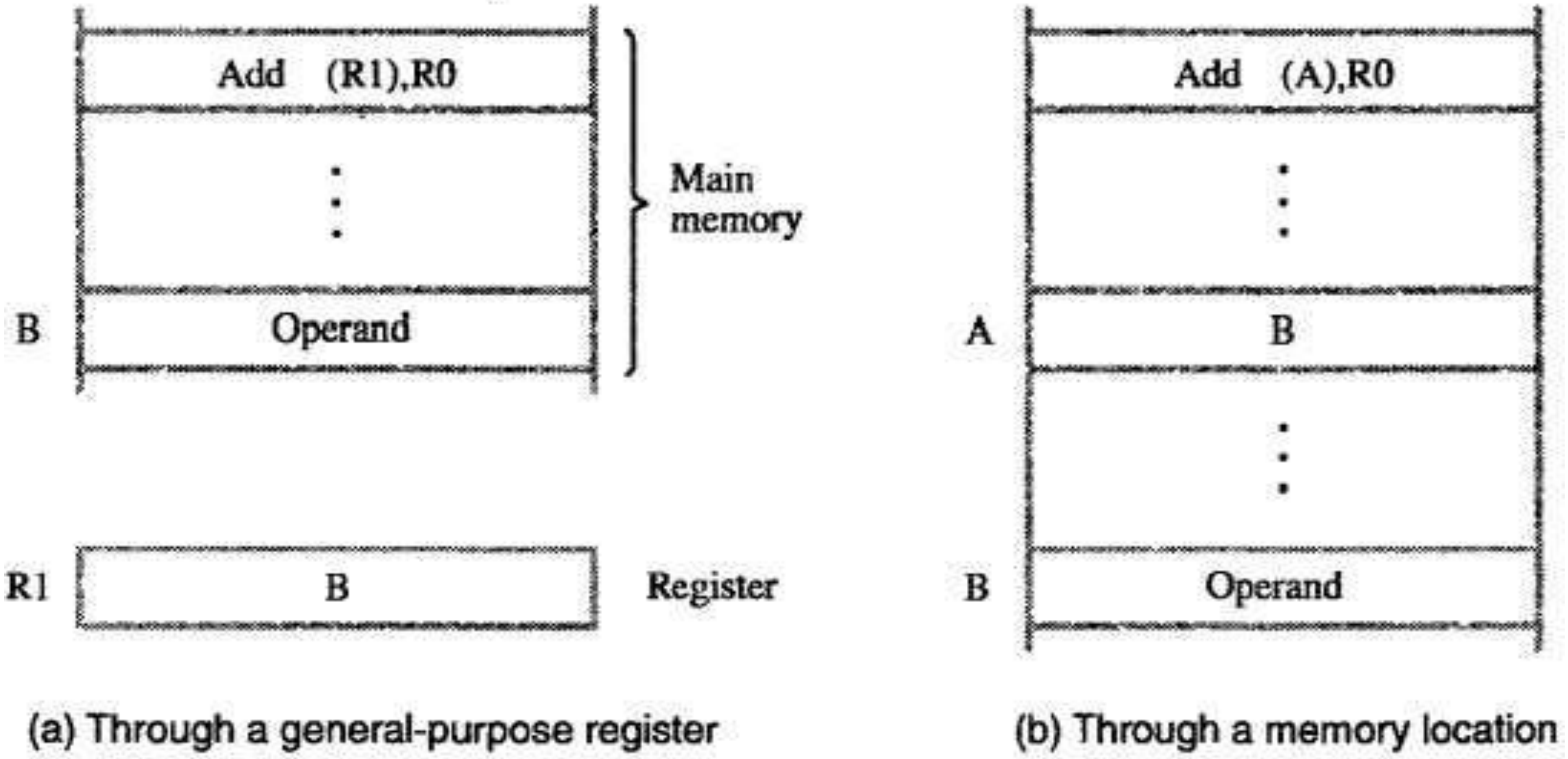


Figure 2.11 Indirect addressing.

The register or memory location that contains the address of an operand is called a *Pointer.* **In 2.11 (a) R1 is the pointer & B is the EA**

Eg- Addition of N No's

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

Indexing and Arrays

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

Index mode — The effective address of the operand is generated by adding a constant value to the contents of a register.

The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an *index register*. We indicate the Index mode symbolically as

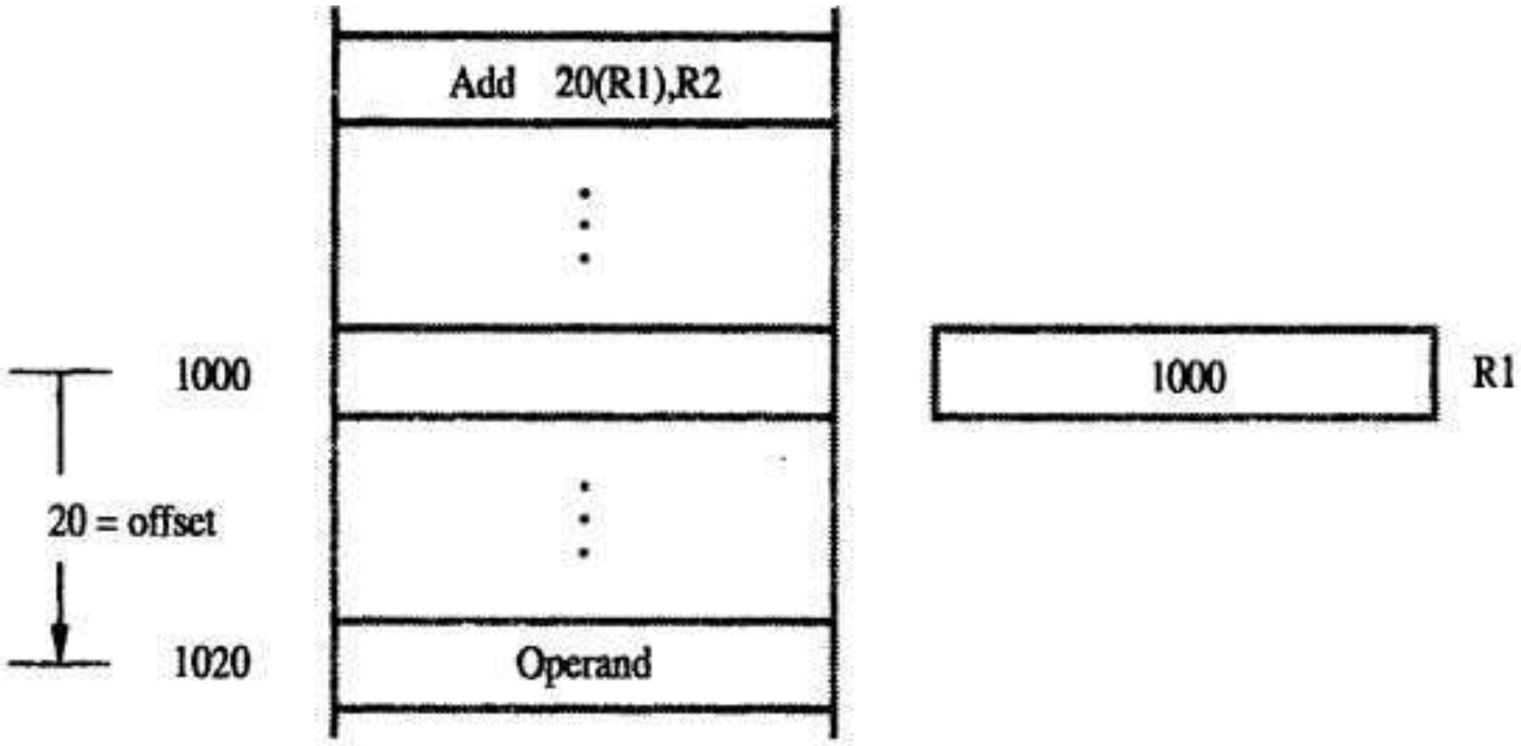
$$X(Ri)$$

where X denotes the constant value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

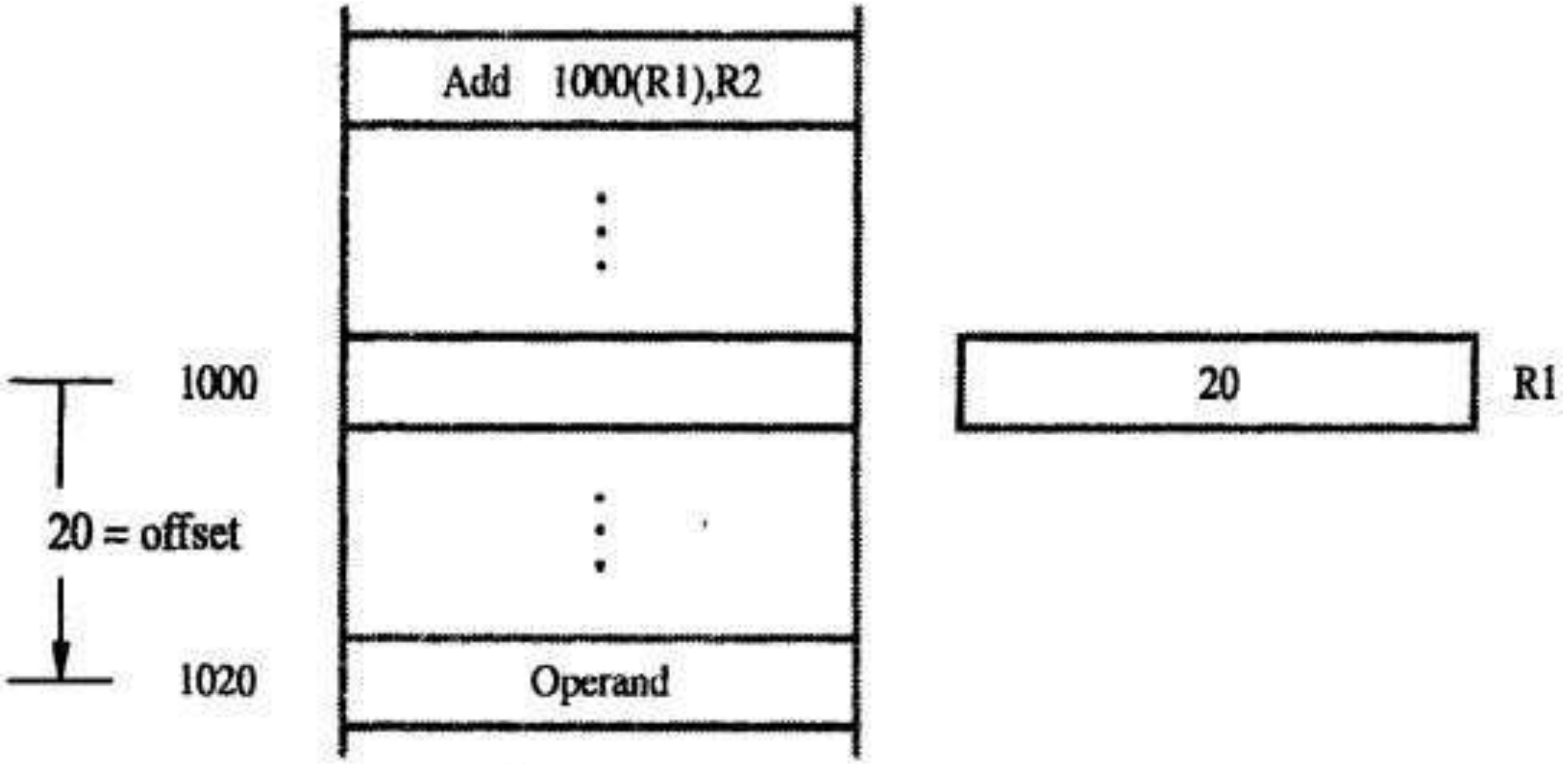
The contents of the index register are not changed in the process of generating the effective address.

the value X defines an *offset* (also called a *displacement*) from this address to the location where the operand is found.



(a) Offset is given as a constant

Figure 2.13 Indexed addressing.



(b) Offset is in the index register

Figure 2.13 Indexed addressing.

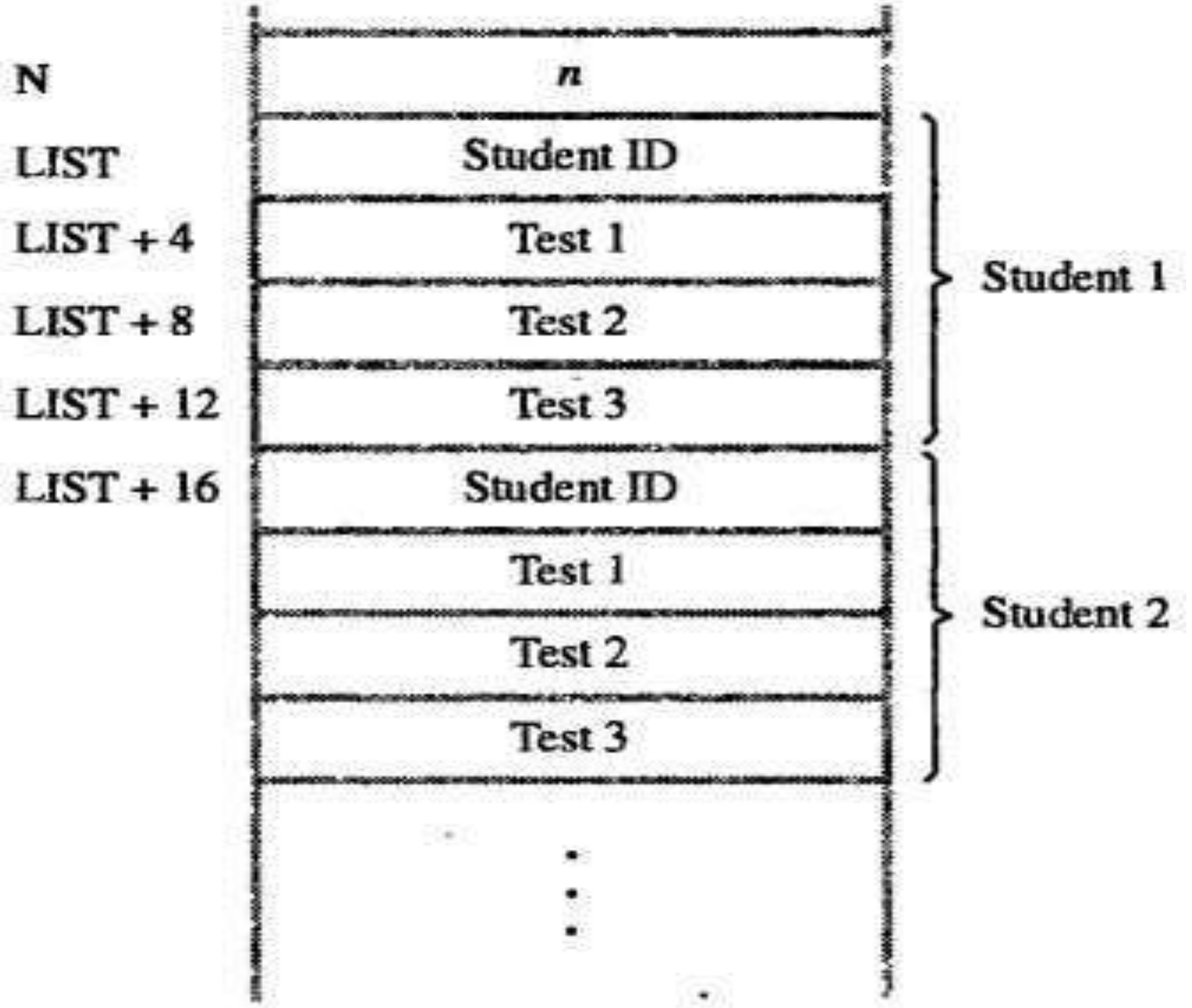


Figure 2.14 A list of students' marks.

```
        Move        #LIST,R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N,R4
LOOP    Add         4(R0),R1
        Add         8(R0),R2
        Add         12(R0),R3
        Add         #16,R0
        Decrement   R4
        Branch>0    LOOP
        Move        R1,SUM1
        Move        R2,SUM2
        Move        R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

We should note that the list in Figure 2.14 represents a two-dimensional array having *n* rows and four columns. Each row contains the entries for one student, and the columns give the IDs and test scores.

Suppose that we wish to compute the sum of all scores obtained on each of the tests and store these three sums in memory locations SUM1, SUM2, and SUM3. A possible program for this task is given in Figure 2.15. In the body of the loop, the program uses the Index addressing mode in the manner depicted in Figure 2.13*a* to access each of the three scores in a student's record. Register R0 is used as the index register. Before the loop is entered, R0 is set to point to the ID location of the first student record; thus, it contains the address LIST.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R1, R2, and R3, which are initially cleared to 0. These scores are accessed using the Index addressing modes 4(R0), 8(R0), and 12(R0). The index register R0 is then incremented by 16 to point to the ID location of the second student. Register R4, initialized to contain the value *n*, is decremented by 1 at the end of each pass through the loop. When the contents of R4 reach 0, all student records have been accessed, and the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R1, R2, and R3, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R0, are not changed when it is used in the Index addressing mode to access the scores. The contents of R0 are changed only by the last Add instruction in the loop, to move from one student record to the next.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

We have introduced the most basic form of indexed addressing. Several variations of this basic form provide for very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

$$(Ri,Rj)$$

The effective address is the sum of the contents of registers R*i* and R*j*. The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

As an example of where this flexibility may be useful, consider again the student record data structure shown in Figure 2.14. In the program in Figure 2.15, we used different index values in the three Add instructions at the beginning of the loop to access different test scores. Suppose each record contains a large number of items, many more than the three test scores of that example. In this case, we would need the ability to replace the three Add instructions with one instruction inside a second (nested) loop. Just as the successive starting locations of the records (the reference points) are maintained in the pointer register R0, offsets to the individual items relative to the contents of R0 could be maintained in another register. The contents of that register would be incremented in successive passes through the inner loop.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

$$X(Ri,Rj)$$

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri,Rj) part of the addressing mode. In other words, this mode implements a three-dimensional array.

Relative Addressing

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general-purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter. Since the addressed location is identified "relative" to the program counter, which always identifies the current execution point in a program, the name Relative mode is associated with this type of addressing.

Relative mode — The effective address is determined by the Index mode using the program counter in place of the general-purpose register R*i*.

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

Branch>0   LOOP

causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

Eg-

Assembly languages allow branch instructions to be written using labels to denote the branch target as shown in Figure 2.12. When the assembler program processes such an instruction, it computes the required offset value, −16 in this case, and generates the corresponding machine instruction using the addressing mode −16(PC).

Additional Modes

The two modes described in this are useful for accessing data items in successive locations in the memory.

- Autoincrement Mode (R*i*)+
- Autodecrement Mode -(R*i*)

Autoincrement mode — The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as

$$(Ri)+$$

Thus, the increment is 1 for byte-sized operand, 2 for 16-bit operands and 4 for 32-bit operands.

Autodecrement mode — The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

We denote the Autodecrement mode by putting the specified register in parentheses, preceded by a minus sign to indicate that the contents of the register are to be decremented before being used as the effective address. Thus, we write

$$-(Ri)$$

These two modes reduces the number of instructions to be executed to perform a specific task & mainly used in stack.

```
            Move        N,R1        }
            Move        #NUM1,R2    } Initialization
            Clear       R0          }
LOOP        Add         (R2)+,R0
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.


Thank You!