Addressing Modes

Carl Hamacher, Zvonko Vranesic and Safwat Zaky, Computer Organization and Embedded Systems, (6e), McGraw Hill Publication, 2017.

Addressing modes

- A program operates on data that reside in the computer’s memory.
- These data can be organized in a variety of ways that reflect the nature of the information and how it is used.
- Programmers use data structures such as lists and arrays for organizing the data used in computations.
- Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures.
- When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations.
- The different ways for specifying the locations of instruction operands are known as **addressing modes.**

Addressing Modes

- **Addressing modes:**
  - Different ways for specifying the locations of instruction operands.
- RISC-style processors basic addressing modes Table 2.1
- The assembler syntax defines the way in which instructions and the addressing modes of their operands are specified

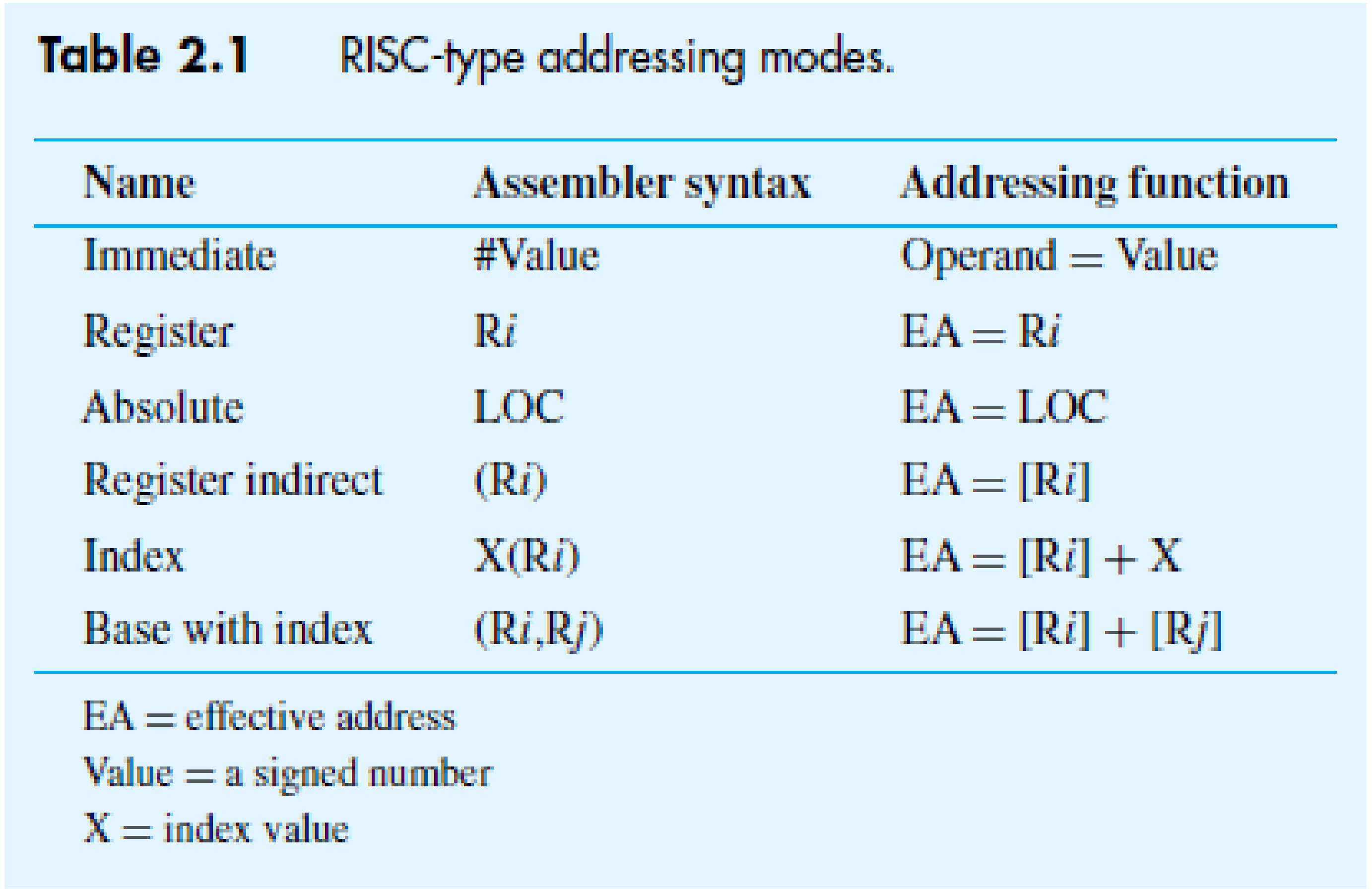


Table 2.1 RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute	LOC	EA = LOC
Register indirect	(Ri)	EA = [Ri]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]

EA = effective address
Value = a signed number
X = index value

Implementation of Variables and Constants

- Register mode
  - The operand is the contents of a processor register; the name of the register is given in the instruction.
  - Example: Add R4, R2, R3
- Absolute mode
  - The operand is in a memory location; the address of this location is given explicitly in the instruction.
  - Load R2, NUM1
- Immediate mode
  - The operand is given explicitly in the instruction.
  - Add R4, R6, $200_{\text{immediate}}$
  - Add R4, R6, #200

Since in a RISC-style processor an instruction must fit in a single word, the number of bits that can be used to give an absolute address is limited, typically to 16 bits if the word length is 32 bits.

Indirection and Pointers

- Indirect mode:
  - The effective address of the operand is the contents of a register that is specified in the instruction
  - Load R2, (R5)

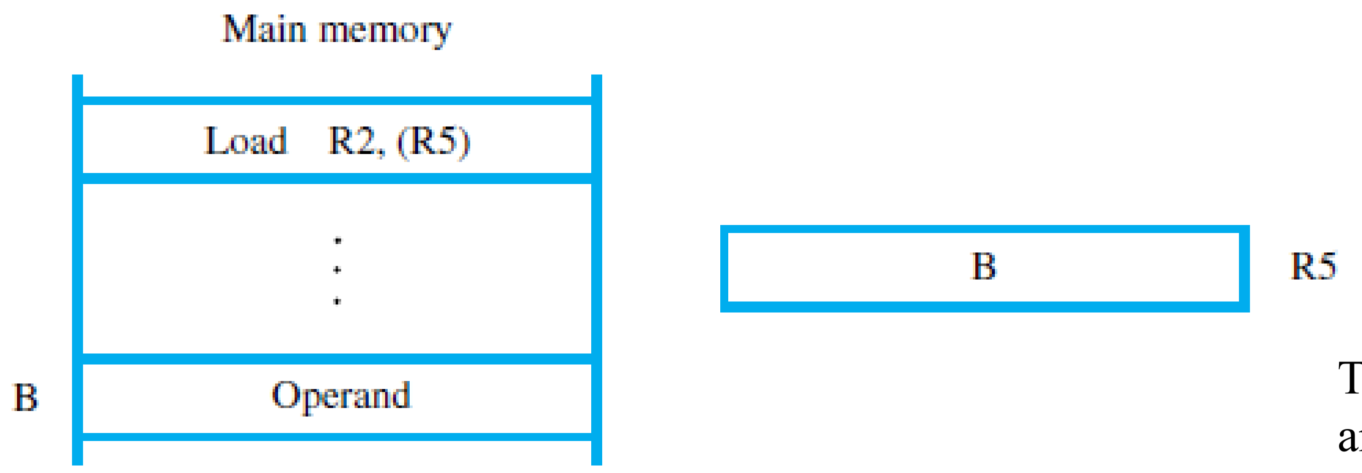


Figure 2.7 Register indirect addressing.

The register acts as a *pointer* to the list, and we say that an item in the list is accessed *indirectly* by using the address in the register. The desired capability is provided by the indirect addressing mode.

Indirection and Pointers

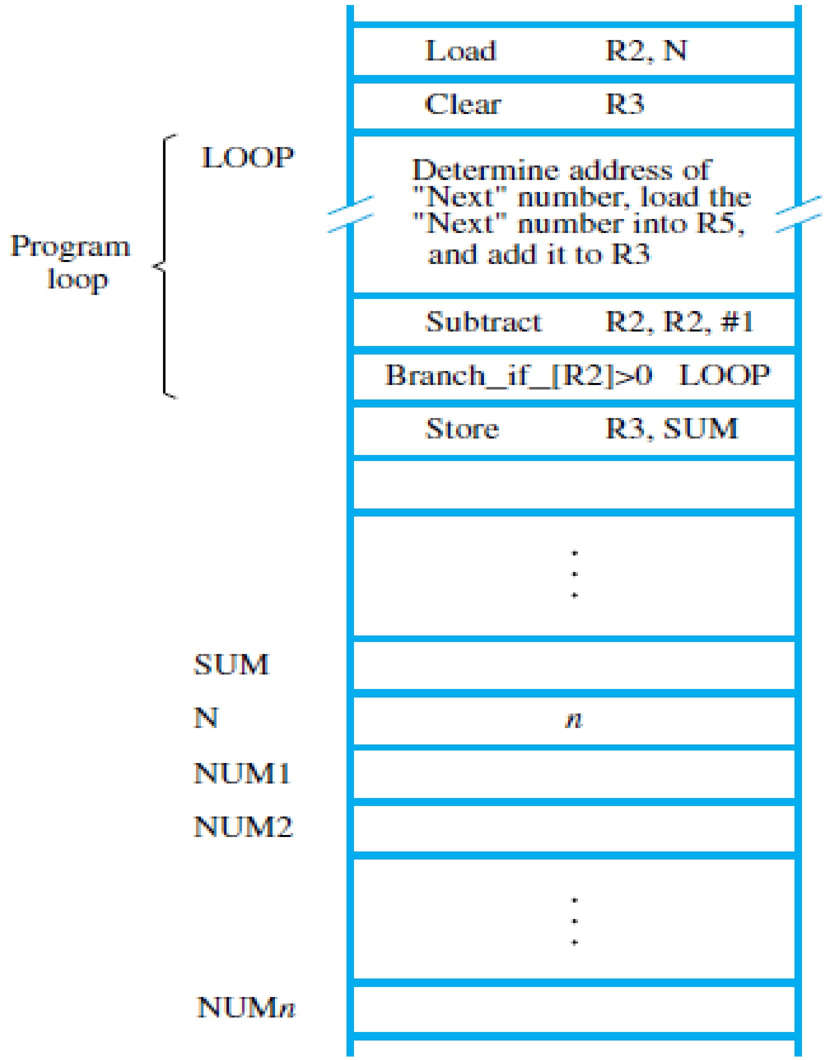


Figure 2.6 Using a loop to add *n* numbers.

```
         Load              R2, N        Load the size of the list.
         Clear             R3           Initialize sum to 0.
         Move              R4, #NUM1    Get address of the first number.
LOOP:    Load              R5, (R4)     Get the next number.
         Add               R3, R3, R5   Add this number to sum.
         Add               R4, R4, #4   Increment the pointer to the list.
         Subtract          R2, R2, #1   Decrement the counter.
         Branch_if_[R2]>0  LOOP         Branch back if not finished.
         Store             R3, SUM      Store the final sum.
```

Figure 2.8 Use of indirect addressing in the program of Figure 2.6.

Indirection and Pointers

Another Example

- C-language statement

  A = *B; where B is a pointer variable and the '*' symbol is the operator for indirect accesses.

- Compiled to:

  Load R2, B // loads the content in B i.e., the address of location where B is pointing at (ex: AAA0)

  Load R3, (R2) //go to location AAA0 and copy the data to R3

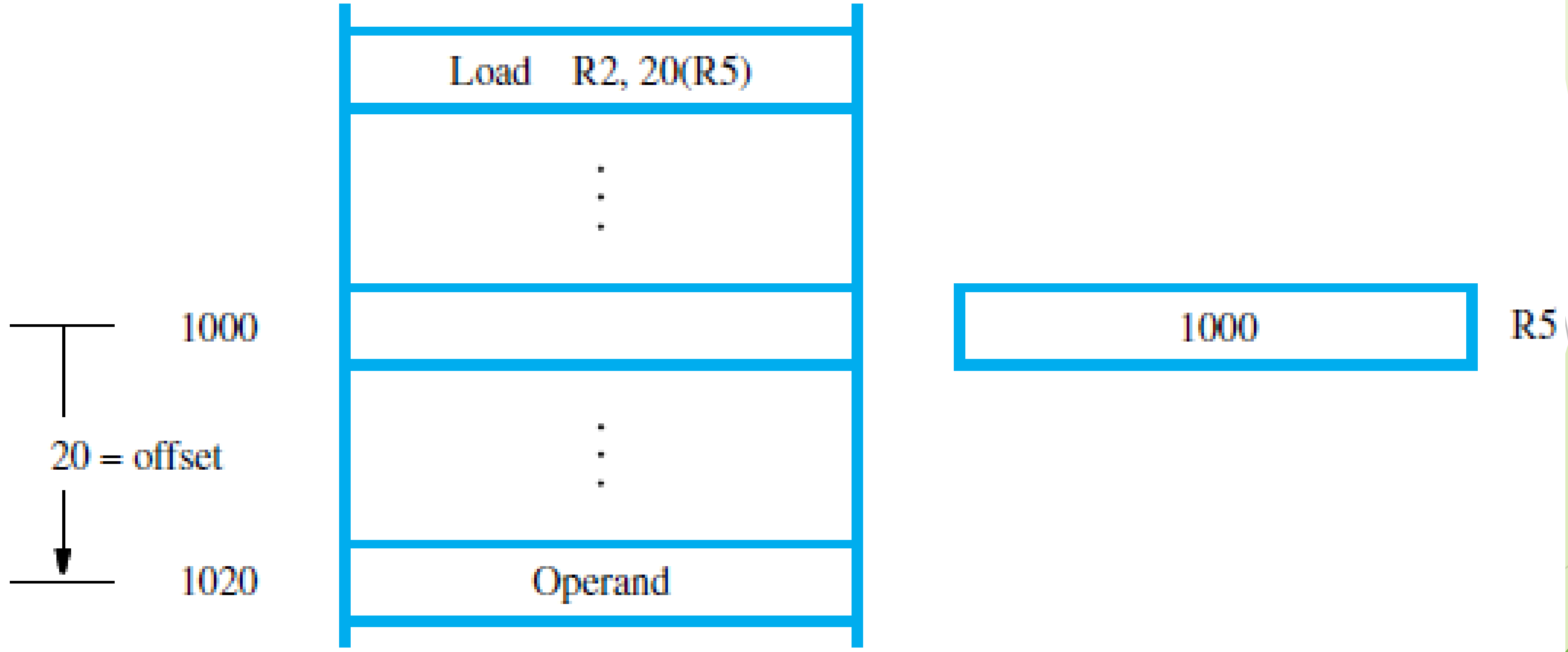
  Store R3, A // store the content of R3 in A

Indexing and Arrays

- Index mode:
  - the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register: The register used in this mode is known as the index register
- X(Ri): EA = X + [Ri]
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.

Index Mode: Types

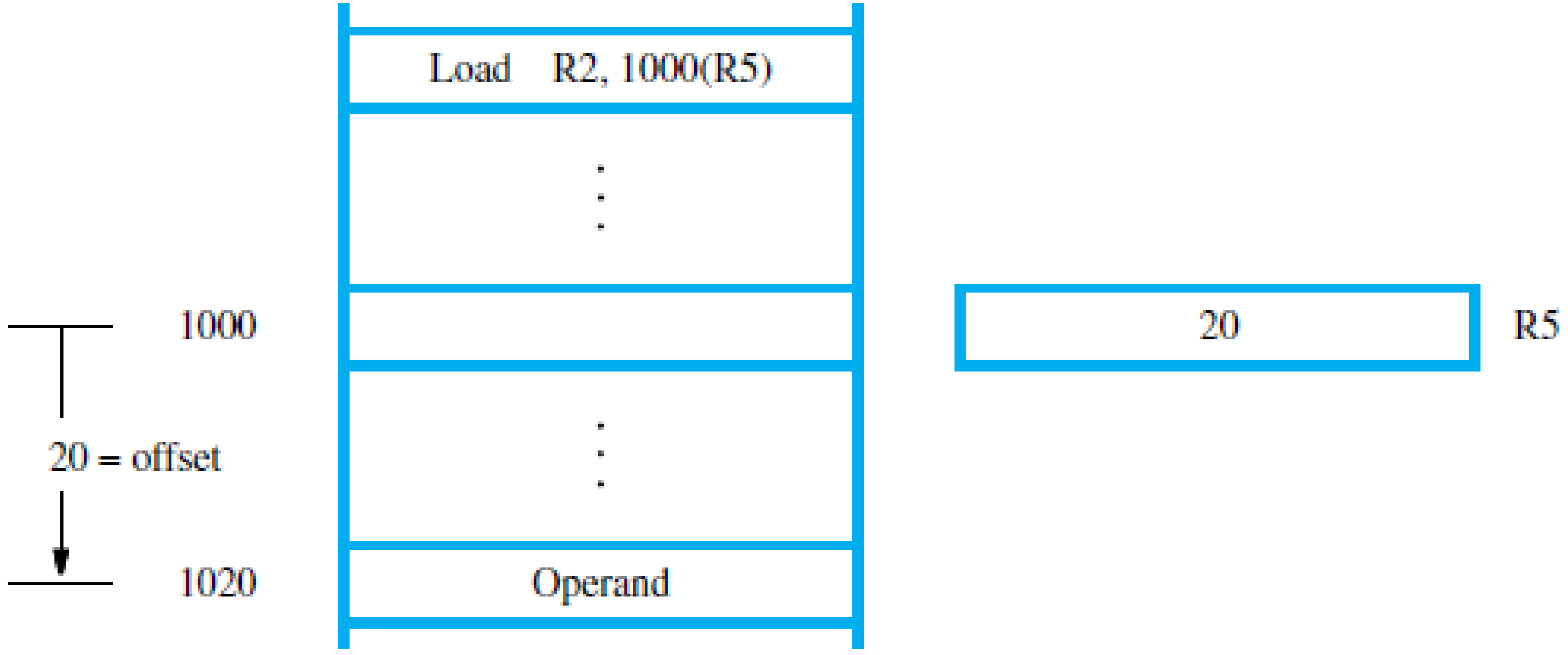
- Offset is given as a constant



(a) Offset is given as a constant

Index Mode: Types

- Offset is in the index register



(b) Offset is in the index register

Indexed Addressing: Example

- 2D array representation

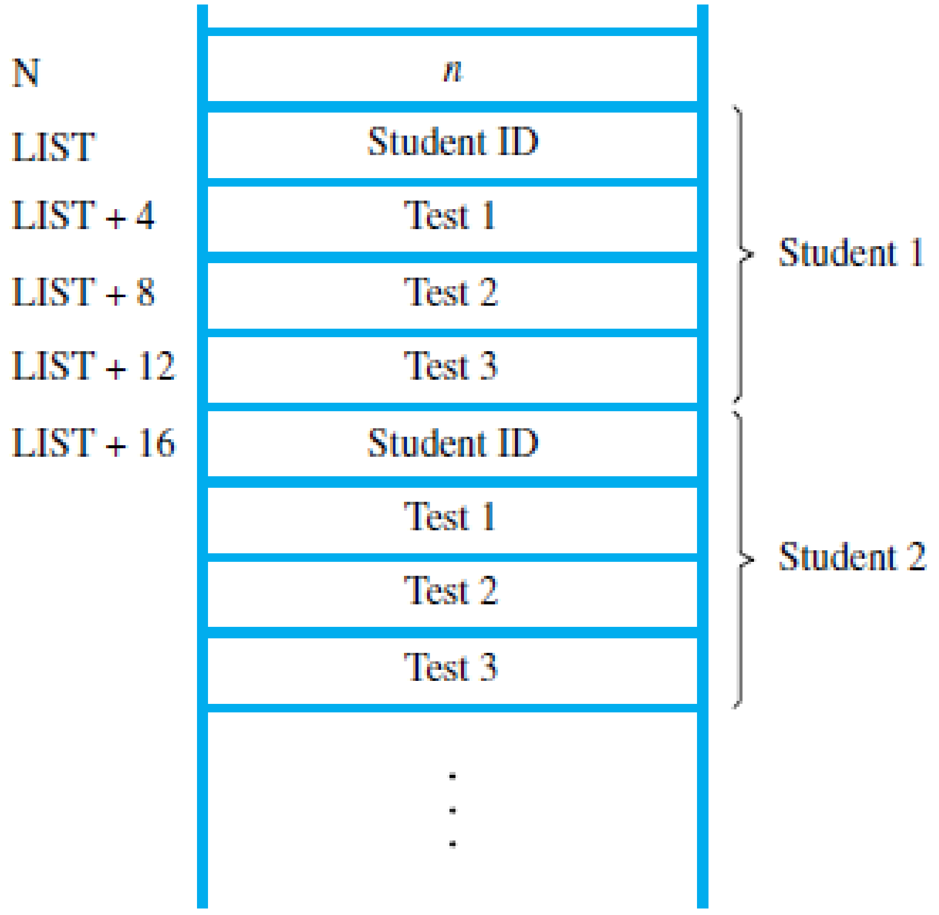


Figure 2.10 A list of students' marks.

Indexed Addressing: Example

```
        Move              R2, #LIST      Get the address LIST.
        Clear             R3
        Clear             R4
        Clear             R5
        Load              R6, N          Load the value n.
LOOP:   Load              R7, 4(R2)      Add the mark for next student's
        Add               R3, R3, R7       Test 1 to the partial sum.
        Load              R7, 8(R2)      Add the mark for that student's
        Add               R4, R4, R7       Test 2 to the partial sum.
        Load              R7, 12(R2)     Add the mark for that student's
        Add               R5, R5, R7       Test 3 to the partial sum.
        Add               R2, R2, #16    Increment the pointer.
        Subtract          R6, R6, #1     Decrement the counter.
        Branch_if_[R6]>0  LOOP           Branch back if not finished.
        Store             R3, SUM1       Store the total for Test 1.
        Store             R4, SUM2       Store the total for Test 2.
        Store             R5, SUM3       Store the total for Test 3.
```

Figure 2.11 Indexed addressing used in accessing test scores in the list in Figure 2.10.

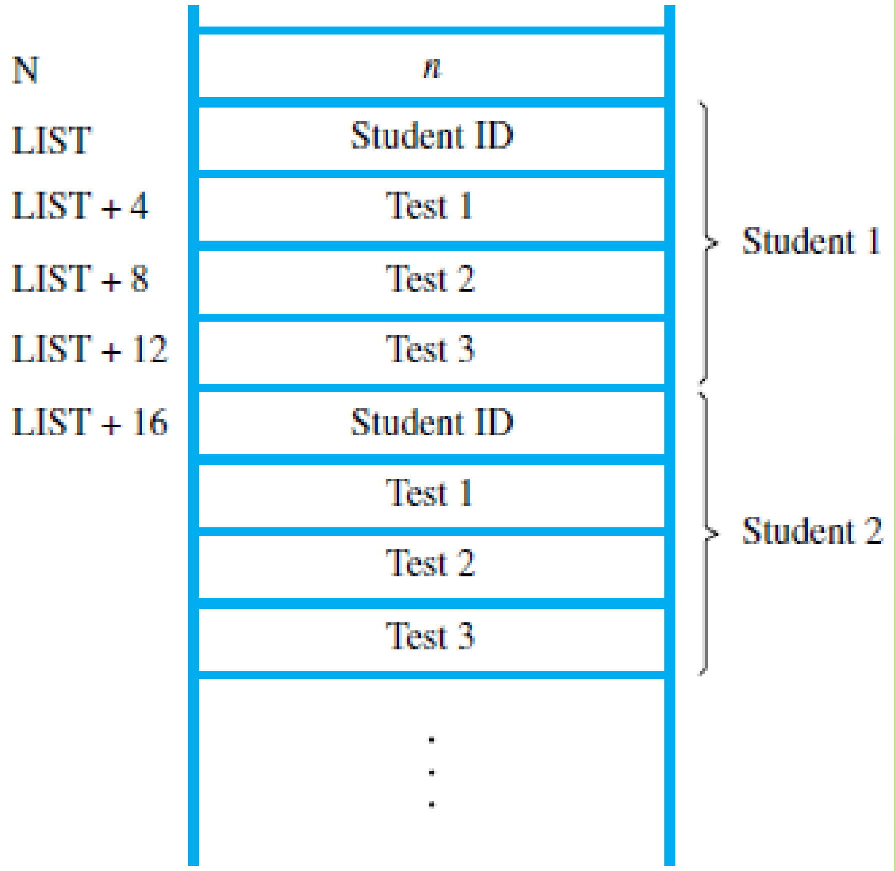


Figure 2.10 A list of students' marks.

Indexed Addressing: Variations

- Base with Index: (R*i*,R*j*)
  - EA = [Ri] + [Rj]

- Base with Index plus constant: $X(R_i, R_j)$
  - $EA = X + [R_i] + [R_j]$

RISC Style

- RISC style is characterized by:
  - Simple addressing modes
  - All instructions fitting in a single word
  - Fewer instructions in the instruction set, because of simple addressing modes
  - Arithmetic and logic operations that can be performed only on operands in processor registers
  - Load/store architecture that does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
  - Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining
  - Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

CISC Style

- CISC style is characterized by:
  - More complex addressing modes
  - More complex instructions, where an instruction may span multiple words
  - Many instructions that implement complex tasks
  - Arithmetic and logic operations that can be performed on memory operands as well as operands in processor registers
  - Transfers from one memory location to another by using a single Move instruction
  - Programs that tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks

CISC Instruction Set

- CISC instruction sets are not constrained to the load/store architecture, in which arithmetic and logic operations can be performed only on operands that are in processor registers.
- Instructions do not necessarily have to fit into a single word.
- Some instructions may occupy a single word, but others may span multiple words
- Most arithmetic and logic instructions use the two-address format

  Operation destination, source
- An Add instruction of type

  Add B, A
- is written as

  $B \leftarrow [A] + [B]$

CISC Instruction Set: Example

- Consider the task of adding two numbers where all three operands may be in memory locations

  C = A + B

- This cannot be done with a single two-address instruction.
- Another two-address instruction is required that copies the contents of one memory location into another.

  Move C, B

- which performs the operation C←[B] (contents of location B is unchanged).
- The operation C←[A] + [B] can now be performed by the two-instruction sequence

  Move C, B

  Add C, A

CISC Instruction Set

- In some CISC processors one operand may be in the memory but the other must be in a register.
- In this case, the instruction sequence for the required task would be

  Move Ri, A

  Add Ri, B

  Move C, Ri
- The general form of the Move instruction is

  Move destination, source
- where both the source and destination may be either a memory location or a processor register

Additional Addressing Modes

- Autoincrement mode: (Ri)+
  - 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 operand in memory

- Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand.
- Increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

- Autodecrement mode: −(Ri)
  - The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand

Additional Addressing Modes

- To push a new item on the stack,

  Subtract SP, #4

  Move (SP), NEWITEM
- just one instruction can be used

  Move −(SP), NEWITEM

- Similarly, to pop an item from the stack,

  Move ITEM, (SP)

  Add SP, #4
- We can use just

  Move ITEM, (SP)+

Additional Addressing Modes

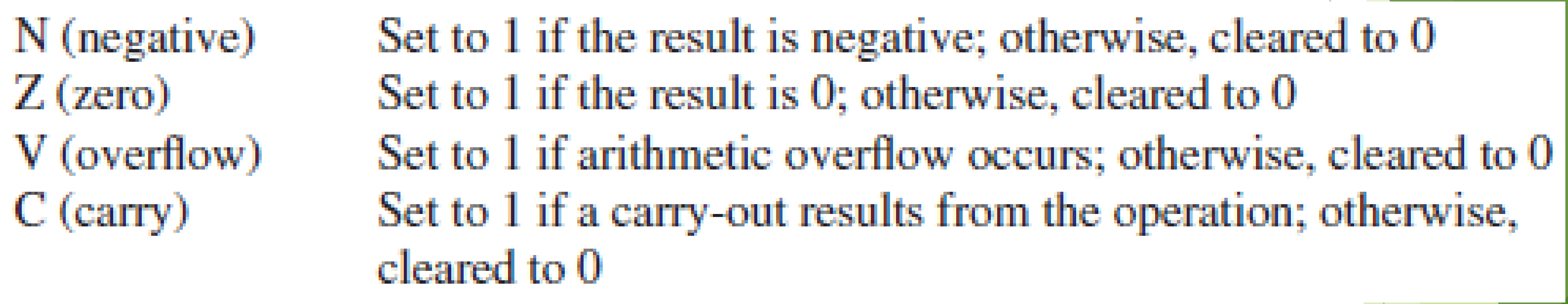
- Relative mode:
  - the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
  - X(PC) – note that X is a signed number

Condition Codes

- Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero
- Maintain the information about these results for use by subsequent conditional branch instructions
- Accomplished by recording the required information in individual bits, often called condition code flags
- These flags are usually grouped together in a special processor register called the condition code register or status register.
- Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed

Condition Codes

- Commonly used flags:

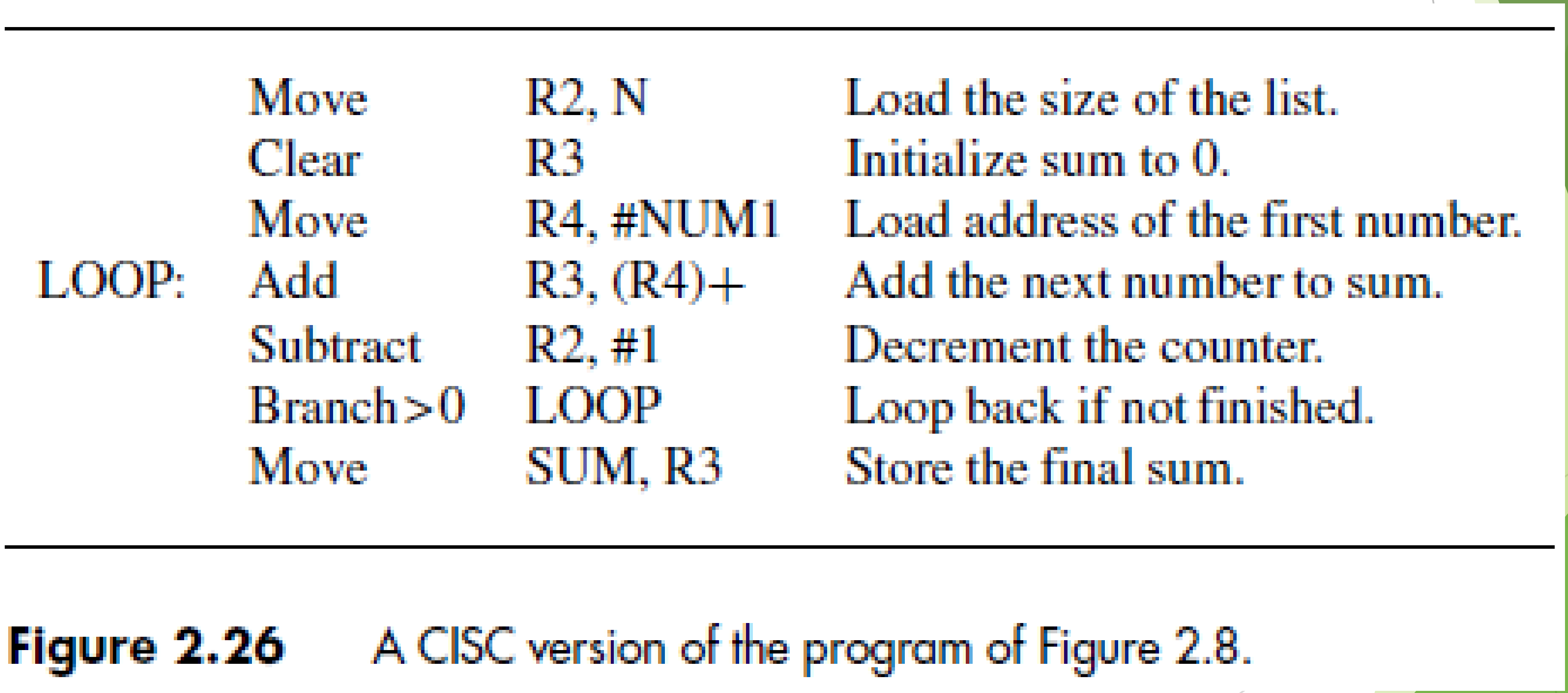


N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

- e.g. Branch>0 LOOP
- This instruction causes a branch if both N and Z are 0, that is, if the result produced by previous arithmetic instruction is neither negative nor equal to zero

Condition Codes

- CISC style programming to add a list of numbers

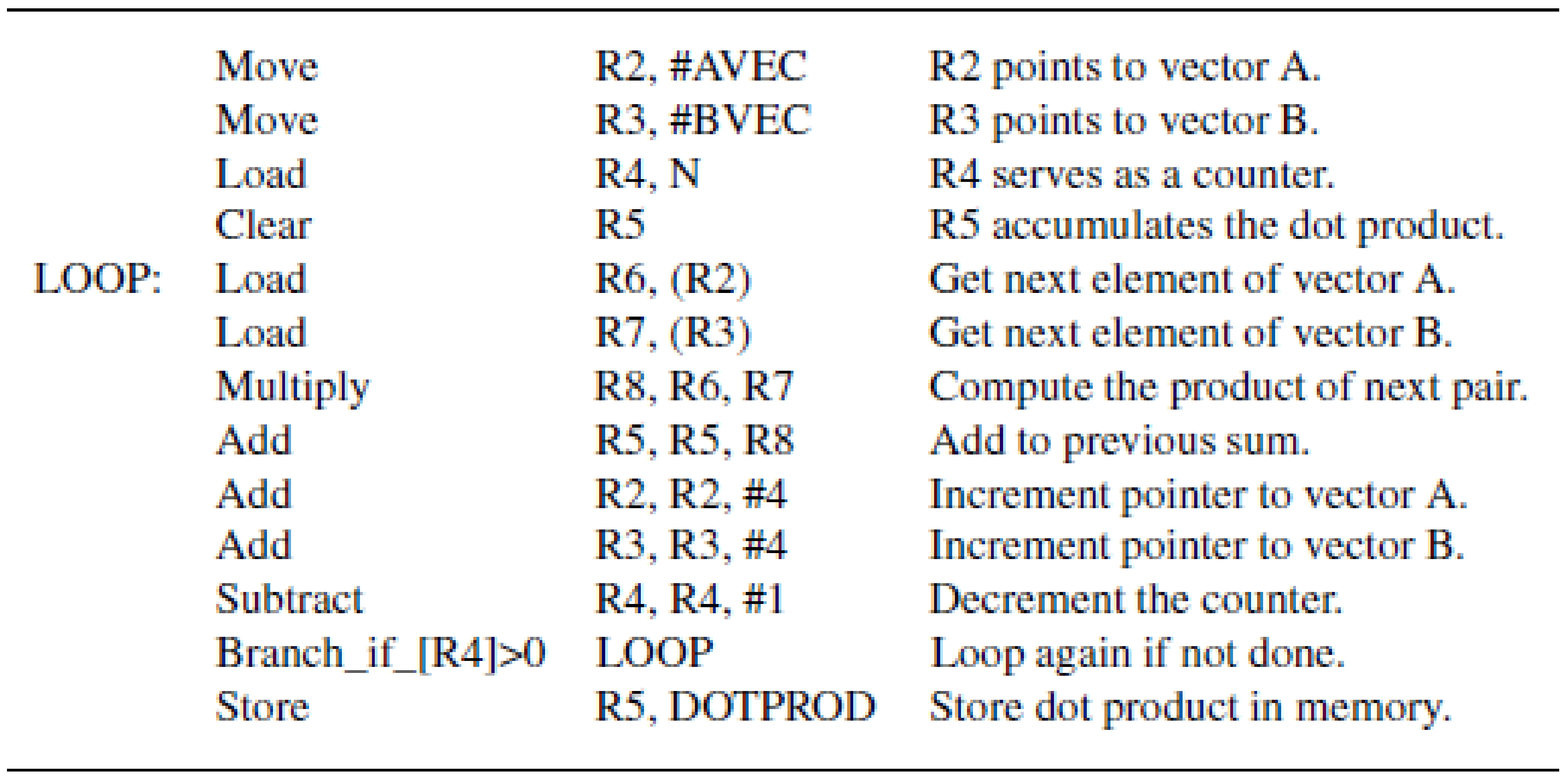


```
        Move        R2, N        Load the size of the list.
        Clear       R3           Initialize sum to 0.
        Move        R4, #NUM1    Load address of the first number.
LOOP:   Add         R3, (R4)+    Add the next number to sum.
        Subtract    R2, #1       Decrement the counter.
        Branch>0    LOOP         Loop back if not finished.
        Move        SUM, R3      Store the final sum.
```

Figure 2.26 A CISC version of the program of Figure 2.8.

Example Program: Vector Dot Product Program (RISC style)

- Dot Product $=\sum_{i=0}^{n-1} A(i) \times B(i)$

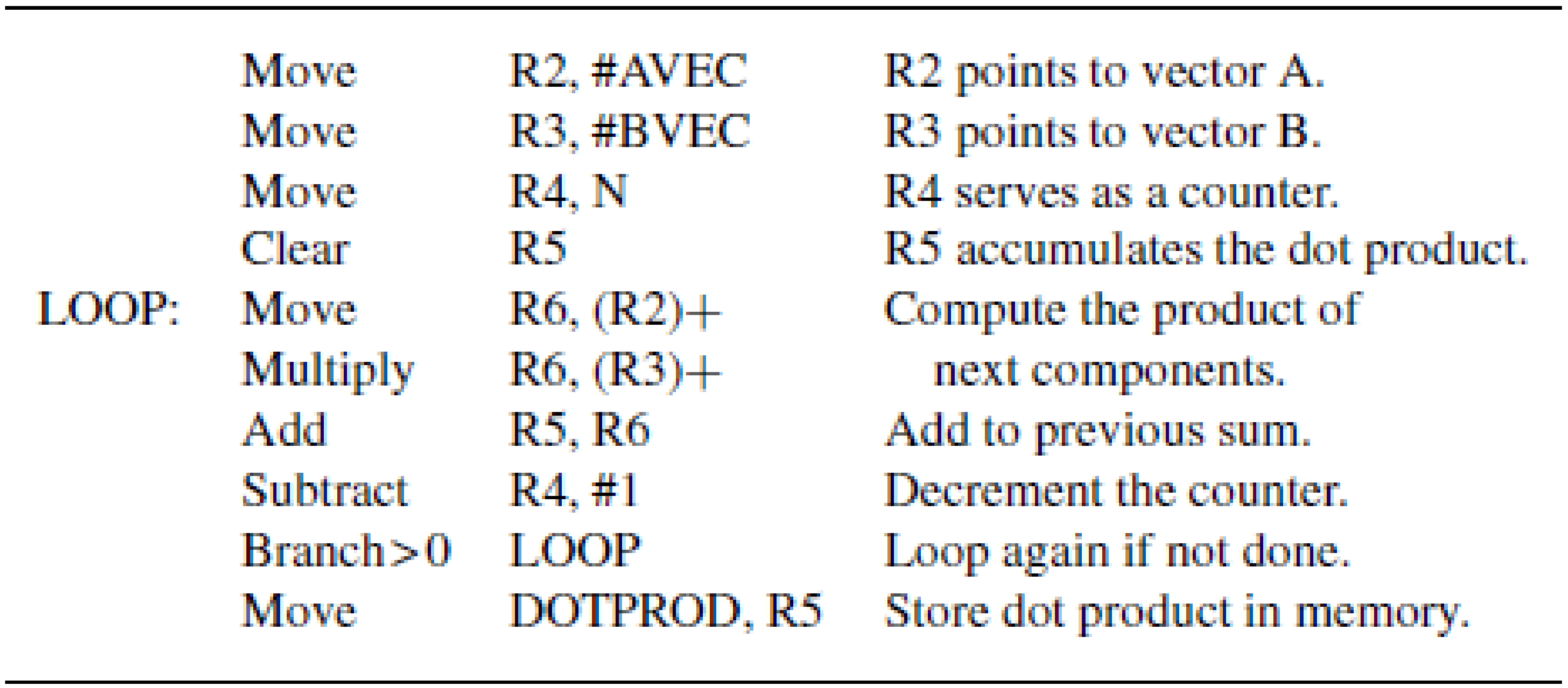


```
         Move              R2, #AVEC       R2 points to vector A.
         Move              R3, #BVEC       R3 points to vector B.
         Load              R4, N           R4 serves as a counter.
         Clear             R5              R5 accumulates the dot product.
LOOP:    Load              R6, (R2)        Get next element of vector A.
         Load              R7, (R3)        Get next element of vector B.
         Multiply          R8, R6, R7      Compute the product of next pair.
         Add               R5, R5, R8      Add to previous sum.
         Add               R2, R2, #4      Increment pointer to vector A.
         Add               R3, R3, #4      Increment pointer to vector B.
         Subtract          R4, R4, #1      Decrement the counter.
         Branch_if_[R4]>0  LOOP            Loop again if not done.
         Store             R5, DOTPROD     Store dot product in memory.
```

Figure 2.27 A RISC-style program for computing the dot product of two vectors.

Example Program: Vector Dot Product Program (CISC style)

- Dot Product $=\sum_{i=0}^{n-1} A(i) \times B(i)$



```
         Move        R2, #AVEC      R2 points to vector A.
         Move        R3, #BVEC      R3 points to vector B.
         Move        R4, N          R4 serves as a counter.
         Clear       R5             R5 accumulates the dot product.
LOOP:    Move        R6, (R2)+      Compute the product of
         Multiply    R6, (R3)+        next components.
         Add         R5, R6         Add to previous sum.
         Subtract    R4, #1         Decrement the counter.
         Branch>0    LOOP           Loop again if not done.
         Move        DOTPROD, R5    Store dot product in memory.
```

Figure 2.28 A CISC-style program for computing the dot product of two vectors.