COMPUTER ORGANIZATION AND ARCHITECTURE

Course Code : CSE 2151

Credits : 04

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)

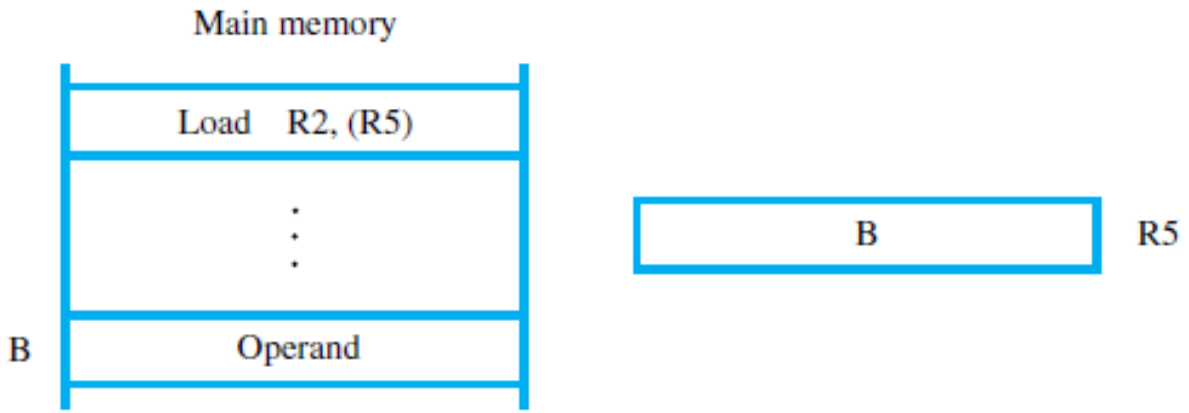


Figure 2.7 Register indirect addressing.

INDIRECTION AND POINTERS

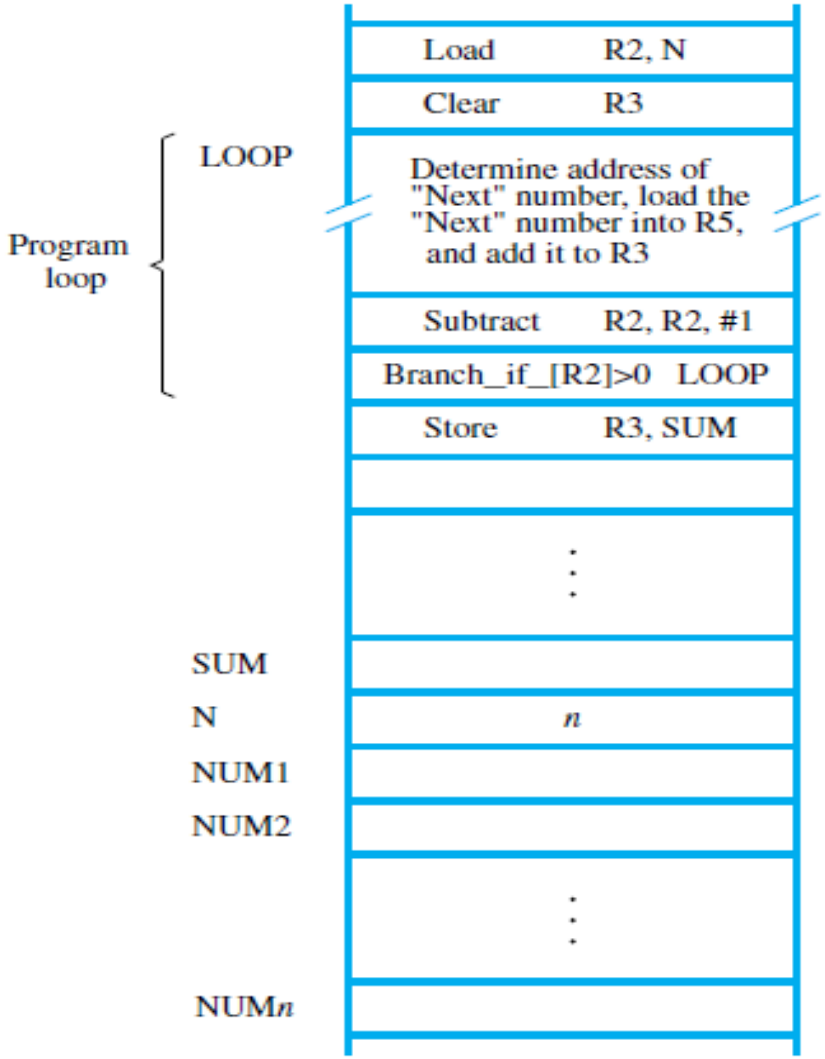


Figure 2.6 Using a loop to add *n* numbers.

```
        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

- C-language statement

```
A = *B;
```

- Compiled to:

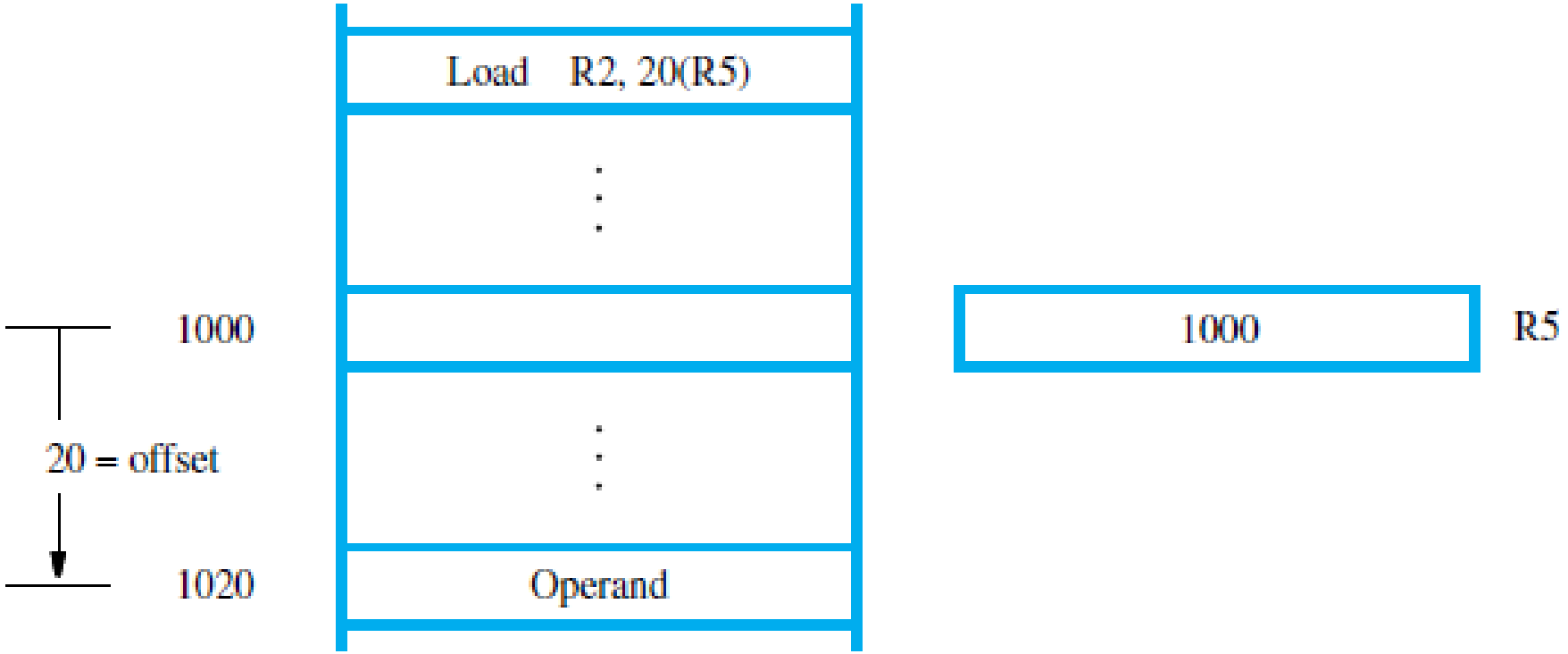
```
Load R2, B        // loads the content in B i.e., the address of location where B is pointing at (ex: AAA0)
Load R3, (R2)     //go to location AAA0 and copy the data to R3
Store R3, A       // store the content of R3 in A
```

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

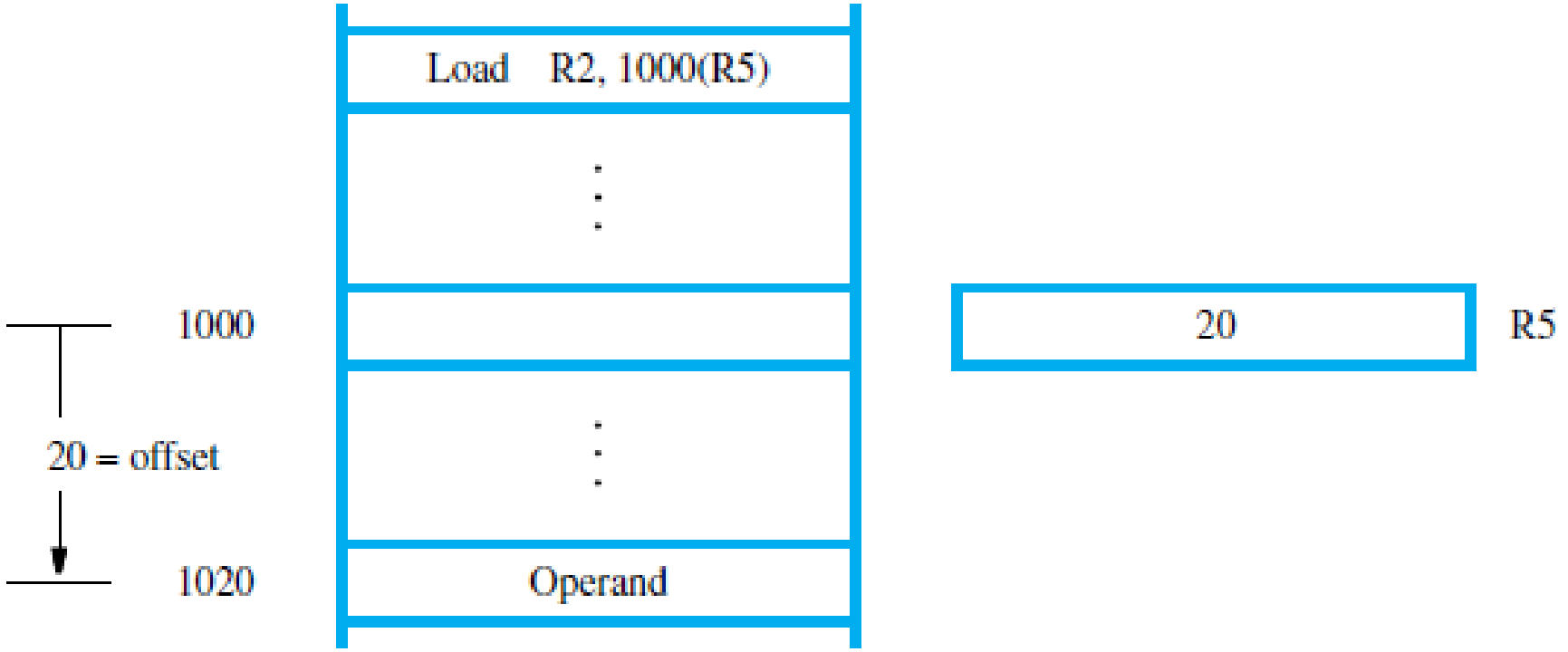
- Offset is given as a constant



(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

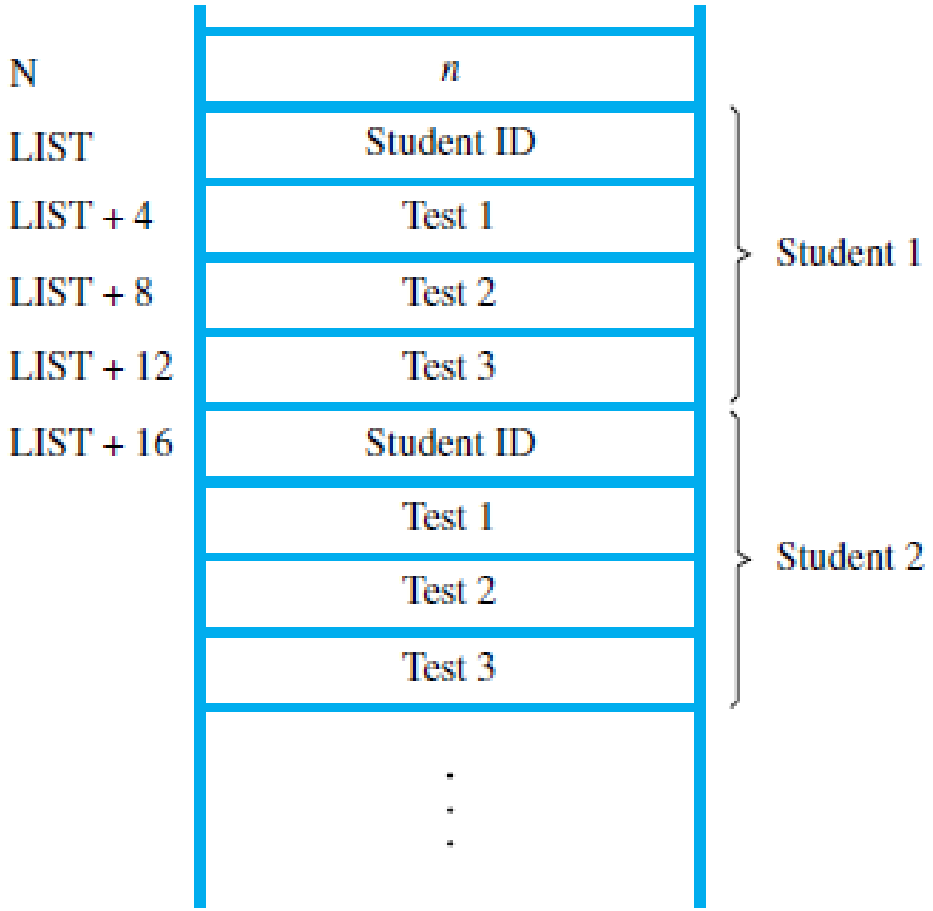


Figure 2.10 A list of students' marks.

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.

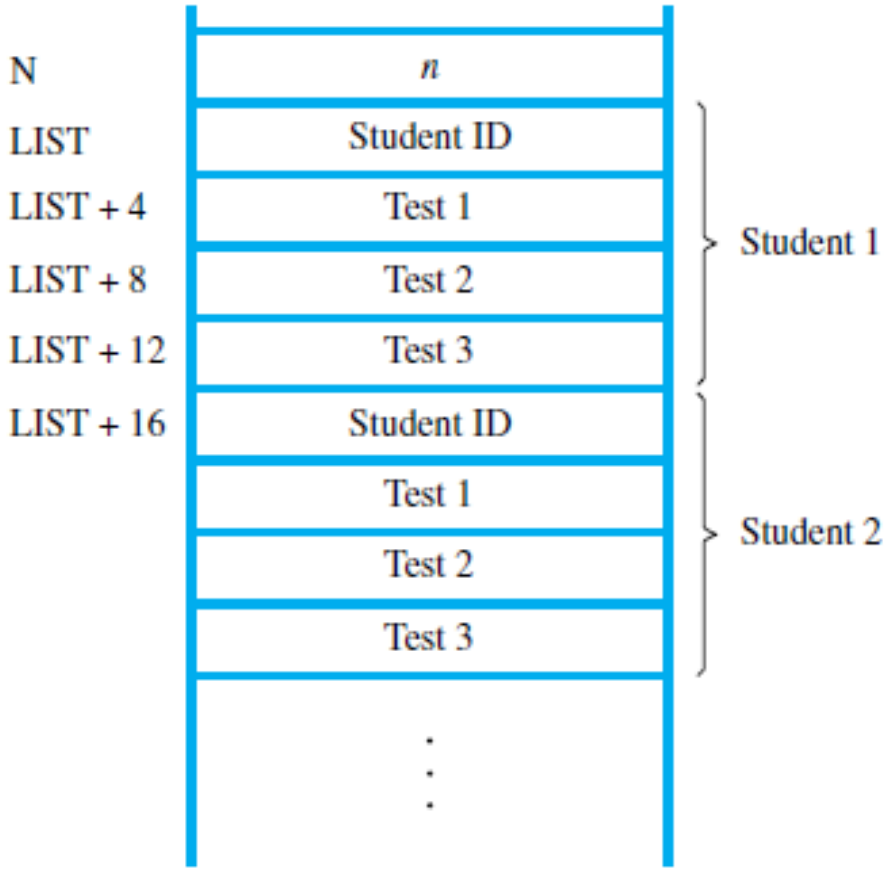


Figure 2.10 A list of students' marks.

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 + $[R_i]$ + $[R_j]$

ADDRESSING MODES

Table 2.1 RISC-type addressing modes.

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

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

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 ← [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: 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: 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.

TOPICS COVERED FROM

- Textbook 1:
  - Chapter 2: 2.4, 2.10, 2.11, 2.12