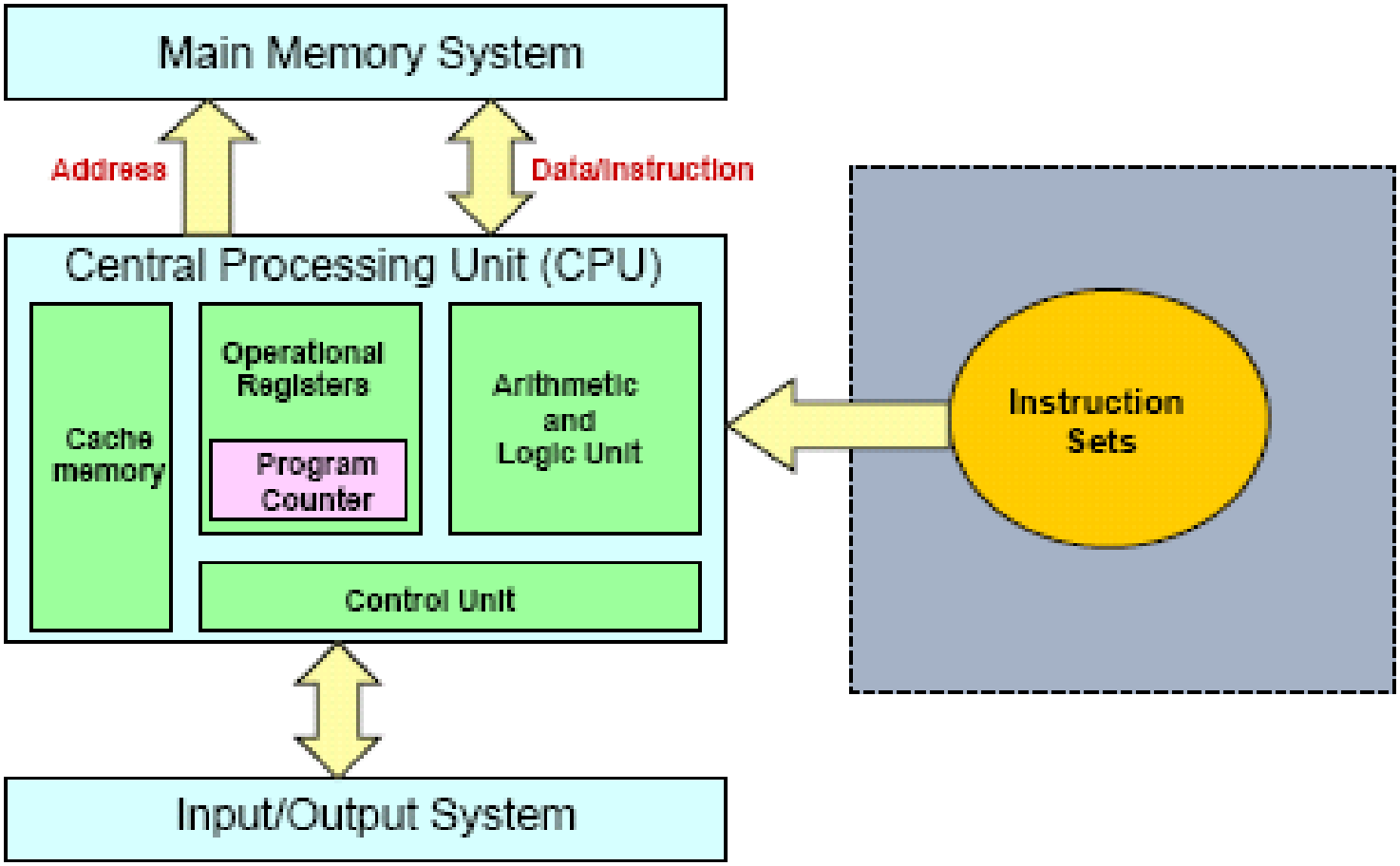
Chapter 2

Machine Instructions and Programs

Outline

- Numbers, Arithmetic Operations, and Characters
- Memory Locations and Addresses
- Memory Operation
- Instructions and Instruction Sequencing
- Addressing Modes
- Assembly Language
- Basic Input/Output Operations
- Stacks and Queues
- Subroutines
- Linked List
- Encoding of Machine Instructions

Content Coverage



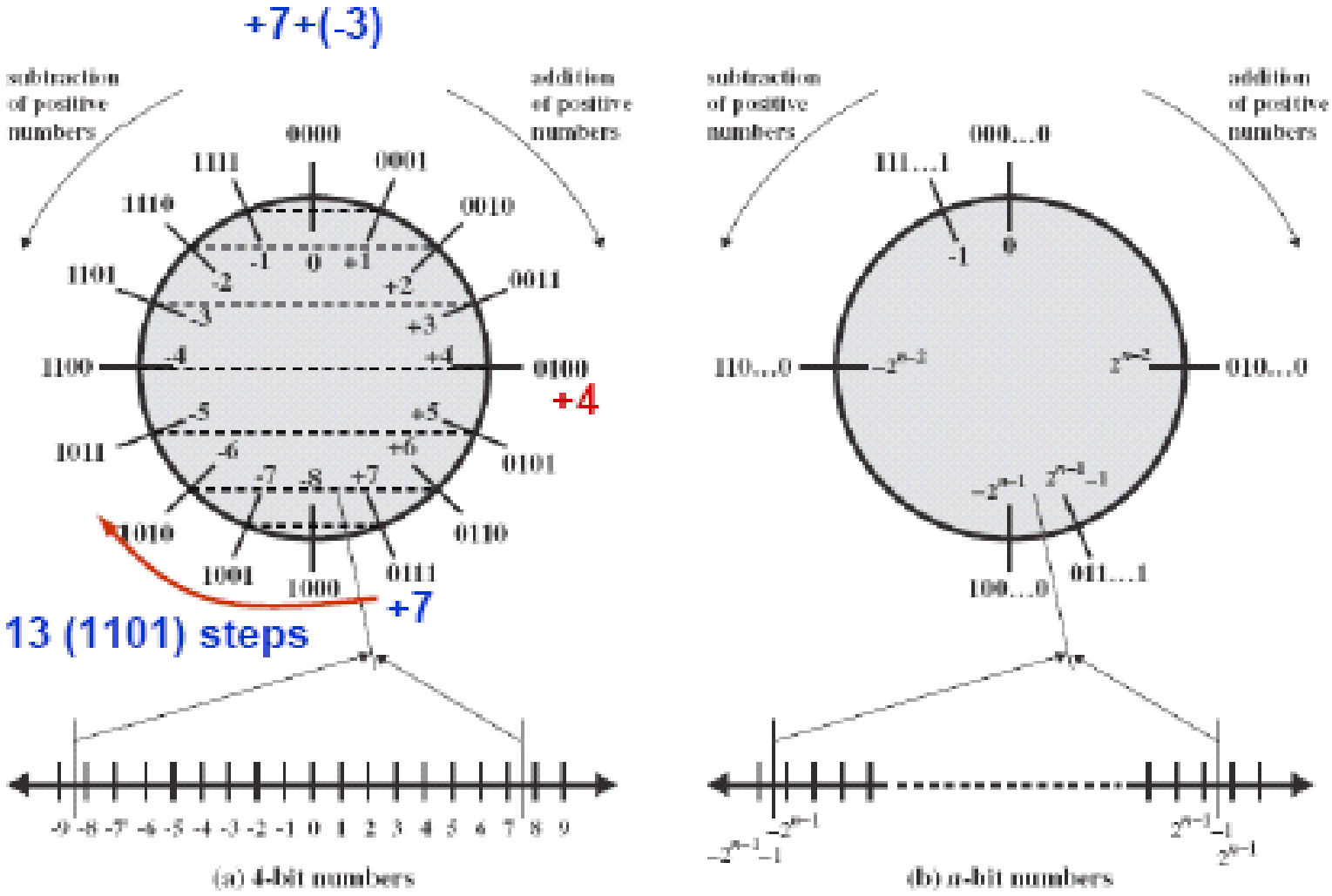
Number Representation

- Consider an *n*-bit vector $B = b_{n-1} \ldots b_1 b_0$, where b_i=0 or 1 for $0 \leq i \leq n-1$
- The vector B can represent unsigned integer values V in the range 0 to $2^n - 1$, where
  - $V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$
- We need to represent positive and negative numbers for most applications
- Three systems are used for representing such numbers
  - Sign-and-magnitude
  - 1's-complement
  - 2's-complement

Number Representation

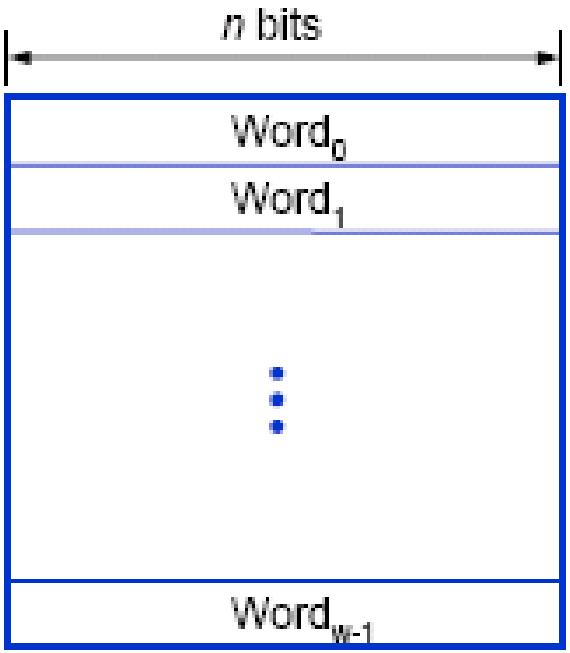
$b_3 b_2 b_1 b_0$	sign and magnitude	1's-complement	2's-complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

2's-Complement System



Memory Words

Memory words



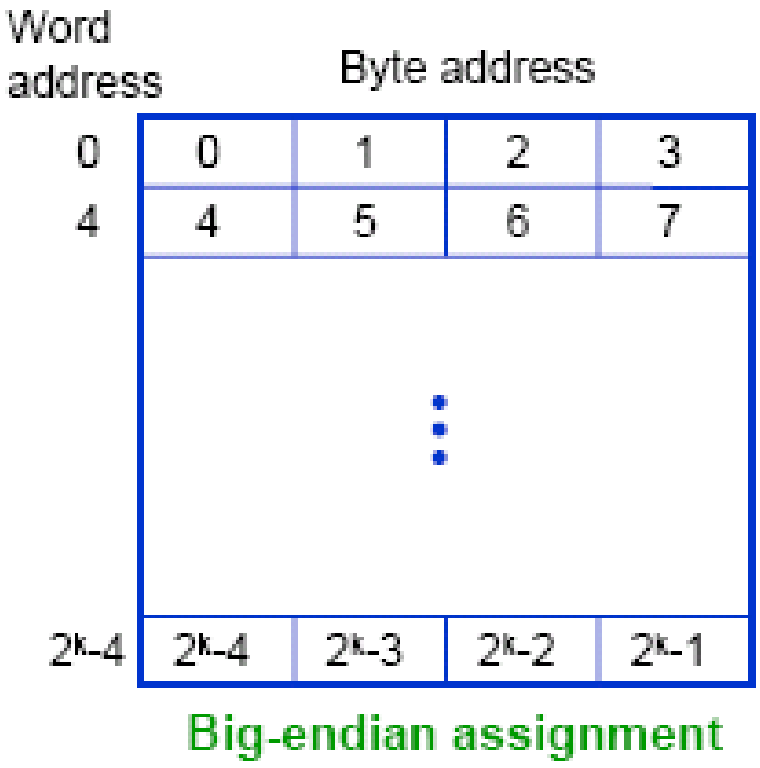
A signed integer

Four characters

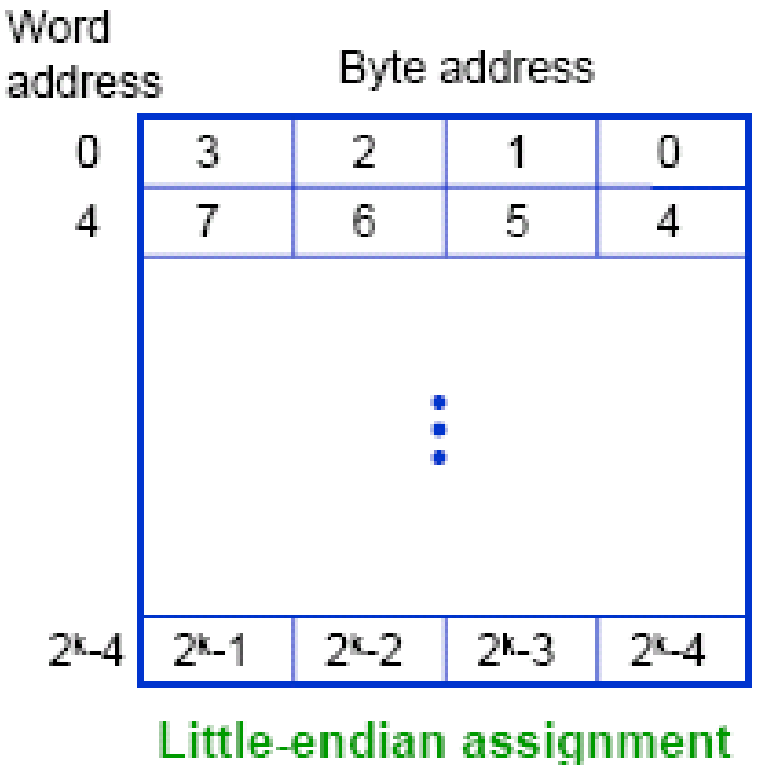
ASCII
character

Big-Endian & Little-Endian Assignments

- Byte addresses can be assigned across words in two ways
  - Big-endian and little-endian



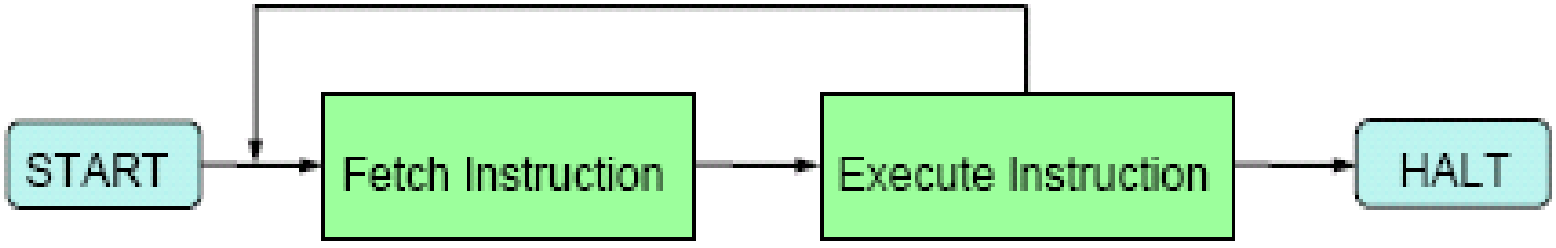
Big-endian assignment



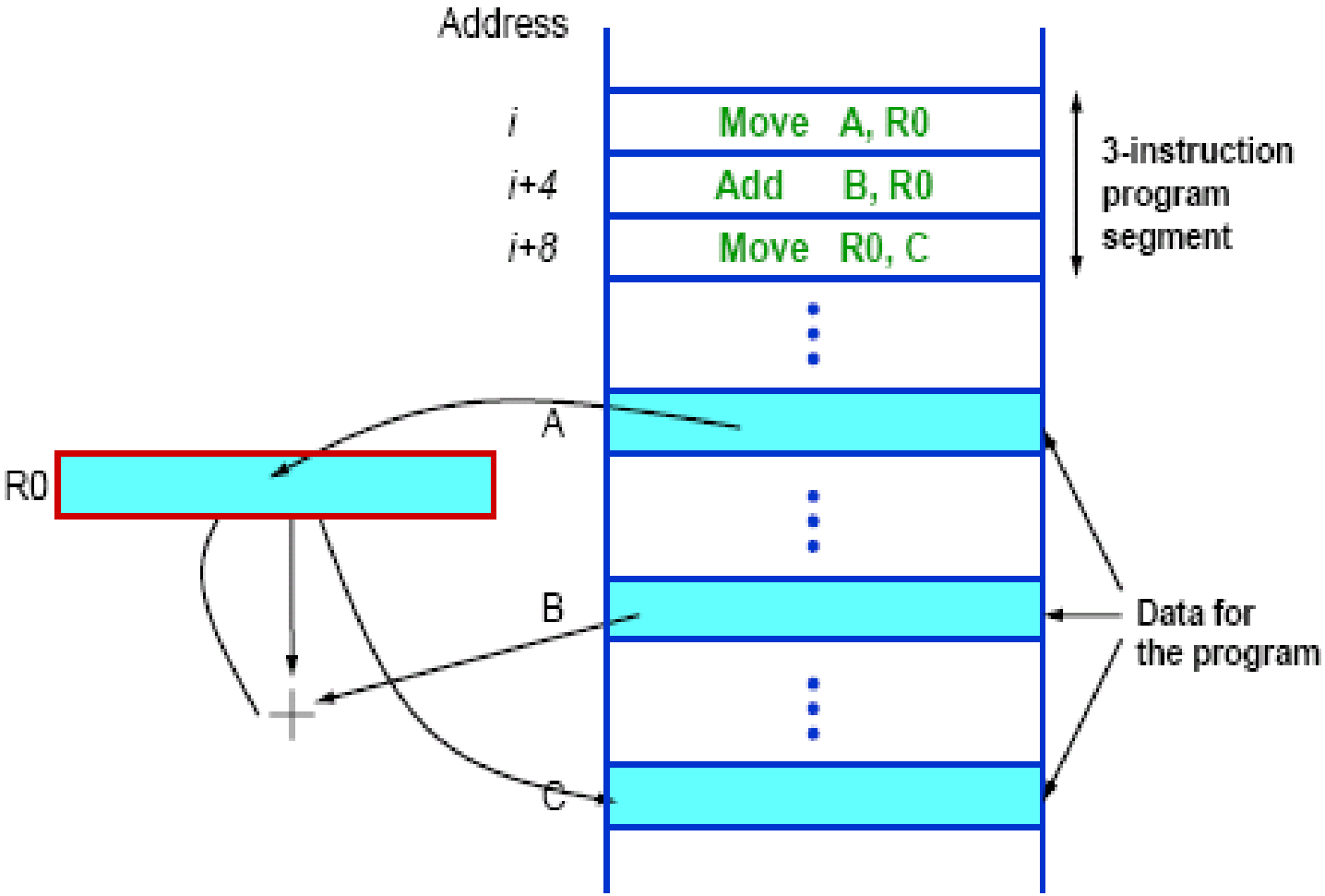
Little-endian assignment

Instruction Execution

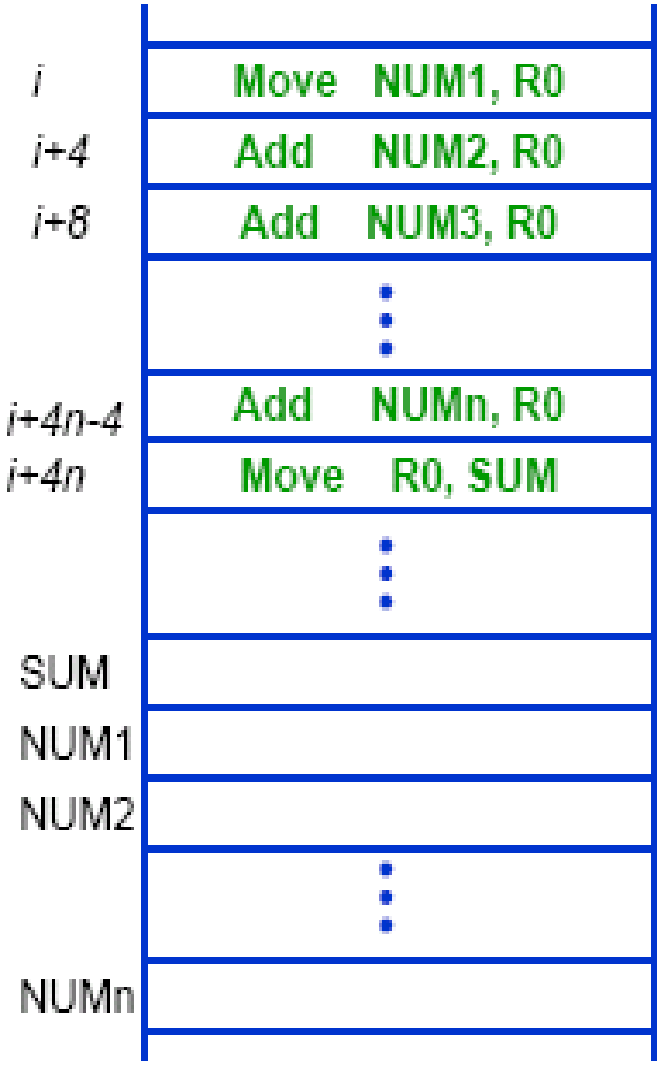
- How a program is executed
  - The processor contains a register called the program counter (PC), which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction must be placed into the PC, then the processor control circuits use the information in the PC to fetch and execute instruction, one at a time, in the order of increasing address
- Basic instruction cycle



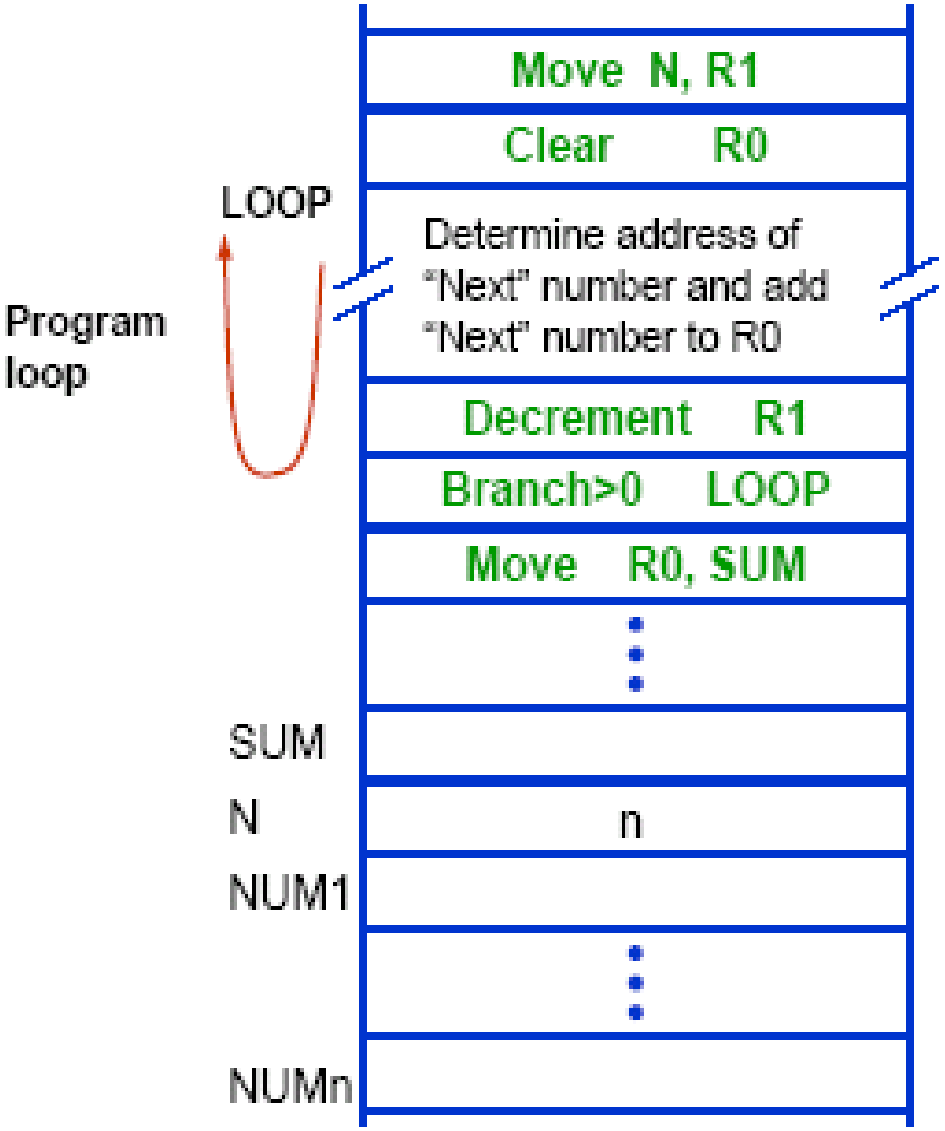
A Program for C ← [A]+[B]



Straight-Line Sequencing



Branching

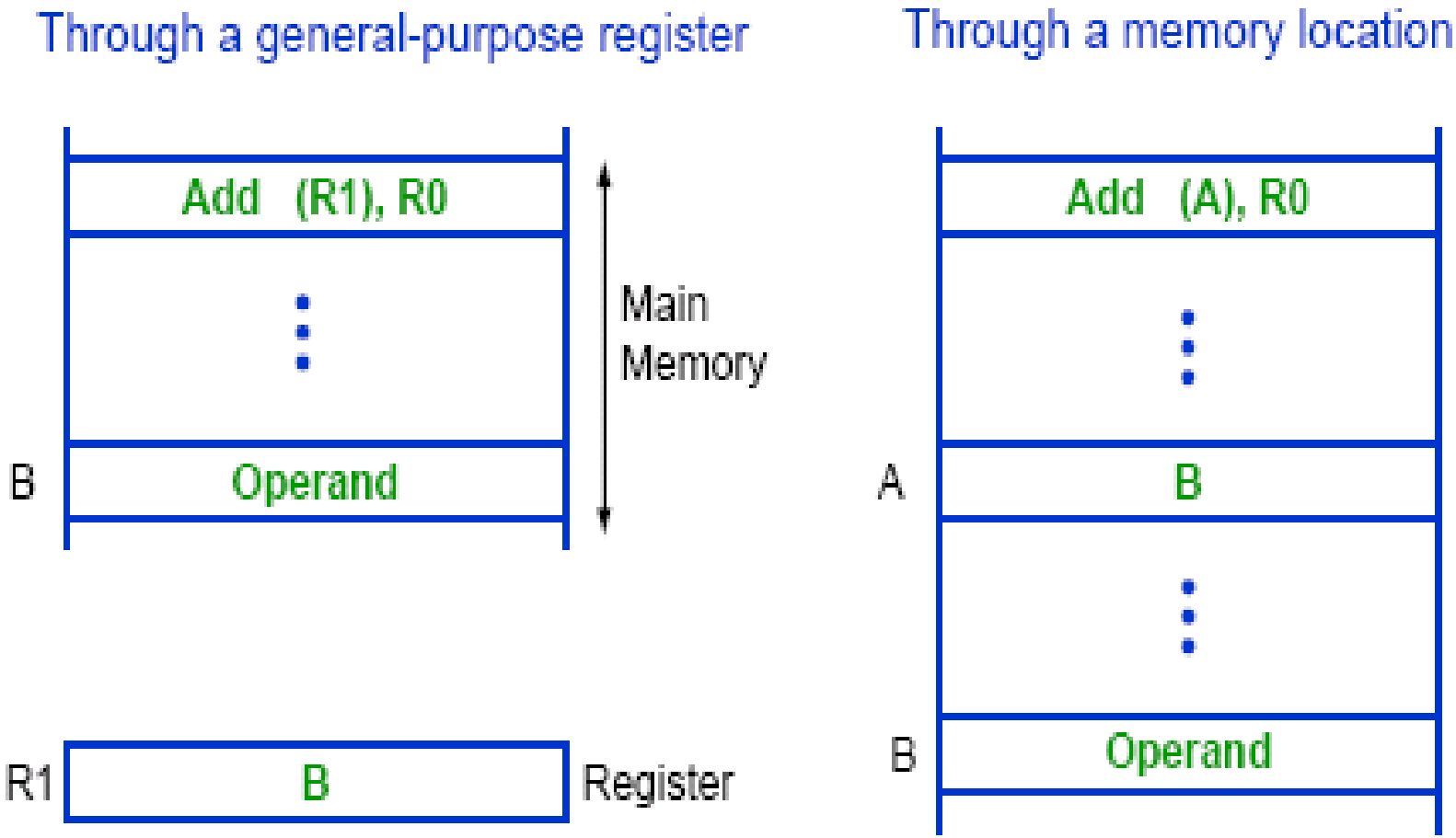

Move N, R1
Clear R0
LOOP
Determine address of "Next" number and add "Next" number to R0
Program loop
Decrement R1
Branch>0 LOOP
Move R0, SUM
SUM
N
n
NUM1
NUMn

Generic Addressing Modes

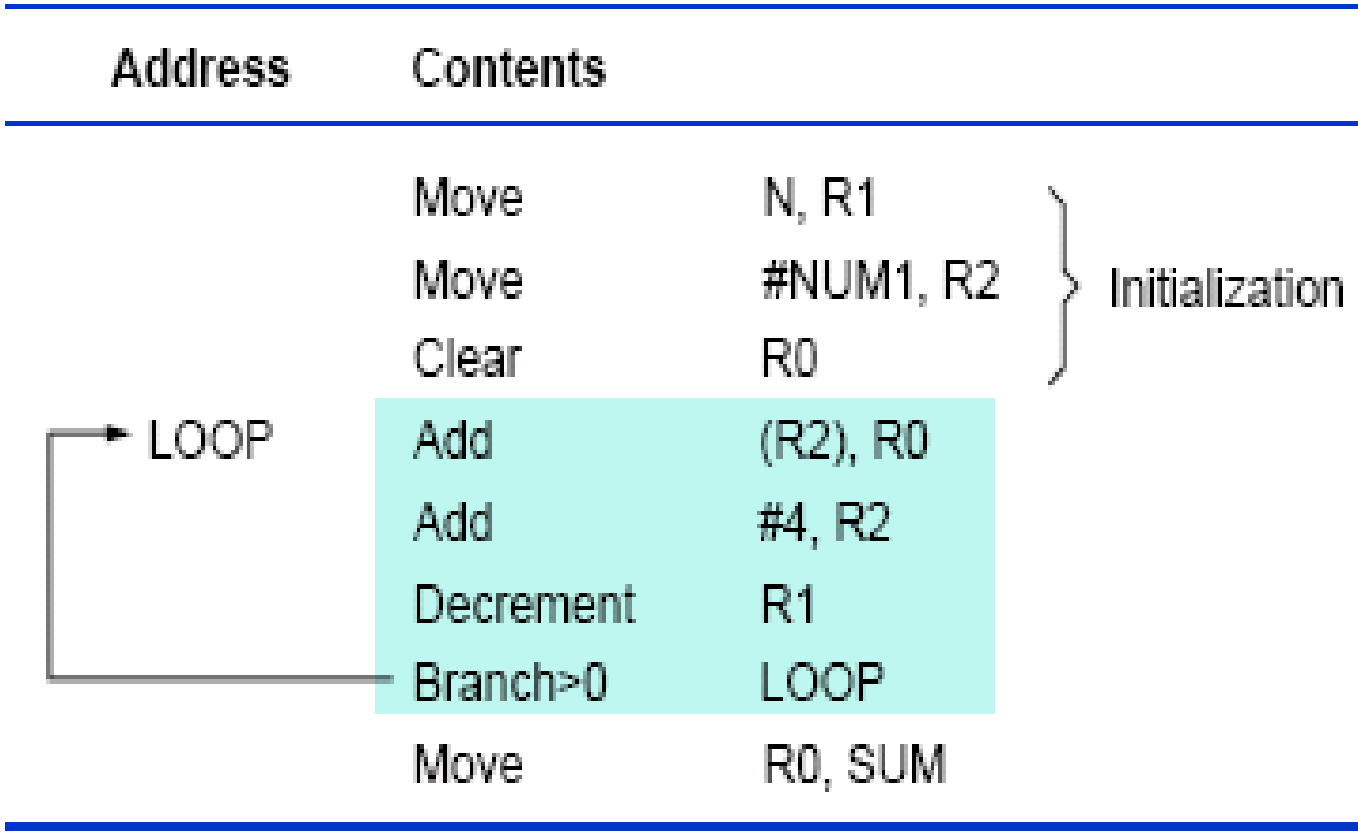
Name	Assembler syntax	Addressing function
Immediate	#Value	Operand=Value
Register	Ri	EA=Ri
Absolute (Direct)	LOC	EA=LOC
Indirect	(Ri)	EA=[Ri]
	(LOC)	EA=[LOC]
Index	X(Ri)	EA=[Ri]+X
Base with index	(Ri, Rj)	EA=[Ri]+[Rj]
Base with index and offset	X(Ri, Rj)	EA=[Ri]+[Rj]+X
Relative	X(PC)	EA=[PC]+X
Autoincrement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

EA: effective address
Value: a signed number

Two Types of Indirect Addressing

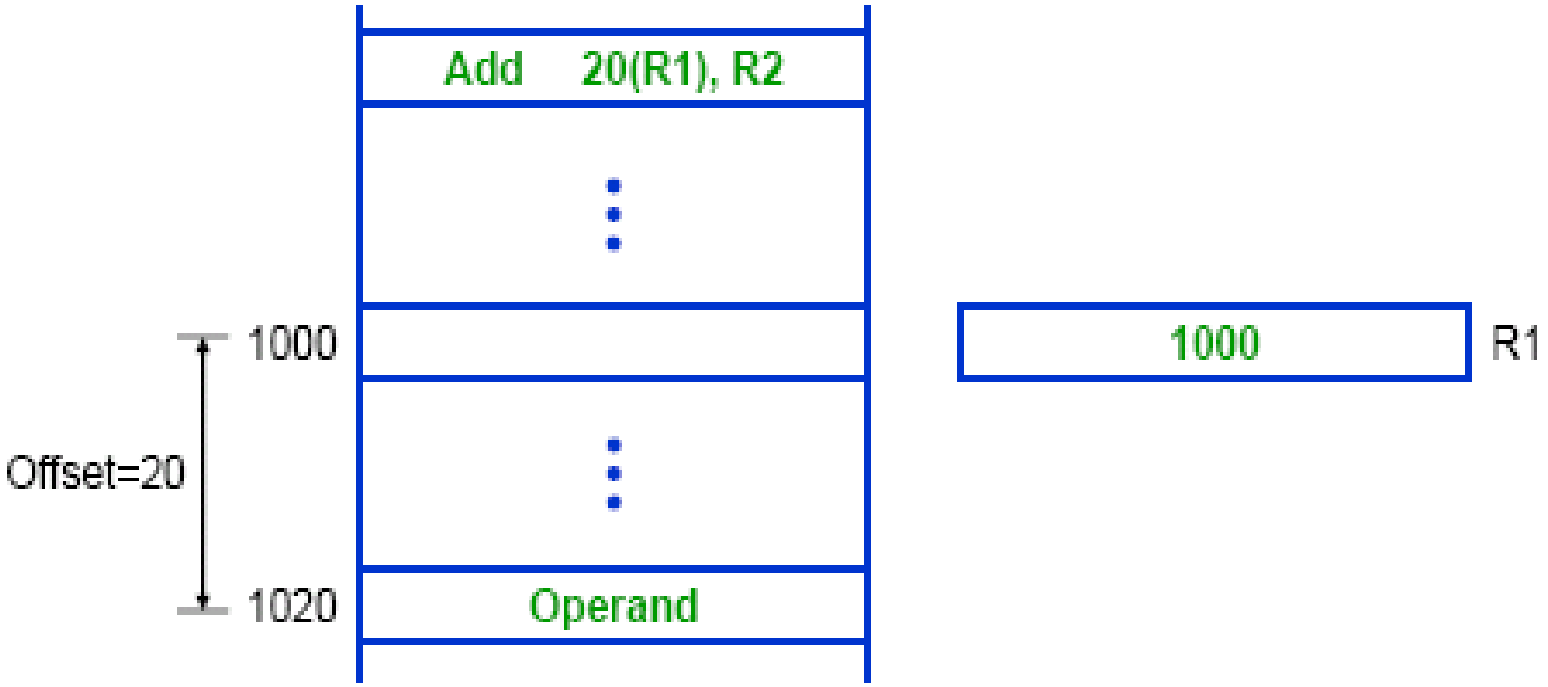


Using Indirect Addressing in a Program



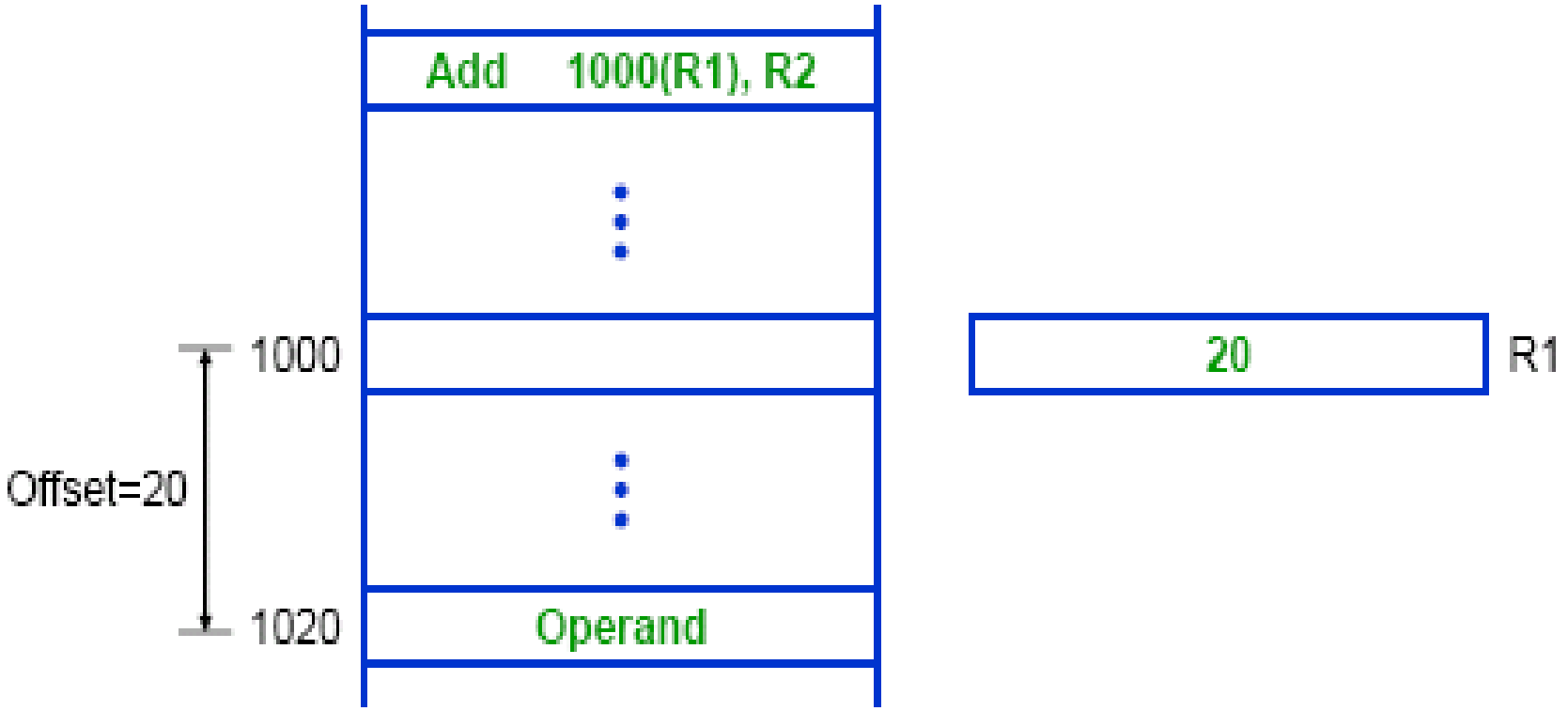
Indexed Addressing

Offset is given as a constant

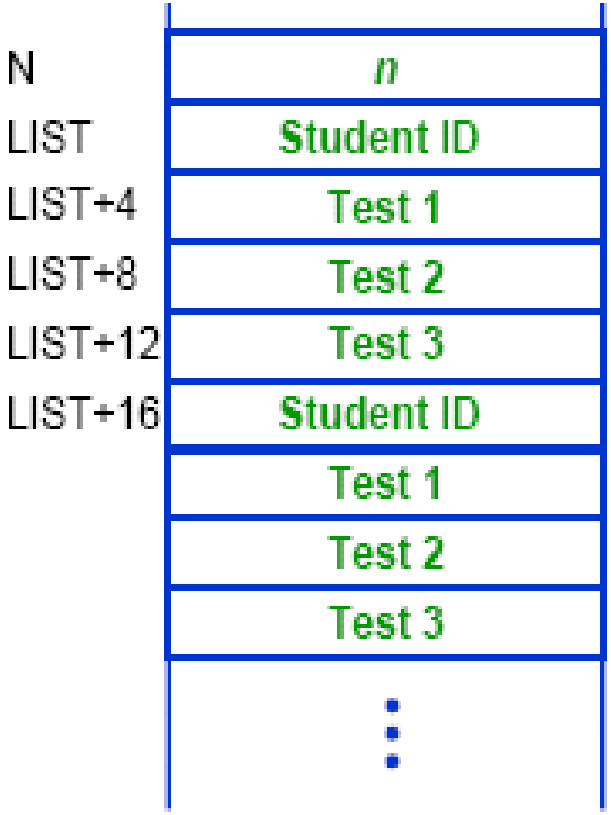


Indexed Addressing

Offset is in the index register

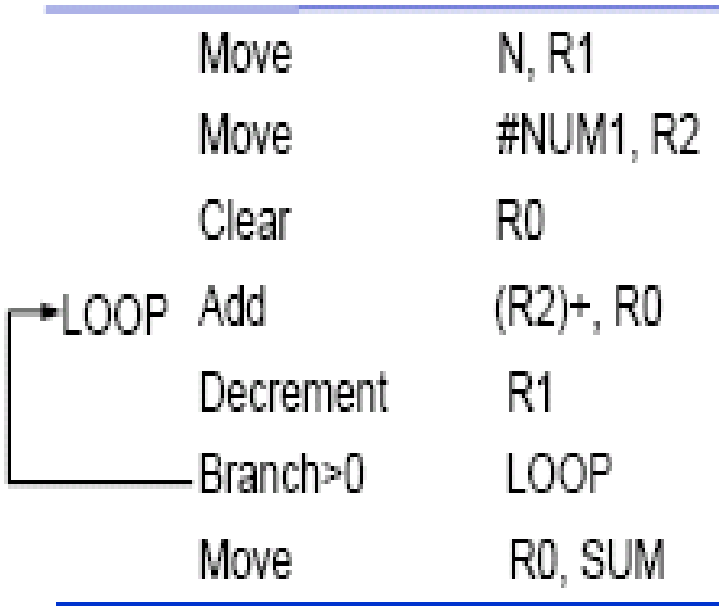


An Example for Indexed Addressing

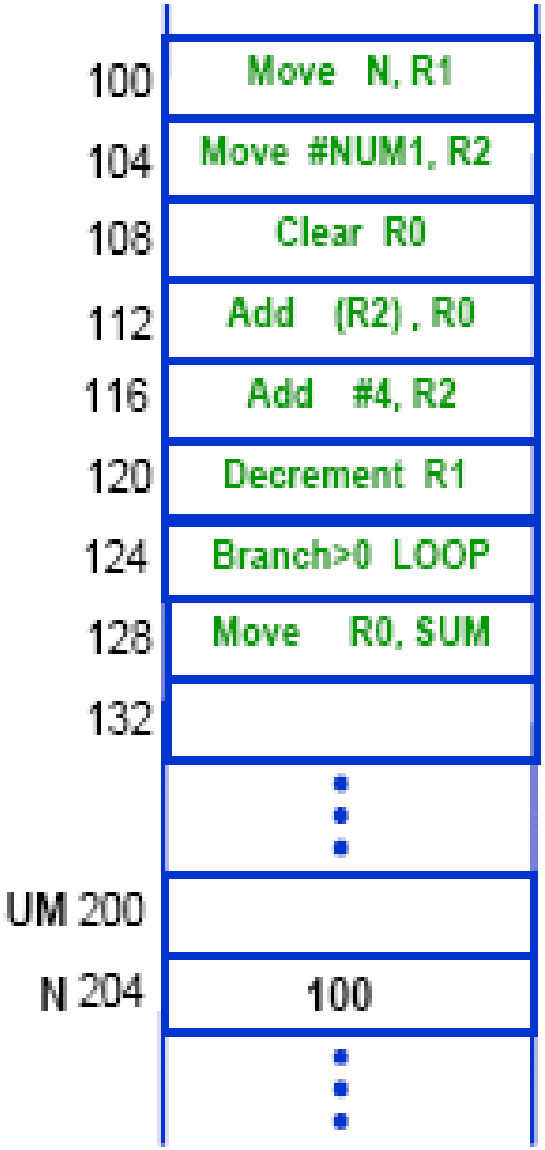



```
        Move      #LIST, R0
        Clear     R1
        Clear     R2
        Clear     R3
        Move      N, R4
LOOP    Add       4(R0), R1
        Add       8(R0), R2
        Add       12(R0), R3
        Add       #16, R0
        Decrement R4
        Branch>0  LOOP
        Move      R1, SUM1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

An Example of Autoincrement Addressing



Assembler



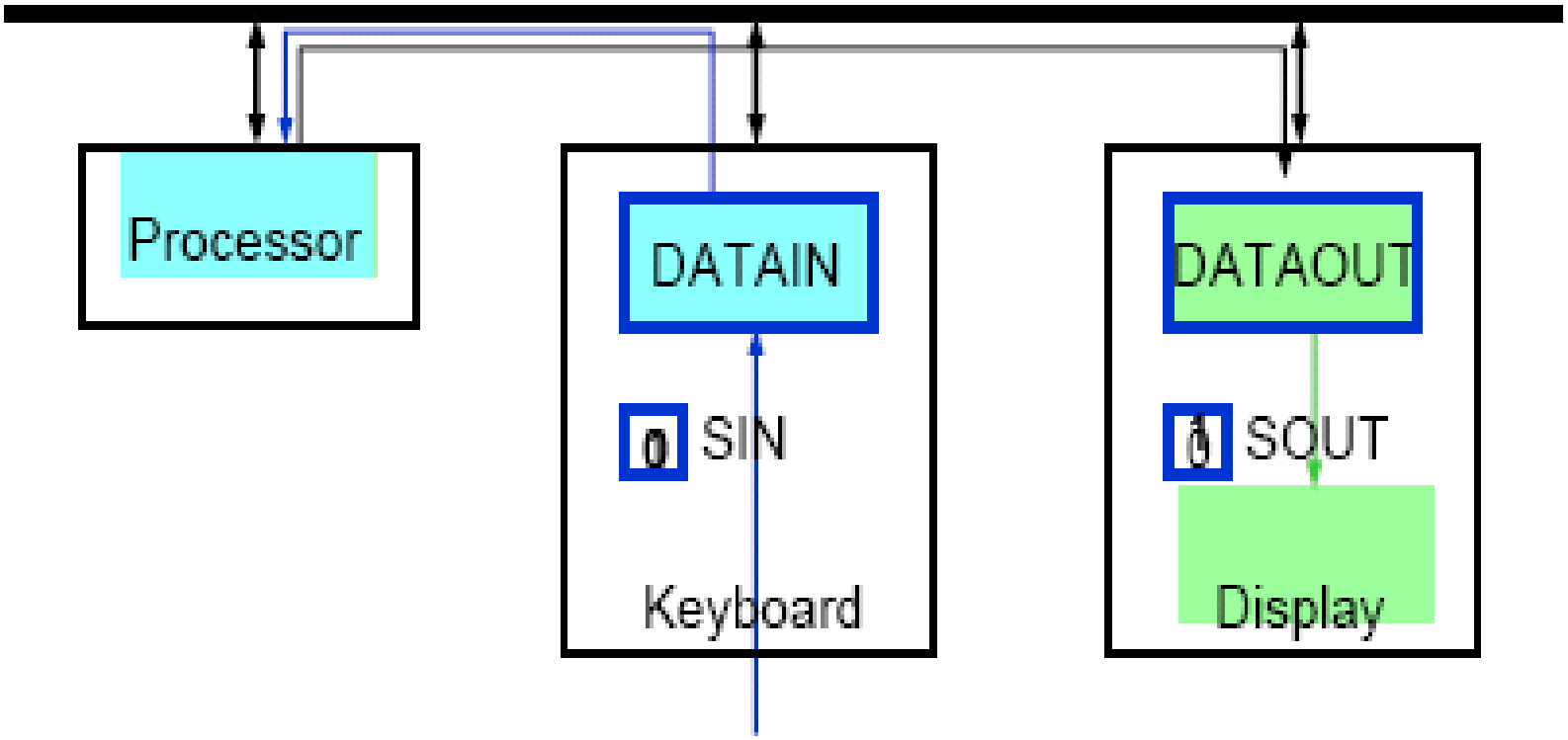
Memory arrangement

Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that	START	MOVE	N, R1
generate		MOVE	#NUM1, R2
machine		CLR	R0
instructions	LOOP	ADD	(R2), R0
		ADD	#4, R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0, SUM
Assembler directives		RETURN	
		END	START

Assembly language representation

Basic Input/Output Operations

- Bus connection for processor, keyboard, and display

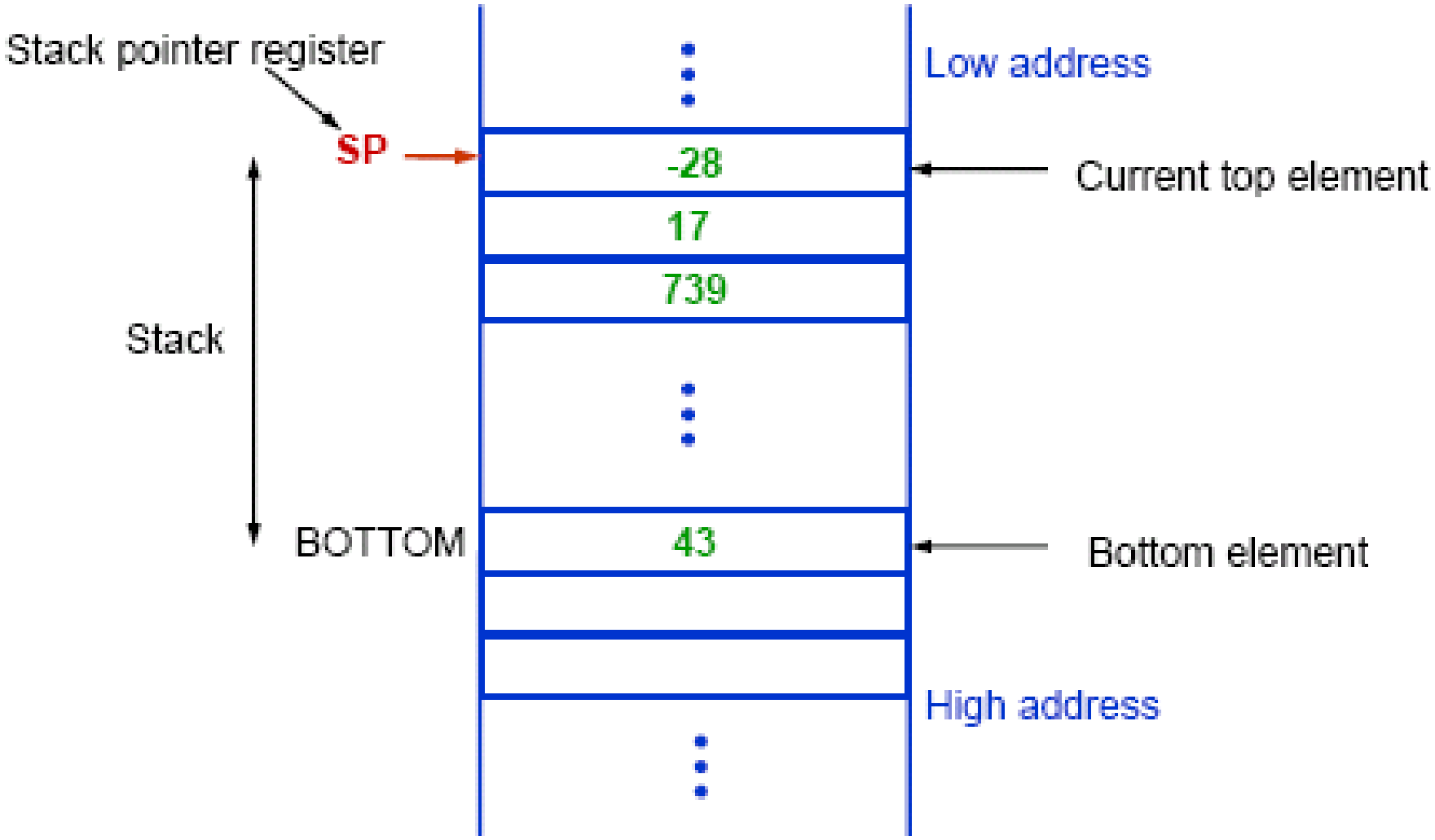


DATAIN, DATAOUT: buffer registers
SIN, SOUT: status control flags

Read and Write Programs

- Assume that bit b_3 in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT, respectively
- Read Loop
  - READWAIT Testbit #3, INSTATUS
    Branch=0 READWAIT
    MoveByte DATAIN, R1
- Write Loop
  - WRITEWAIT Testbit #3, OUTSTATUS
    Branch=0 WRITEWAIT
    MoveByte R1, DATAOUT

A Stack of Words in the Memory



Push and Pop Operations

- Assume that a byte-addressable memory with 32-bit words
- The push operation can be implemented as

```
Subtract   #4, SP
Move       NEWITEM, (SP)
```

- The pop operation can be implemented as

```
Move    (SP), ITEM
Add     #4, SP
```

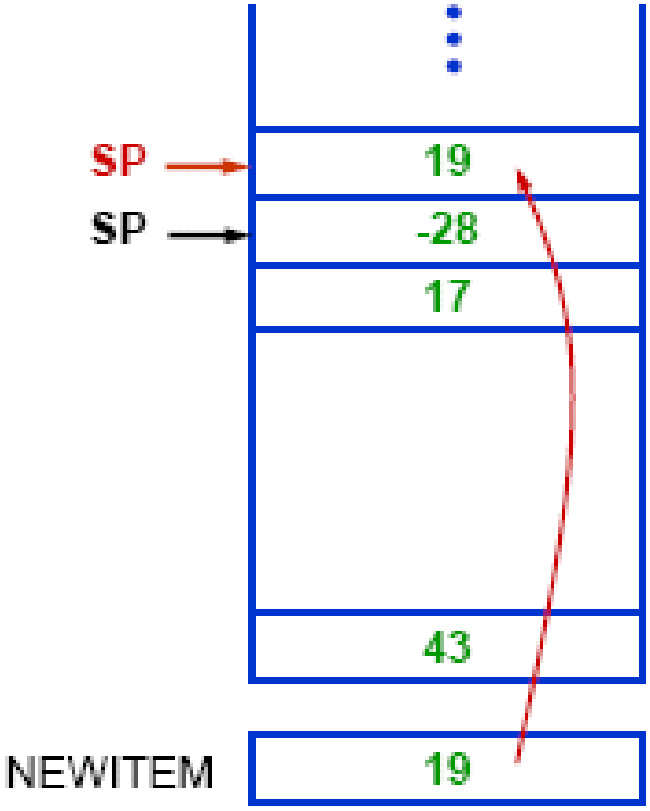
- If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be implemented by the single instruction

```
Move  NEWITEM, -(SP)
```

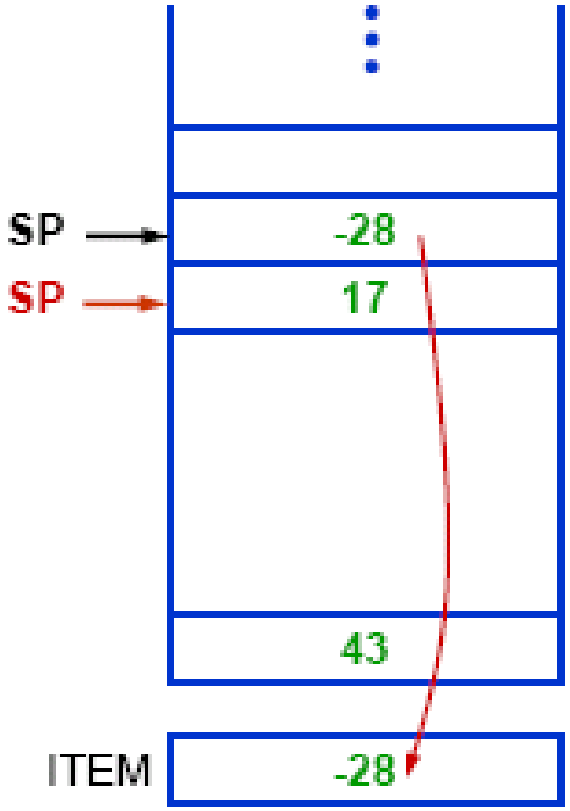
- And the pop operation can be implemented as

```
Move (SP)+, ITEM
```

Examples



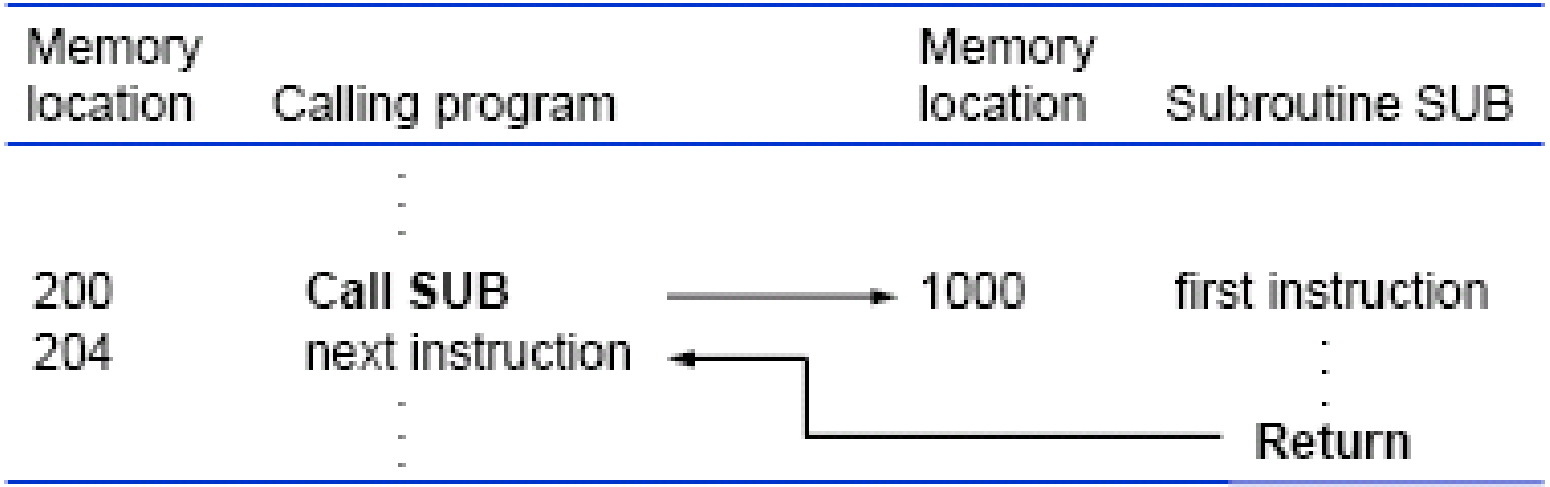
Push operation



Pop operation

Subroutines

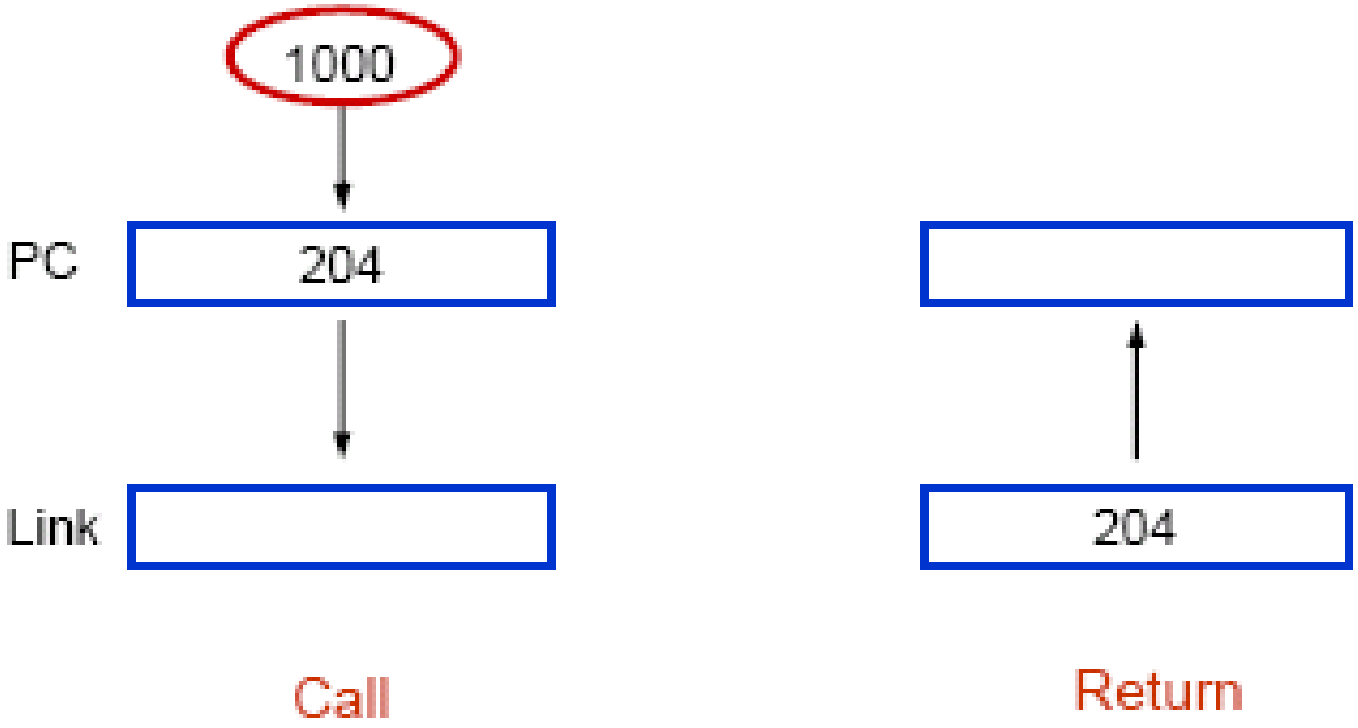
- In a given program, it is often necessary to perform a particular subtask many times on different data values. Such a subtask is called a *subroutine*.



- The location where the calling program resumes execution is the location pointed by the updated PC while the Call instruction is being executed. Hence the contents of the PC must be saved by the Call instruction to enable correct return to the calling program

Subroutine Linkage

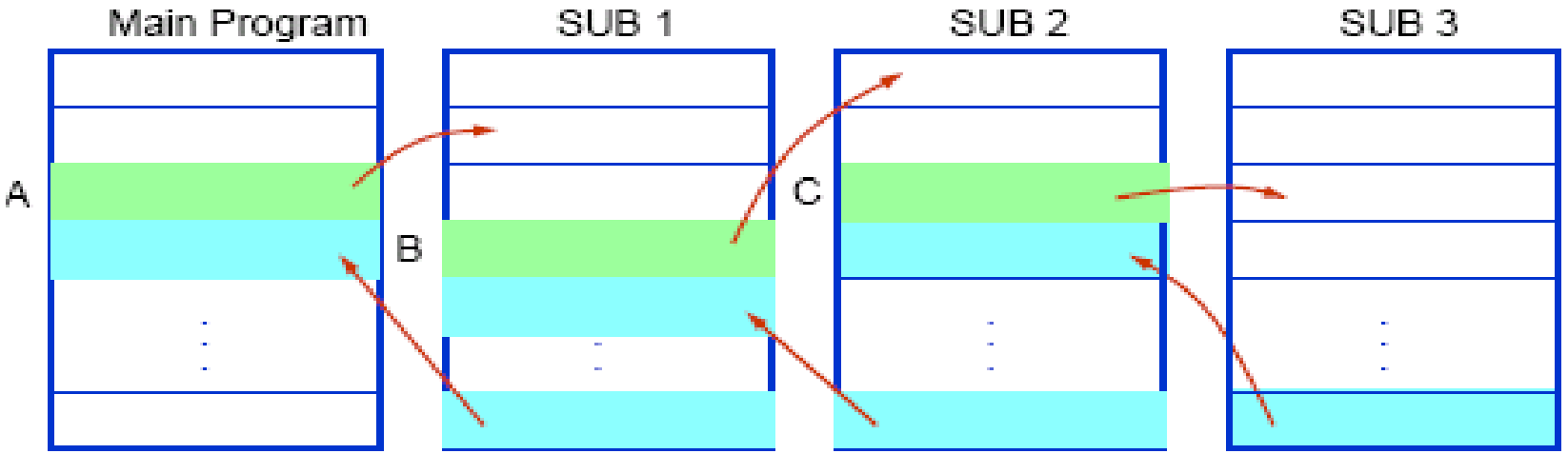
- The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method
- Subroutine linkage using a link register

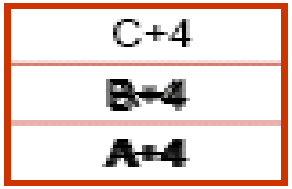


Subroutine Nesting

- A common programming practice, called subroutine nesting, is to have one subroutine call another
- Subroutine nesting call be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it
- The return address needed for this first returns is the last one generated in the nested call sequence. That is, return addresses are generated and used in a last-in-first-out order
- Many processors do this by using a stack pointer and the stack pointer points to a stack called the processor stack

An Example of Subroutine Nesting

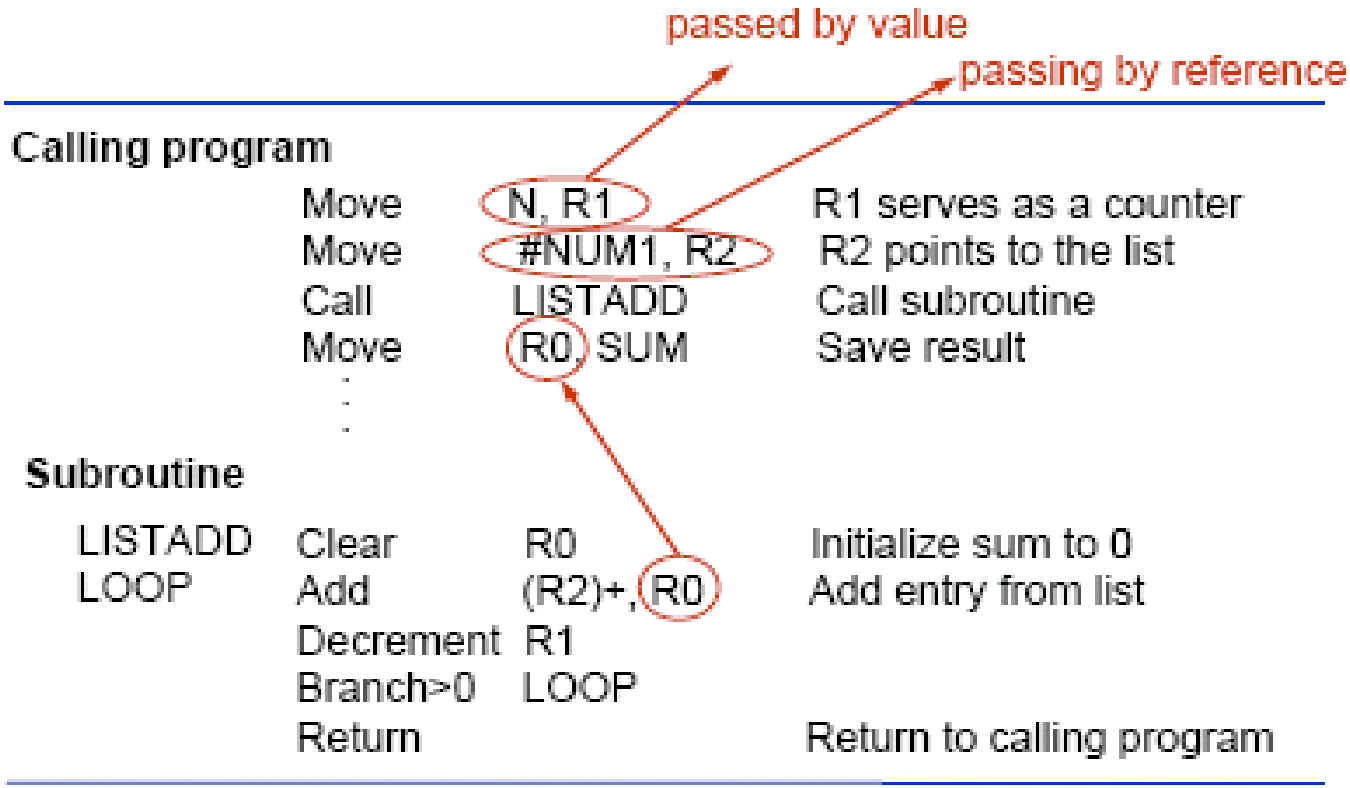




Parameter Passing

- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the result of computation
- The exchange of information between a calling program and a subroutine is referred to as parameter passing
- Parameter passing approaches
  - The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine
  - The parameters may be placed on the processor stack used for saving the return address

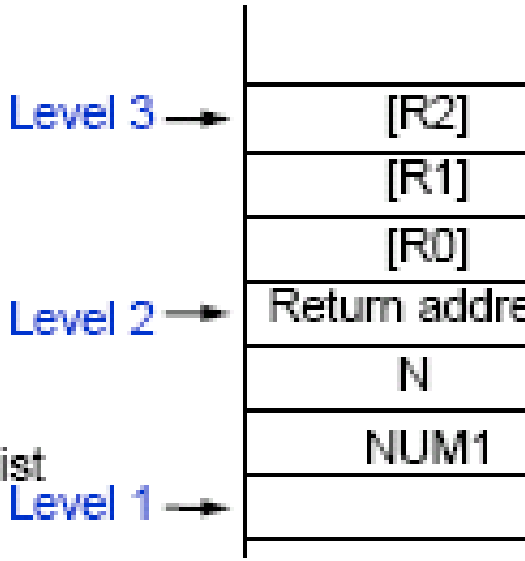
Passing Parameters with Registers



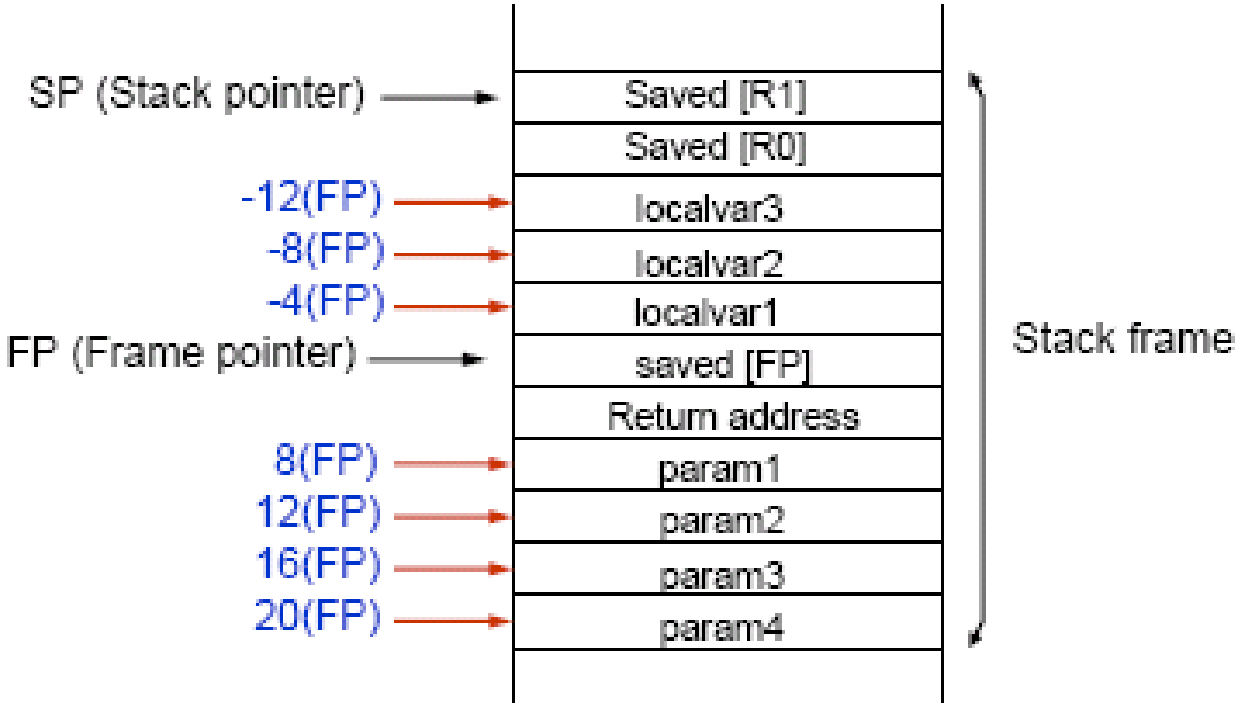
Passing Parameters with Stack

Assume top of stack is at level 1 below.

```
                Move        #NUM1, -(SP)     Push parameters onto stack
                Move        N, -(SP)
                Call        LISTADD          Call subroutine
                                             (top of stack at level 2)
                Move        4(SP), SUM       Save result
                Add         #8, SP           Restore top of stack
                                             (top of stack at level 1)
                            .
                            .
                            .
LISTADD         MoveMultiple R0-R2, -(SP)    Save registers
                                             (top of stack at level 3)
                Move        16(SP), R1       Initialize counter to N.
                Move        20(SP), R2       Initialize pointer to the list
                Clear       R0               Initialize sum to 0
LOOP            Add         (R2)+, R0        Add entry from list
                Decrement   R1
                Branch>0    LOOP
                Move        R0, 20(SP)       Put result on the stack
                MoveMultiple (SP)+, R0-R2    Restore registers
                Return                       Return to calling program
```

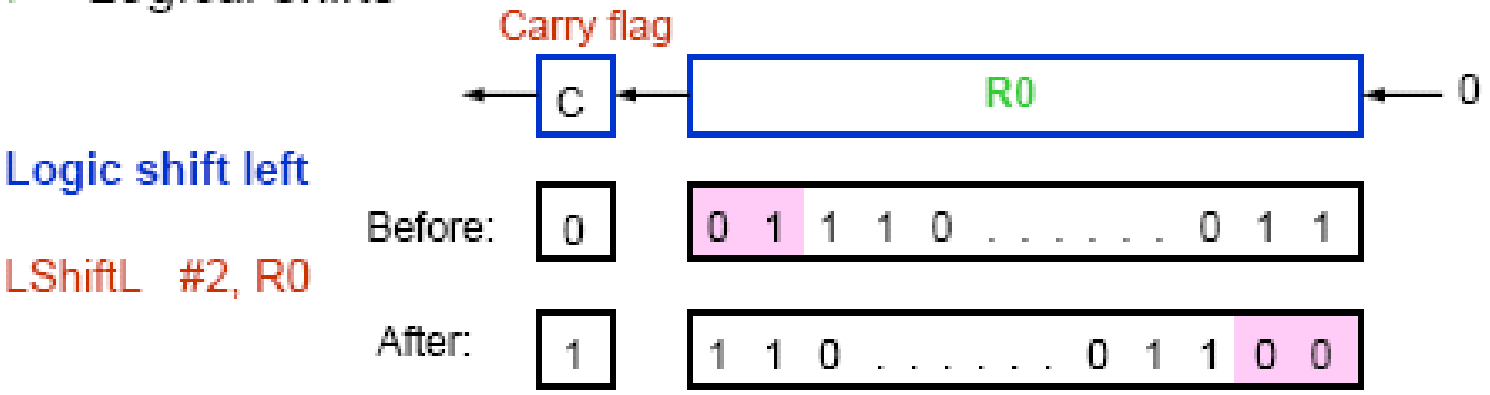


Stack Frame

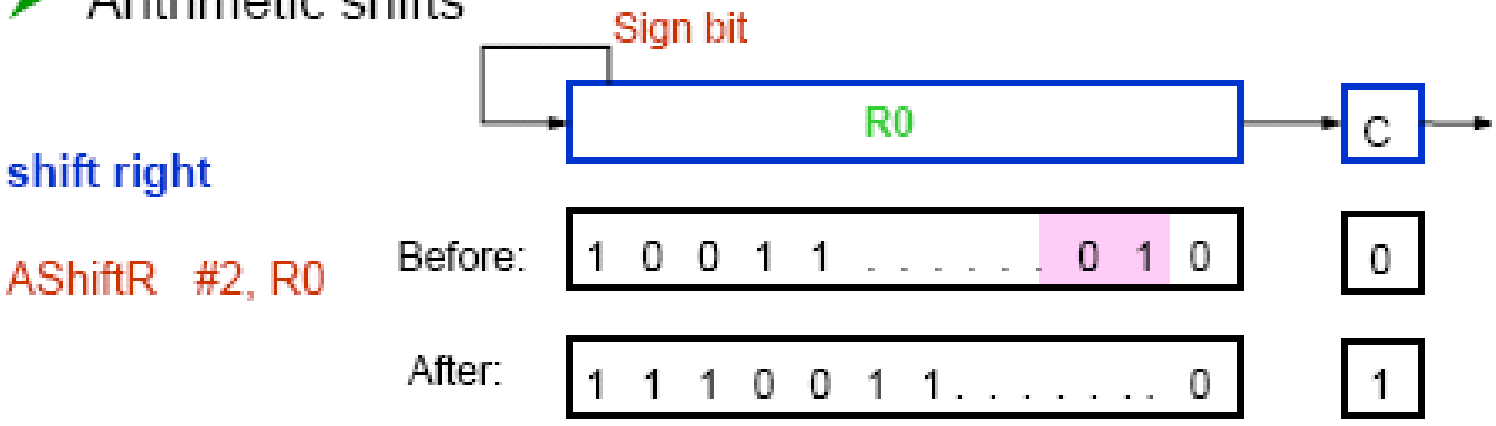

SP (Stack pointer)
Saved [R1]
Saved [R0]
-12(FP)
localvar3
-8(FP)
localvar2
-4(FP)
localvar1
FP (Frame pointer)
saved [FP]
Return address
8(FP)
param1
12(FP)
param2
16(FP)
param3
20(FP)
param4
Stack frame

Shift Instructions

- Logical shifts

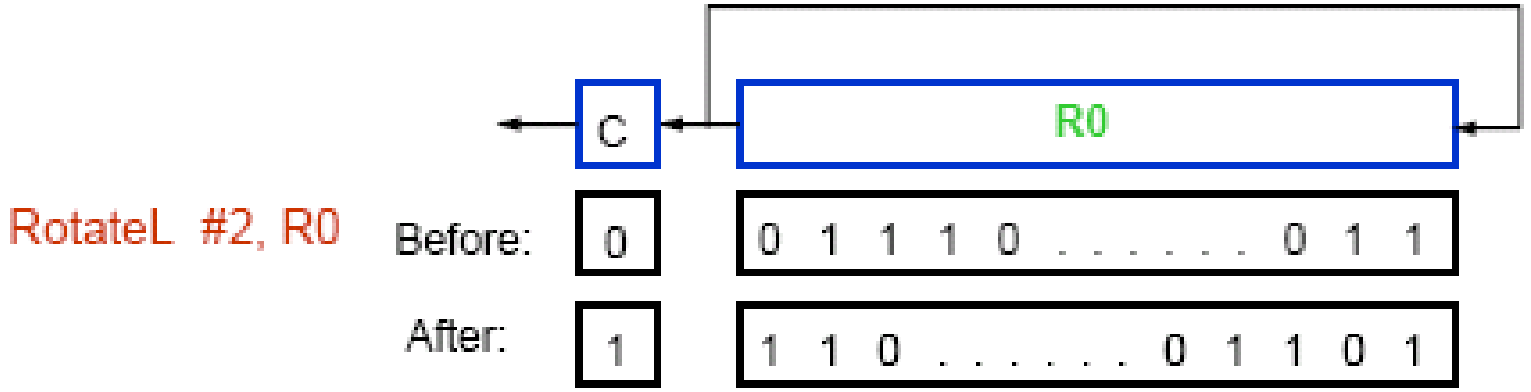


- Arithmetic shifts

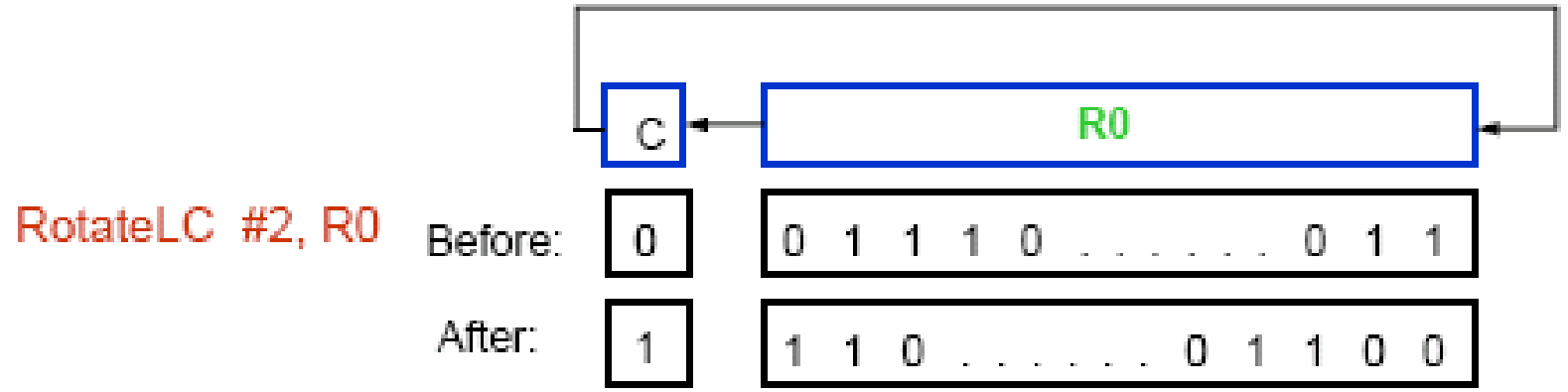


Rotate Instructions

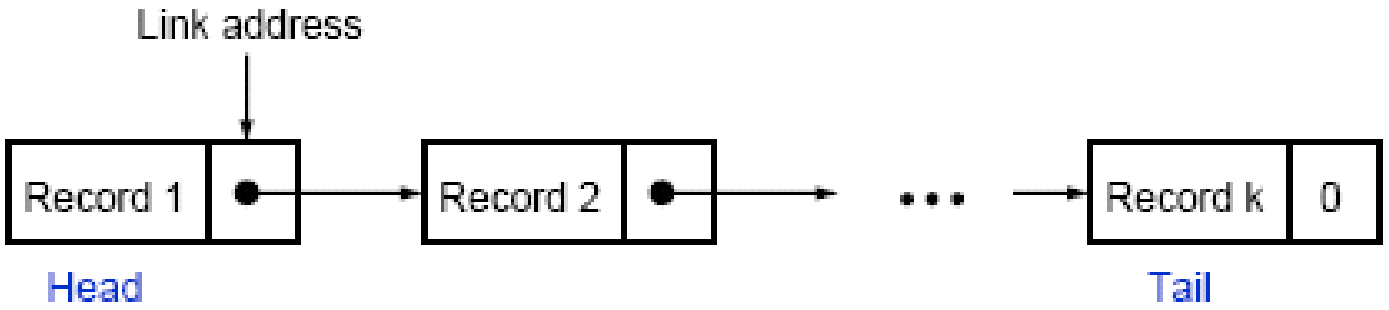
- Rotate left without carry



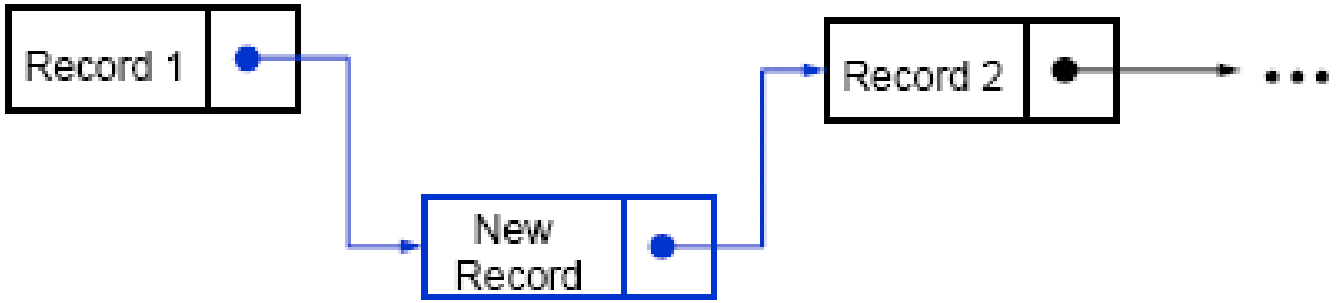
- Rotate left with carry



Linked List

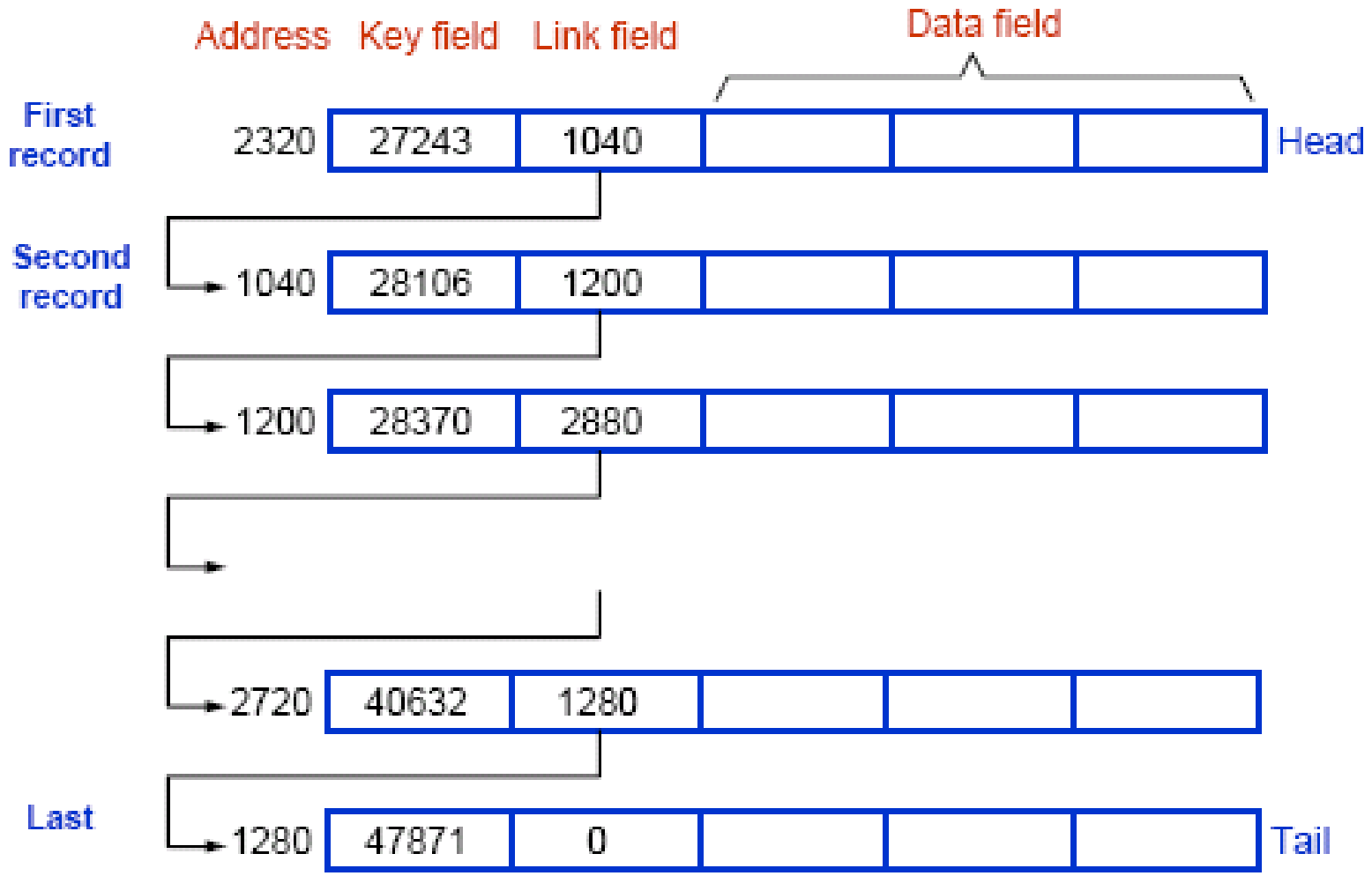


Linking structure

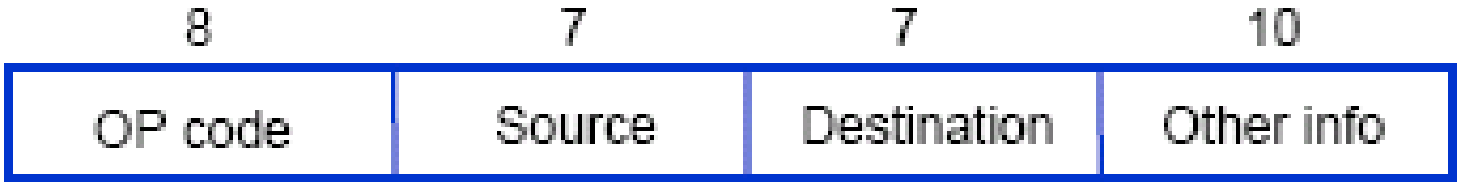


Inserting a new record

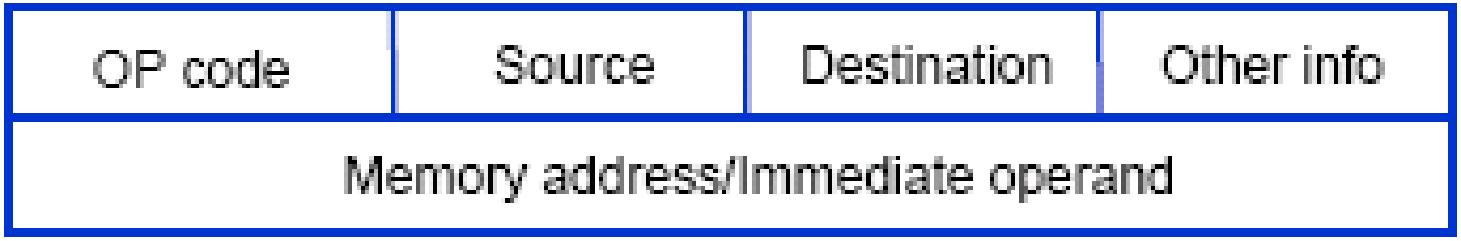
A List of Student Test Scores



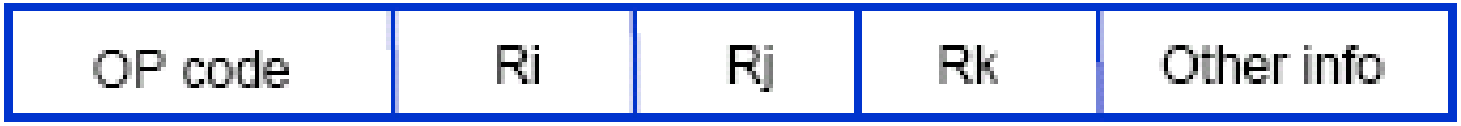
Encoding Instructions into 32-bit Words



One-word instruction



Two-word instruction



Three-operand instruction

Encoding Instructions into 32-bit Words

- But, what happens if we want to specify a memory operand using the Absolute addressing mode?
- The instruction Move R2, LOC
  - Require 18 bits to denote the OP code, the addressing modes, and the register
  - The leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient
- If we want to be able to give a complete 32-bit address in the instruction, an instruction must have two words
- If we want to handle this type of instructions: Move LOC1, LOC2
  - An instruction must have three words

CISC & RISC

- Using multiple words, we can implement quite complex instructions, closely resembling operations in high-level programming language
- The term *complex instruction set computer* (CISC) has been used to refer to processors that use instruction sets of this type
- The restriction that an instruction must occupy only one word has led to a style of computers that have become known as *reduced instruction set computer* (RISC)