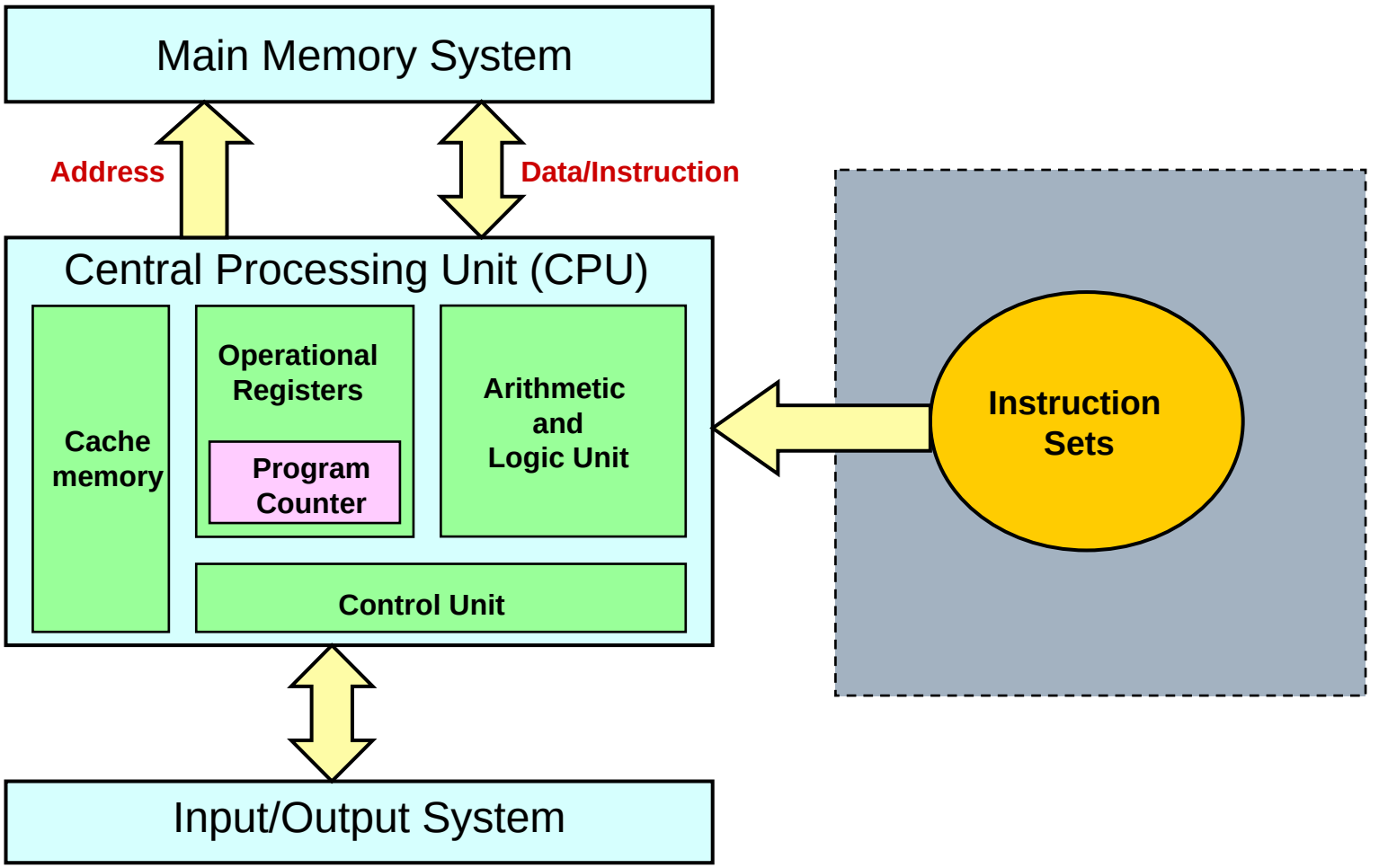
Chapter 3
Machine Instructions & Programs

Jin-Fu Li
Department of Electrical Engineering
National Central University
Jungli, Taiwan

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 $0 \le i \le n-1$
- 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 Systems

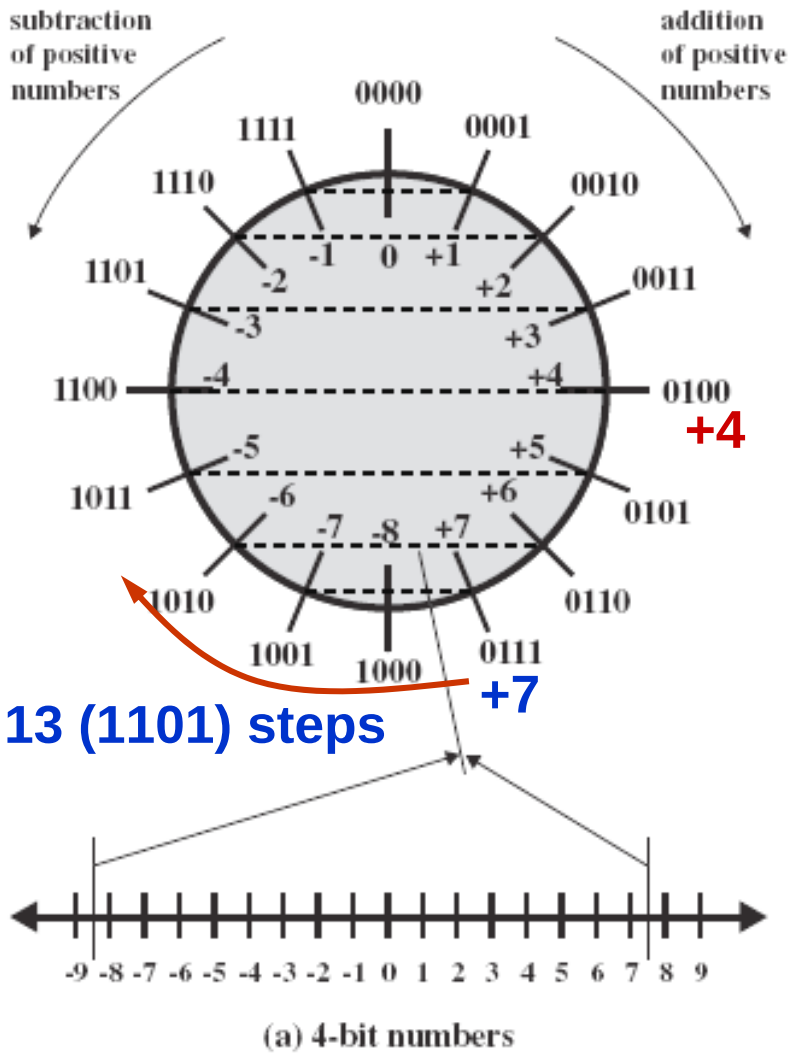
- In *sign-and-magnitude system*
  - Negative values are represented by changing the most significant bit from 0 to 1
- In *1's-complement system*
  - Negative values are obtained by complementing each bit of the corresponding positive number
  - The operation of forming the 1's-complement of a given number is equivalent to subtracting that number from 2^n-1
- In *2's-complement system*
  - The operation of forming the 2's-complement of a given number is done by subtracting that number from 2^n
- The 2's-complement of a number is obtained by adding 1 to 1's-complement of that number

An Example of Number Representations

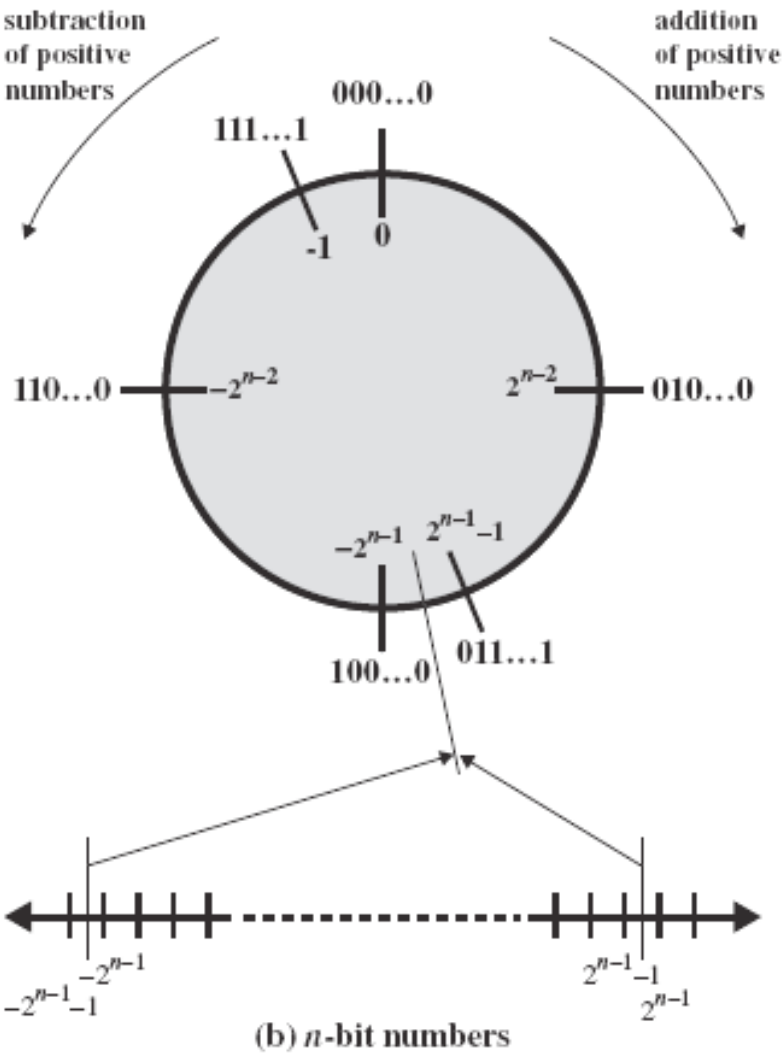
$b_3b_2b_1b_0$	sign and magnitude	1's-complement	2's-complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

2's-Complement System

+7+(-3)



(a) 4-bit numbers



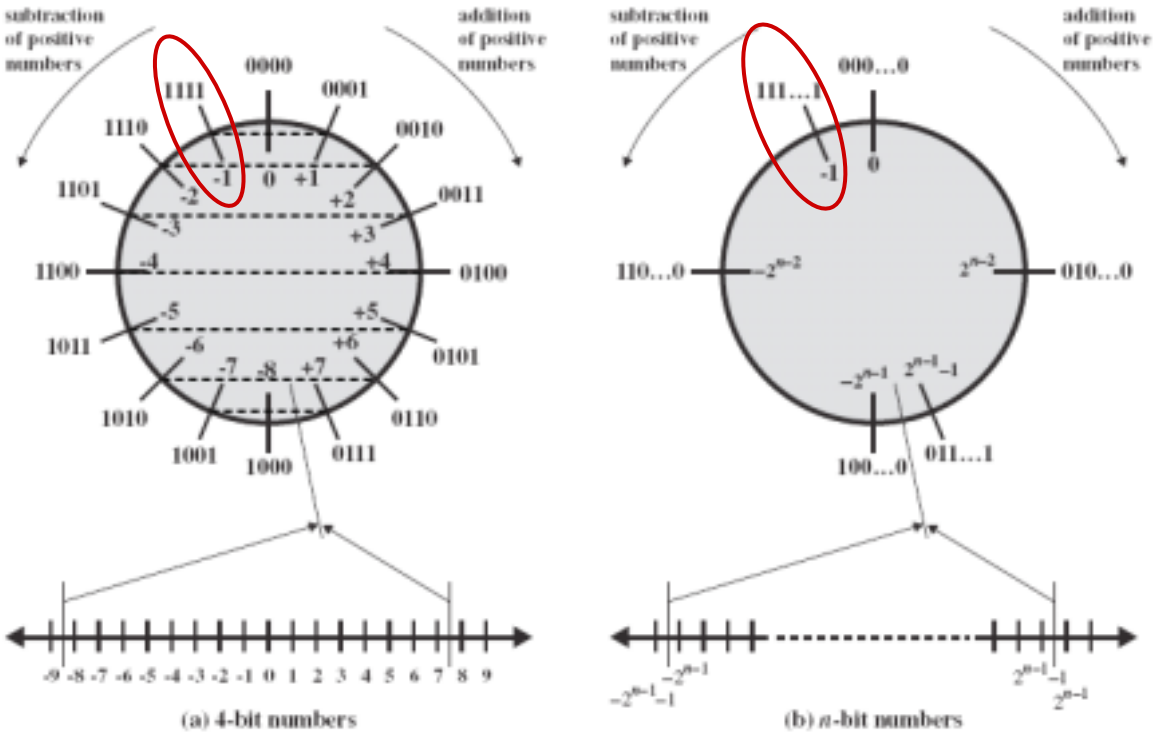
(b) *n*-bit numbers

Addition of Numbers in 2's Complement

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Sign Extension of 2's Complement

- Sign extension
  - To represent a signed number in 2's complement form using a larger number of bits, repeat the sign bit as many times as needed to the left



Memory Locations

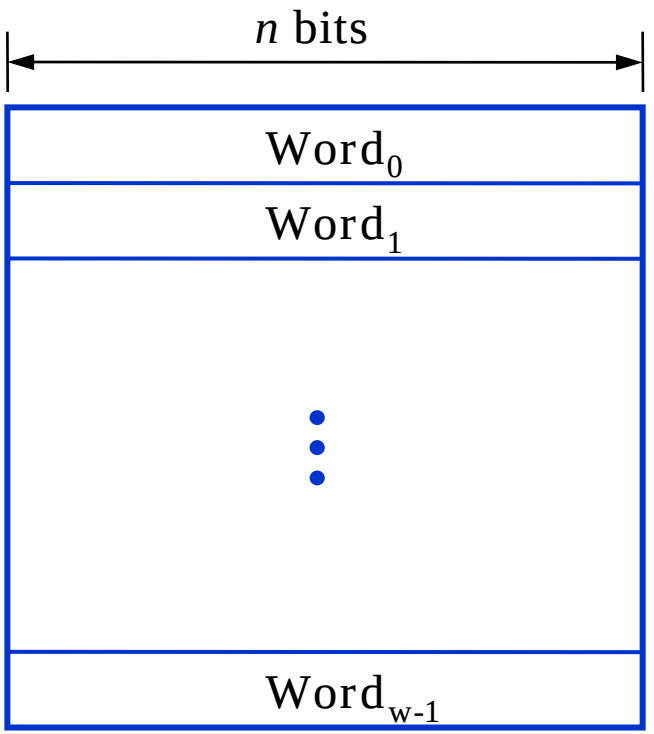
- A memory consists of cells, each of which can store a bit of binary information (0 or 1)
- Because a single bit represents a very small amount of information
  - Bits are seldom handled individually
- The memory usually is organized so that a group of *n* bits can be stored or retrieved in a single, basic operation
  - Each group of n bits is referred to as a *word* of information, and *n* is called the *word length*
  - A unit of 8 bits is called a byte
- Modern computers have word lengths that typically range from 16 to 64 bits

Memory Addresses

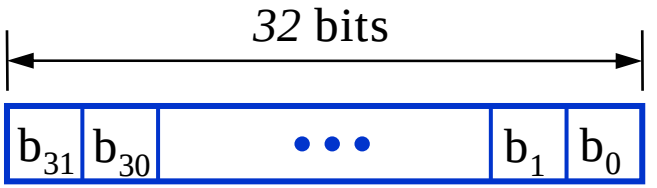
- Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each item location
- It is customary to use numbers from 0 to 2^k-1 as the address space of successive locations in the memory
  - K denotes address
  - 2^k-1 denotes address space of memory locations
- For example, a 24-bit address generates an address space of 2^{24} (16,777,216) locations
- Terminology
  - 2^{10}: 1K (kilo)
  - 2^{20}: 1M (mega)
  - 2^{30}: 1G (giga)
  - 2^{40}: 1T (tera)

Memory Words

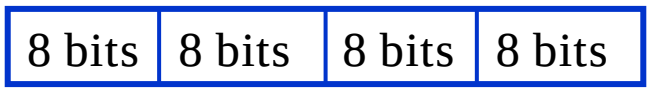
Memory words



A signed integer



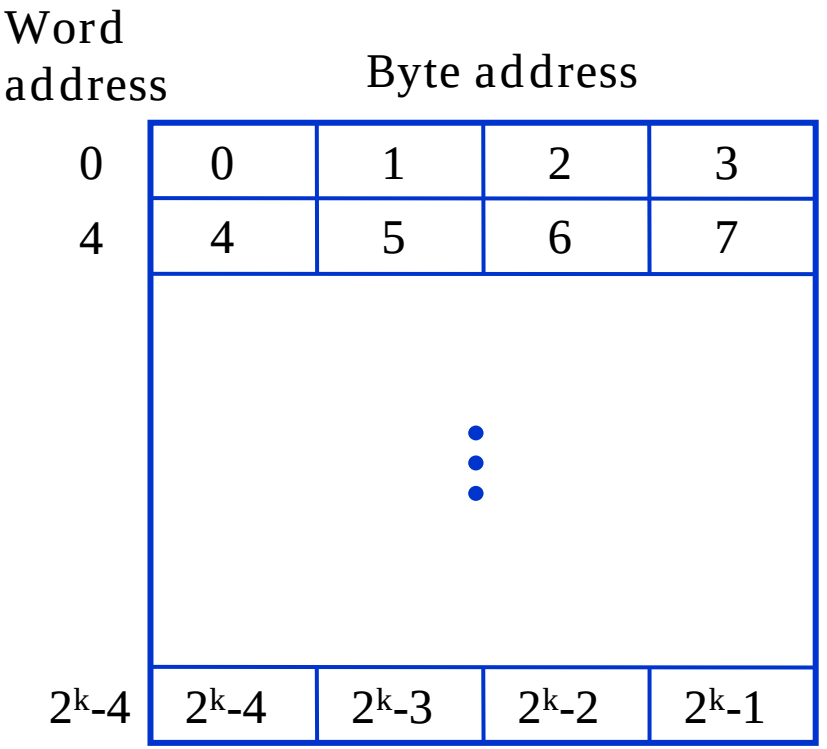
Four characters



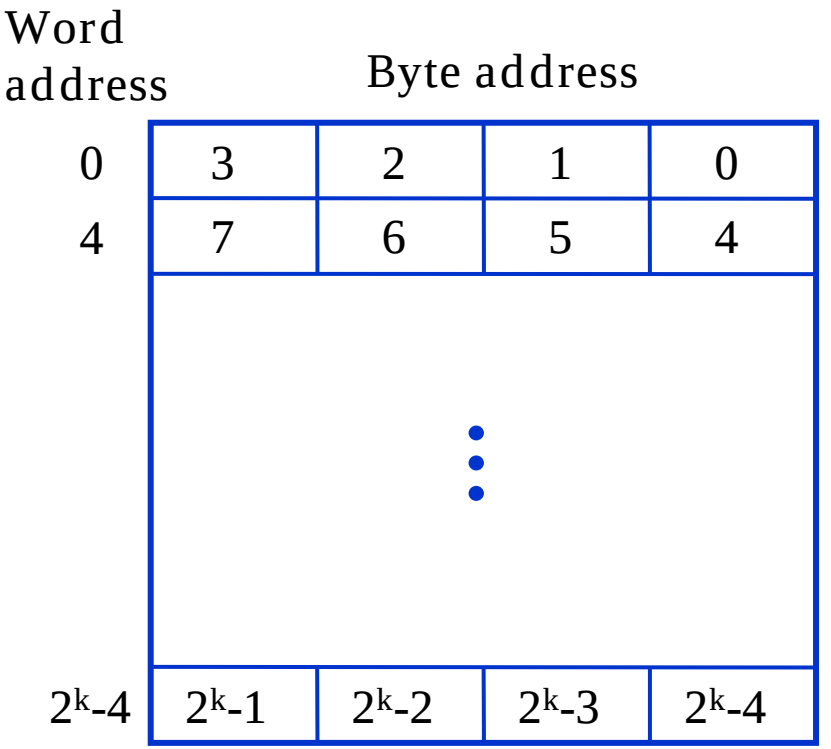
ASCII
character

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

Memory Operation

- Random access memories must have two basic operations
  - Write: writes a data into the specified location
  - Read: reads the data stored in the specified location
- In machine language program, the two basic operations usually are called
  - Store: write operation
  - Load: read operation
- The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged
- The Store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location

Instructions

- 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
- Register transfer notation
  - The contents of a location are denoted by placing square brackets around the name of the location
  - For example, R1←[LOC] means that the contents of memory location LOC are transferred into processor register R1
  - As another example, R3←[R1]+[R2] means that adds the contents of registers R1 and R2, and then places their sum into register R3

Assembly Language Notation

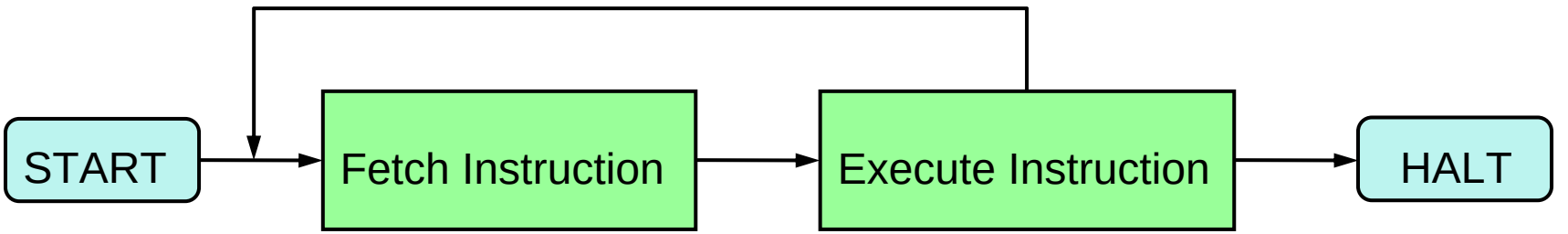
- Types of instructions
  - Zero-address instruction
  - One-address instruction
  - Two-address instruction
  - Three-address instruction
- Zero-address instruction
  - For example, store operands in a structure called a **pushdown stack**
- One-address instruction
  - Instruction form: **Operation Destination**
  - For example, **Add A**: add the contents of memory location A to the contents of the accumulator register and place the sum back into the accumulator
  - As another example, **Load A**: copies the contents of memory location A into the accumulator

Assembly Language Notation

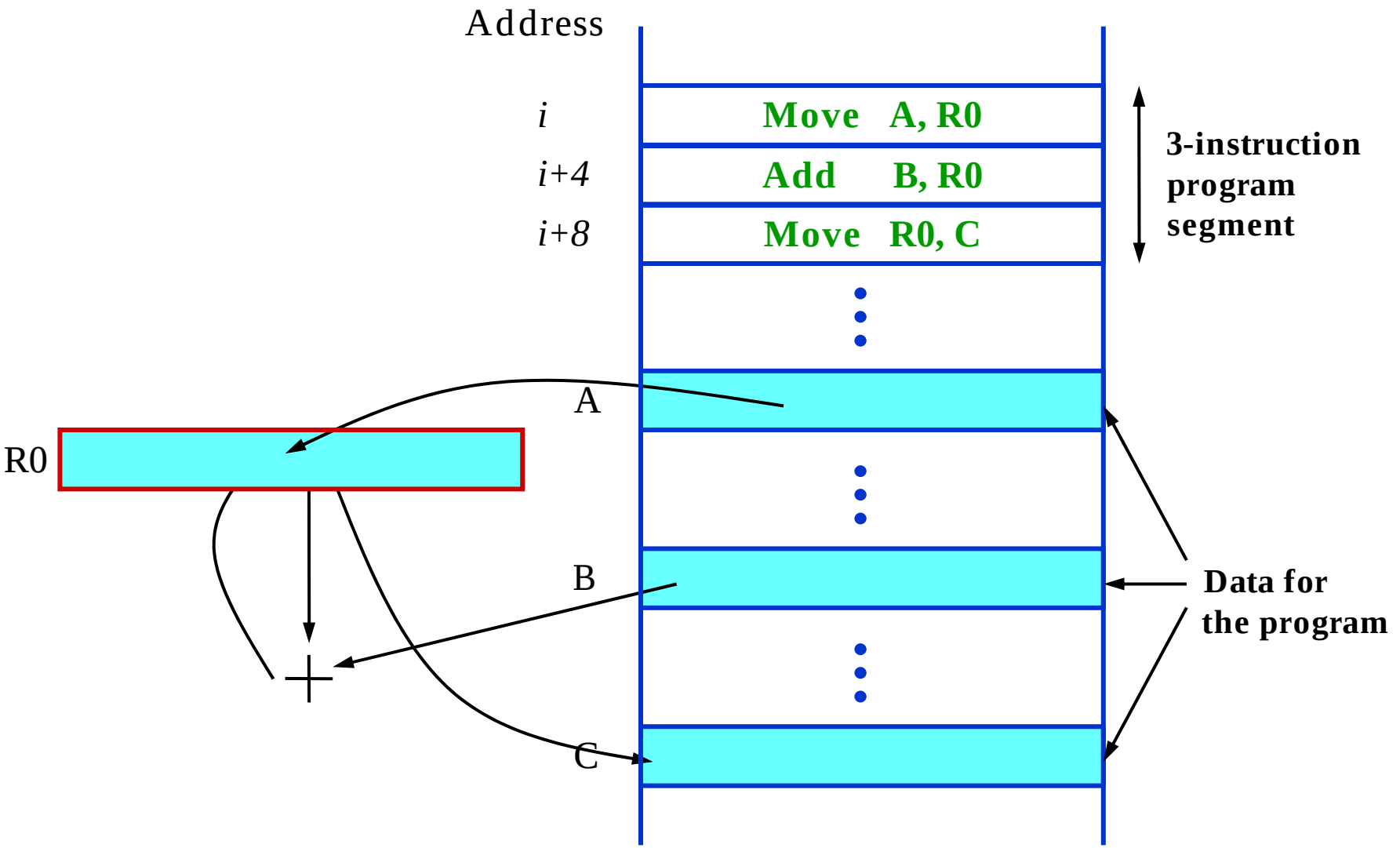
- Two-address instruction
  - Instruction form: **Operation Source, Destination**
  - For example, **Add A, B**: performs the operation B←[A]+[B]. When the sum is calculated, the result is sent to the memory and stored in location B
  - As another example, **Move B, C**: performs the operation C←[B], leaving the contents of location B unchanged
- Three-address instruction
  - Instruction form: **Operation Source1, Source2, Destination**
  - For example, **Add A, B, C**: adds A and B, and the result is sent to the memory and stored in location C
  - If k bits are needed to specify the memory address of each operand, the encoded form of the above instruction must contain 3k bits for addressing purposes in addition to the bits needed to denote the Add operation

Instruction Execution

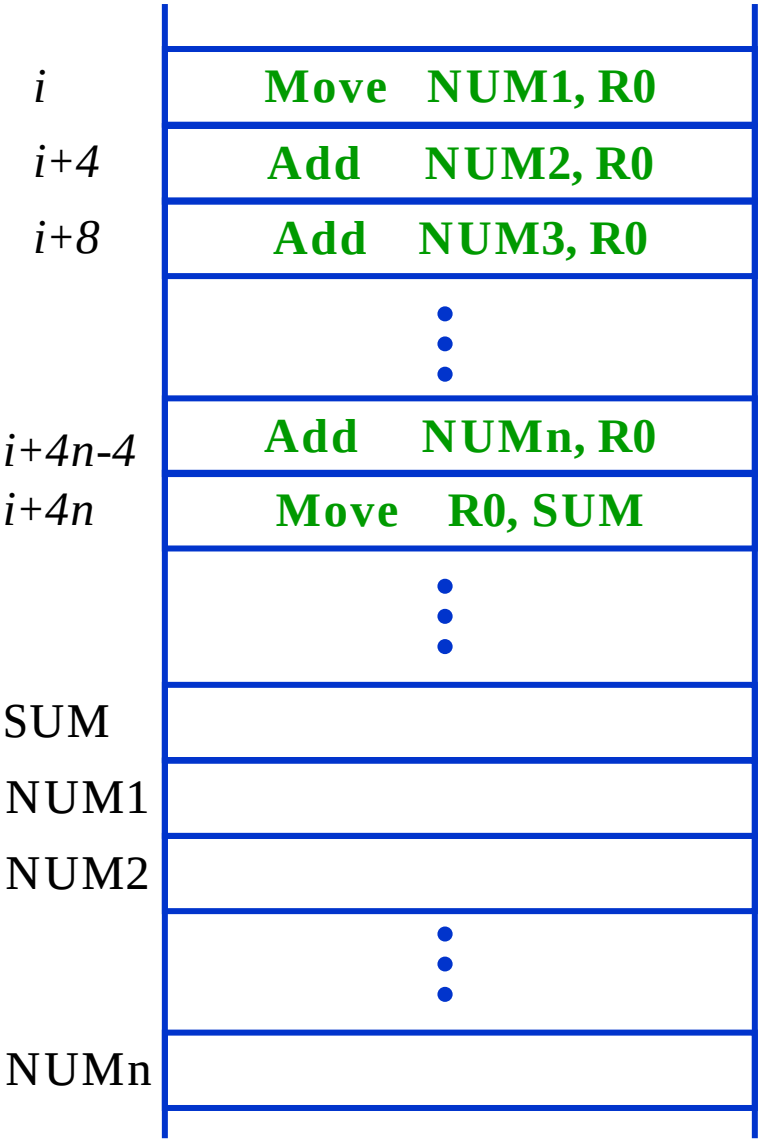
- How a program is executed
  - The processor contains a register called the program counter (PC), which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction must be placed into the PC, then the processor control circuits use the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



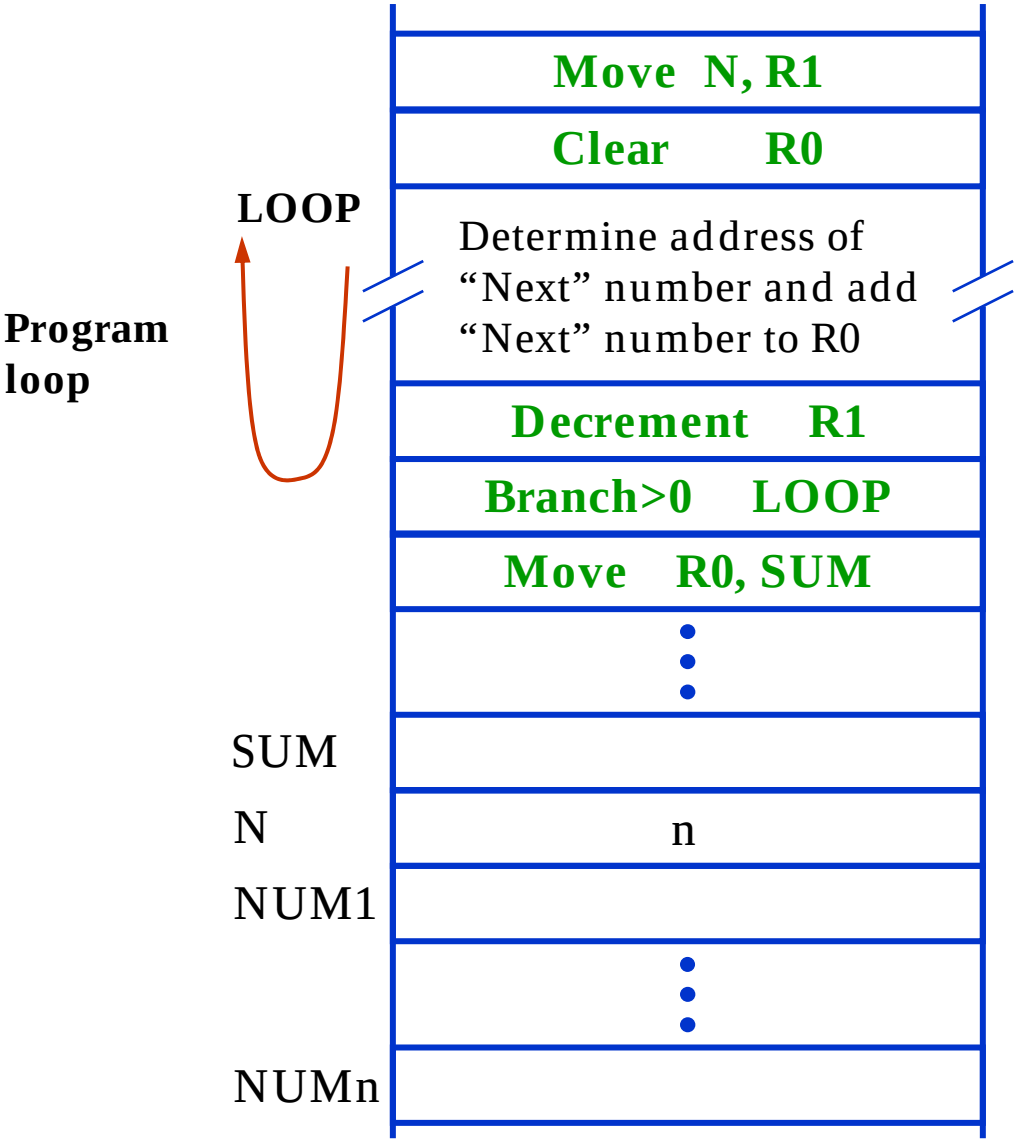
A Program for C←[A]+[B]



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

Condition Codes

- The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recoding required information in individual bits, often called *condition code flags*
- Four commonly used flags are
  - N (negative): set to 1 if the results 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
- N and Z flags caused by an arithmetic or a logic operation, V and C flags caused by an arithmetic operation

Addressing Modes

- Programmers use data structures to represent the data used in computations. These include lists, linked lists, array, queues, and so on
- A high-level language enables the programmer to use constants, local and global variables, pointers, and arrays
- When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities in the instruction set of the computer
- The different ways in which the location of an operand is specified in an instruction are referred to as *addressing modes*

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

Register, Absolute and Immediate Modes

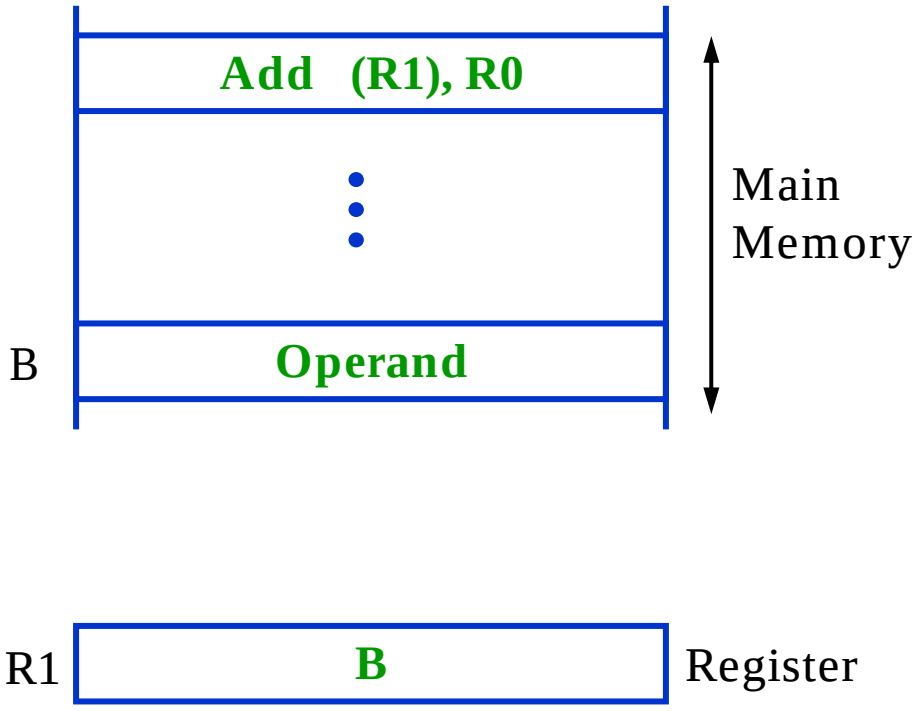
- Register mode: the operand is the contents of a processor register; the name (address) of the register is given in the instruction
  - For example, **Add Ri, Rj** (adds the contents of Ri and Rj and the result is stored in Rj)
- Absolute mode: the operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called Direct)
  - For example, **Move LOC, R2** (moves the content of the memory with address LOC to the register R2)
  - The Absolute mode can represent global variables in a program. For example, a declaration such as Integer A, B;
- Immediate mode: the operand is given explicitly in the instruction

Indirection and Pointers

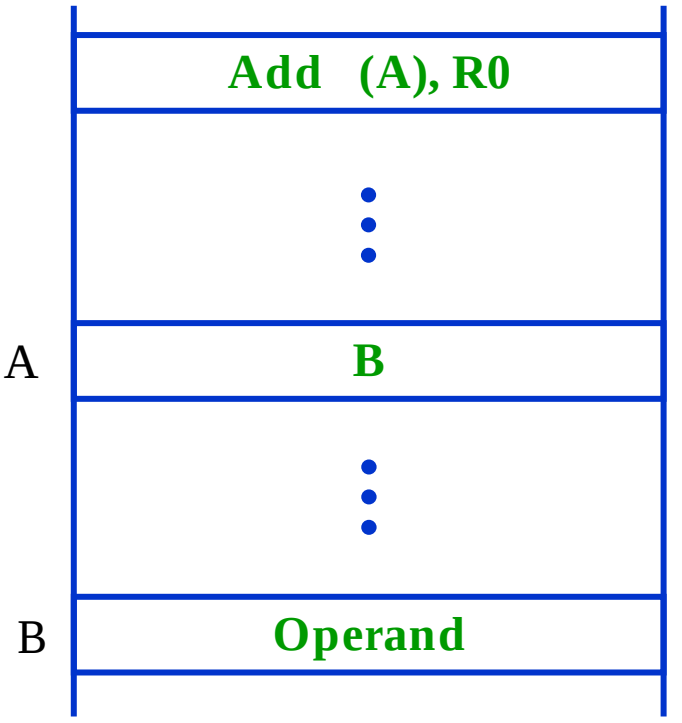
- Indirect mode: the effective address of the operand is the contents of a register or memory location whose address appears in the instruction
- Indirection is denoted by placing the name of the register or the memory address given in the instruction in parentheses
- The register or memory location that contains the address of an operand is called a pointer

Two Types of Indirect Addressing

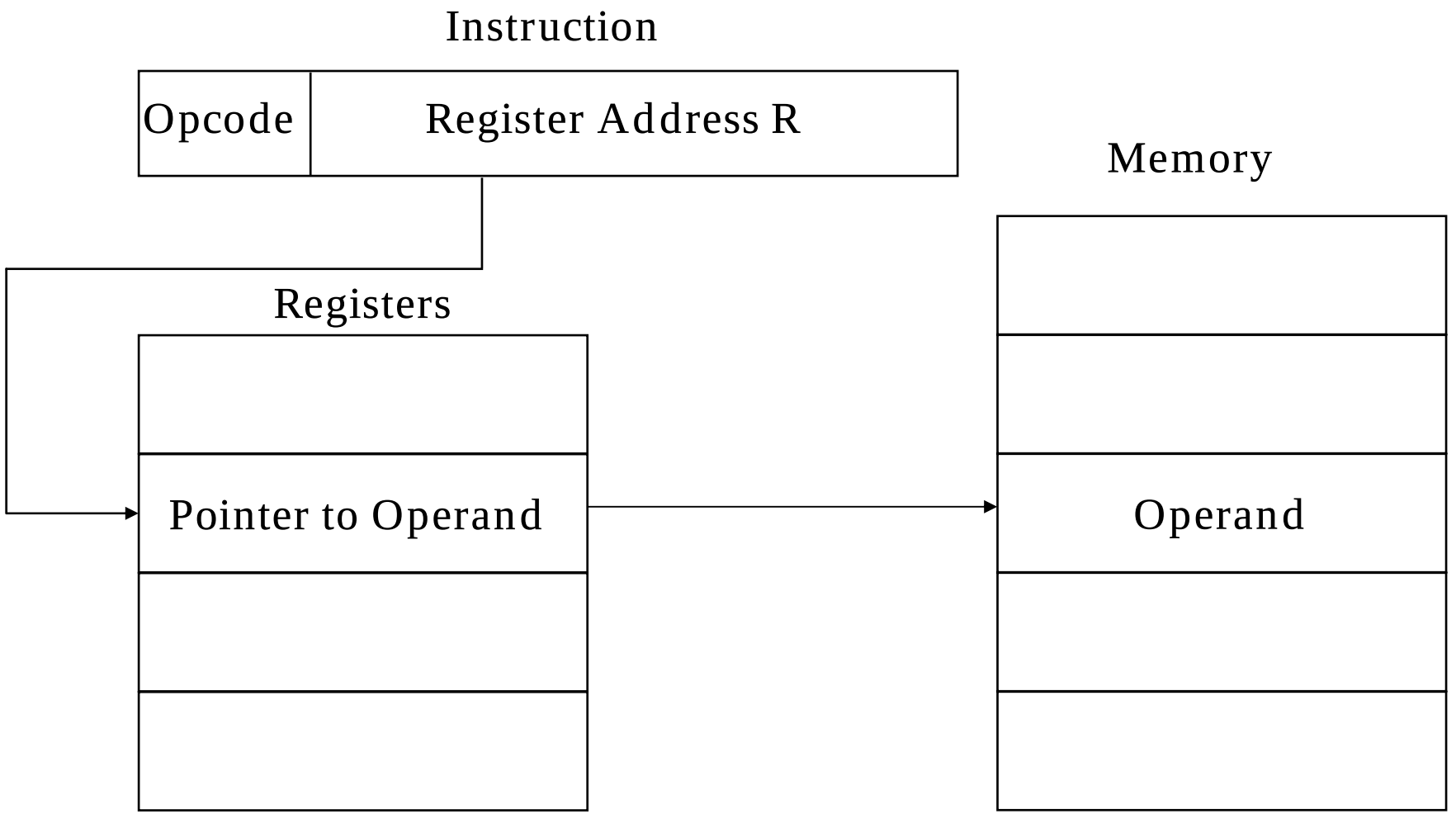
Through a general-purpose register



Through a memory location



Register Indirect Addressing Diagram



Using Indirect Addressing in a Program

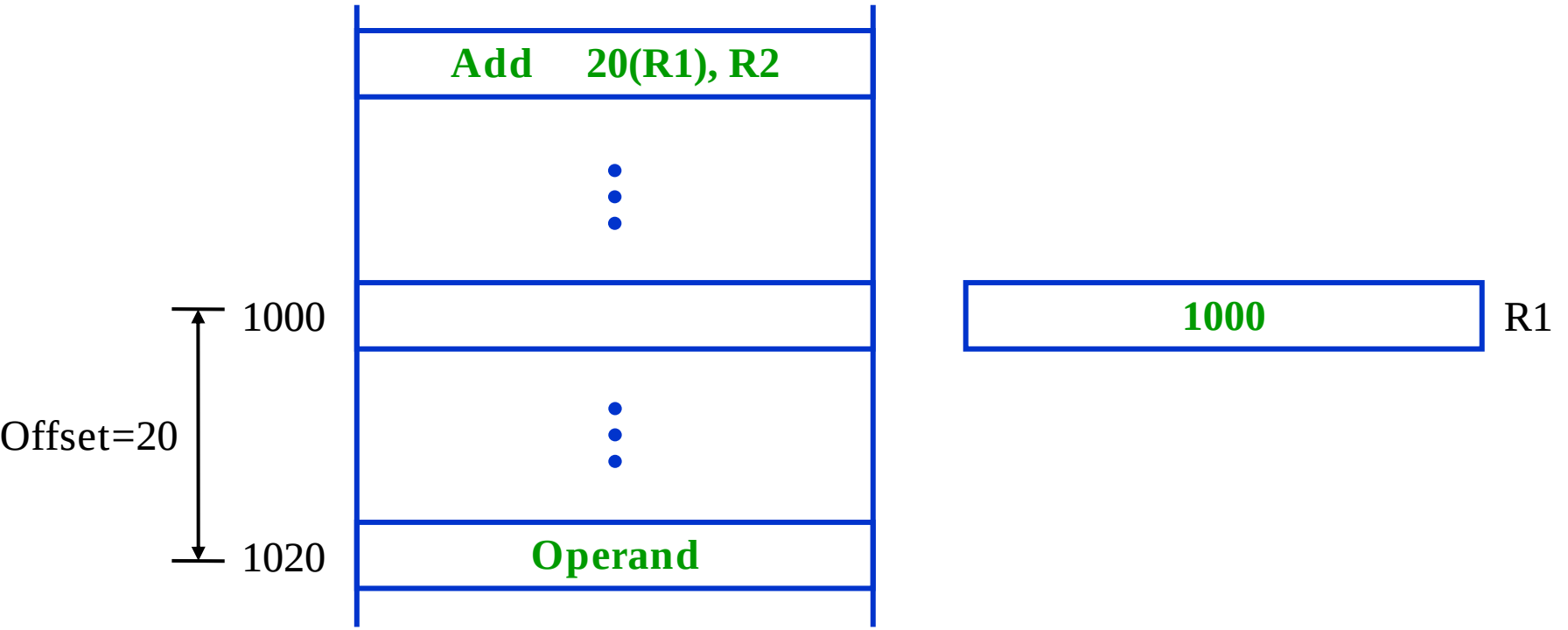
Address	Contents		
	Move	N, R1	Initialization
	Move	#NUM1, R2	
	Clear	R0	
LOOP	Add	(R2), R0	
	Add	#4, R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0, SUM	

Indexing 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. It is referred to as an index register
  - The index mode is useful in dealing with lists and arrays
  - We denote the Index mode symbolically as X(Ri), where X denotes the constant value contained in the instruction and Ri 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

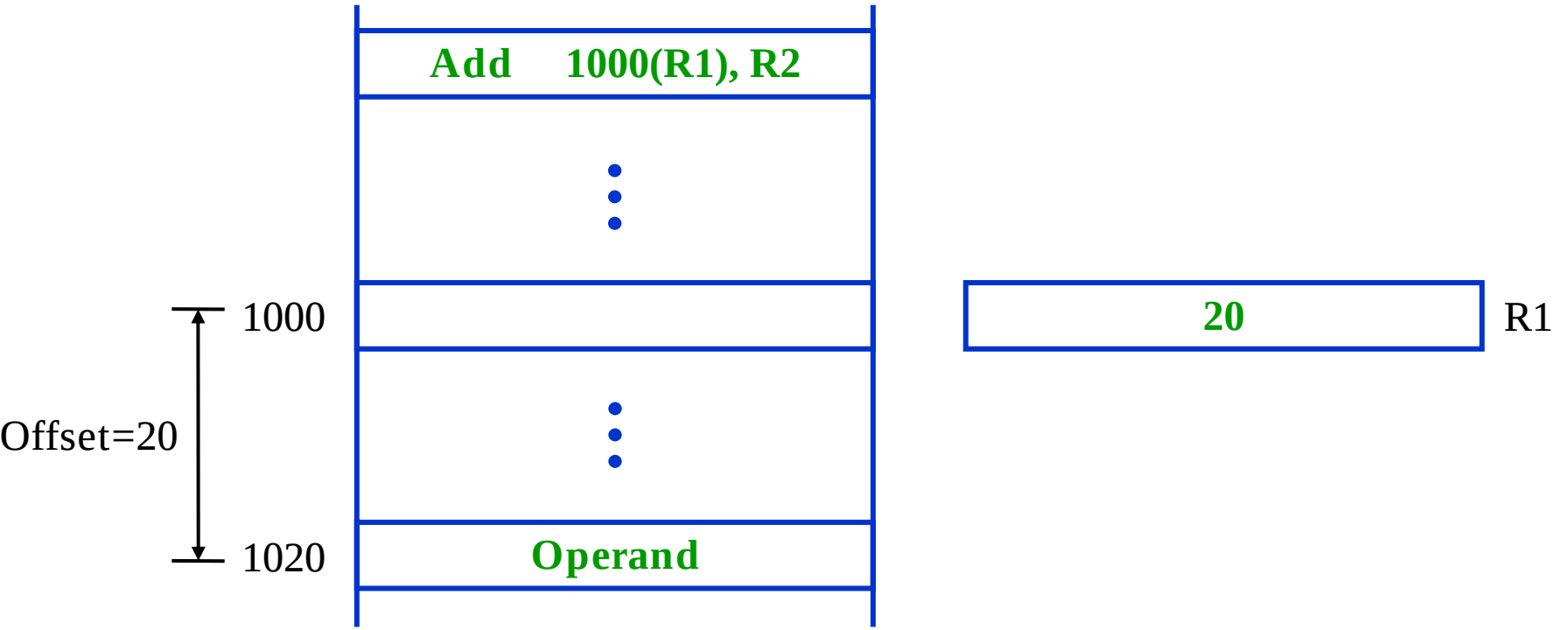
Indexed Addressing

Offset is given as a constant

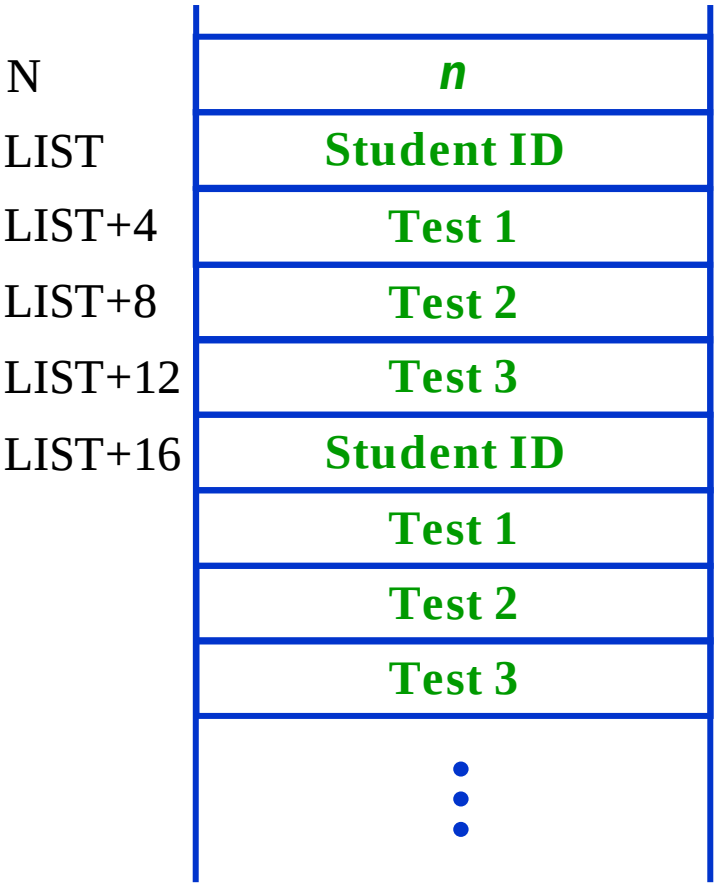


Indexed Addressing

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
```

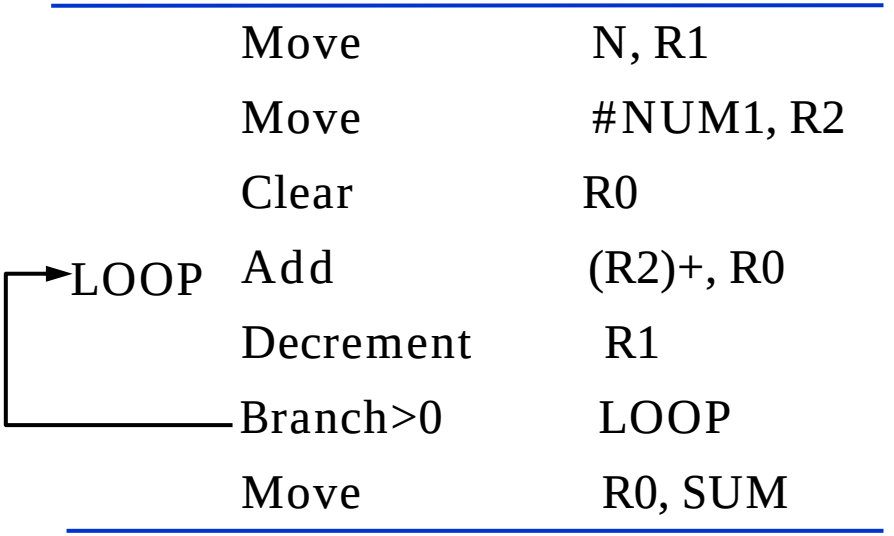
Variations of Indexed Addressing Mode

- 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 Ri and Rj
  - The second register is usually called the base register
  - This mode implements a two-dimensional array
- Another version of the Index mode use two registers plus a constant, which can be denoted as X(Ri,Rj)
  - The effective address is the sum of the constant X and the contents of registers Ri and Rj
  - This mode implements a three-dimensional array

Additional Modes

- 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
  - The Autoincrement mode is denoted as (Ri)+
- 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
  - The Autodecrement mode is denoted as –(Ri)

An Example of Autoincrement Addressing



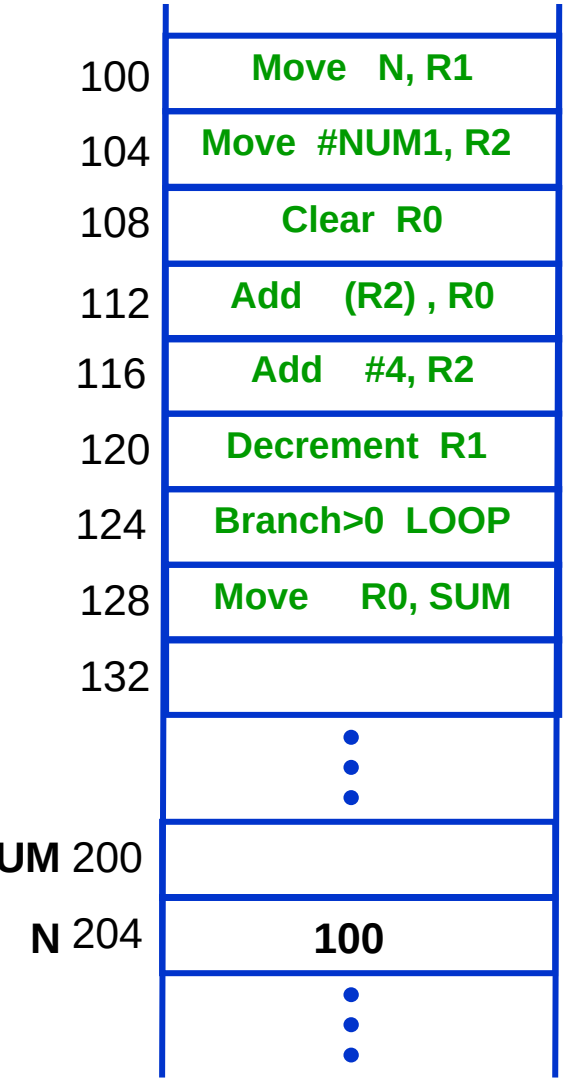
Assembly Language

- A complete set of symbolic names and rules for their use constitute a programming language, generally referred to as an assembly language
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*
- When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program
- The user program in its original alphanumeric text format is called a *source program*, and the assembled machine language program is called an *object program*

Assembler Directives

- In addition to providing a mechanism for representing instructions in a program, the assembly language allows the programmer to specify other information needed to translate the source program into the object program
- Suppose that the name SUM is used to represent the value 200. The fact may be conveyed to the assembler program through a statement such as
  - **SUM EQU 200**
- This statement does not denote an instruction that will be executed when the object program is run; it will not even appear in the object program
  - Such statements, called *assembler directives* (or *commands*)

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

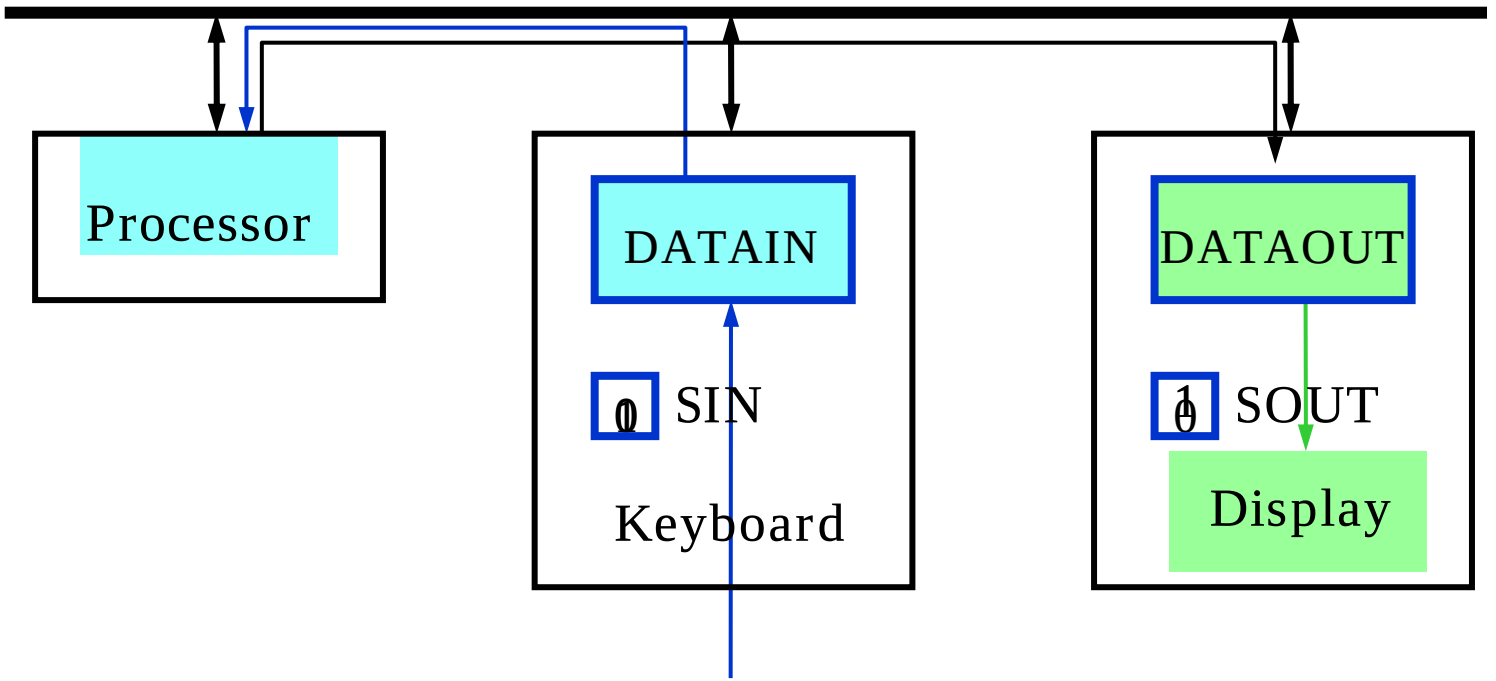
Assembly language representation

Number Notation

- When dealing with numerical values, most assemblers allow numerical values to be specified in different ways
- For example, consider the number 93, which is represented by the 8-bit binary number 01011101. If the value is to be used as immediate operand,
  - It can be given as a decimal number, as in the instruction ADD #93, R1
  - It can be given as a binary number, as in the instruction ADD #%01011101,R1 (a binary number is identified by a prefix symbol such as percent sign)
  - It also can be given as a hexadecimal number, as in the instruction ADD #$5D, R1 (a hexadecimal number is identified by a prefix symbol such as dollar sign)

Basic Input/Output Operations

- Bus connection for processor, keyboard, and display



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

Wait Loop

- In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and I/O device
- Wait loop for Read operation
  - READWAIT Branch to READWAIT if SIN=0
    Input from DATAIN to R1
- Wait loop for Write operation
  - WRITEWAIT Branch to WRITEWAIT if SOUT=0
    Output from R1 to DATAOUT
- We assume that the initial state of SIN is 0 and the initial state of SOUT is 1

Memory-Mapped I/O

- Many computers use an arrangement called memory-mapped I/O in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT
- Thus no special instructions are needed to access the contents of these registers; data can be transferred between these registers and the processor using instructions that we have discussed, such as Move, Load, or Store
- Also, the status flags SIN and SOUT can be handled by including them in device status registers, one for each of the two devices

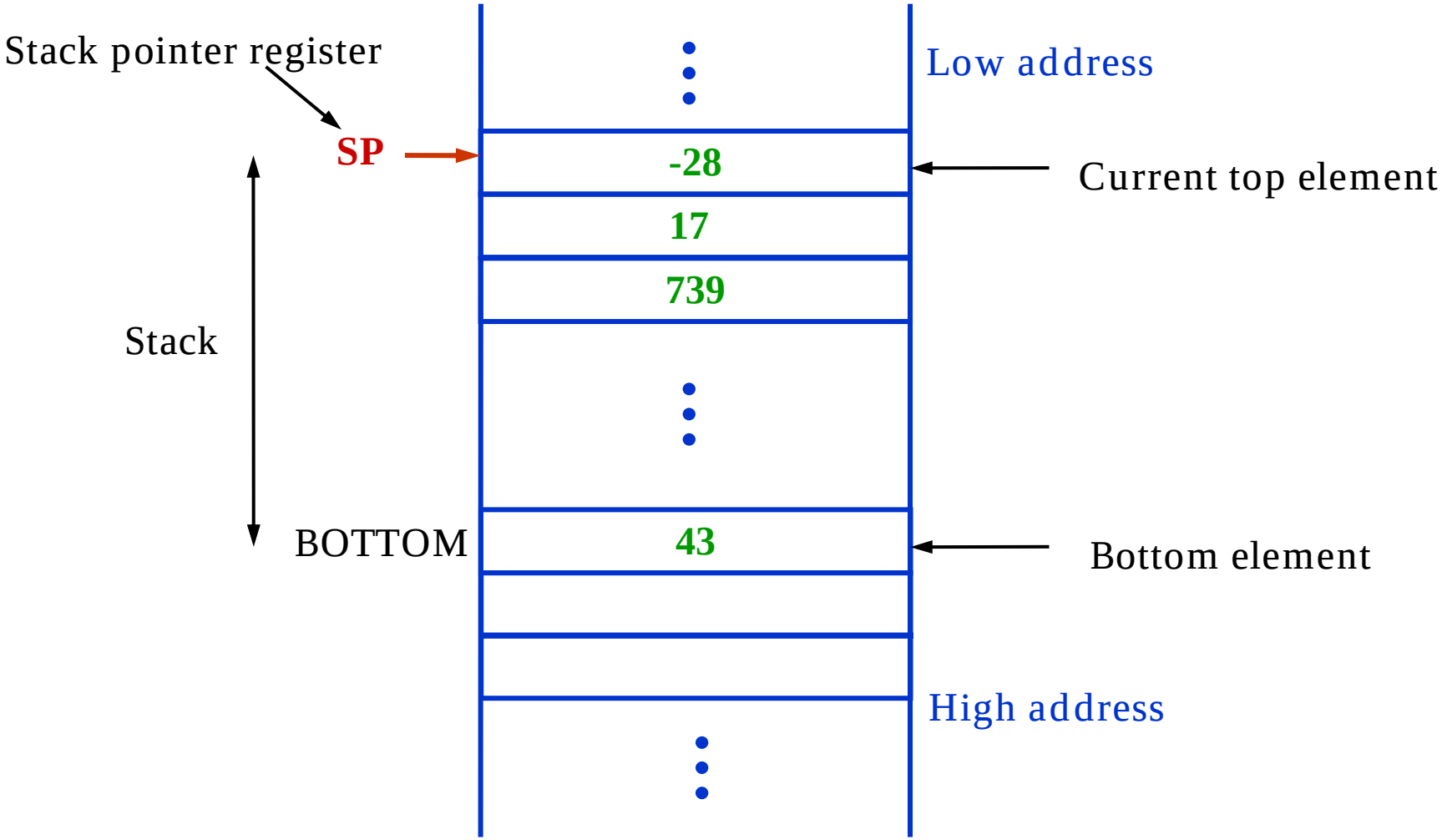
Read and Write Programs

- 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

Stacks and Queues

- A stack is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only
  - It is also called a last-in-first-out (LIFO) stack
  - A stack has two basic operations: push and pop
  - The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.
- Another useful data structure that is similar to the stack is called a queue
  - Data are stored in and retrieved from a queue on a first-in-first-out (FIFO) basis
  - Two pointers are needed to keep track of the two ends of the queue

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
```

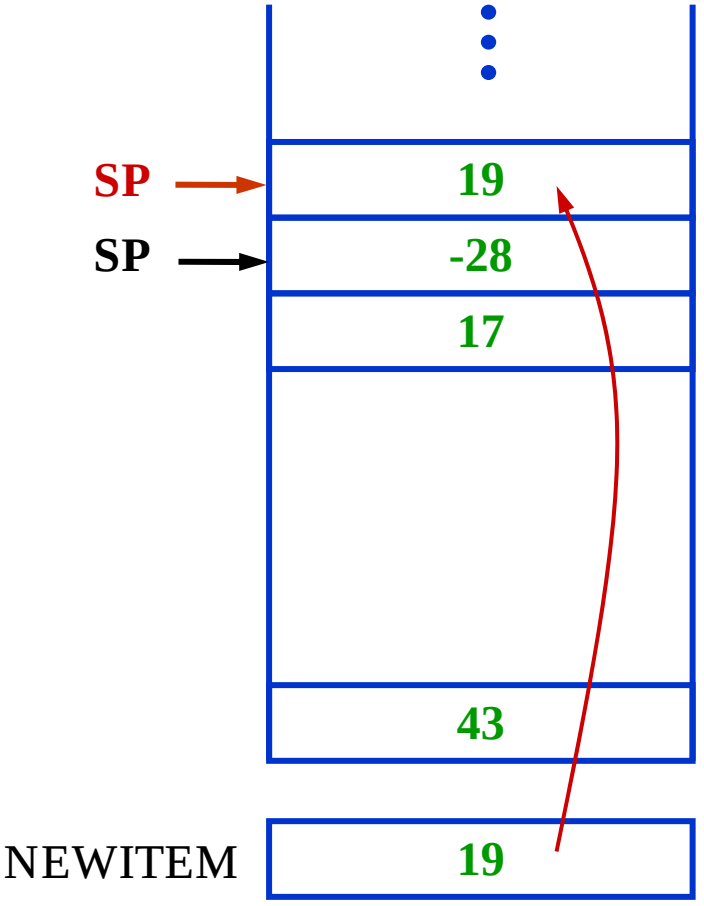
- If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be implemented by the single instruction

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

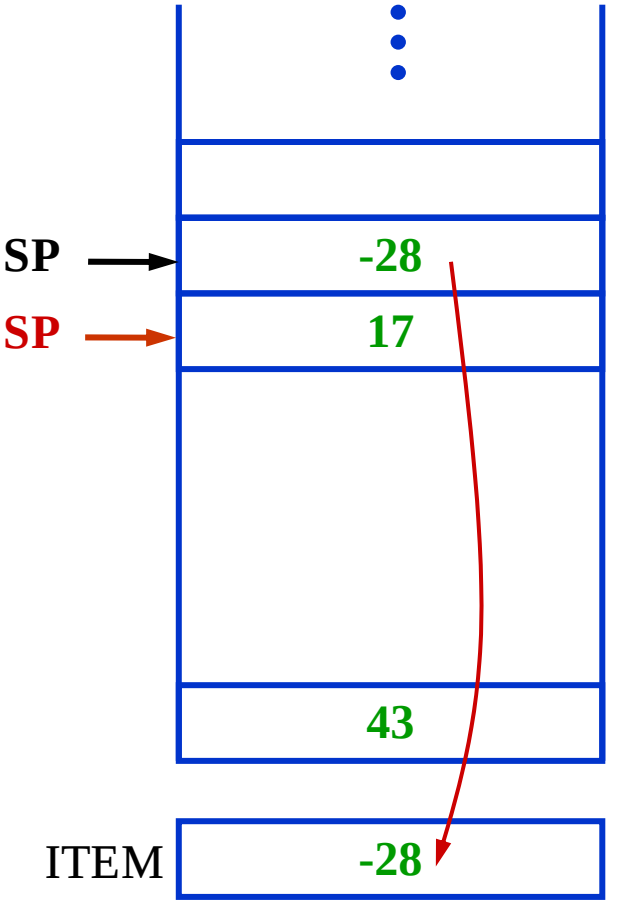
- And the pop operation can be implemented as

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

Examples



Push operation



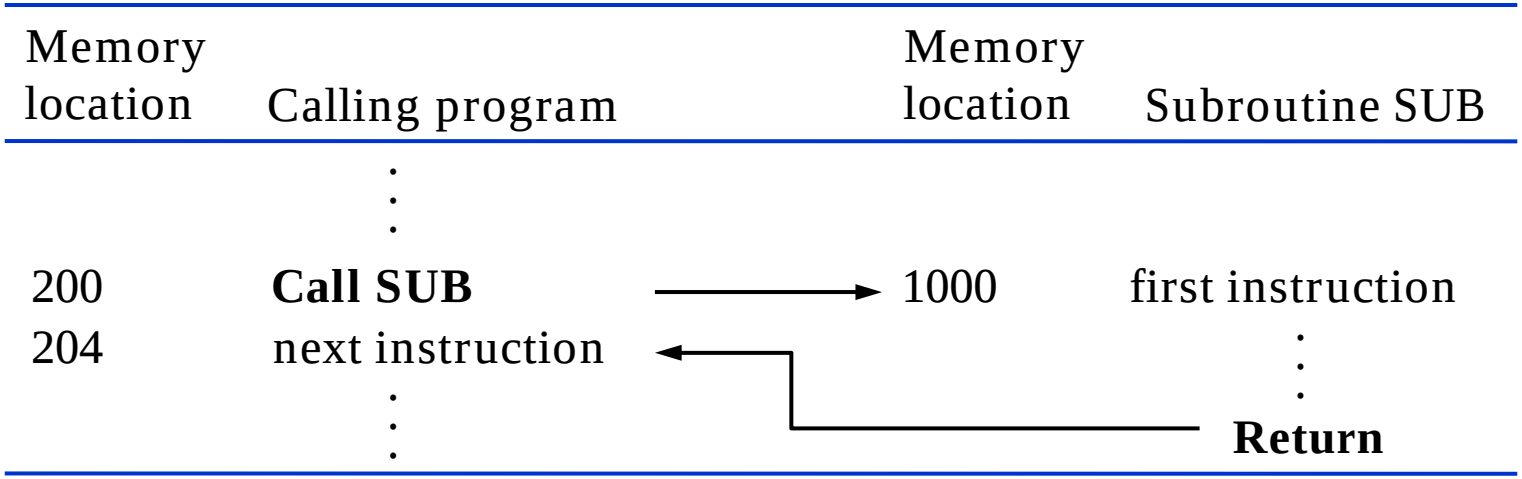
Pop operation

Checking for Empty and Full Errors

- When a stack is used in a program, it is usually allocated a fixed amount of space in the memory
  - We must avoid pushing an item onto the stack when the stack has reached in its maximum size, i.e., the stack is full
  - On the other hand, we must avoid popping an item off the stack when the stack has reached in its minimum size, i.e., the stack is empty
- Routine for a safe pop or a safe push operation
  - Compare src, dst
  - Perform [dst]-[src]
  - Sets the condition code flags according to the result

Subroutines

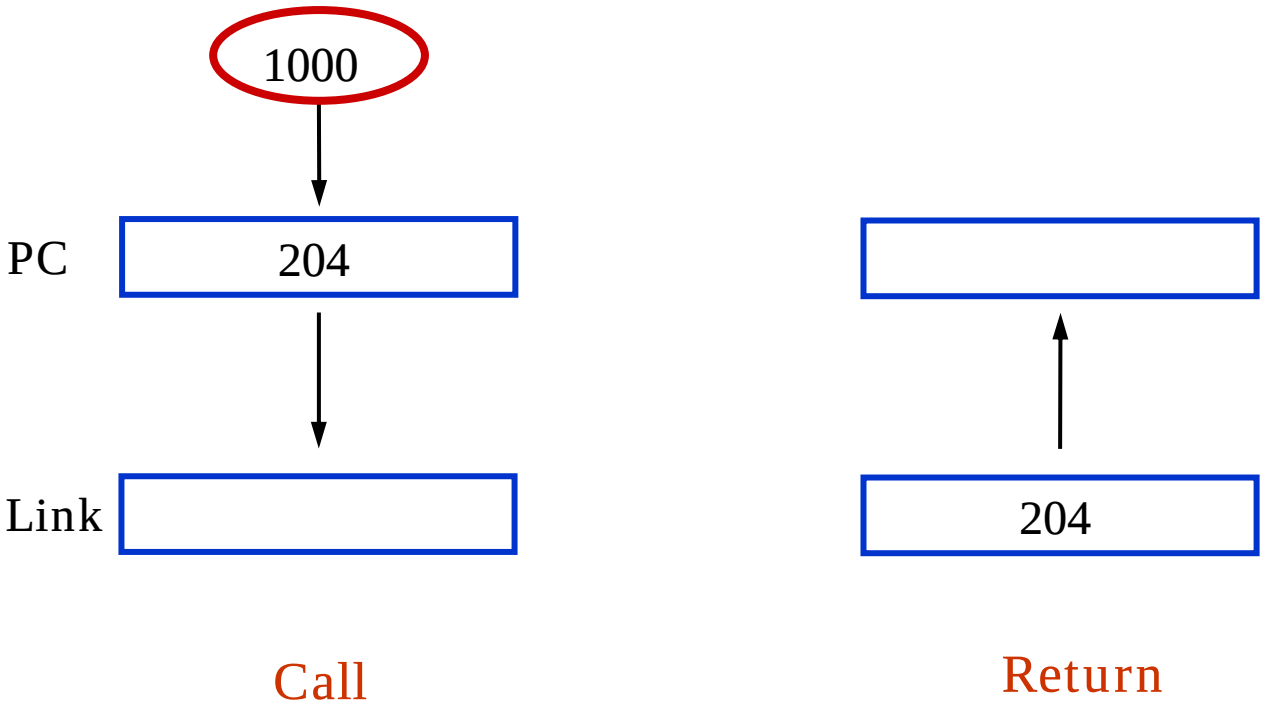
- In a given program, it is often necessary to perform a particular subtask many times on different data values. Such a subtask is called a *subroutine*.



- 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

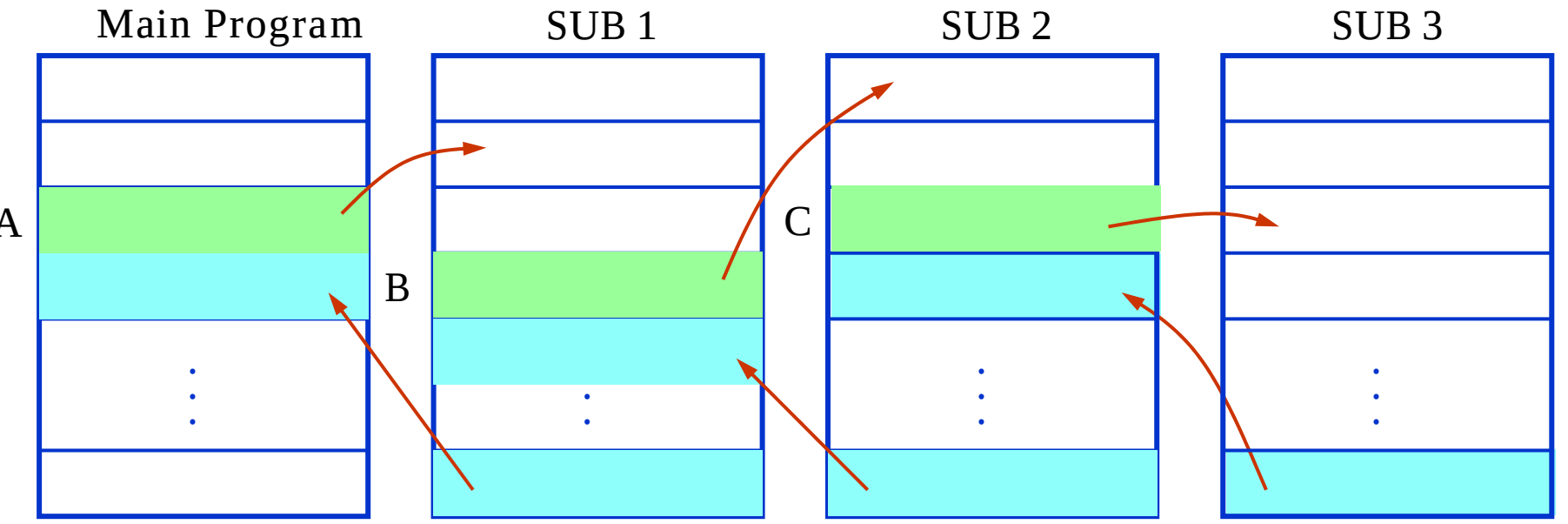
- The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method
- Subroutine linkage using a link register



Subroutine Nesting

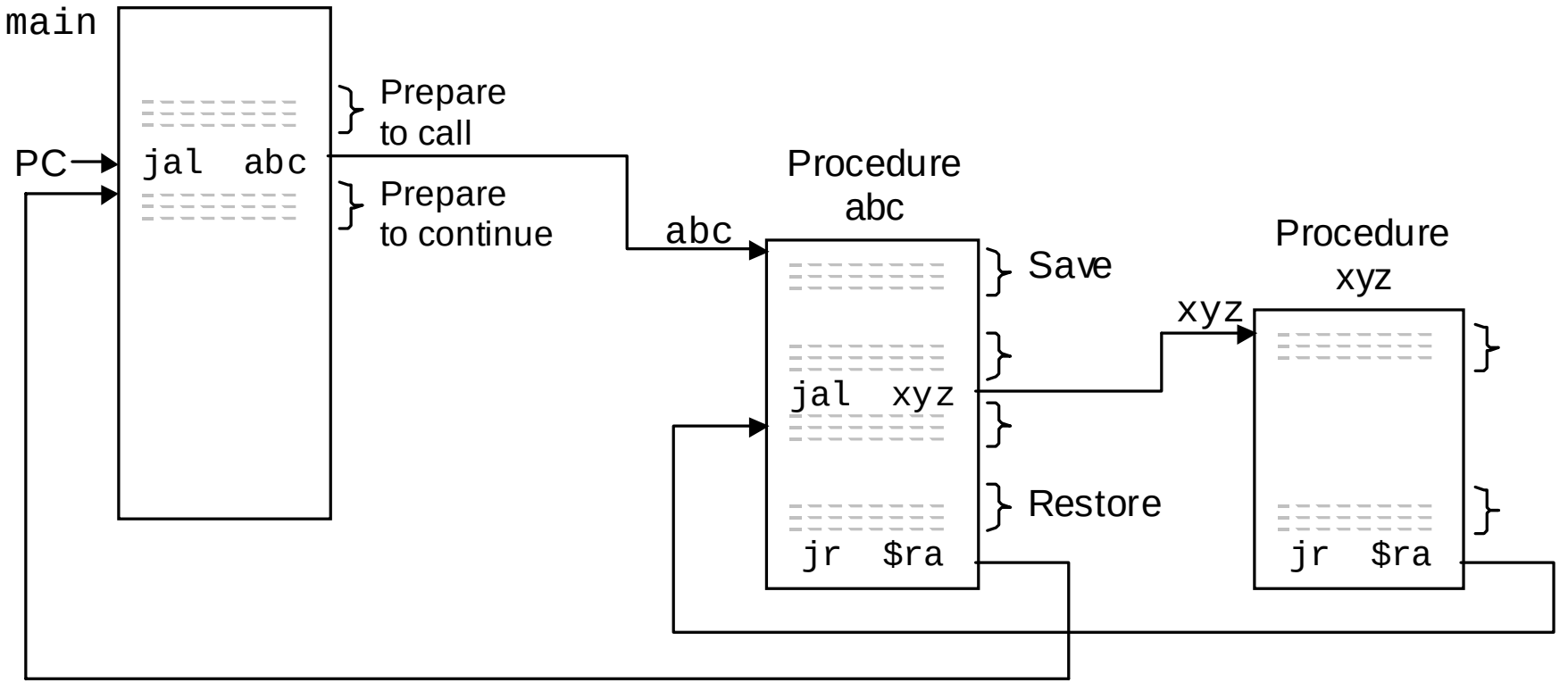
- A common programming practice, called subroutine nesting, is to have one subroutine call another
- Subroutine nesting call be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it
- The return address needed for this 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

Example of Subroutine Nesting



C+4
B+4
A+4

Example of Subroutine Nesting

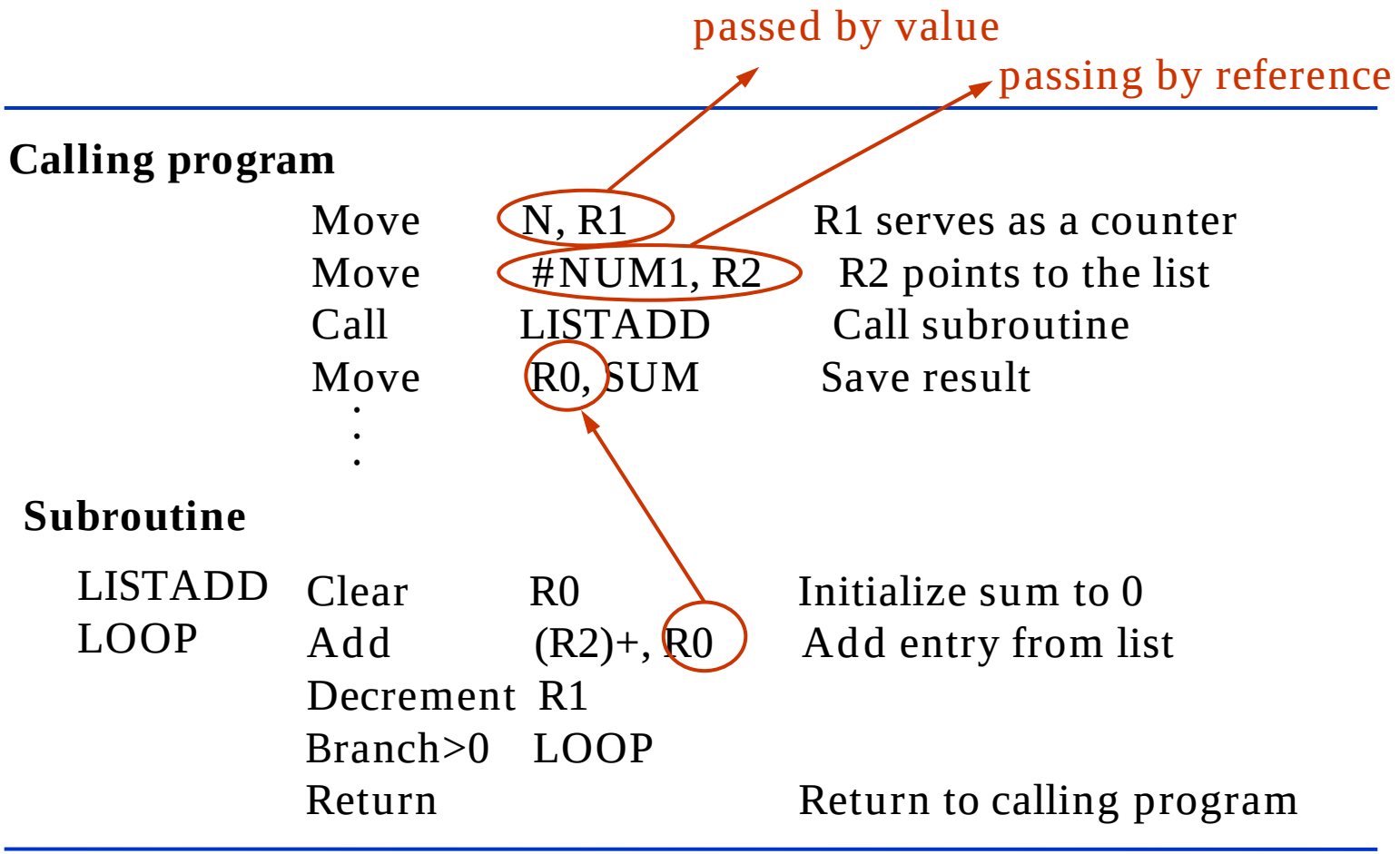


[Source: B. Parhami, UCSB]

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

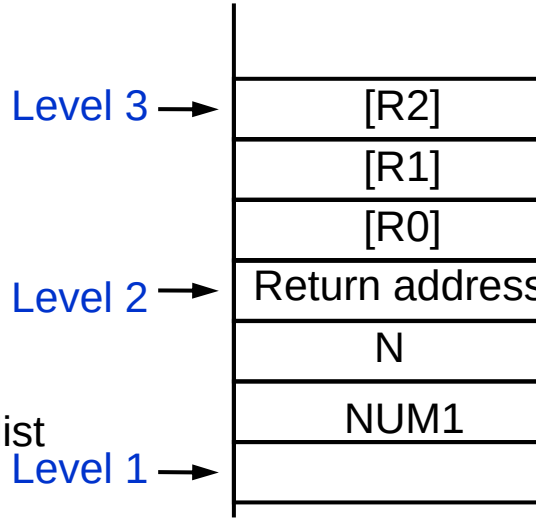
Passing Parameters with Registers



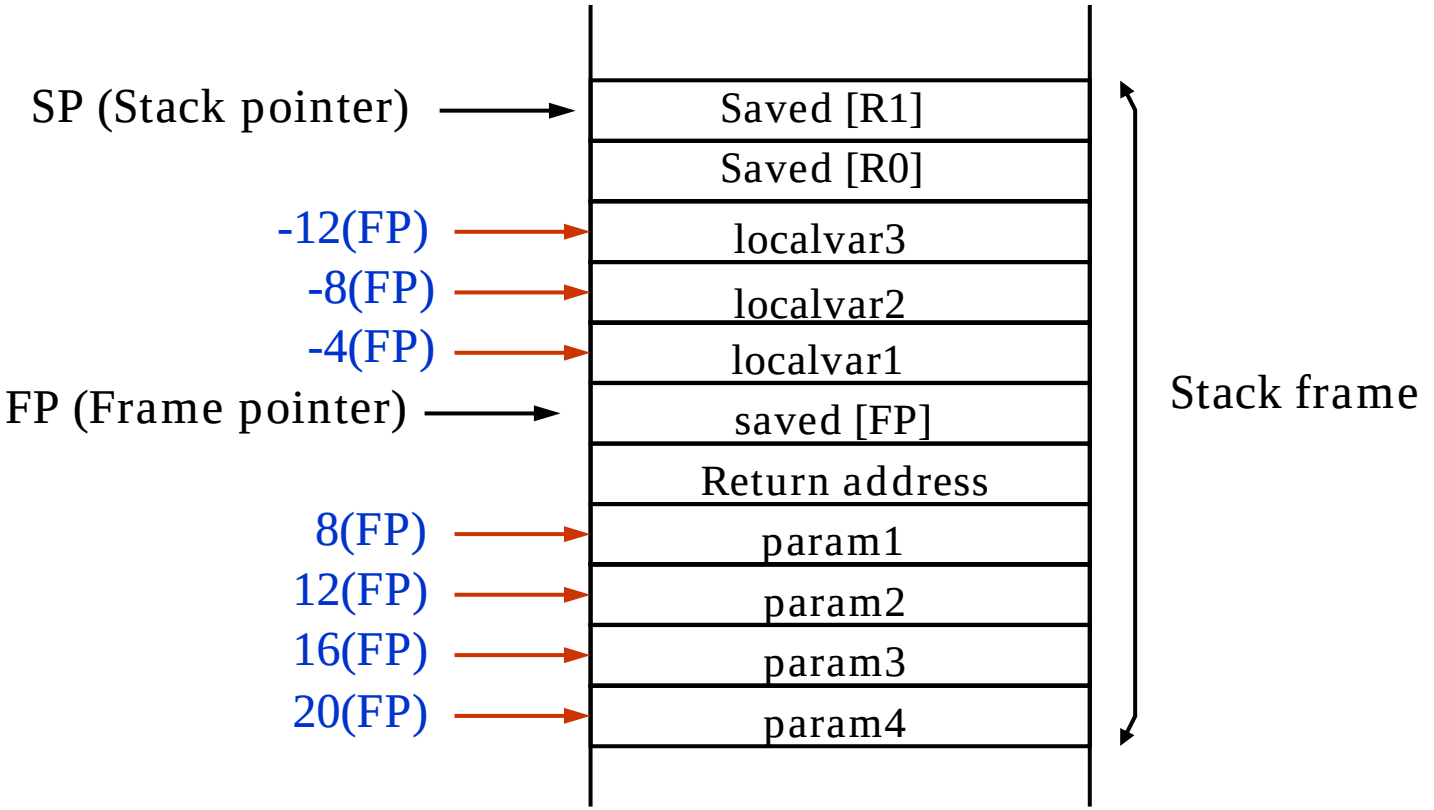
Passing Parameters with Stack

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

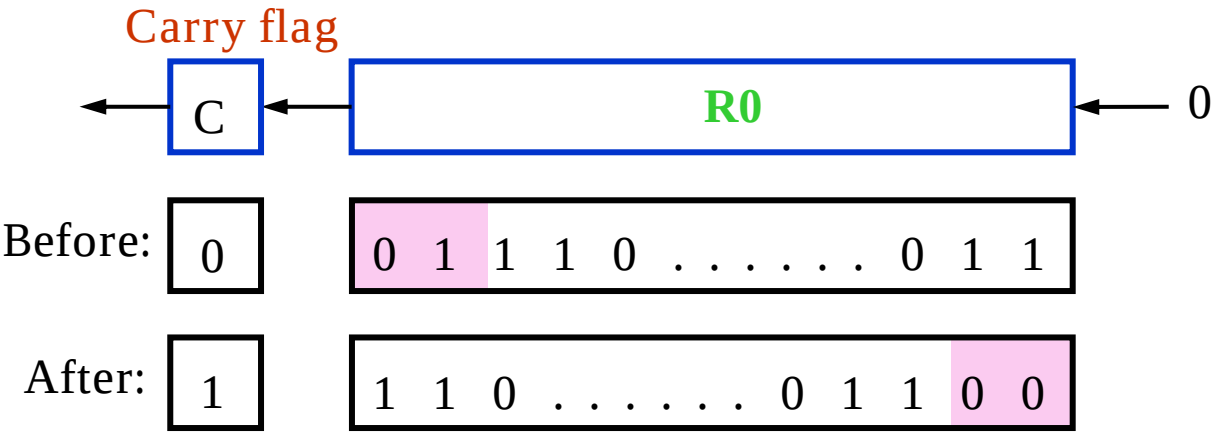

SP (Stack pointer)
Saved [R1]
Saved [R0]
-12(FP)
localvar3
-8(FP)
localvar2
-4(FP)
localvar1
FP (Frame pointer)
saved [FP]
Return address
8(FP)
param1
12(FP)
param2
16(FP)
param3
20(FP)
param4
Stack frame

Shift Instructions

- Logical shifts

Logic shift left

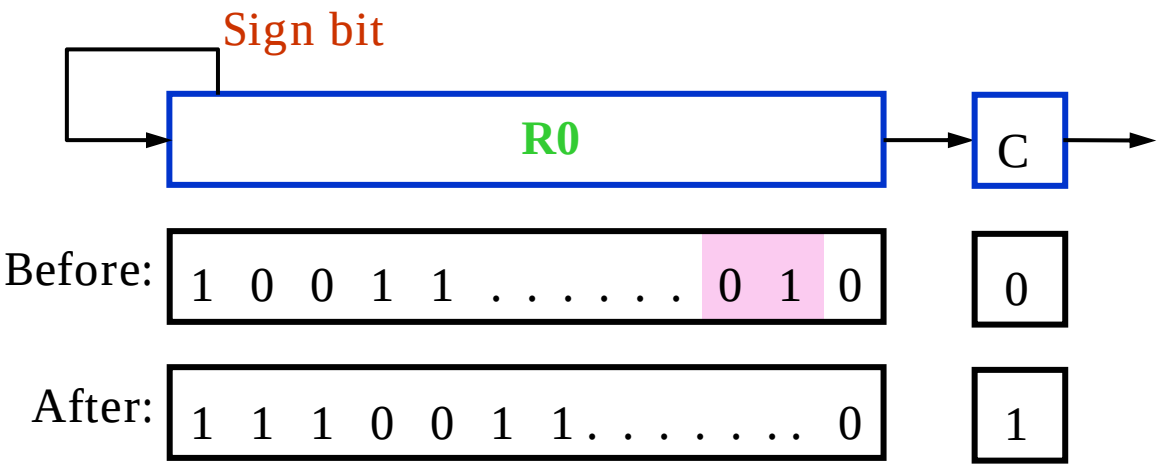
LShiftL #2, R0



- Arithmetic shifts

shift right

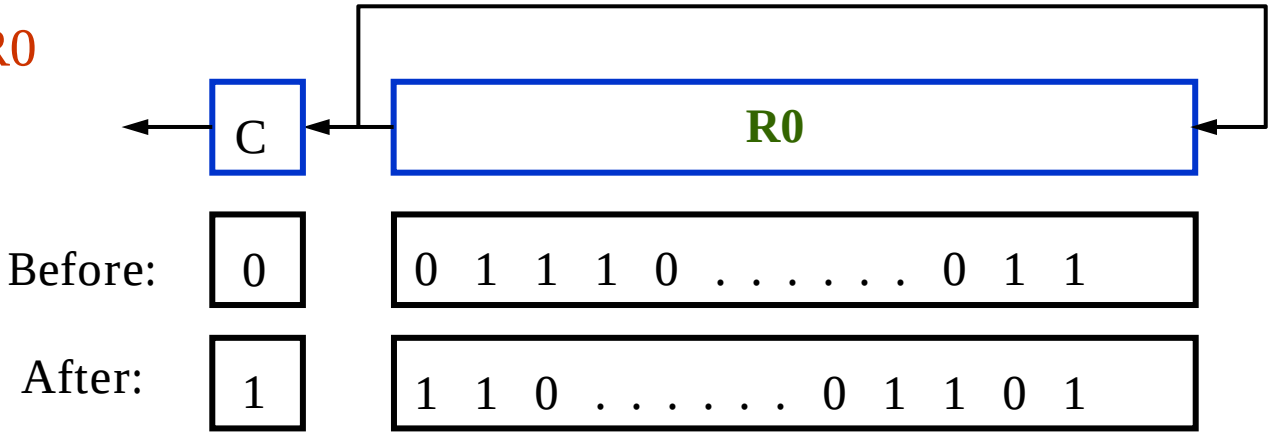
AShiftR #2, R0



Rotate Instructions

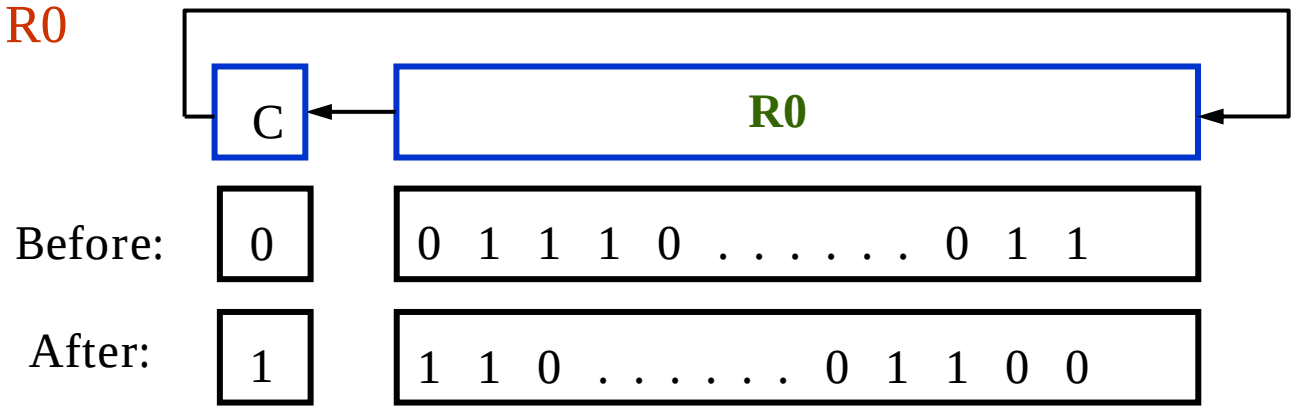
- Rotate left without carry

RotateL #2, R0

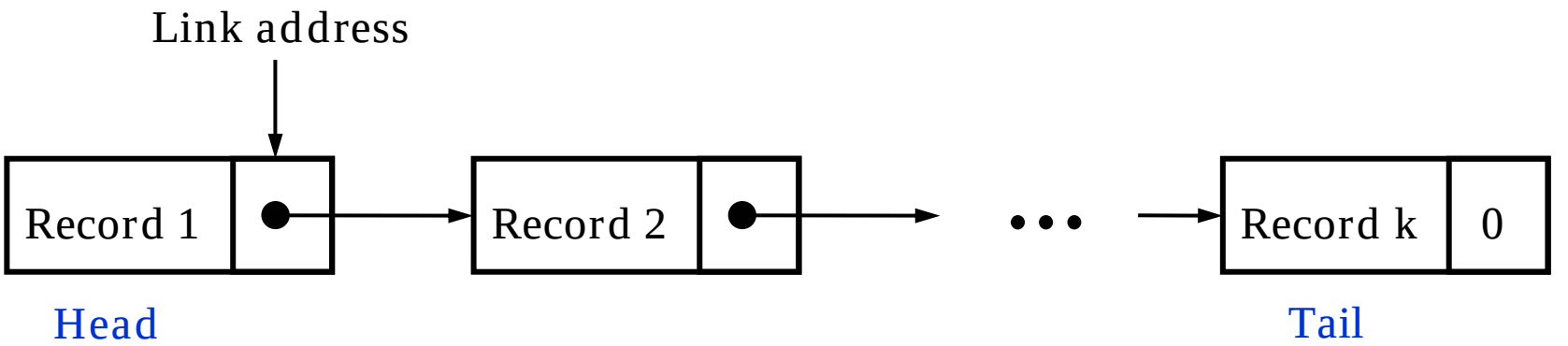


- Rotate left with carry

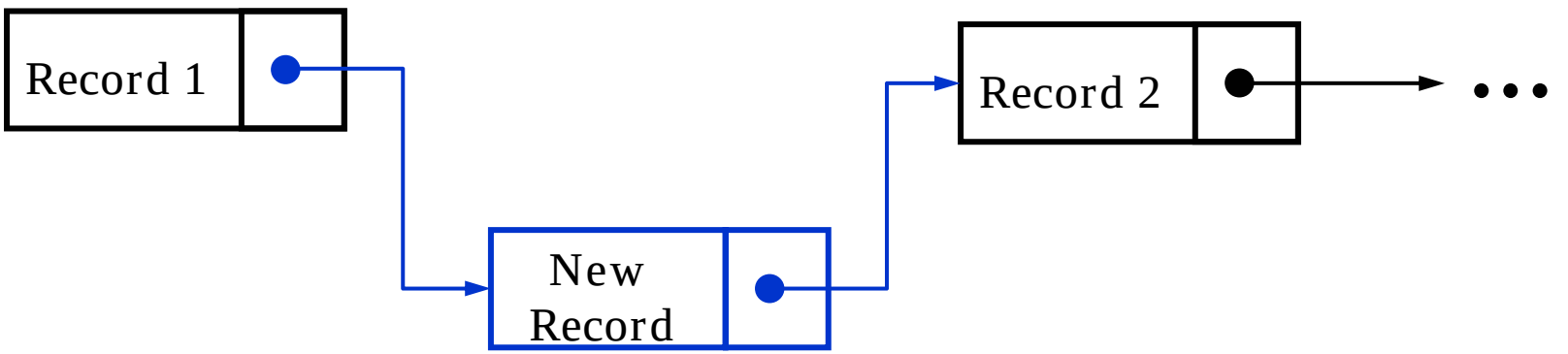
RotateLC #2, R0



Linked List

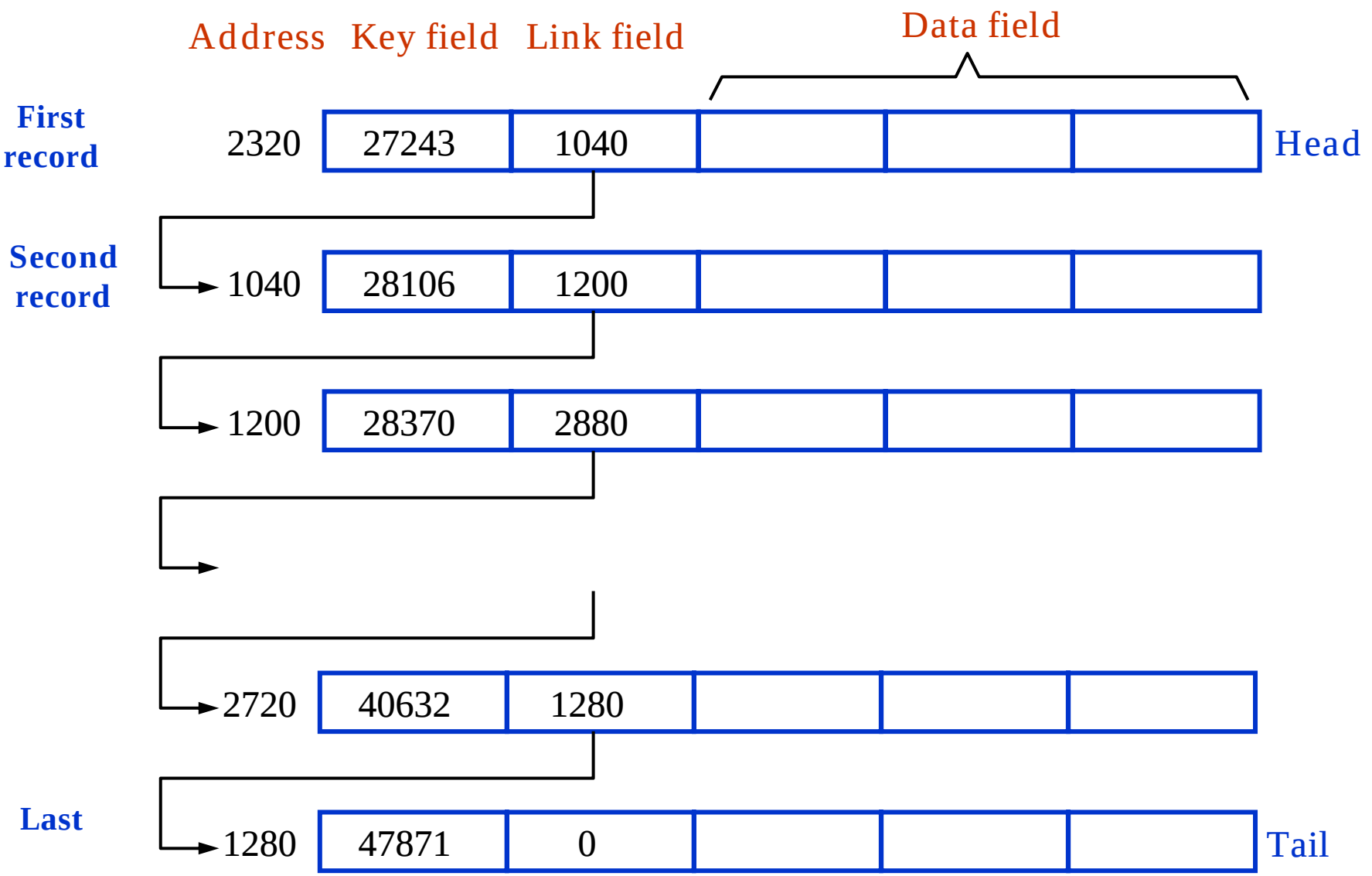


Linking structure



Inserting a new record

A List of Student Test Scores



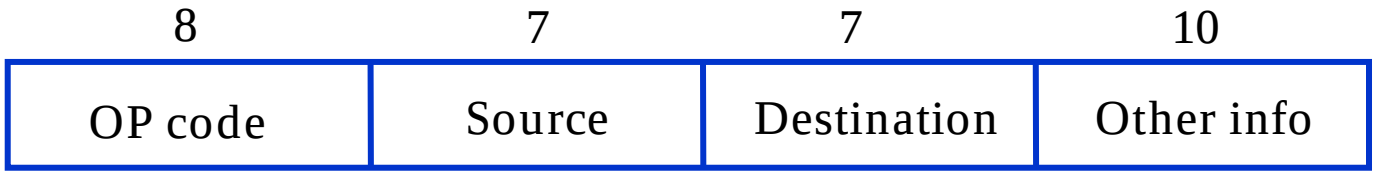
Encoding of Machine Instructions

- To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as machine instructions. The instructions that use symbolic names and acronyms are called assembly language instructions, which are converted into the machine instructions using assembler program
- For a given instruction, the type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the OP code
- In addition to the OP code, the instruction has to specify the source and destination registers, and addressing mode, etc,

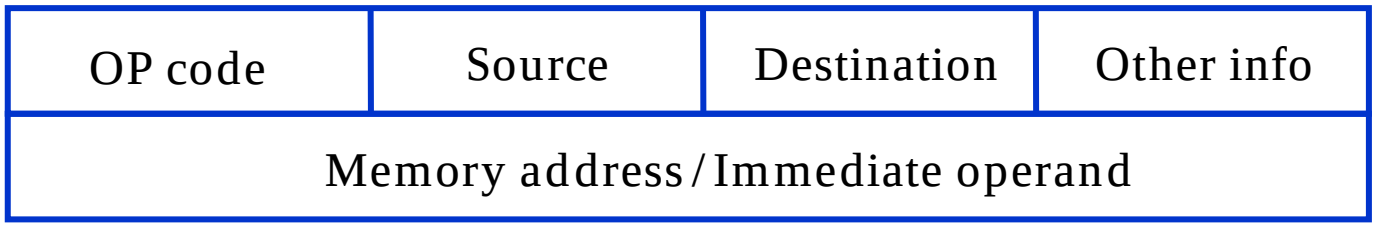
Examples

- Assume that 8 bits are allocated for OP code, and 4 bits are needed to identify each register, and 6 bits are needed to specify an addressing mode
- The instruction Move 24(R0), R5
  - Require 16 bits to denote the OP code and the two registers
  - Require 6 bits to choose the addressing mode
  - Only 10 bits are left to give the index value
- The instruction LshiftR #2, R0
  - Require 18 bits to specify the OP code, the addressing modes, and the register
  - This limits the size of the immediate operand to what is expressible in 14 bits
- In the two examples, the instructions can be encoded in a 32-bit word.

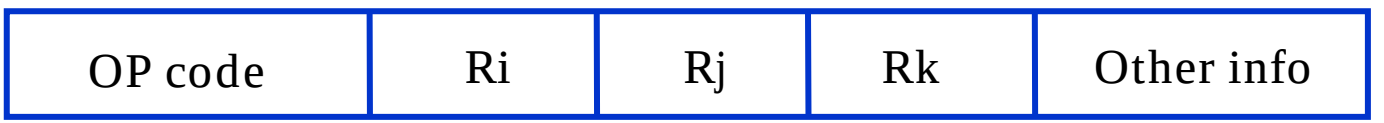
Encoding Instructions into 32-bit Words



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)