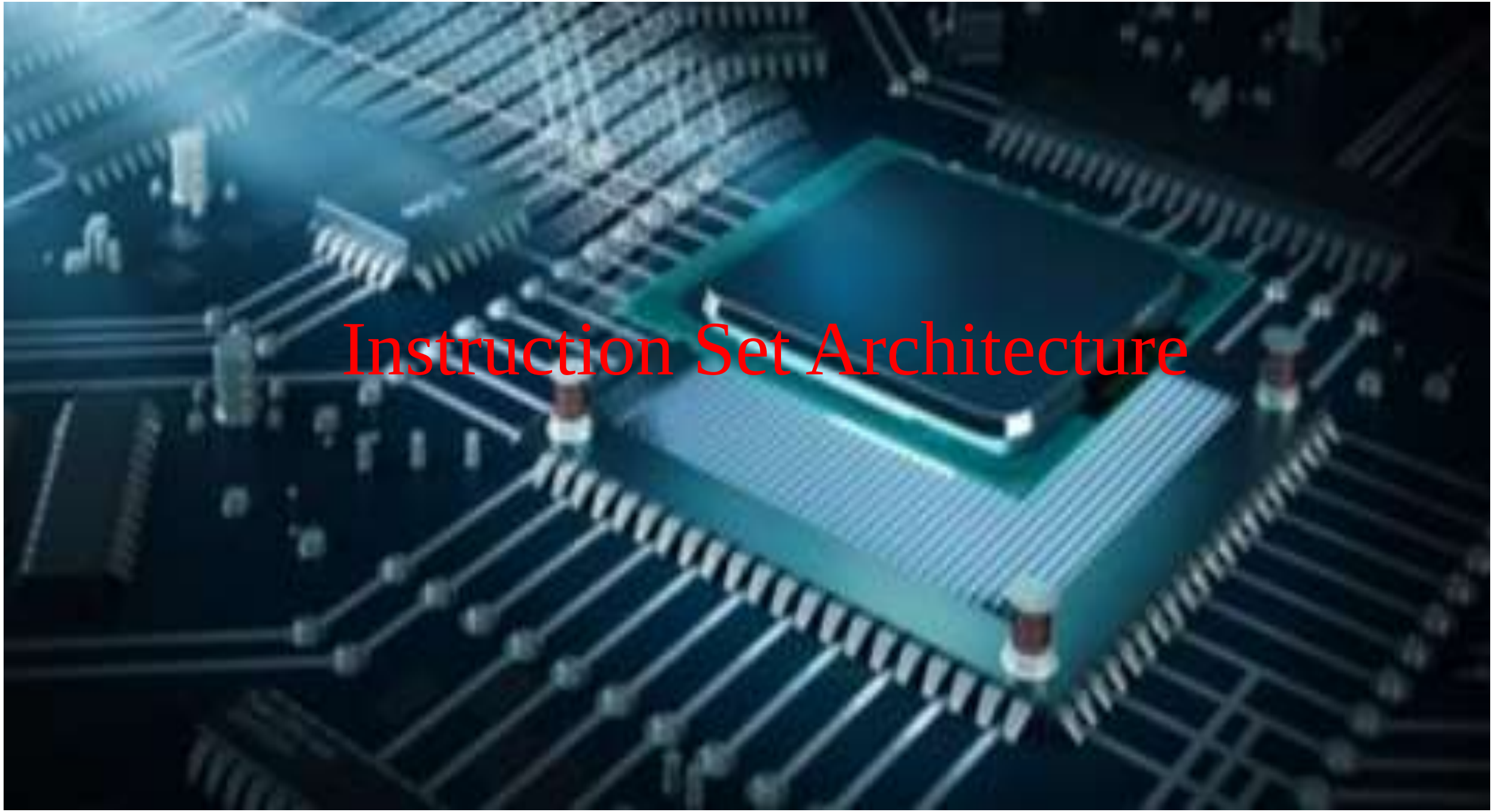

Instruction Set Architecture

What is Instruction Set?

ISA is an interface between hardware and software.

What is Instruction Set?

- It is collection of instruction that execute by processor.
- The instruction set are commands for the processor, to tell it what it needs to do.
- guidelines in low level language(Assembly language) that tells how to execute a function in a system
- ISA is machine understand able form that assists in interaction between hardware and software ,makes both compatible with one another.

What is Instruction Set?

- To perform operations in computers ,instructions (Assembly language mnemonics) are written in machine understandable form on ROM that processor execute . These instructions known as ISA.
- The **Instruction Set Architecture** (ISA) defines how a microprocessor is programmed at the machine level.
- **IS** depicts how to perform a specific task.
- Different architectures(Intel/AMD/core2Duo) have their own sets of instructions, syntax, data types, and addressing modes at the machine level.

Instruction Set Architecture

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software.

The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.

The ISA provides the only way through which a user is able to interact with the hardware.

It can be viewed as a programmer's manual because it's the portion of the machine that's visible to the assembly language programmer, the compiler writer, and the application programmer.

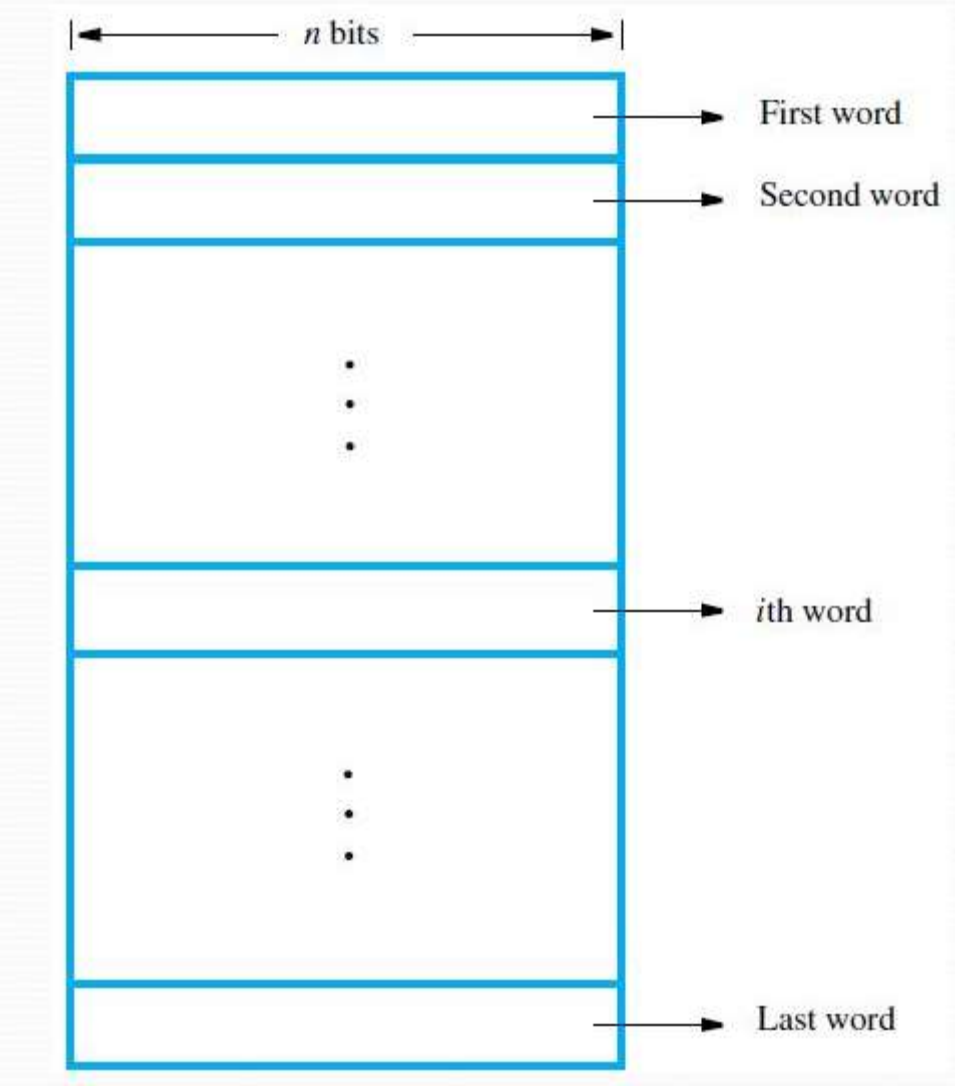
Memory Locations and Addresses

- The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually.
- The usual approach is to deal with them in groups of fixed size.

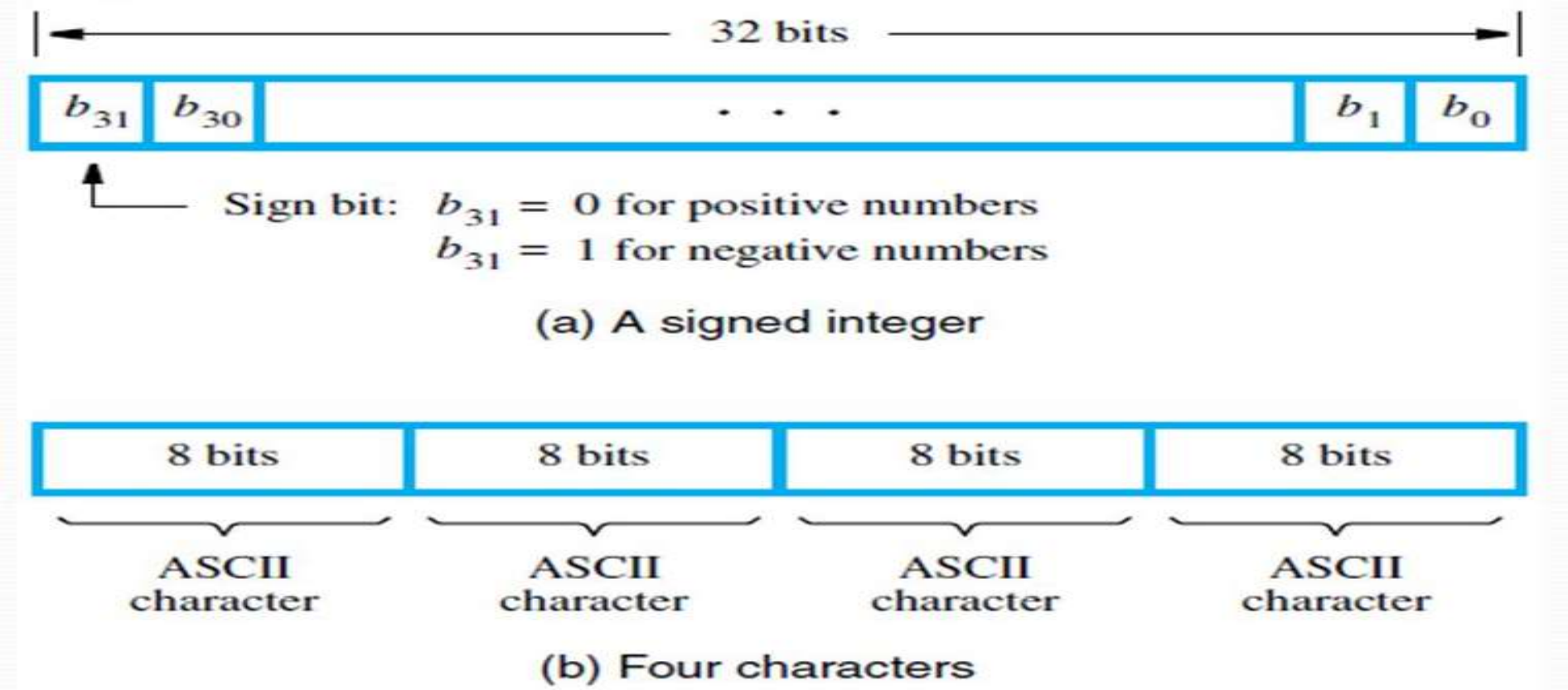
For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation.

Each group of n bits is referred to as a word of information, and n is called the word length.

The memory of a computer can be schematically represented as a collection of words, as shown in Figure .



- Modern computers have word lengths that typically range from 16 to 64 bits.
- If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, A unit of 8 bits is called a byte. as shown in Fig



- Machine instructions may require one or more words for their representation.
- 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 location.

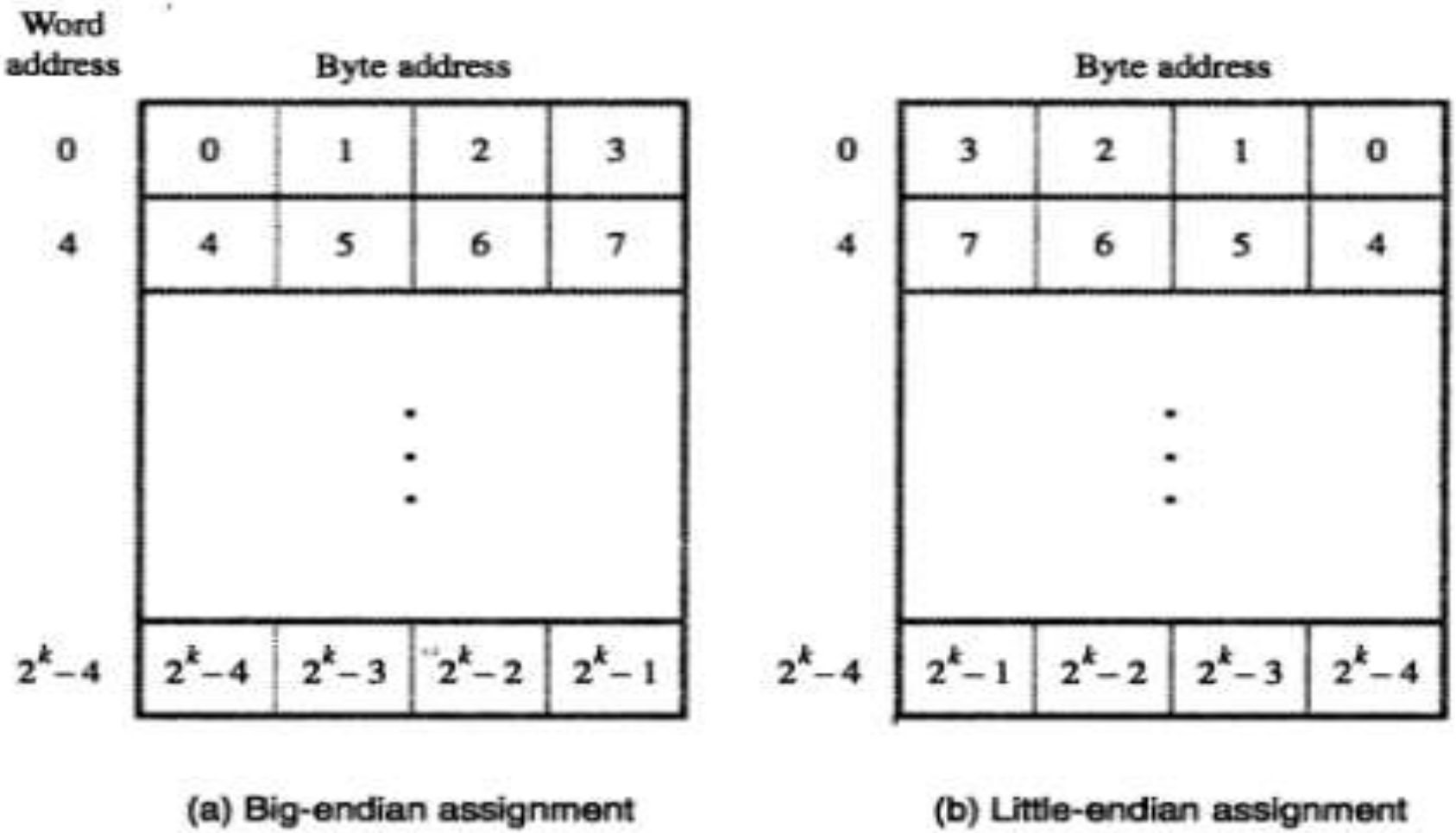
- It is customary to use numbers from 0 to 2^k − 1, for some suitable value of k, as the addresses of successive locations in the memory. Thus, the memory can have up to 2k addressable locations.
- The 2^k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 2^24 (16,777,216) locations.
- This number is usually written as 16M (16 mega), where 1M is the number 2^20 (1,048,576). A 32-bit address creates an address space of 2^32 or 4G (4 giga) locations, where 1G is 2^30.
- Other notational conventions that are commonly used are K (kilo) for the number 2^10 (1,024), and T (Tera) for the number 2^40.

Byte Addressability

- Byte addressing in hardware architectures supports accessing individual bytes.
- If each byte has a unique address, we have byte addressing.
- A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory.
- Byte locations have addresses 0, 1, 2,.... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8,..., with each word consisting of four bytes

Big-Endian and Little-Endian Assignments

- There are two ways that byte addresses can be assigned across words.
- The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word.
- The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.
- The words “more significant” and “less significant” are used in relation to the weights (powers of 2) assigned to bits when the word represents a number.
- Both little-endian and big-endian assignments are used in commercial machines.
- In both cases, byte addresses 0, 4, 8,..., are taken as the addresses of successive words in the memory of a computer with a 32-bit word length.
- These are the addresses used when accessing the memory to store or retrieve a word.



(a) Big-endian assignment

(b) Little-endian assignment

Word Alignment

- Words are said to be Aligned in memory if they begin at a byte-address that is a multiple of the number of bytes in a word.
- For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is

  If the word length is 16(2 bytes), aligned words begin at byte-addresses 0, 2, 4 . . . . .
  If the word length is 64(2^3bytes), aligned words begin at byte-addresses 0, 8, 16 . . . . .

ACCESSING NUMBERS, CHARACTERS

• A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte-address.

• There are two ways to indicate the length of the string:

1) A special control character with the meaning “end of string” can be used as the last character in the string.

2) A separate memory word location or register can contain a number indicating the length of the string in bytes.

Memory Operations

- Both program instructions and data operands are stored in the memory.
- To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor.
- Operands and results must also be moved between the memory and the processor.

Two basic operations involving the memory are needed, namely,

Read

Write

The Read operation transfers a copy of the contents of a specific memory location to the processor.

The memory contents remain unchanged.

To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read.

The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location.

To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location.

The memory then uses the address and data to perform the write

Instruction And Instruction sequencing

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen.

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

- We need to describe the transfer of information from one location in the computer to another.
- Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem.

- Most of the time, we identify a location by a symbolic name standing for its hardware binary address.
- For example. names for the address's of memory locations may be LOC, PLACE, A,VAR2:processor register names may be RO, R5; and I/O register names may he DATAIN, OUTSTATUS and so on.
- The contents of a location are denoted by placing square brackets around the name of the location.

Thus, the expression

R1 <--- [LOC]

means that the contents of memory location LOC are transfcrred into processor register RI.

As another example,

- consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3.
- This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

- This type of notation is known as Register Transfer Notation (RTN).

Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, over writing the old contents of that location.

In computer jargon, the words “transfer” and “move” are commonly used to mean “copy.” Transferring data from a source location A to a destination location B means that the contents of location A are read and then written into location B.

In this operation, only the contents of the destination will change. The contents of the source will stay the same

Assembly-Language Notation

We need another type of notation to represent machine instructions and programs. For this, we use assembly language.

For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

```
Load R2, LOC
```

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten.

The name Load is appropriate for this instruction, because the contents read from a memory location are loaded into a processor register.

The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

```
Add R4, R2, R3
```

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

Instruction Set: CISC And RISC

- The architectural designs of CPU are RISC (Reduced instruction set computing) and CISC (Complex instruction set computing). CISC has the ability to execute addressing modes or multi-step operations within one instruction set.

CISC

- The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.
- Computers based on the CISC architecture are designed to decrease the memory cost. Because, the large programs need more storage, thus increasing the memory cost and large memory becomes more expensive.
- To solve these problems, the number of instructions per program can be reduced by embedding the number of operations in a single instruction, thereby making the instructions more complex.

Examples of CISC PROCESSORS

- IBM 370/168 – It was introduced in the year 1970. CISC design is a 32 bit processor and four 64-bit floating point registers.
- VAX 11/780 – CISC design is a 32-bit processor and it supports many numbers of addressing modes and machine instructions which is from Digital Equipment Corporation.

- Intel 80486 – It was launched in the year 1989 and it is a CISC processor, which has instructions varying lengths from 1 to 11 and it will have 235 instructions

characteristics of CISC architecture

- Instruction-decoding logic will be Complex.
- One instruction is required to support multiple addressing modes.
- Less chip space is enough for general purpose registers for the instructions that are operated directly on memory

RISC Architecture

- RISC (Reduced Instruction Set Computer) is used in portable devices due to its power efficiency. For Example, Apple iPod and Nintendo DS.
- RISC is a type of microprocessor architecture that uses highly-optimized set of instructions. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program Pipelining is one of the unique feature of RISC.
- It is performed by overlapping the execution of several instructions in a pipeline fashion. It has a high performance advantage over CISC.

RISC architecture characteristics

- Simple Instructions are used in RISC architecture.
- RISC helps and supports few simple data types and synthesize complex data types.
- RISC permits any register to use in any context.
- One Cycle Execution Time

Introduction to RISC Instruction Sets

Two key characteristics of RISC instruction sets are:

• Each instruction fits in a single word.

• A load/store architecture is used, in which

– Memory operands are accessed only using Load and Store instructions.

– All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word

- At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer.
- This task is done by Load instructions which copy the contents of a memory location into a processor register.
- Load instructions are of the form

  Load destination, source
- or more specifically

  Load processor_register, memory_location
- Let us now consider a typical arithmetic operation.
- The operation of adding two numbers is a fundamental capability in any computer.

  The statement C = A + B
- The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action to take place in the computer.

  C ← [A]+[B]

- To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed.
- This result is then sent back to the memory and stored in location C.
- The required action can be accomplished by a sequence of simple machine instructions.
- We choose to use registers R2, R3, and R4 to perform the task with four instructions:

Load R2, A

Load R3, B

Add R4, R2, R3

Store R4, C

Add is a three-operand, or a three-address, instruction of the form

Add destination, source1, source2

The Store instruction is of the form

Store source, destination

- where the source is a processor register and the destination is a memory location.
- Observe that in the Store instruction the source and destination are specified in the reverse order from the Load instruction; this is a commonly used convention.
- Note that we can accomplish the desired addition by using only two registers, R2 and R3, if one of the source registers is also used as the destination for the result. In this case the addition would be performed as

```
Add R3, R2, R3
```

and the last instruction would become

```
Store R3, C
```

Instruction Execution and Straight-Line Sequencing

We used the task C = A + B, implemented as C ← [A] + [B], as an example. Figure

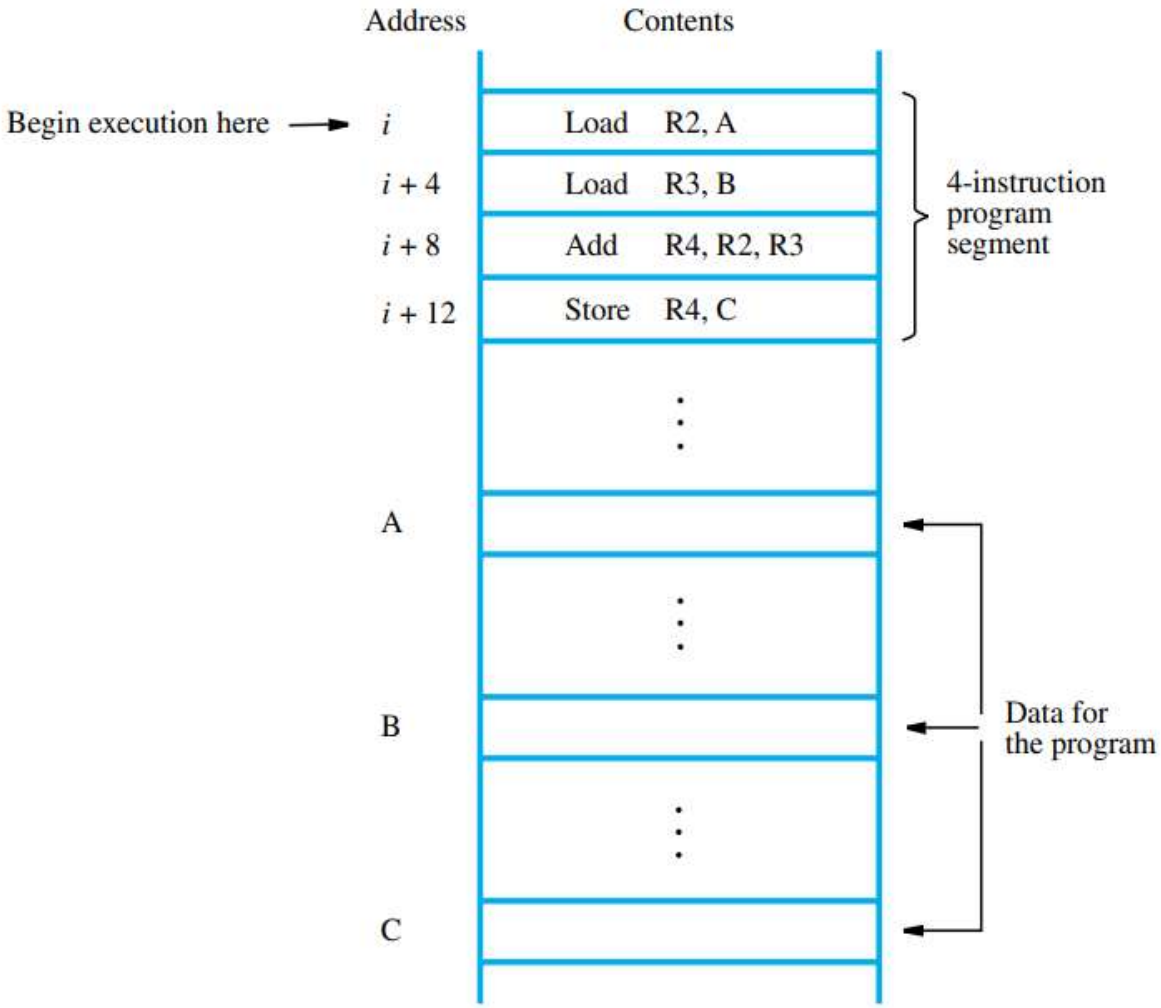


Figure 2.4 A program for C ← [A] + [B].

- Figure shows a possible program segment for this task as it appears in the memory of a computer.
- We assume that the word length is 32 bits and the memory is byte-addressable.
- The four instructions of the program are in successive word locations, starting at location i.
- Since each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses i + 4, i + 8, and i + 12.

Let us consider how this program is executed.

- The processor contains a register called the program counter (PC), which holds the address of the next instruction to be executed.
- To begin executing a program, the address of its first instruction (i in our example) must be placed into the PC.
- Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing.
- During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location i + 12 is executed, the PC contains the value i + 16, which is the address of the first instruction of the next program segment.

Branching

Consider the task of adding a list of n numbers. The program outlined in Figure 2.5 is a generalization of the program in above Figure 2.4.

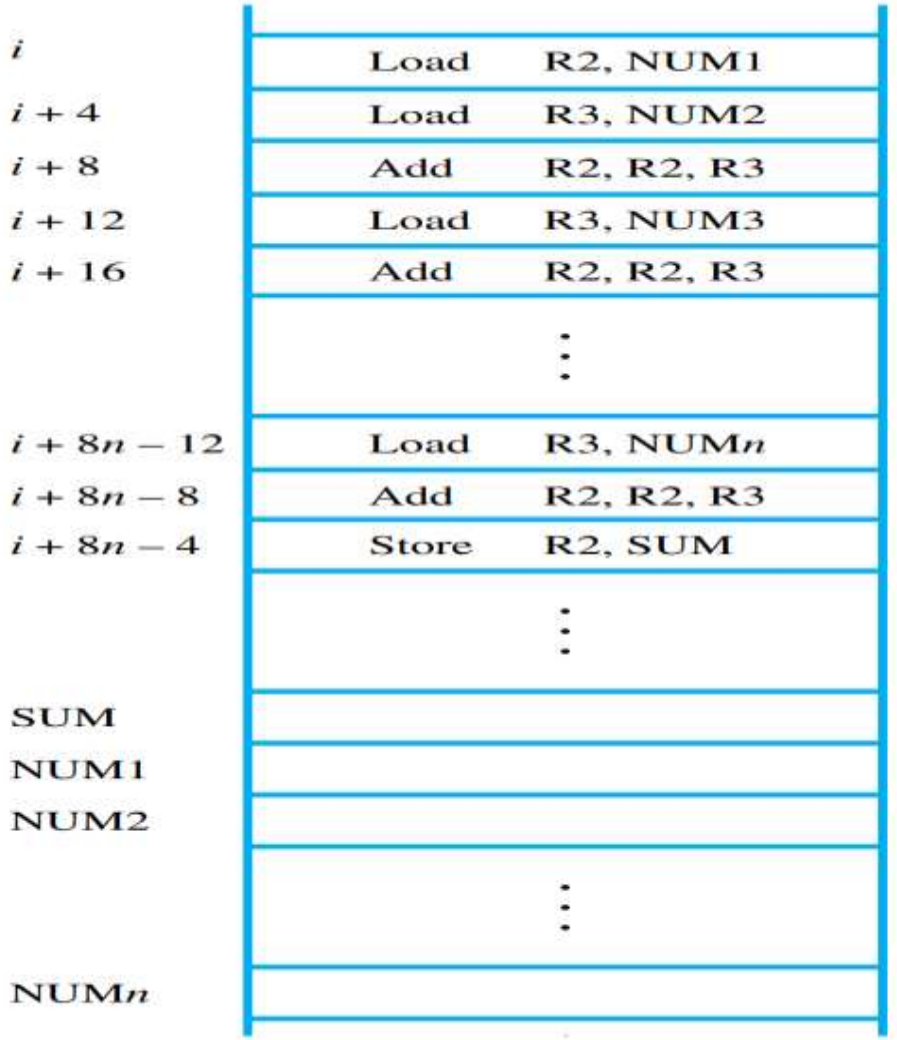


Figure 2.5 A program for adding *n* numbers.

- The addresses of the memory locations containing the n numbers are symbolically given as NUM1, NUM2,..., NUMn, and separate Load and Add instructions are used to add each number to the contents of register R2.
- After all the numbers have been added, the result is placed in memory location SUM.
- Instead of using a long list of Load and Add instructions, as in Figure 2.5, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum.
- To add all numbers, the loop has to be executed as many times as there are numbers in the list
- Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R2 is used as a counter to determine the number of times the loop is executed.
- Hence, the contents of location N are loaded into register R2 at the beginning of the program. Then, within the body of the loop, the instruction reduces the contents of R2 by 1 each time through the loop. Execution of the loop is repeated as long as the contents of R2 are greater than zero.

Subtract R2, R2, #1

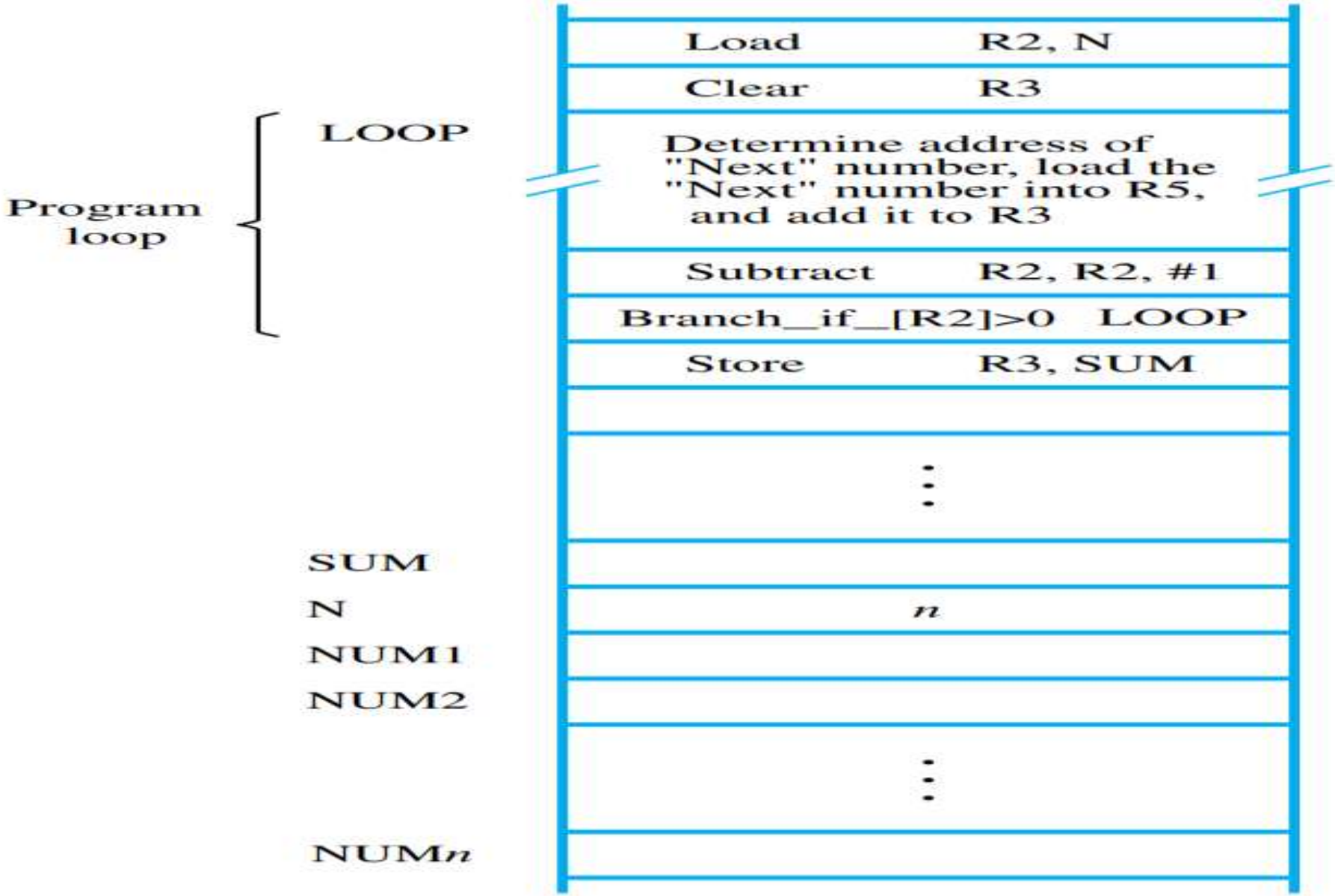


Figure 2.6 Using a loop to add *n* numbers.

- Branch instructions:-This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order.
- A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.
- In the program, the instruction Branch_if_[R2]>0 LOOP is a conditional branch instruction that causes a branch to location LOOP if the contents of register R2 are greater than zero.
- One way of implementing conditional branch instructions is to compare the contents of two registers and then branch to the target instruction if the comparison meets the specified requirement.
- For example, the instruction that implements the action Branch_if_[R4]>[R5] LOOP
- May be written in generic assembly language as Branch_greater_than R4, R5, LOOP
- or using an actual mnemonic as BGT R4, R5, LOOP

Addressing Modes

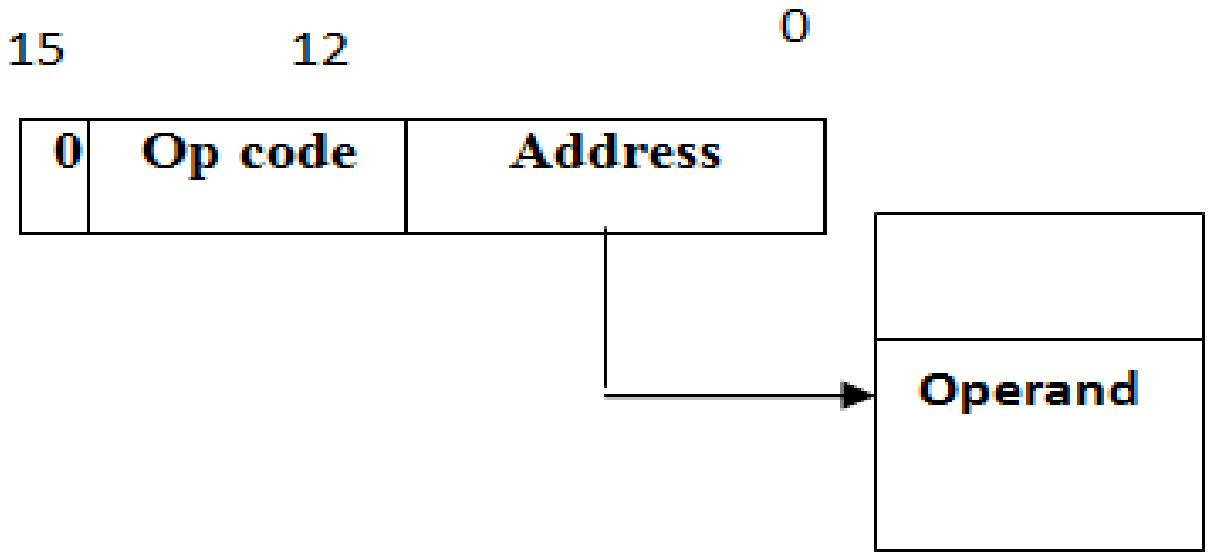
- **The operation field of an instruction specifies the operation to be performed.**
- **This operation must be executed on some data stored in computer registers.**
- **The different ways of specifying the location of an operand in an instruction are called as addressing modes.**

- **The following arte different types of addressing modes**

1. **Direct Addressing Mode**
2. **In-Direct Addressing Mode**
3. **Relative Addressing Mode**
4. **Index Addressing Mode**
5. **Register Addressing Mode**
6. **Register-Index Addressing Mode**
7. **Auto-Increment Addressing Mode**
8. **Auto-Decrement Addressing Mode**

Direct Addressing Mode

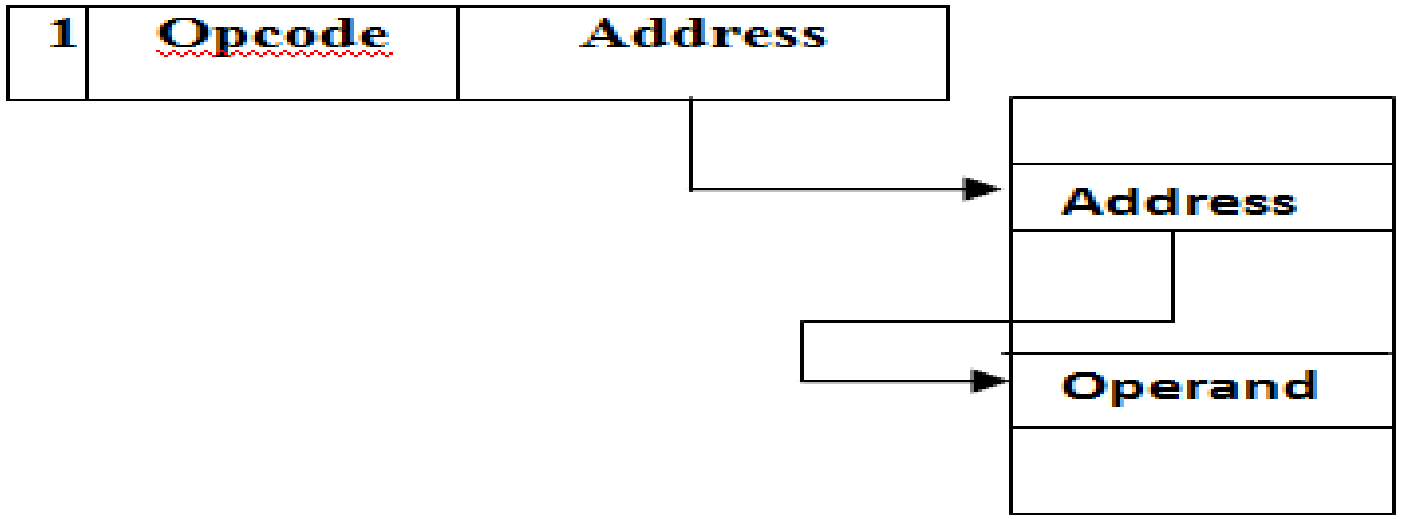
- Direct Addressing: in the instruction code mode(I) bit either (0 or 1) where I=0 for then perform direct addressing mode



- It uses direct address of operands. I.e. address part of the instruction specifies a memory address where the data is stored.

In-Direct Addressing Mode

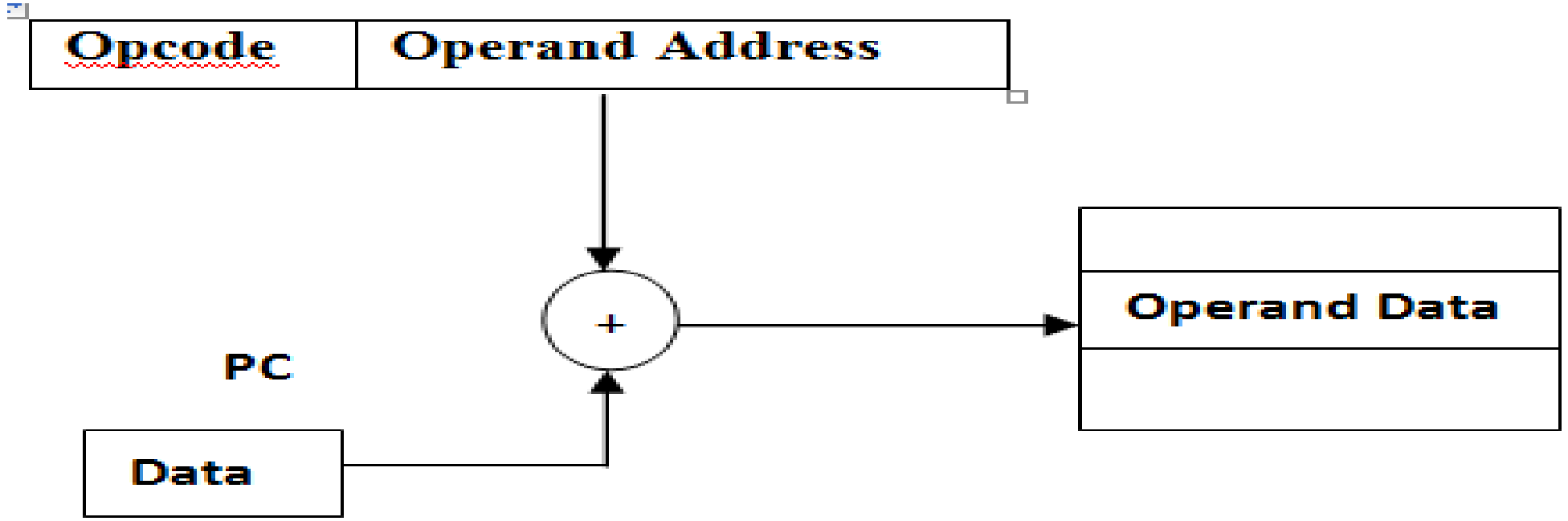
- In-Direct Addressing: in the instruction code mode(I) bit either (0 or 1) where I=1 for then perform direct addressing mode



- It uses indirect address of operands. I.e. address part of the instruction specifies another memory location where the data is stored.

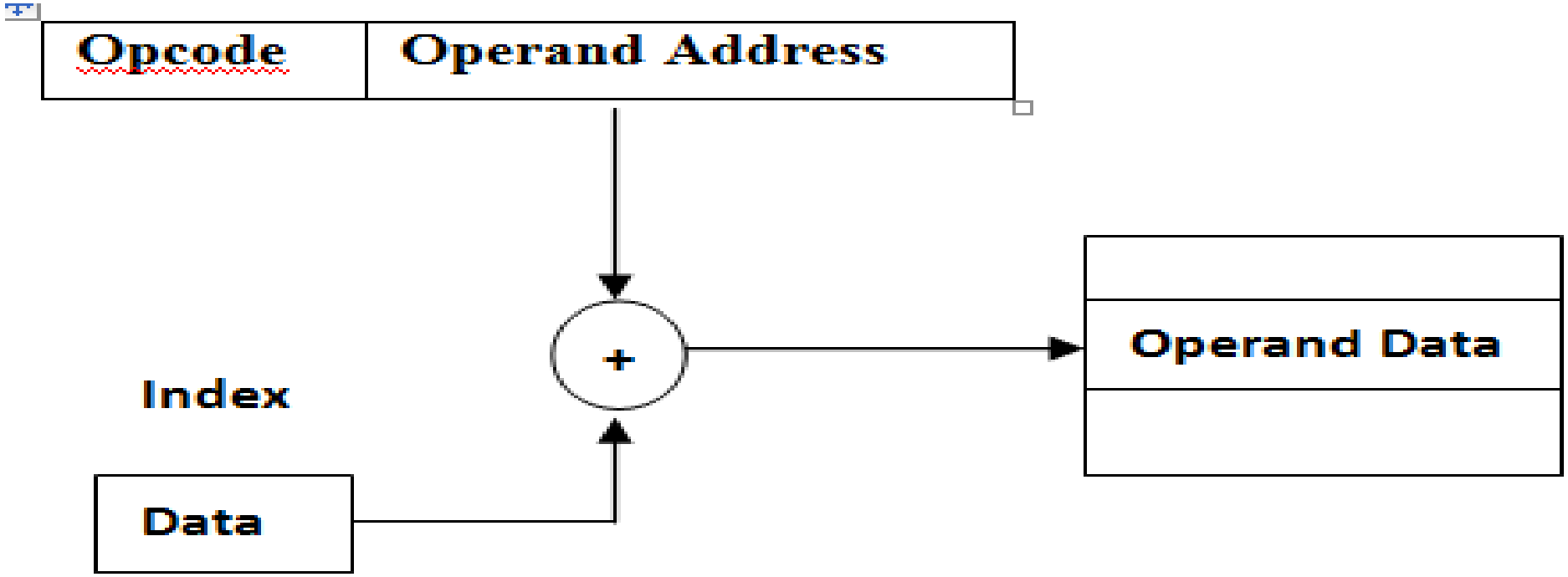
Relative Addressing Mode

- In this addressing mode, the instruction contains a relative address which is to be added to the contents of program counter to get the effective address.
- Effective Address = (PC)+ Address specified in the instruction



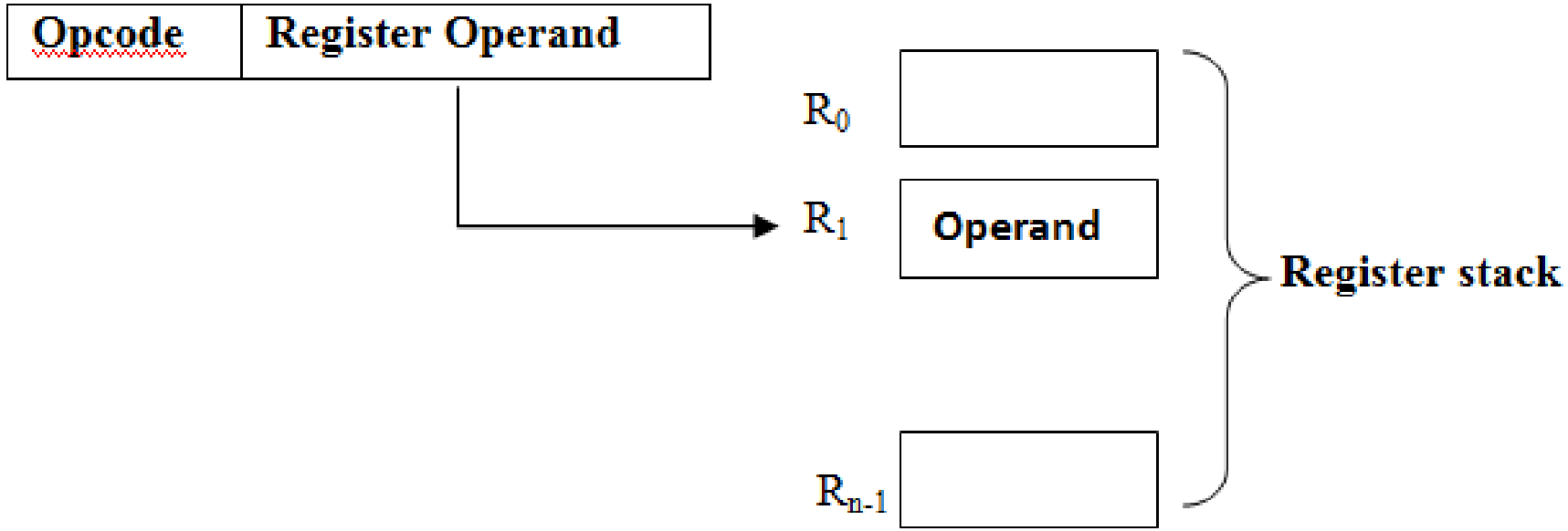
Indexed Addressing Mode

- In this addressing mode the content of index register is added to the address of the instruction
- Effective Address = index register + Address Specified in the instruction



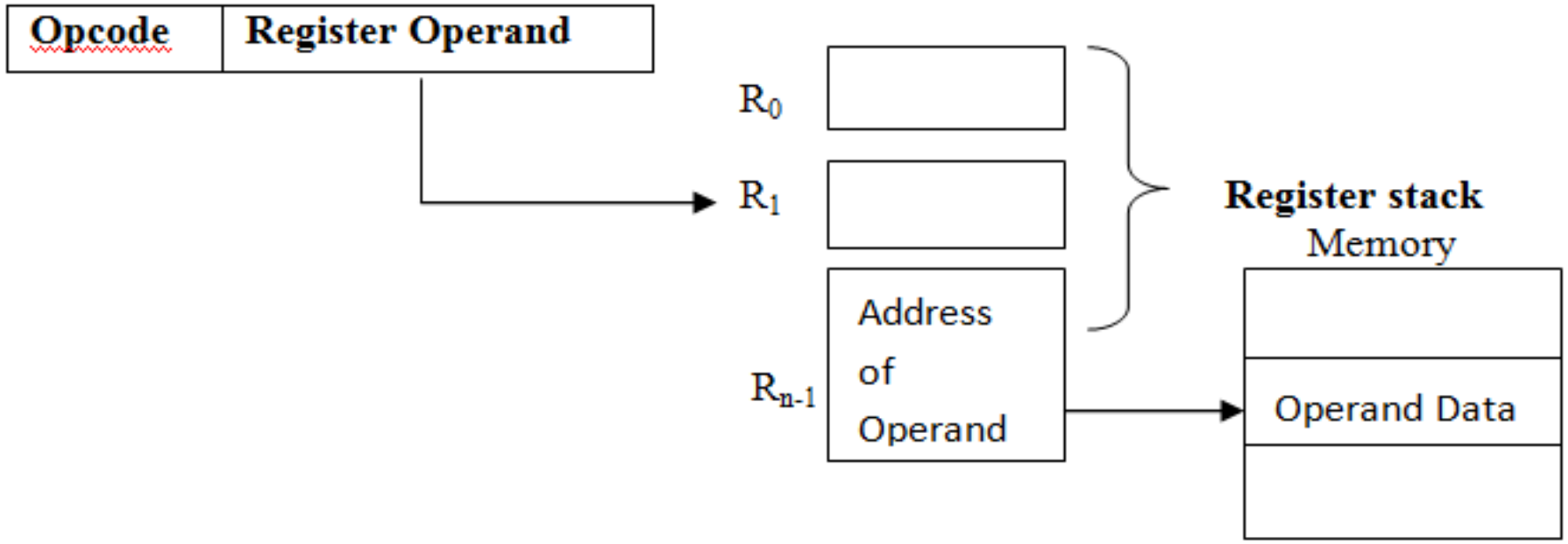
Register Addressing Mode

- In this addressing mode content of register is added to AC Register no need effective address



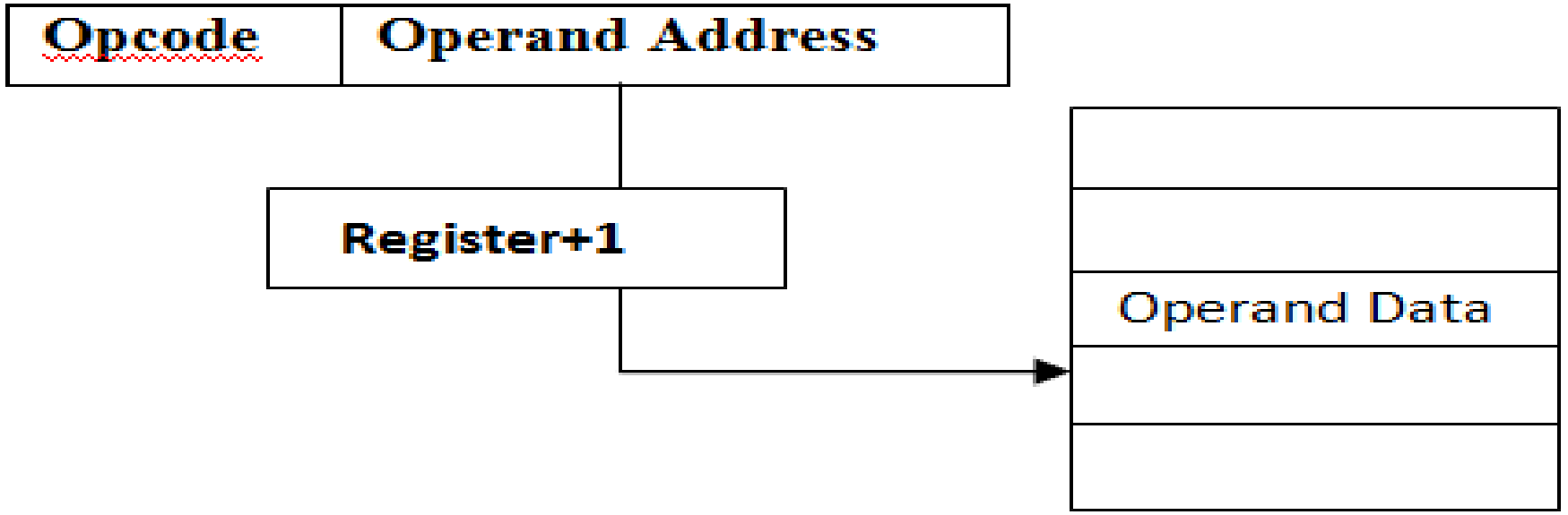
Register-Index Addressing Mode

- In this addressing mode the content of the register gives the address part of the operand in the memory
- Effective Address = content of the register



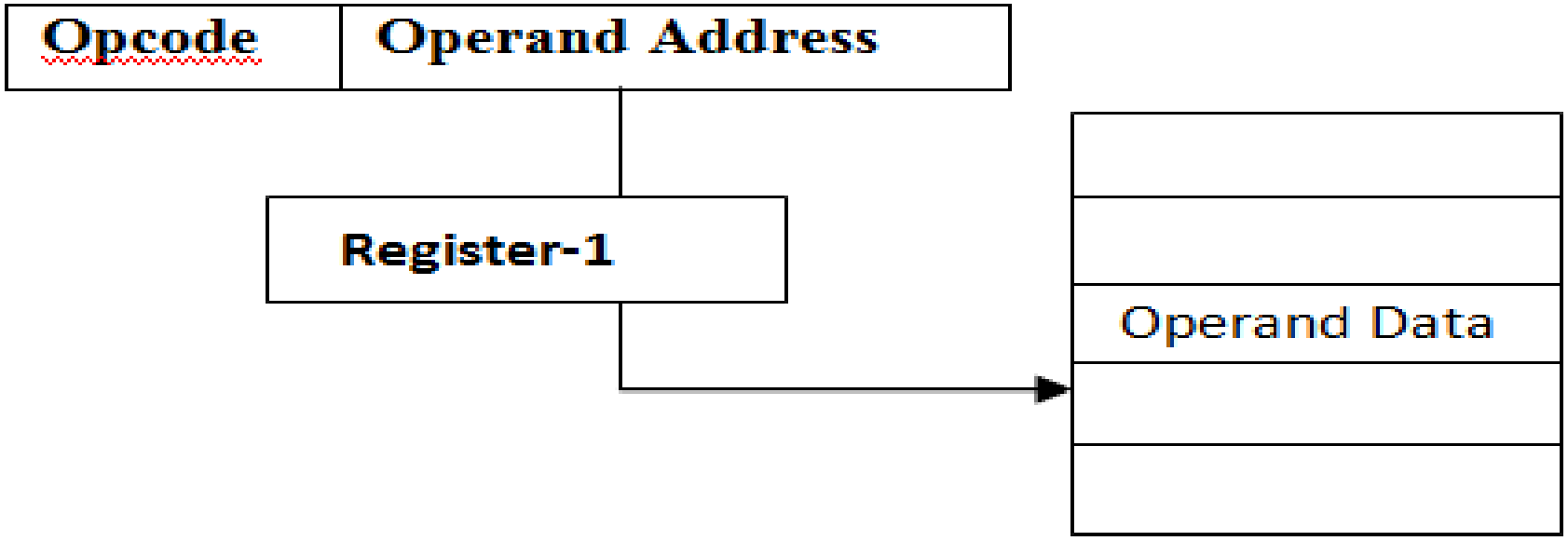
Auto-Increment Addressing Mode

- In this addressing mode the content of the register is incremented by 1
- Effective address= content of the register+1



Auto-Decrement Addressing Mode

- In this addressing mode the content of the register is Decrement by 1
- Effective address= content of the register-1



ADDRESS	LOAD TO AC
ADDRESS=500	
399	450
400	700
401	600
. . . .	. . . .
500	800
600	900
700	325
800	300

PC=200

R=400

XR=100

AC register

Address register	Effective address	Content of AC Register
Direct address Mode	500	800
In-Direct address Mode	800	300
Relative Addressing Mode	200+500=700	325
Index addressing Mode	100+500=600	900
Register Addressing Mode	-	400
Register Index Addressing Mode	400	700
Auto Increment Addressing Mode	400+1=401	600
Auto Decrement Addressing Mode	400-1=399	450

Addressing Modes

- The addressing mode is the method to specify the operand of an instruction. The job of a microprocessor is to execute a set of instructions stored in memory to perform a specific task. Operations require the following:
- The operator or opcode which determines what will be done
- The operands which define the data to be used in the operation

In general a program operates on data that reside in the computer's memory.

The different ways for specifying the locations of instruction operands are known as 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

Relative	X(PC)	EA = [PC] + X
Auto increment	(Ri)+	EA = [Ri]; Increment Ri
Auto decrement	-(Ri)	Decrement Ri; EA = [Ri]

Immediate Addressing mode

In this mode, the operand is specified in the instruction itself. An immediate mode instruction has an operand field rather than the address field.

- Operand is part of instruction , Operand = operand field

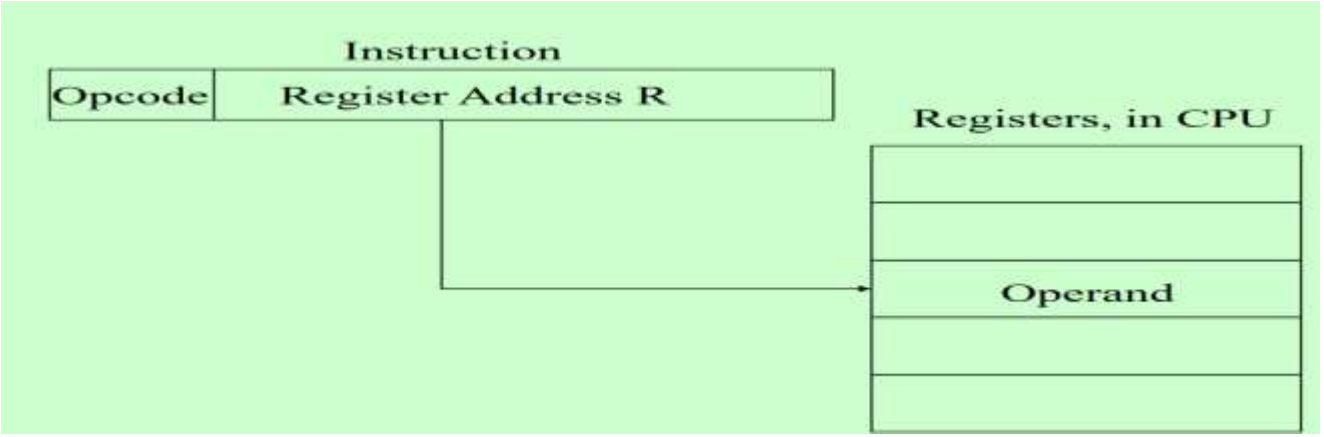
e.g. ADD 5

Add 5 to contents of accumulator , 5 is the operand

- No memory reference to access data and Fast.
- Range of operands limited to # of bits in operand field (< word size)

Register Mode

- In this mode the operand is stored in the register and this register is present in CPU. The instruction has the address of the Register where the operand is stored.



Operand is held in register named in address filed :- EA = R

ADD R will increment the value stored in the accumulator by the content of register R.

- AC ← AC + [R]

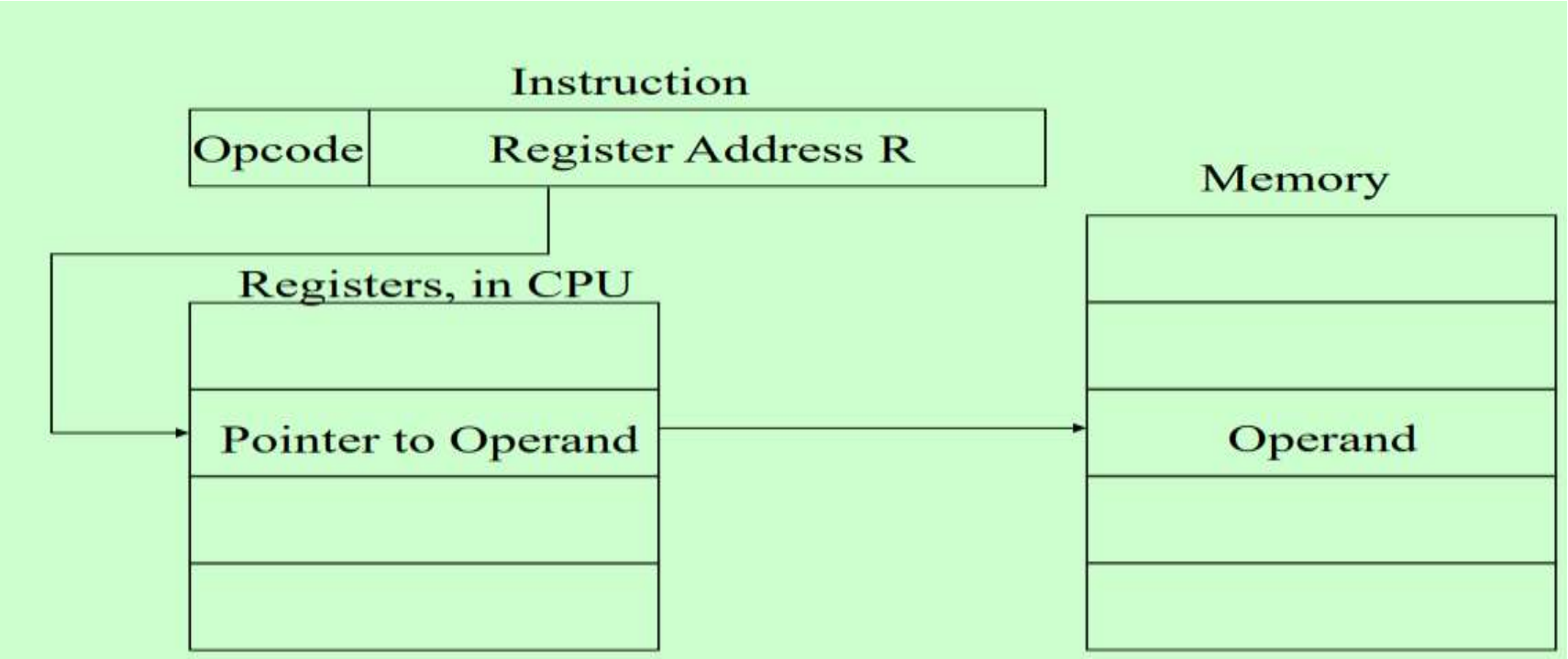
Advantages:

- Very small address field -Shorter instructions and Faster instruction fetch.
- Faster memory access to operand(s)

Disadvantage: Very limited address space

Register Indirect Addressing mode

In this mode, the instruction specifies the register whose contents give us the address of operand which is in memory. Thus, the register contains the address of operand rather than the operand itself.



Operand is in memory cell pointed to by contents of register R

- ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

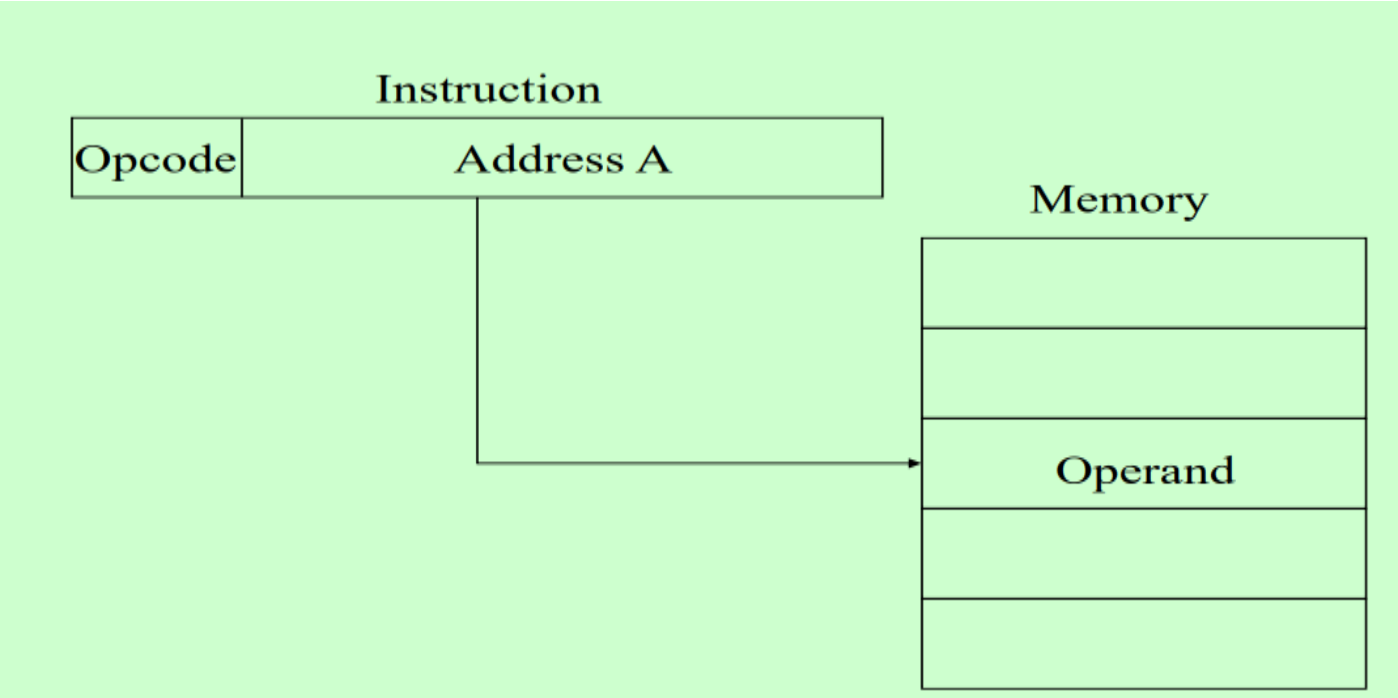
$$AC \leftarrow AC + [[R]]$$

Absolute (Direct) Addressing Mode

In this mode, effective address of operand is present in instruction itself.

Single memory reference to access data.

No additional calculations to find the effective address of the operand.



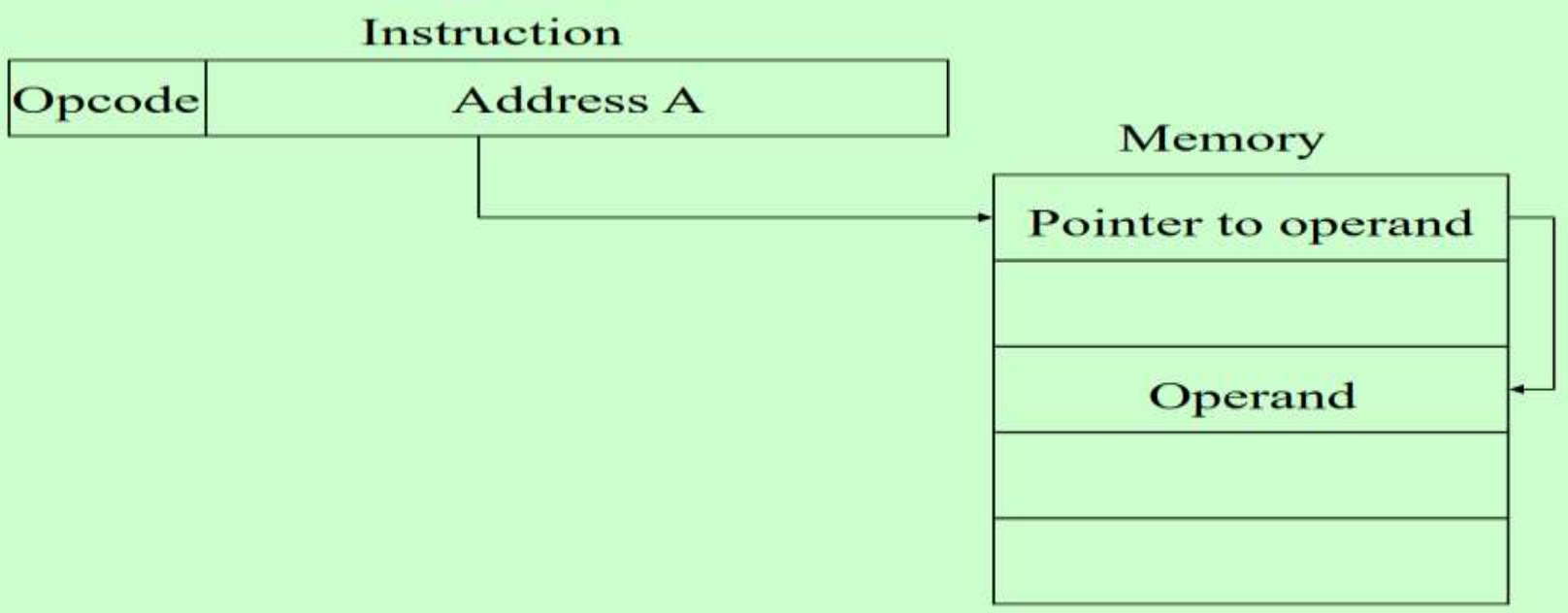
Effective address (EA) = address field (A)

ADD X will increment the value stored in the accumulator by the value stored at memory location X.

$$AC \leftarrow AC + [X]$$

Indirect Addressing Mode

- In this, the address field of instruction gives the address where the effective address is stored in memory. This slows down the execution, as this includes multiple memory lookups to find the operand.



- EA = (A)

ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.

$$AC \leftarrow AC + [[X]]$$

Index Addressing Mode

- In the indexed addressing mode, the content of a given index register gets added to an instruction's address part so as to obtain the effective address. Here, the index register refers to a special CPU register that consists of an index value. An instruction's address field defines the beginning address of any data array present in memory.
- The index addressing mode is helpful whenever the instructions in a program access an array or large ranges of memory addresses. The effective address, in such a mode, is generated when we add a constant to the content of the register
- Here is a symbolic representation of the index addressing mode: X(R)
- The effective address here is denoted as : EA = X + (R)

Consider the instruction given below, for instance:

Load R2, A

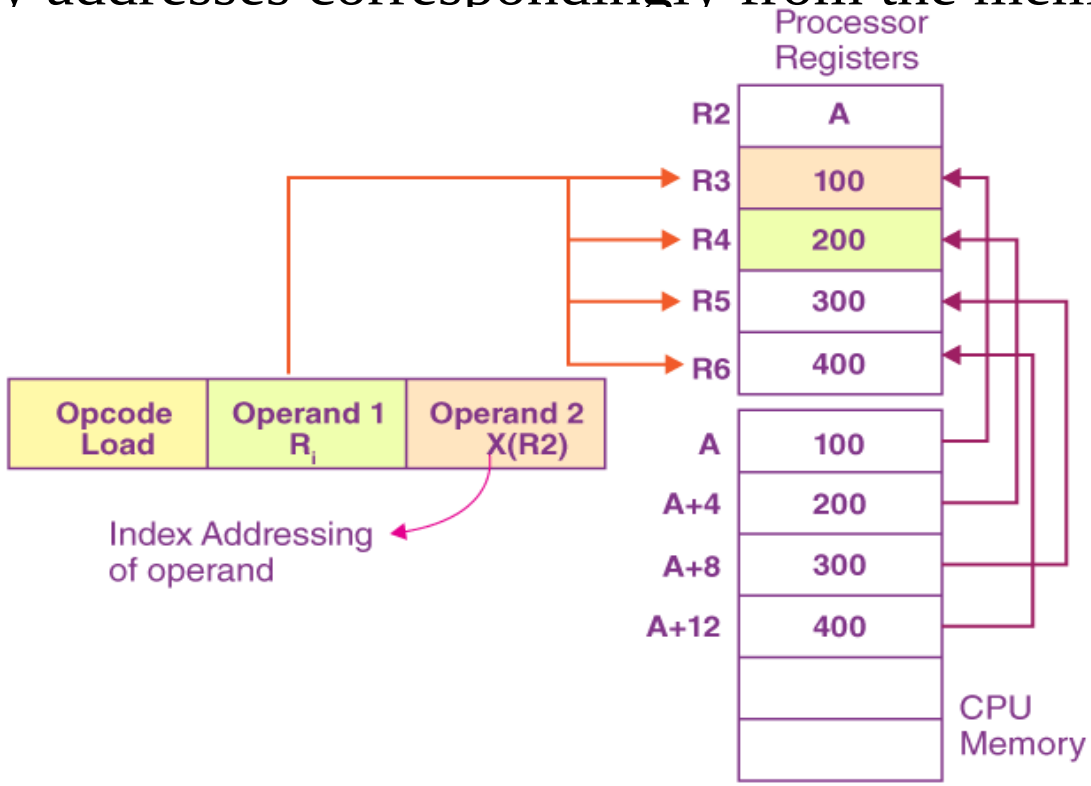
Load R3, (R2)

Load R4, 4(R2)

Load R5, 8(R2)

Load R6, 12(R2)

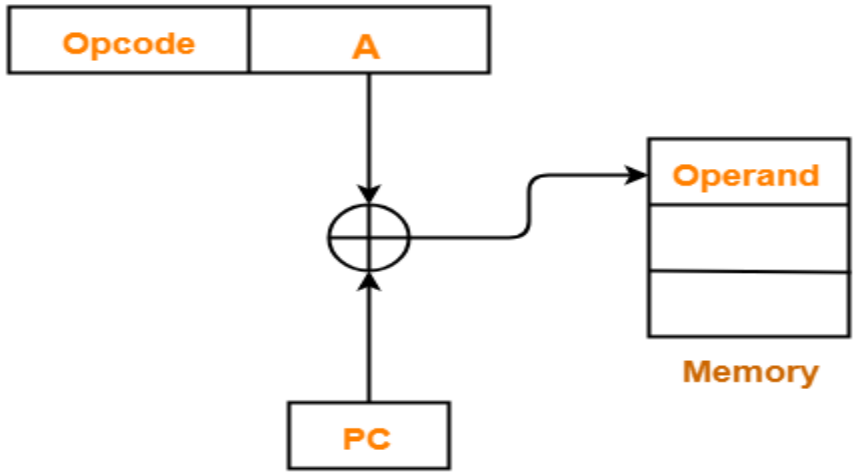
The instructions given above will load the R3, R4, R5, R6 registers, along with the contents that are present at the successive memory addresses correspondingly from the memory location A.



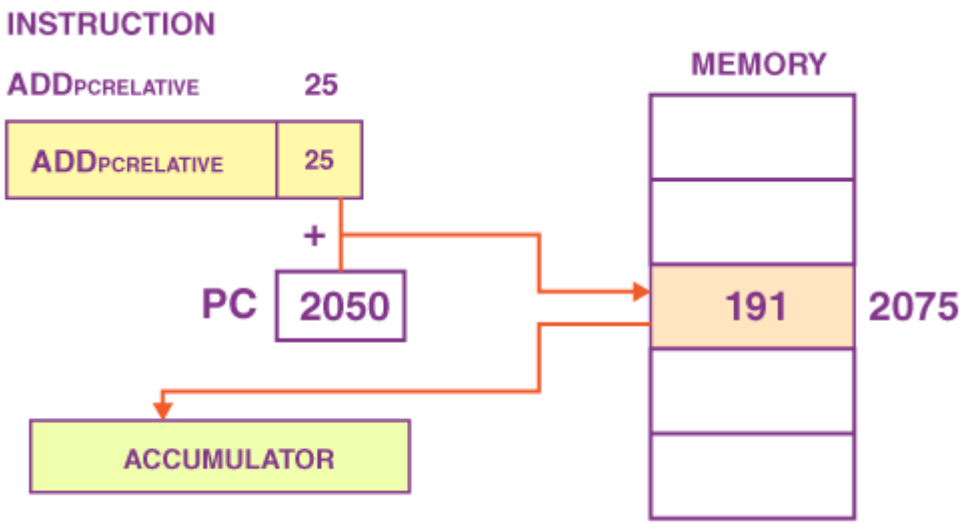
Relative Addressing Mode

- In this addressing mode, Effective address of the operand is obtained by adding the content of program counter with the address part of the instruction.
- Effective Address= Content of Program Counter + Address part of the instruction
- Here is a symbolic representation of the relative address mode:X(PC)

- Here is how the effective address for it:EA = X + (PC)
- Since the operand addresses here are found relative to a program counter, it is known as the relative address mode.



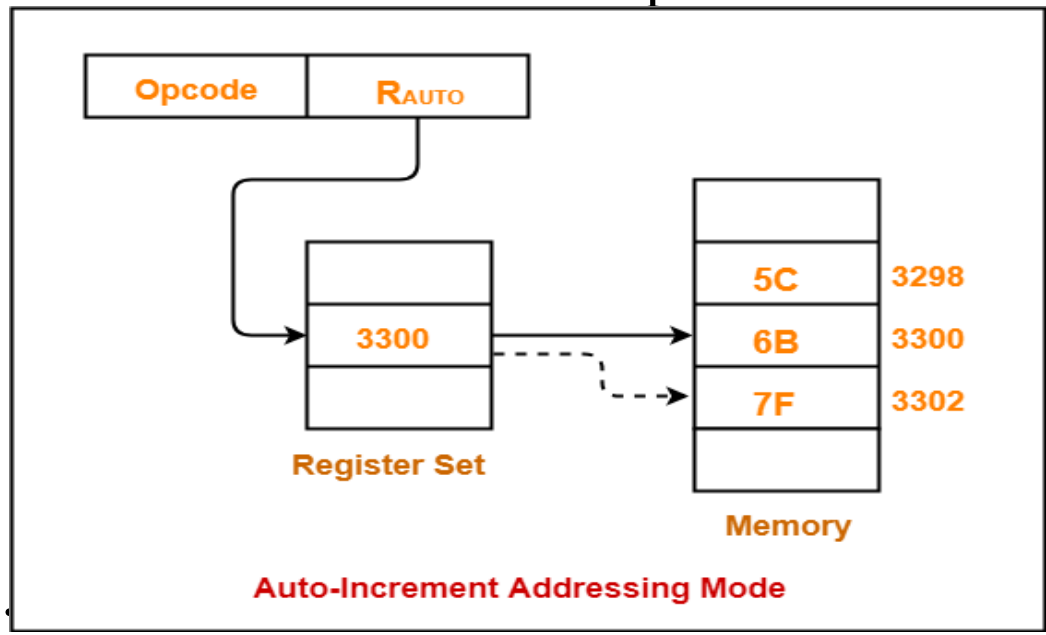
Relative Addressing Mode



- Program counter (PC) always contains the address of the next instruction to be executed.
- After fetching the address of the instruction, the value of program counter immediately increases.
- The value increases irrespective of whether the fetched instruction has completely executed or not.

Auto Increment Addressing Mode

- In the case of auto increment mode, the content present in the register is initially incremented, and then the content that is incremented in the register is used in the form of an effective address.
- Once the content present in the register in the auto-increment addressing mode is accessed by its instruction, the content of the register is incremented to refer to the next operand.
- Symbolically, we can represent it as follows:(R)+
- Assume operand size = 2 bytes.



Here,

- After fetching the operand 6B, the instruction

register R_{AUTO} will be automatically incremented by 2.

- Then, updated value of R_{AUTO} will be 3300 + 2 = 3302.
- At memory address 3302, the next operand will be found.

Auto Decrement Addressing Mode

- The auto decrement addressing mode is just the opposite of the auto increment mode. In the case of auto decrement mode, the content present in the register is initially decremented, and then the content that is decremented in the register is used in the form of an effective address.
- It is presented symbolically as :-(R)
- The auto decrement mode helps us in implementing the structure of the stack.

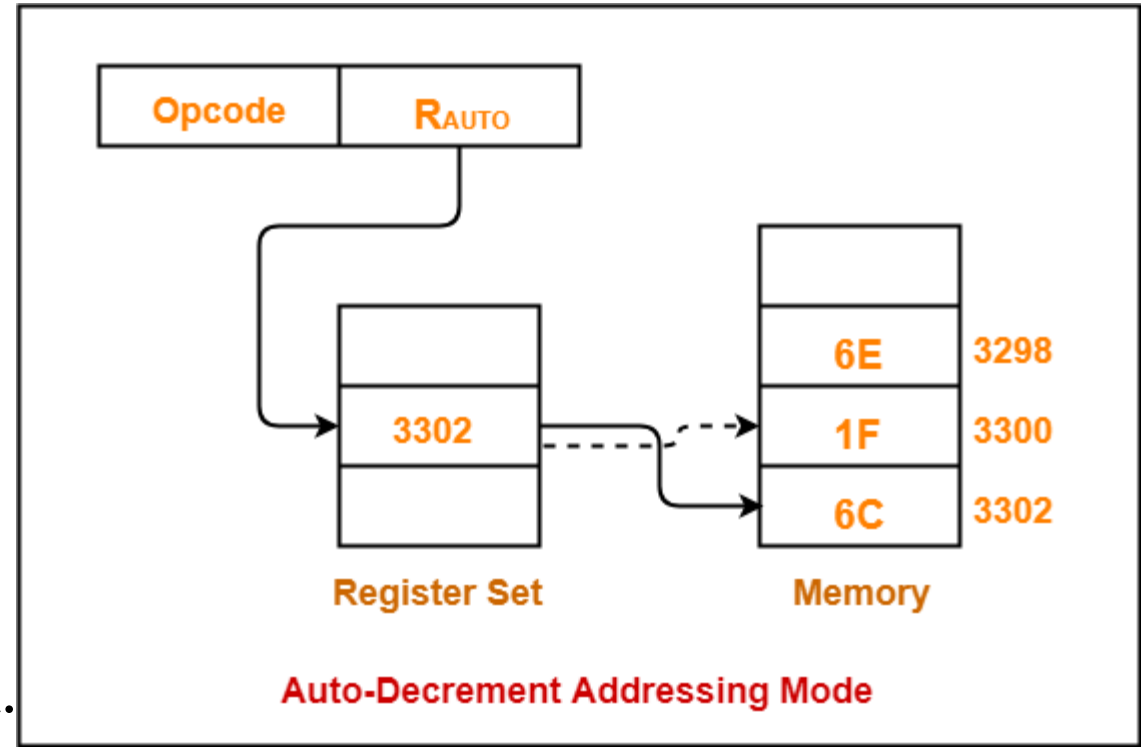
Assume operand size = 2 bytes.

Here,

- First, the instruction register R_{AUTO} will be decremented by 2.
- Then, updated value of R_{AUTO} will be

$$3302 - 2 = 3300.$$

- At memory address 3300, the operand will be found.



Indirection and Pointers

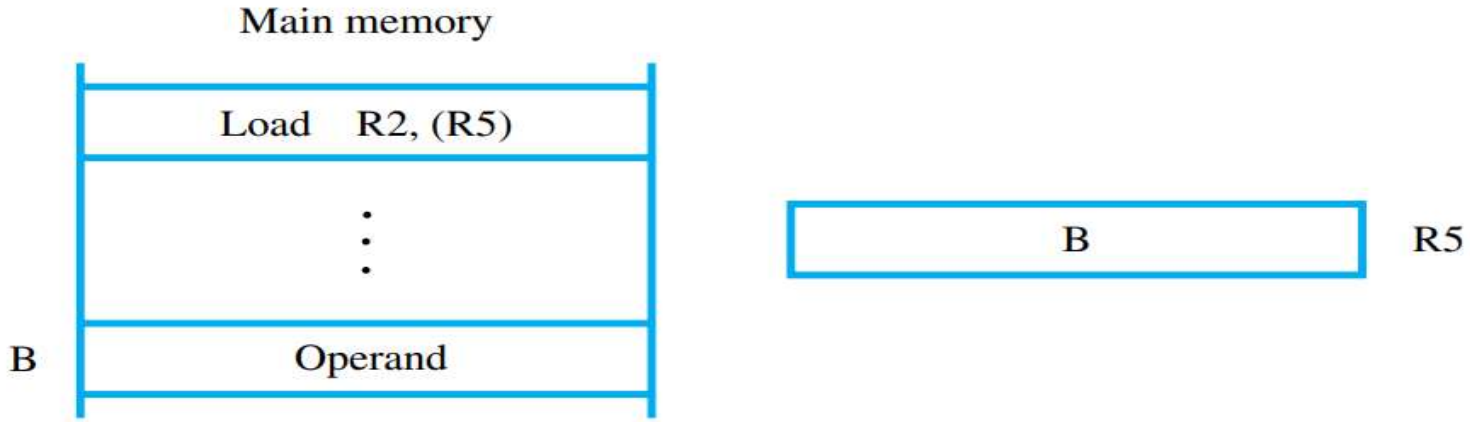


Figure 2.7 Register indirect addressing.

To execute the Load instruction in Figure 2.7, the processor uses the value B, which is in register R5, as the effective address of the operand. It requests a Read operation to fetch the contents of location B in the memory.

The value from the memory is the desired operand, which the processor loads into register R2. Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.

For example, register R5 in Figure 2.7 serves as a pointer for the Load instruction to load an operand from the memory into register R2. At one time, R5 may point to location B in memory.

Later, the program may change the contents of R5 to point to a different location, in which case the same Load instruction will load the value from that location into R2. Thus, a program segment that includes this Load instruction is conveniently reused with only a change in the pointer value.

```
        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.

Figure 2.8. Register R4 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R4.

The initialization section of the program loads the counter value n from memory location N into R2. Then, it uses the Clear instruction to clear R3 to 0. The next instruction uses the Immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R4.

Observe that we cannot use the Load instruction to load the desired immediate value, because the Load instruction can operate only on memory source operands. Instead, we use the Move instruction

Move R4, #NUM1

In many RISC-type processors, one general-purpose register is dedicated to holding a constant value zero. Usually, this is register R0. Its contents cannot be changed by a program instruction. We will assume that R0 is used in this manner in our discussion of RISC-style processors. Then, the above Move instruction can be implemented as Add R4, R0, #NUM1

As another example of pointers,

consider the C-language statement A = *B;

where B is a pointer variable and the ‘*’ symbol is the operator for indirect accesses.

This statement causes the contents of the memory location pointed to by B to be loaded into memory location A.

The statement may be compiled into

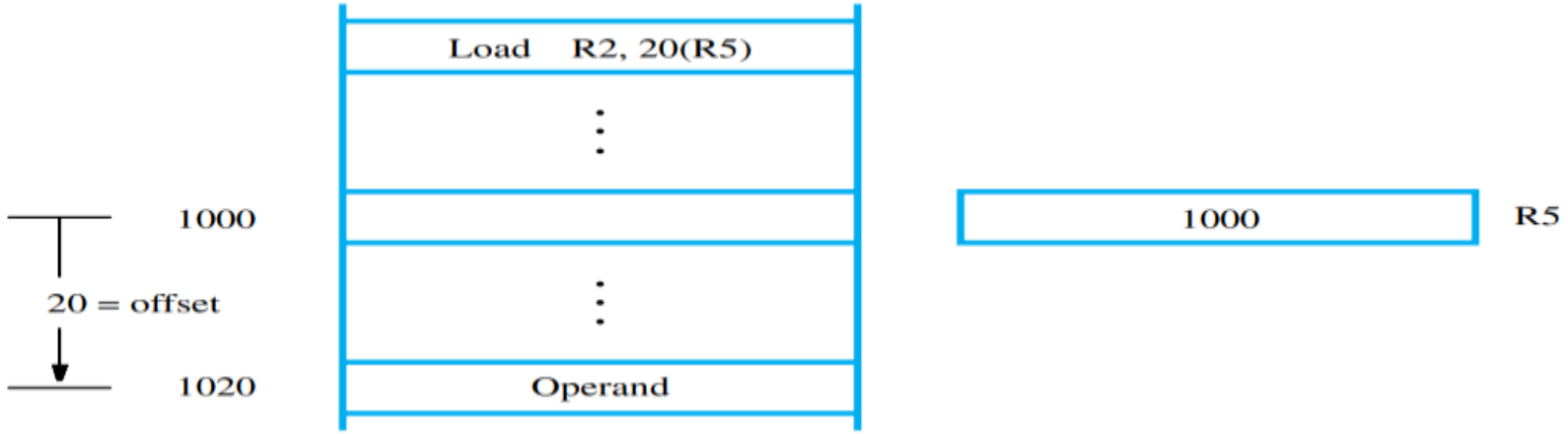
Load R2, B

Load R3, (R2)

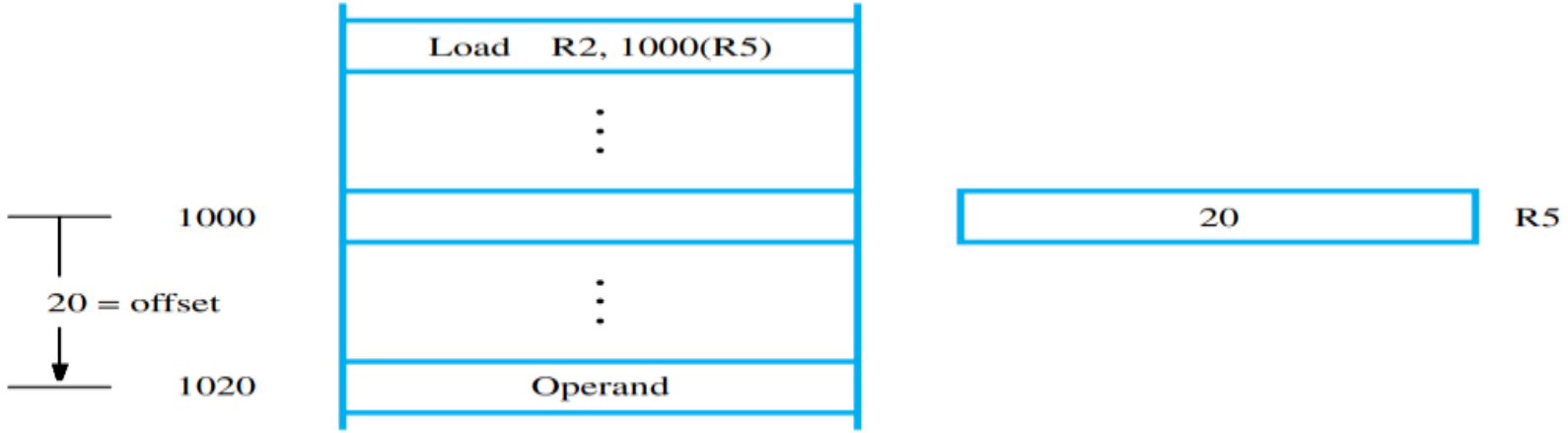
Store R3, A

Indexing and Arrays

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



(a) Offset is given as a constant



(b) Offset is in the index register

STACK

- Stack is a storage structure that stores information in such a way that the last item stored is the first item retrieved. It is based on the principle of LIFO (Last-in-first-out). The stack in digital computers is a group of memory locations with a register that holds the address of top of element. This register that holds the address of top of element of the stack is called Stack Pointer.

Stack Operations

- The two operations of a stack are: Push: Inserts an item on top of stack.
  Pop: Deletes an item from top of stack.
- Implementation of Stack

In digital computers, stack can be implemented in two ways : Register Stack and Memory Stack.

- Register Stack

A stack can be organized as a collection of finite number of registers that are used to store temporary information during the execution of a program. The stack pointer (SP) is a register that holds the address of top of element of the stack.

- Memory Stack

A stack can be implemented in a random access memory (RAM) attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. The starting memory location of the stack is specified by the processor register as stack pointer.

Figure 2.14 shows an example of a stack of word data items

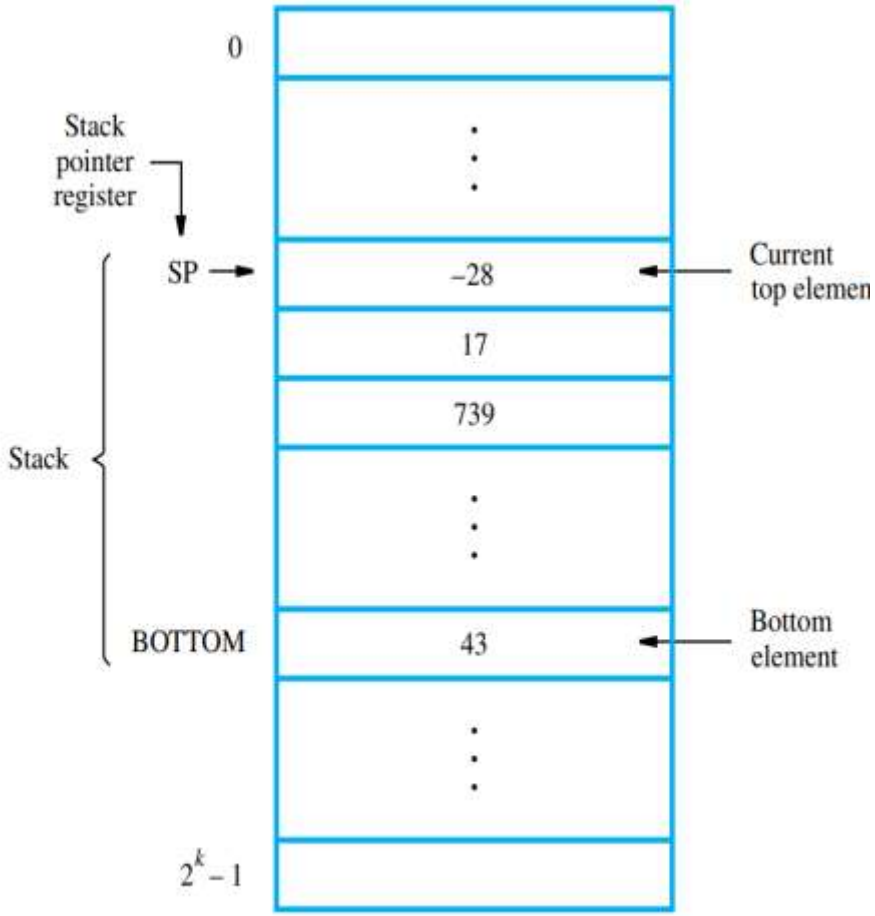


Figure 2.14 A stack of words in the memory.

The stack contains numerical values, with 43 at the bottom and −28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time.

If we assume a byte-addressable memory with a 32-bit word length,

The push operation can be implemented as

```
Subtract SP, SP, #4
Store Rj, (SP)
```

The pop operation can be implemented as

```
Load Rj, (SP)
Add SP, SP, #4
```

These two instructions load (pop) the top value from the stack into register Rj and then increment the stack pointer by 4 so that it points to the new top element.

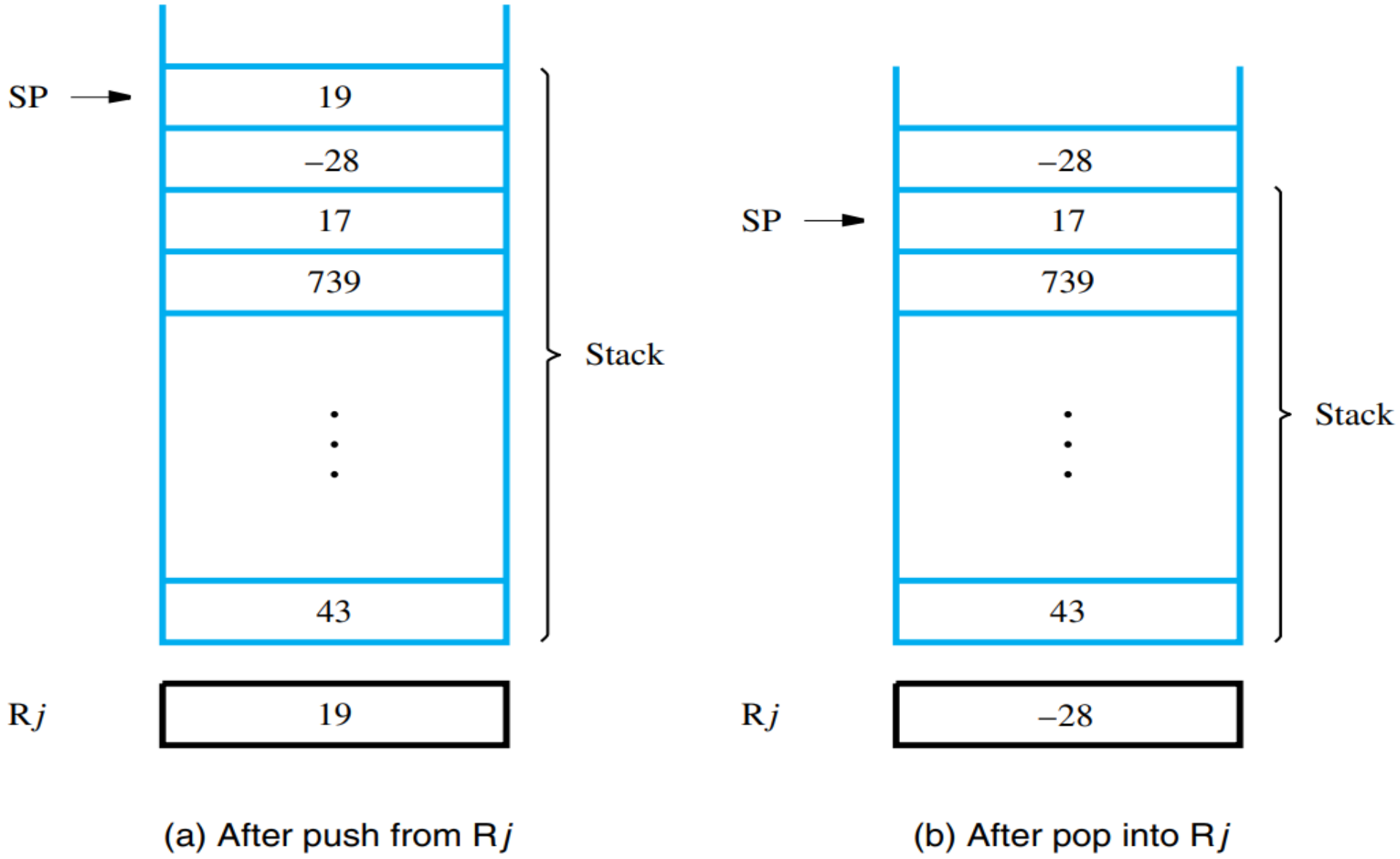


Figure 2.15 Effect of stack operations on the stack in Figure 2.14.

Subroutines

- In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a subroutine.
- The CALL instruction interrupts the flow of a program by passing control to an internal or external subroutine. An internal subroutine is part of the calling program. An external subroutine is another program.
- The RETURN instruction returns control from a subroutine back to the calling program and optionally returns a value.
- The subroutine may be called from different places in a calling program, provision must be made for program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed.
- The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the link register.

The Call instruction is just a special branch instruction that performs the following operations:

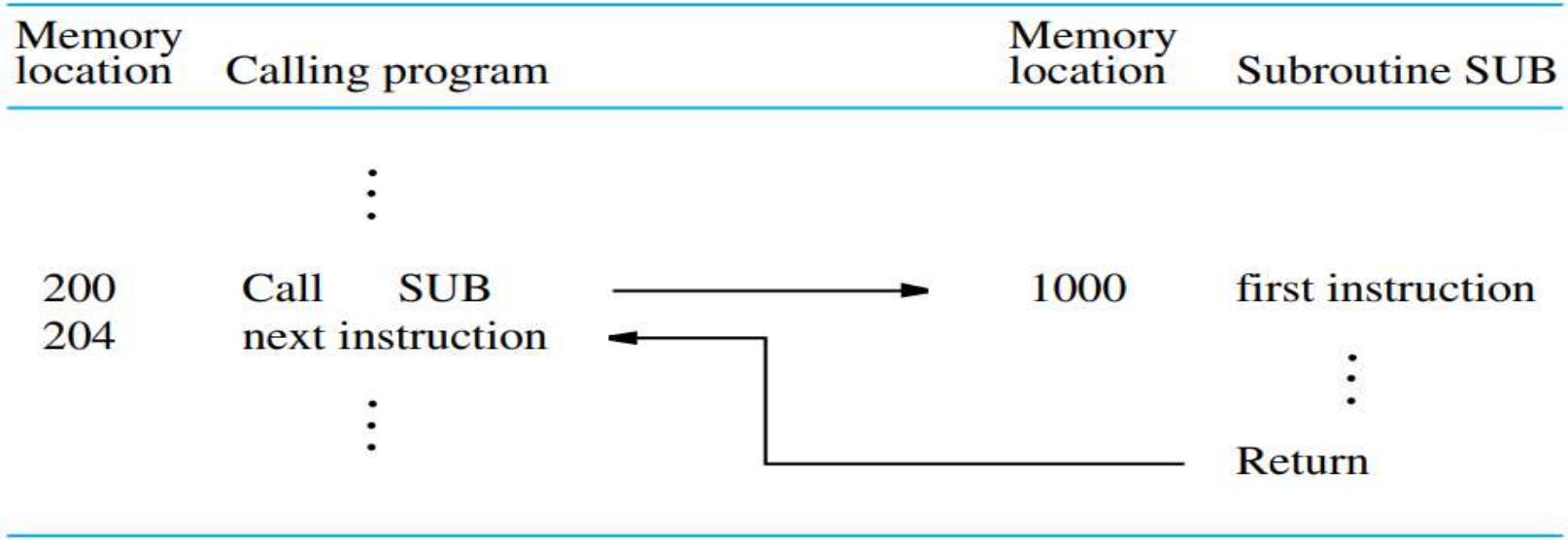
• Store the contents of the PC in the link register

• Branch to the target address specified by the Call instruction

The Return instruction is a special branch instruction that performs the operation

• Branch to the address contained in the link register

Figure illustrates how the PC and the link register are affected by the Call and Return instructions.



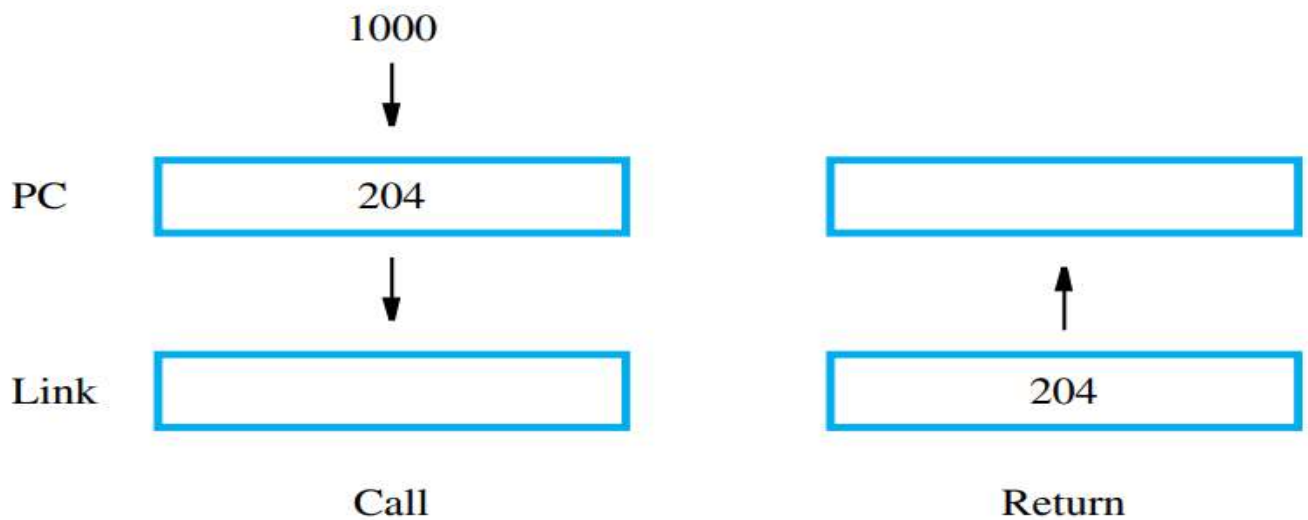
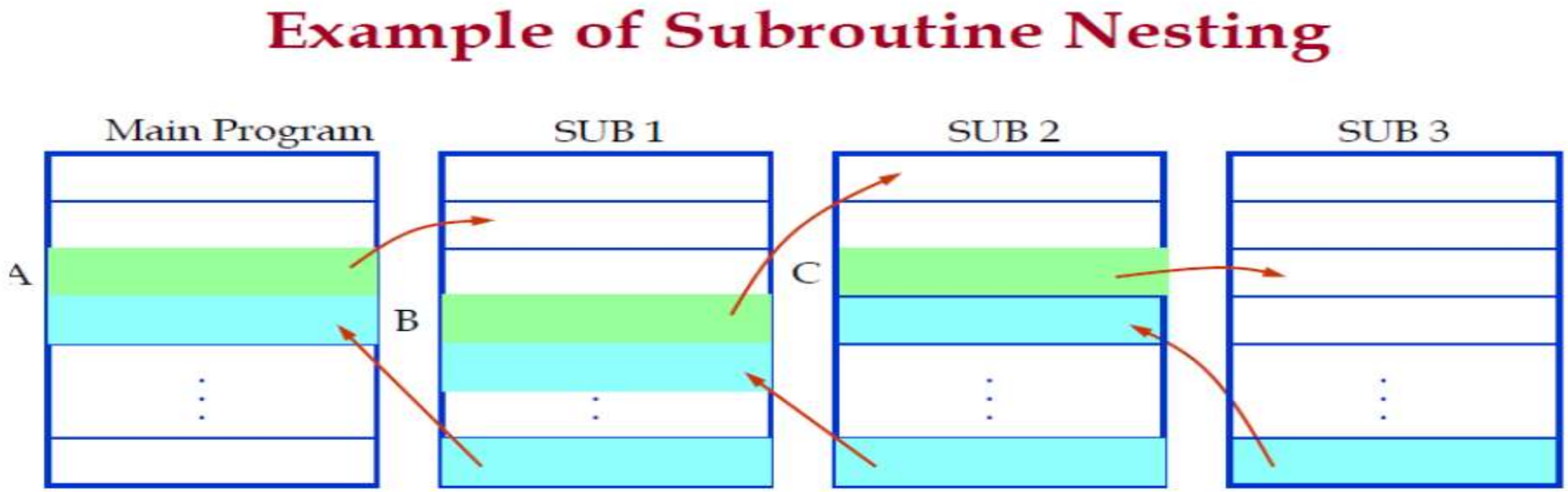


Figure 2.16 Subroutine linkage using a link register.

Subroutine Nesting and the Processor Stack

- A nested subroutine is a subroutine that is called from within some other subroutine.
- In this case, the return address of the second call is also stored in the link register, overwriting its previous contents.
- Hence, it is essential to save the contents of the link register in some other location before calling another subroutine.
- Otherwise, the return address of the first subroutine will be lost.



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, which are the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing.
- Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack.
- Passing parameters through processor registers is straightforward and efficient.

Calling program

```
                Load        R2, N           Parameter 1 is list size.
                Move        R4, #NUM1       Parameter 2 is list location.
                Call        LISTADD         Call subroutine.
                Store       R3, SUM         Save result.
                ⋮
```

Subroutine

```
LISTADD:        Subtract    SP, SP, #4      Save the contents of
                Store       R5, (SP)          R5 on the stack.
                Clear       R3              Initialize sum to 0.
LOOP:           Load        R5, (R4)        Get the next number.
                Add         R3, R3, R5      Add this number to sum.
                Add         R4, R4, #4      Increment the pointer by 4.
                Subtract    R2, R2, #1      Decrement the counter.
                Branch_if_[R2]>0 LOOP
                Load        R5, (SP)        Restore the contents of R5.
                Add         SP, SP, #4
                Return                      Return to calling program.
```

Figure 2.17 Program of Figure 2.8 written as a subroutine; parameters passed through registers.

Assume top of stack is at level 1 in Figure 2.19.

```
                Move                R2, #NUM1       Push parameters onto stack.
                Subtract            SP, SP, #4
                Store               R2, (SP)
                Load                R2, N
                Subtract            SP, SP, #4
                Store               R2, (SP)
                Call                LISTADD         Call subroutine
                                                      (top of stack is at level 2).
                Load                R2, 4(SP)       Get the result from the stack
                Store               R2, SUM           and save it in SUM.
                Add                 SP, SP, #8      Restore top of stack
                                                      (top of stack is at level 1).
                :

LISTADD:        Subtract            SP, SP, #16     Save registers
                Store               R2, 12(SP)
                Store               R3, 8(SP)
                Store               R4, 4(SP)
                Store               R5, (SP)          (top of stack is at level 3).
                Load                R2, 16(SP)      Initialize counter to n.
                Load                R4, 20(SP)      Initialize pointer to the list.
                Clear               R3              Initialize sum to 0.
LOOP:           Load                R5, (R4)        Get the next number.
                Add                 R3, R3, R5      Add this number to sum.
                Add                 R4, R4, #4      Increment the pointer by 4.
                Subtract            R2, R2, #1      Decrement the counter.
                Branch_if_[R2]>0    LOOP
                Store               R3, 20(SP)      Put result in the stack.
                Load                R5, (SP)        Restore registers.
                Load                R4, 4(SP)
                Load                R3, 8(SP)
                Load                R2, 12(SP)
                Add                 SP, SP, #16       (top of stack is at level 2).
                Return                              Return to calling program.
```

Figure 2.18 Program of Figure 2.8 written as a subroutine; parameters passed on the stack.

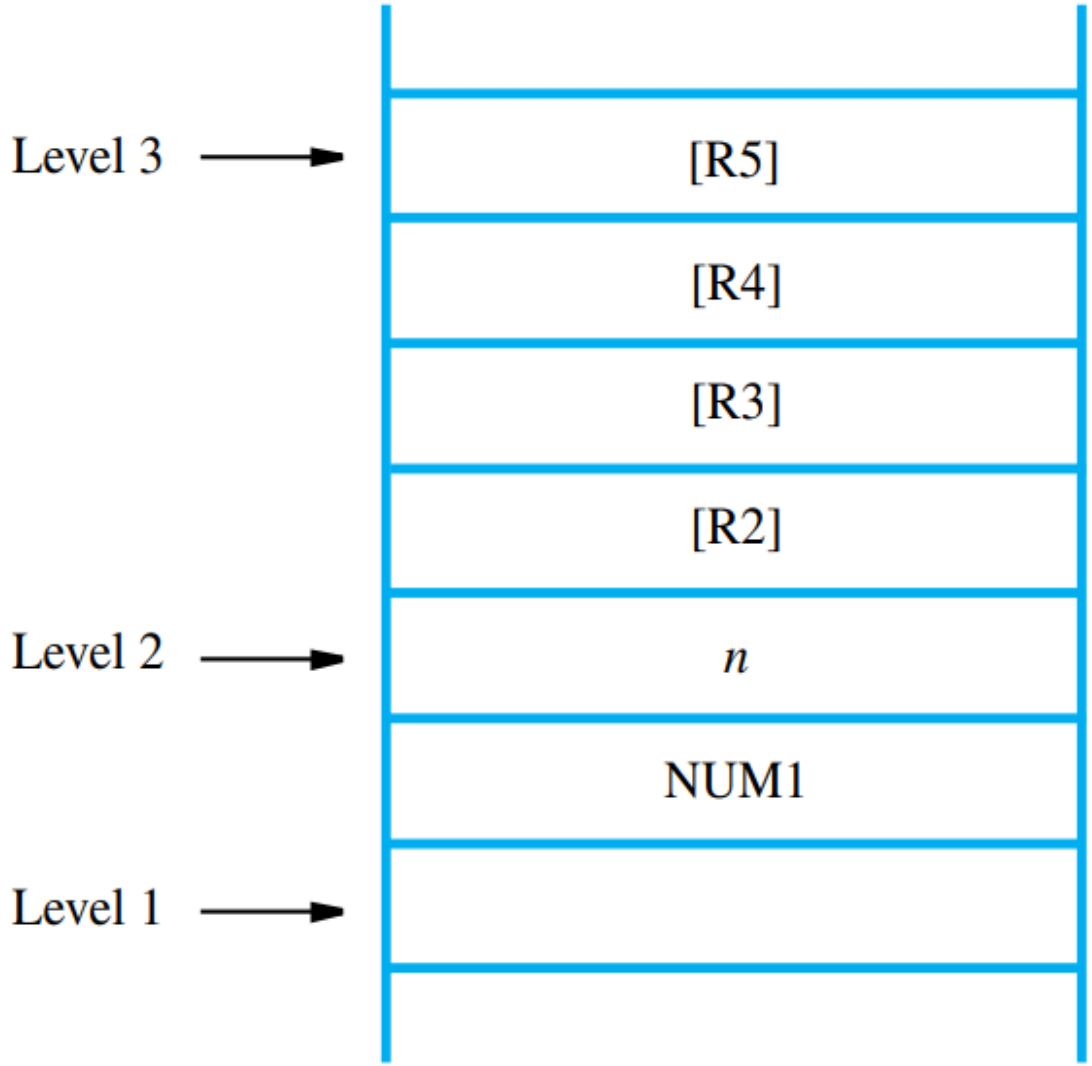


Figure 2.19 Stack contents for the program in Figure 2.18.

The Stack Frame

Each function has local memory associated with it to hold incoming parameters, local variables, and (in some cases) temporary variables. This region of memory is called a stack frame and is allocated on the process' stack.

In other words, it can be considered the collection of all information on the stack pertaining to a subroutine call.

Stack frames are only existent during the runtime process.

A frame pointer contains the base address of the function's frame. The code to access local variables within a function is generated in terms of offsets to the frame pointer.

The stack may change during the execution of a function as values are pushed or popped off the stack (such as pushing parameters in preparation to calling another function).

The frame pointer doesn't change throughout the function.

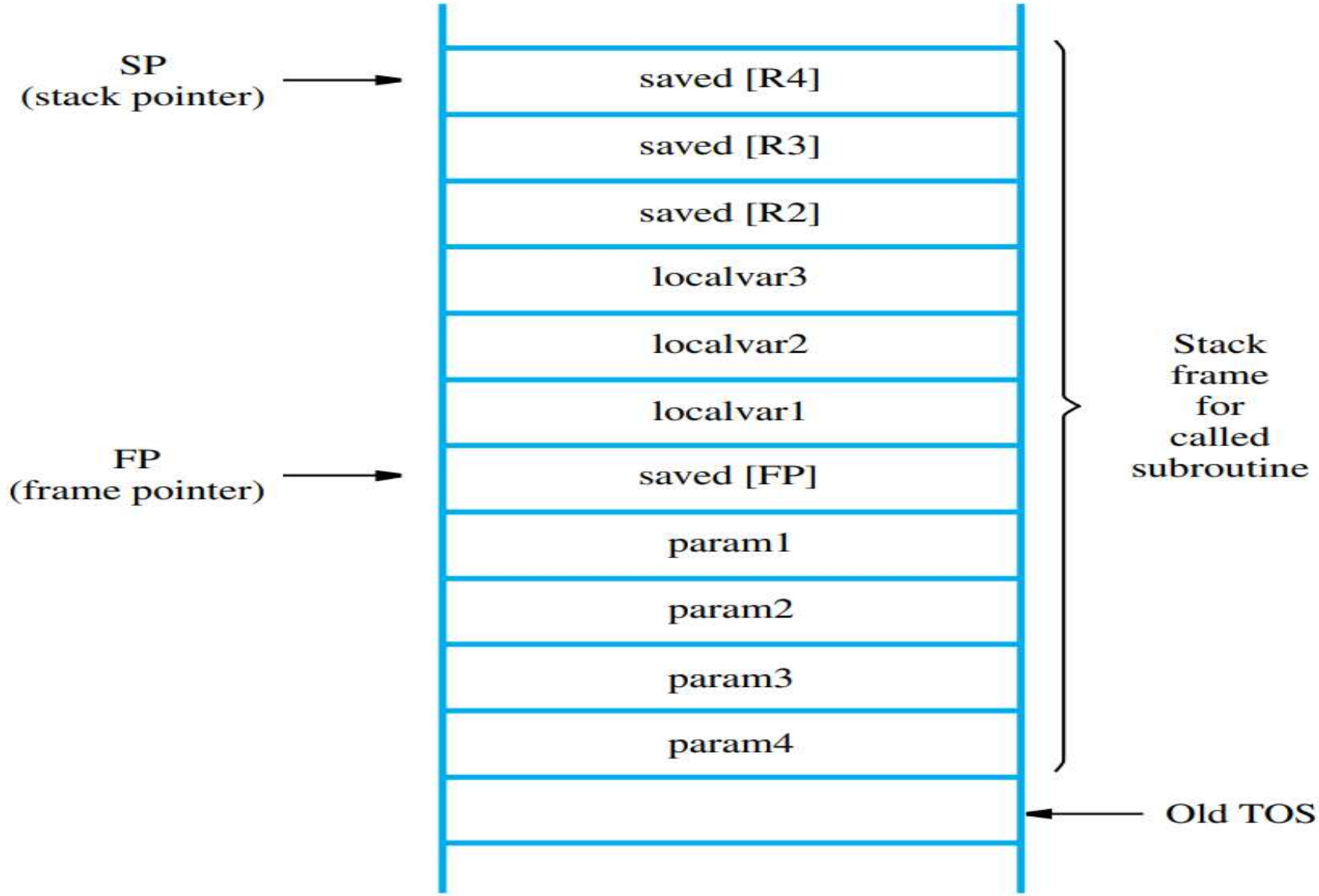


Figure 2.20 A subroutine stack frame example.

Additional Instructions

We introduce a few more instructions that are found in most instruction sets.

- Logic Instructions
- Shift and Rotate Instructions
- Multiplication and Division

Logic Instructions

Logic operations such as AND, OR, and NOT, applied to individual bits, are the basic building blocks of digital circuits.

It is also useful to be able to perform logic operations in software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel.

For example : And R4, R2, R3

The instruction computes the bit-wise AND of operands in registers R2 and R3, and leaves the result in R4.

An immediate form of this instruction may be

And R4, R2, #Value

where Value is a 16-bit logic value that is extended to 32 bits by placing zeros into the 16 most-significant bit positions.

Example 1**:**

MOV AX, 10100011b // Copy the binary value 10100011 to register AX

MOV BX, 00111101b // Copy the binary value 00111101 to register BX

AND AX, BX ; Perform an AND operation on the values in registers AX and BX. Store the output in the accumulator

The AND operation is performed as shown below:

Operand1 (AX): 1010 0011

Operand2 (BX): 0011 1101

Operand1(AX): 0010 0001

Logical OR

MOV AX, 10100011b
MOV BX, 00111101b
OR AX, BX
The AND operation is performed as shown below:
Operand1 (AX): 1010 0011
Operand2 (BX): 0011 1101
———————————-
Operand1(AX): 1011 1111

Logical NOT

MOV AL, 10100011b
NOT AL
The NOT operation is performed as shown below:
Operand1 (AL): 1010 0011
———————————-
Operand1(AL): 0101 1100

Shift and Rotate Instructions

Shift instructions can perform two basic types of shift operations; the logical shift and the arithmetic shift. Also, each of these operations can be performed to the right or to the left.

SHL - Shift Left.

SHR - Shift Right.

SAL – Shift Arithmetic Left.

SAR – Shift Arithmetic Right.

ROL – Rotate Left.

ROR – Rotate Right.

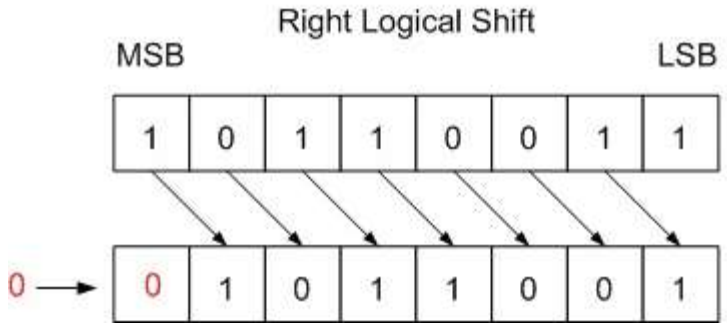
RCL - Rotate Carry Left.

RCR - Rotate Carry Right.

SHR : Shift Right

A Right Logical Shift of one position moves each bit to the right by one. The least significant bit is discarded and the vacant MSB is filled with zero.

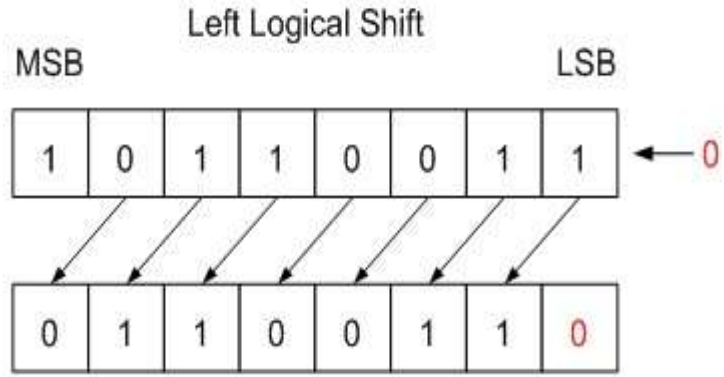
- The output of the shift right is divides the given number by 2.



SHL - Shift Left

A Left Logical Shift of one position moves each bit to the left by one. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded.

- The output of the shift left is multiplies the given number by 2

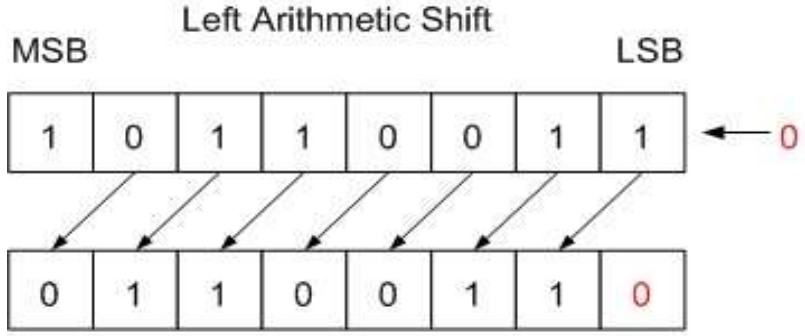


Arithmetic Shift

SAL – Shift Arithmetic Left

A Left Arithmetic Shift of one position moves each bit to the left by one. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded. It is identical to Left Logical Shift.

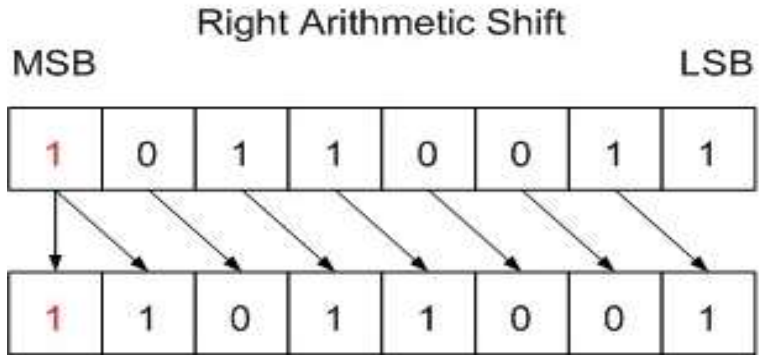
The output of the shift Arithmetic left is multiplies the given number by 2.



SAR – Shift Arithmetic Right.

A Right Arithmetic Shift of one position moves each bit to the right by one. The least significant bit is discarded and the vacant MSB is filled with the value of the previous (now shifted one position to the right) MSB. The signed bit is unchanged in this shift(so, the MSB is doesn't changed).

- The output of the shift Arithmetic right is divides the given number by 2.



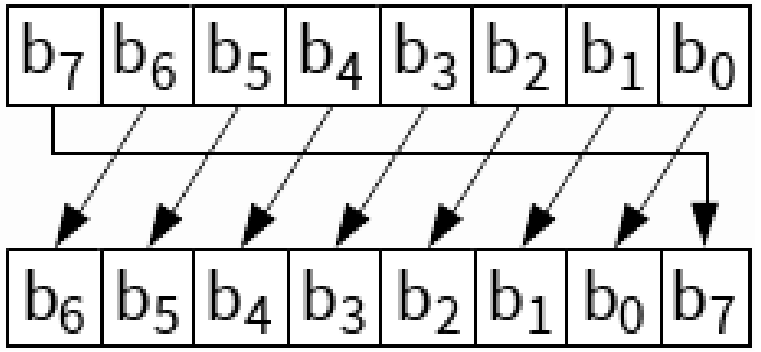
In this arithmetic shifts we take the negative number then we use the 2's complement to conversion of given number

Rotate Instructions

ROL – Rotate Left

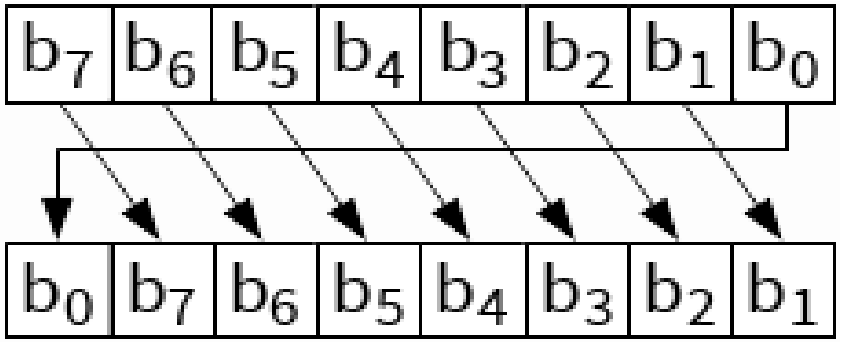
The ROL instruction rotates the mentioned bits in the register to the left side one by one such that leftmost bit that is being rotated is again stored as the rightmost bit in the register(MSB) is stored the LSB position.

- Rotate shifts (Circular shifts) are often used for cryptographic applications and are suitable when it is desirable to not lose any bit values.



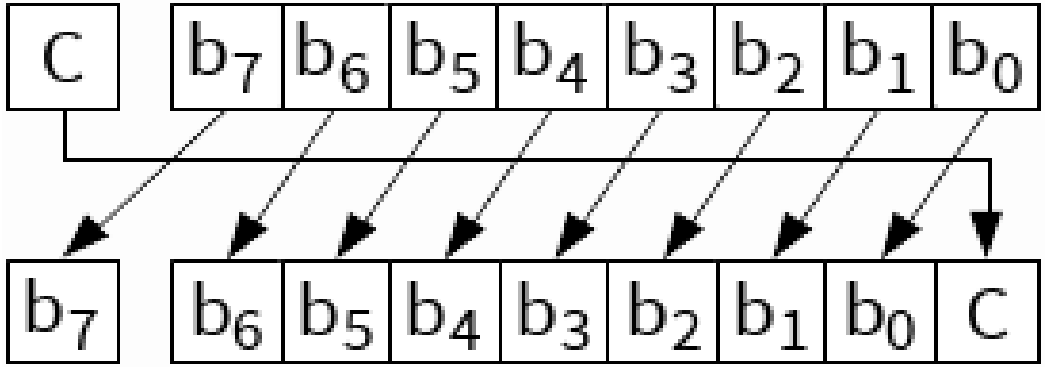
ROR - Rotate Right

The ROR instruction rotates the mentioned bits in the register to the right side one by one such that rightmost bit that is being rotated is again stored as the leftmost bit in the register(LSB) is stored the MSB position.



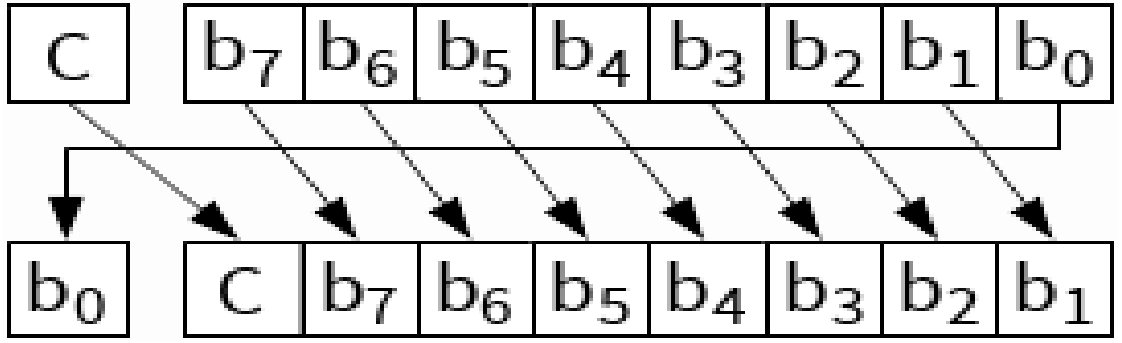
RCL : Rotate Carry Left

This instruction rotates the mentioned bits in the register to the left side one by one such that leftmost bit that is being rotated it is stored in the Carry Flag (CF), and the bit in the CF moved as the LSB in the register.



RCR : Rotate Carry Right

This instruction rotates the mentioned bits in the register to the right side such that rightmost bit that is being rotated it is stored in the Carry Flag (CF), and the bit in the CF moved as the MSB in the register.



Multiplication and Division

Two signed integers can be multiplied or divided by machine instructions.

The instruction Multiply Rk, Ri, Rj

performs the operation $Rk \leftarrow [Ri] \times [Rj]$

The product of two n-bit numbers can be as large as 2n bits. Therefore, the answer will not necessarily fit into register Rk.

A number of instruction sets have a Multiply instruction that computes the low-order n bits of the product and places it in register Rk, as indicated.

This is sufficient if it is known that all products in some particular application task will fit into n bits.

To accommodate the general 2n-bit product case, some processors produce the product in two registers, usually adjacent registers Rk and R(k + 1), with the high-order half being placed in register R(k + 1).

An instruction set may also provide a signed integer Divide instruction

Divide Rk, Ri, Rj

which performs the operation $Rk \leftarrow [Rj]/[Ri]$

placing the quotient in Rk. The remainder may be placed in R(k + 1), or it may be lost.

Encoding of Machine Instructions

Assembly-language instructions symbolically express the actions that must be performed by the processor circuitry. To be executed in a processor, assembly-language instructions must be converted by the assembler program into machine instructions that are encoded in a compact binary pattern.

Let us now examine how machine instructions may be formed.

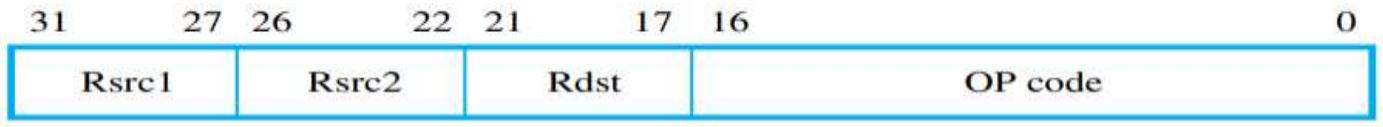
Add Rdst, Rsrc1, Rsrc2

The Add instruction is representative of a class of three-operand instructions that use operands in processor registers. Registers Rdst, Rsrc1, and Rsrc2 hold the destination and two source operands.

If a processor has 32 registers, then it is necessary to use five bits to specify each of the three registers in such instructions.

If each instruction is implemented in a 32-bit word, the remaining 17 bits can be used to specify the OP code that indicates the operation to be performed.

A possible format is shown in Figure



(a) Register-operand format

Now consider instructions in which one operand is given using the Immediate addressing mode,

such as Add Rdst, Rsrc, #Value

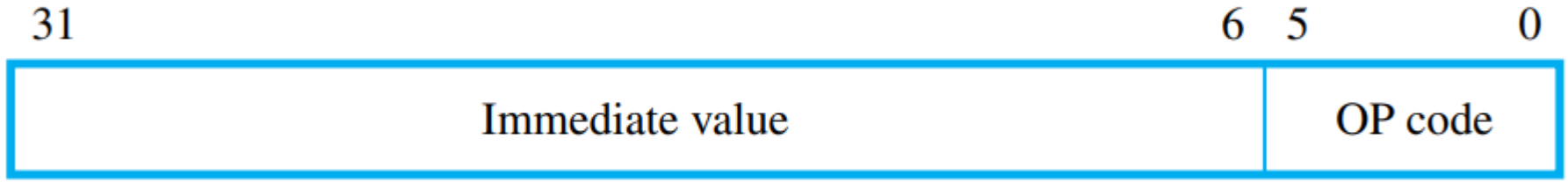
Of the 32 bits available, ten bits are needed to specify the two registers. The remaining 22 bits must give the OP code and the value of the immediate operand.

The most useful sizes of immediate operands are 32, 16, and 8 bits. Since 32 bits are not available, a good choice is to allocate 16 bits for the immediate operand.

This leaves six bits for specifying the OP code. A possible format is presented in Figure

(b) Immediate-operand format

- Finally, we should consider the Call instruction, which is used to call a subroutine. It only needs to specify the OP code and an immediate value that is used to determine the address of the first instruction in the subroutine.
- If six bits are used for the OP code, then the remaining 26 bits can be used to denote the immediate value.
- This gives the format shown in Figure



(c) Call format