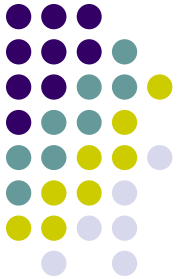
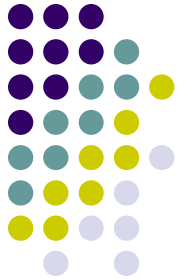
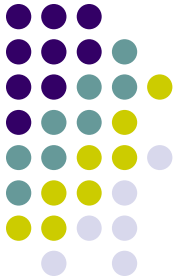


Chapter 2
Machine Instructions and Programs

Objectives

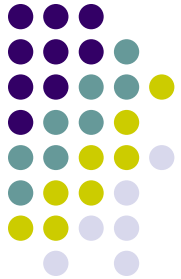


- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2’s-complement system.
- Addressing methods for accessing register and memory operands.



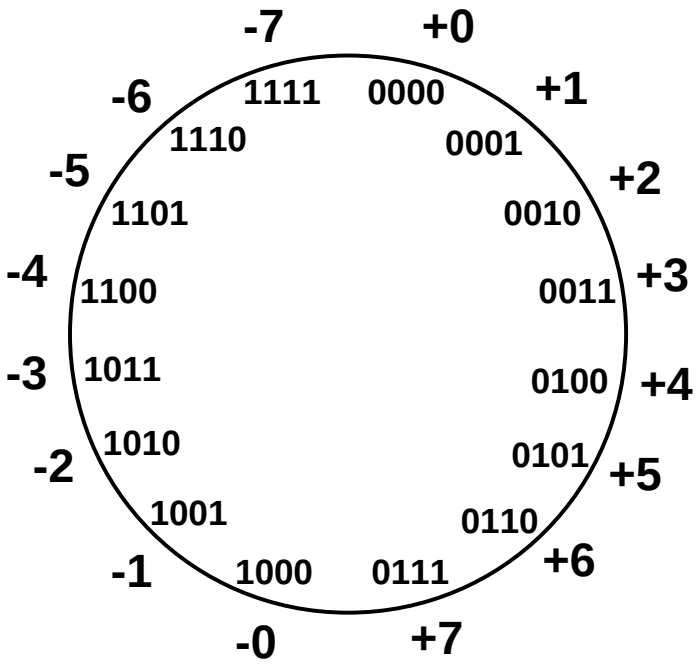
Number, Arithmetic Operations, and Characters

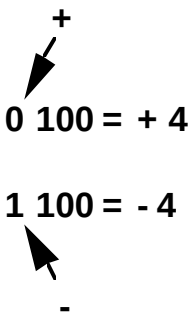
Signed Integer Representations



- 3 major representations:
  - Sign and magnitude
  - One’s complement
  - Two’s complement
- Assumptions for the Next Example:
  - 4-bit machine word
  - 16 different values can be represented
  - Roughly half are positive, half are negative

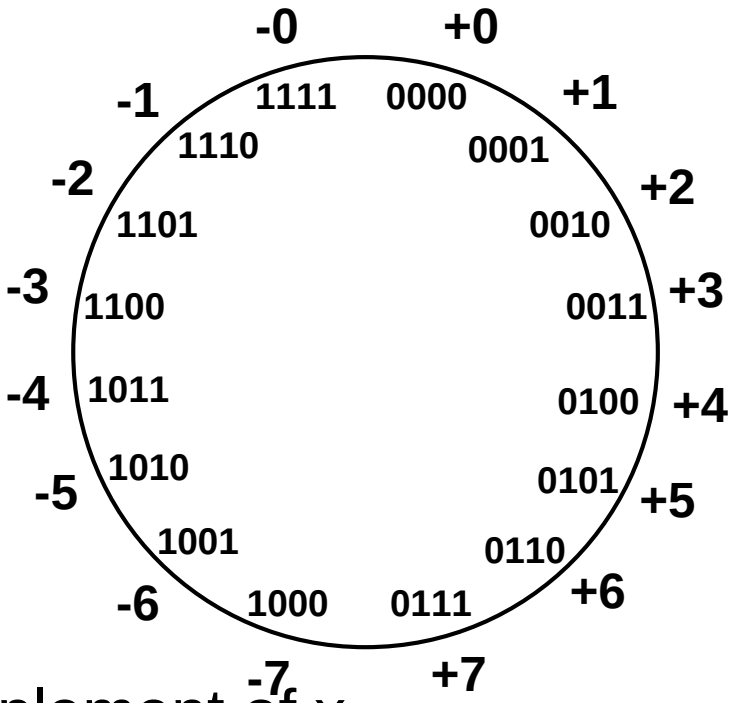
Sign and Magnitude Representation

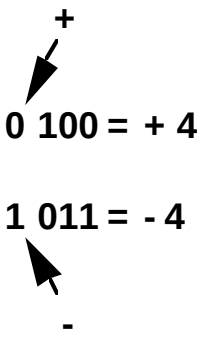




High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Number range for n bits = +/- $(2^{n-1} - 1)$
Problems: Two representations for 0 (0000 is +0, 1000 is –0) (see the number wheel)
Some Complexities in Addition, Subtraction

One's Complement Representation

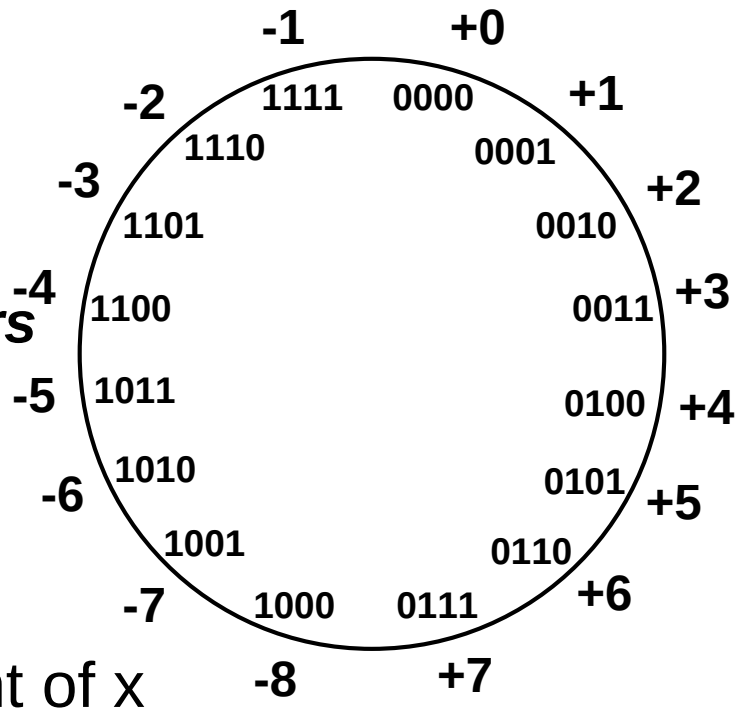


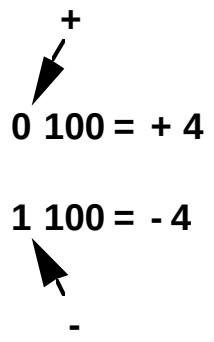


- - x = 1's complement of x
- 1's complement is invert 0 to 1 and 1 to 0
- Two representations for 0 (0000 is +0, 1111 is -0). causes some problems Some complexities in addition, subtraction
- Subtraction (X-Y) implemented by addition & 1's complement (x – y = X + 1's complement of Y = X + Y')

Two's Complement Representation

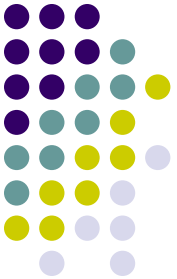
like 1's comp except negative numbers shifted one position clockwise





- - x = 2's complement of x
- 2's complement is just 1's complement + 1
- Only one representation for 0 (0000 => 1111+1 => 10000 => 0000 in 4 bits, ignore the carry out / MSB 1)
- Addition, Subtraction Very Simple

Binary, Signed-Integer Representations (Self Study)



Page 28

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

Figure 2.1. Binary, signed-integer representations.

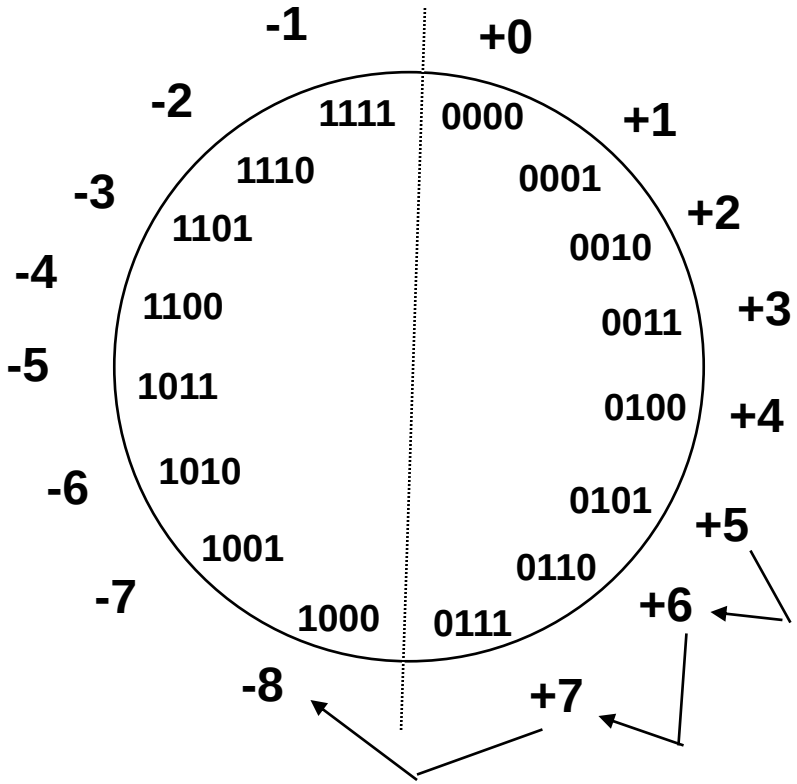
2's-Complement Add and Subtract Operations(Self Study)

Page 31

(a)	0 0 1 0	(+2)		(b)	0 1 0 0	(+4)
	+ 0 0 1 1	(+3)			+ 1 0 1 0	(- 6)
	0 1 0 1	(+5)			1 1 1 0	(- 2)
(c)	1 0 1 1	(- 5)		(d)	0 1 1 1	(+7)
	+ 1 1 1 0	(- 2)			+ 1 1 0 1	(- 3)
	1 0 0 1	(- 7)			0 1 0 0	(+4)
(e)	1 1 0 1	(- 3)	⇒		1 1 0 1	
	- 1 0 0 1	(- 7)			+ 0 1 1 1	
					0 1 0 0	(+4)
(f)	0 0 1 0	(+2)	⇒		0 0 1 0	
	- 0 1 0 0	(+4)			+ 1 1 0 0	
					1 1 1 0	(- 2)
(g)	0 1 1 0	(+6)	⇒		0 1 1 0	
	- 0 0 1 1	(+3)			+ 1 1 0 1	
					0 0 1 1	(+3)
(h)	1 0 0 1	(- 7)	⇒		1 0 0 1	
	- 1 0 1 1	(- 5)			+ 0 1 0 1	
					1 1 1 0	(- 2)
(i)	1 0 0 1	(- 7)	⇒		1 0 0 1	
	- 0 0 0 1	(+1)			+ 1 1 1 1	
					1 0 0 0	(- 8)
(j)	0 0 1 0	(+2)	⇒		0 0 1 0	
	- 1 1 0 1	(- 3)			+ 0 0 1 1	
					0 1 0 1	(+ 5)

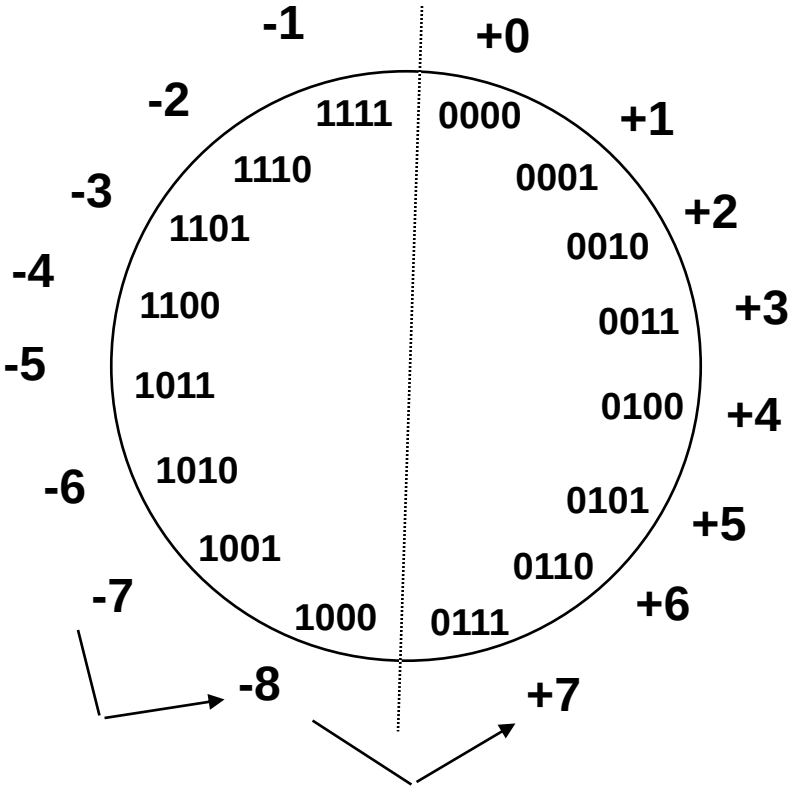
Figure 2.4. 2's-complement Add and Subtract operations.

Overflow Condition - Add two positive numbers to get a negative number or two negative numbers to get a positive number



5 + 3 = -8

0101 + 0011 = 1000



-7 - 2 = +7

1001 + 1110 = 10111 => 0111

X

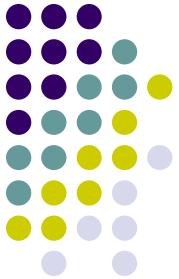
Overflow Condition – Carry in to MSB ≠ Carry out from MSB

		0 1 1 1←carry-in		1 0 0 0	
Overflow	5	0 1 0 1	-7	1 0 0 1	Overflow
	3	0 0 1 1	-2	1 1 1 0	
	-8	1 0 0 0	7	1 0 1 1 1	
		0 0 0 0		1 1 1 1	
No overflow	5	0 1 0 1	-3	1 1 0 1	No overflow
	2	0 0 1 0	-5	1 0 1 1	
	7	0 1 1 1	-8	1 1 0 0 0	

Two Ways to detect Overflow: (1) when carry-in to the MSB (most significant bit) does not equal carry out from MSB

(2) Add two positive numbers to get a negative number

or, Add two negative numbers to get a positive number



Memory Locations, Addresses, and Operations

Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which stores 1 bit.
- Data is usually accessed in *n*-bit groups, called a "**word**".
- *n* is called **word length**.
- Typically n=32 or 64 bits etc. (such systems called 32-bit systems, like: 32-bit CPU or 64-bit OS)

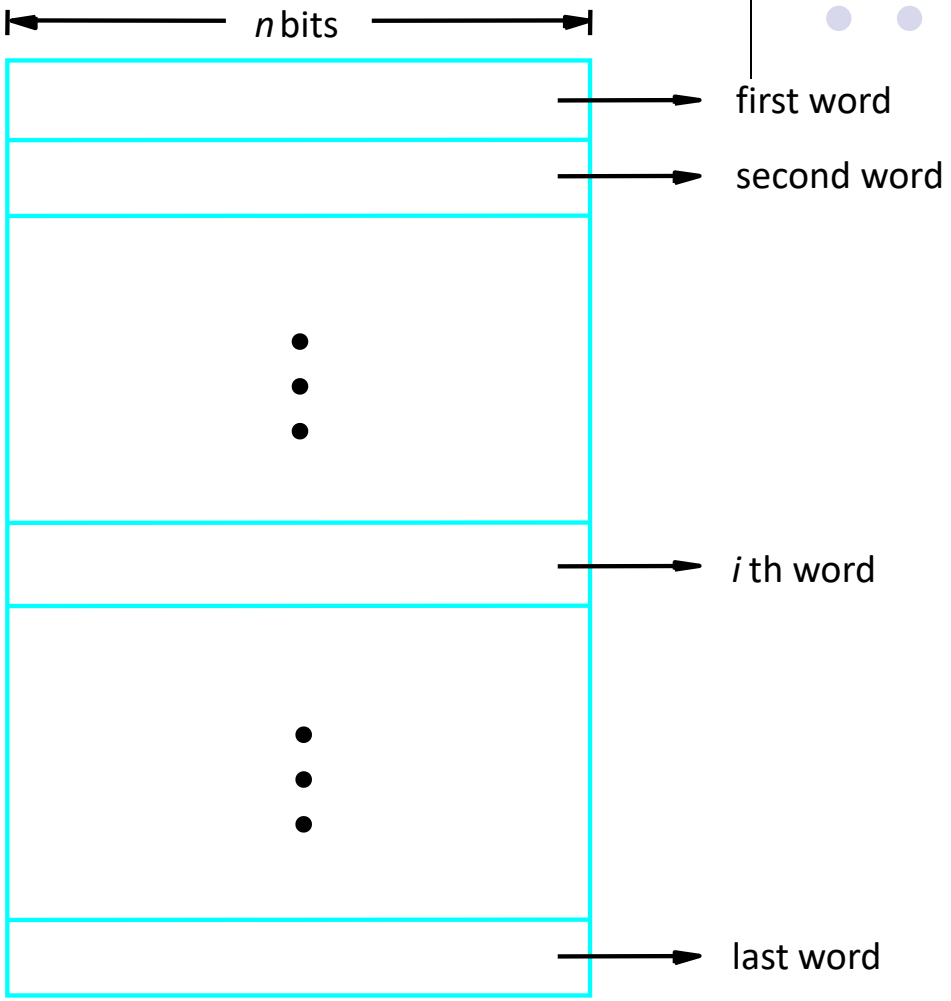
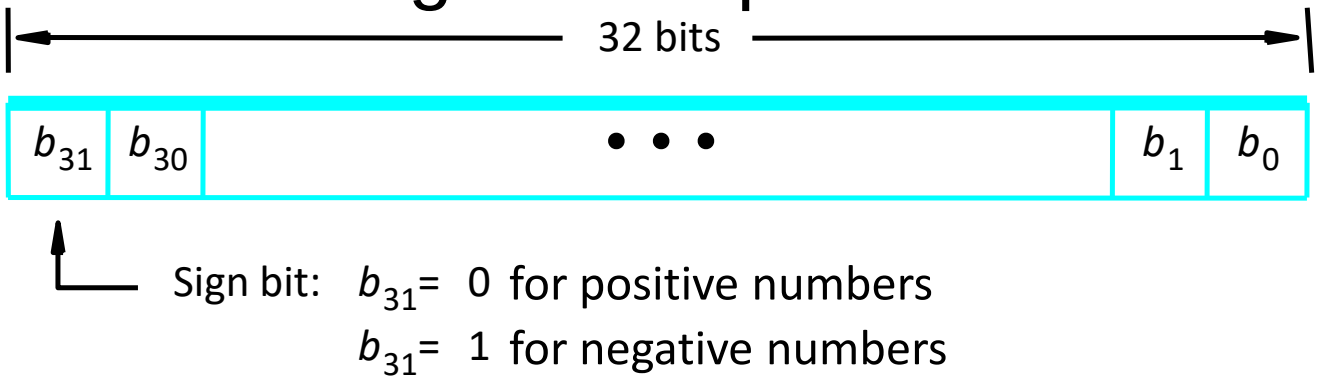


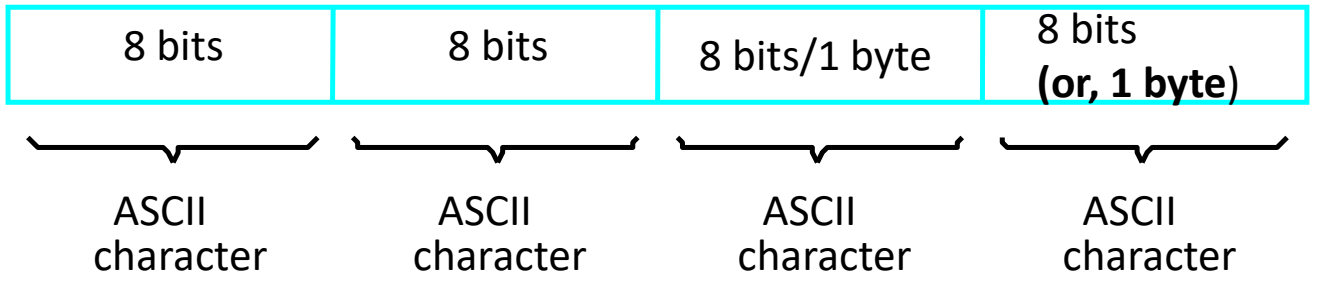
Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example

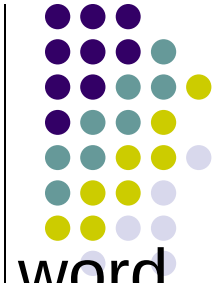


(a) A signed integer



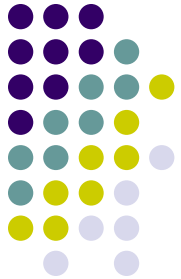
(b) Four characters

Memory Location, Addresses, and Operation



- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location needed.
- Each byte (8-bit group) in the memory are addressable! => This is called **byte addressable!**
- A *k*-bit addressed memory chip has 2^k memory locations, namely 0 – 2^k-1, called **memory space**. (example: 4 bit => addresses 0000 to 1111 = 0 to 15 = 0 to 2^4-1)
- 1K(kilo)=2^{10}=1024;1MB(Megabyte)=1024KB(kilobyte)=2^{10}*2^{10}=2^{20}bytes
- 1GB (Gigabytes)=1024Megabytes=2^{10}*2^{20}bytes= 2^{30}bytes
- **24-bit memory:**2^{24}= 2^4*2^{20}=16*1Mega= 16M (1Mega=2^{20})
- **32-bit memory:** 2^{32} = 4GB (1GB (gigabytes) = 2^{30} bytes 4GB=2^{32} bytes because 4G= 4*1G = 2^2*2^{30} = 2^{32})
- 1T(tera)=2^{40}, (after this, peta=2^{50},exa=2^{60},zetta=2^{70},yotta

Memory Location, Addresses, and Operation



- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory **byte-addressable memory.**
- **Byte locations have addresses 0, 1, 2, … If word length is 32 bits (4 bytes), then successive words are located at addresses 0, 4, 8,…**

Big-Endian and Little-Endian Assignment of Memory Addresses

Big-Endian: higher (bigger) byte addresses are used for the least significant bytes of the word (hint: bigger address ends it (i.e. at rightmost / LSB)

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word

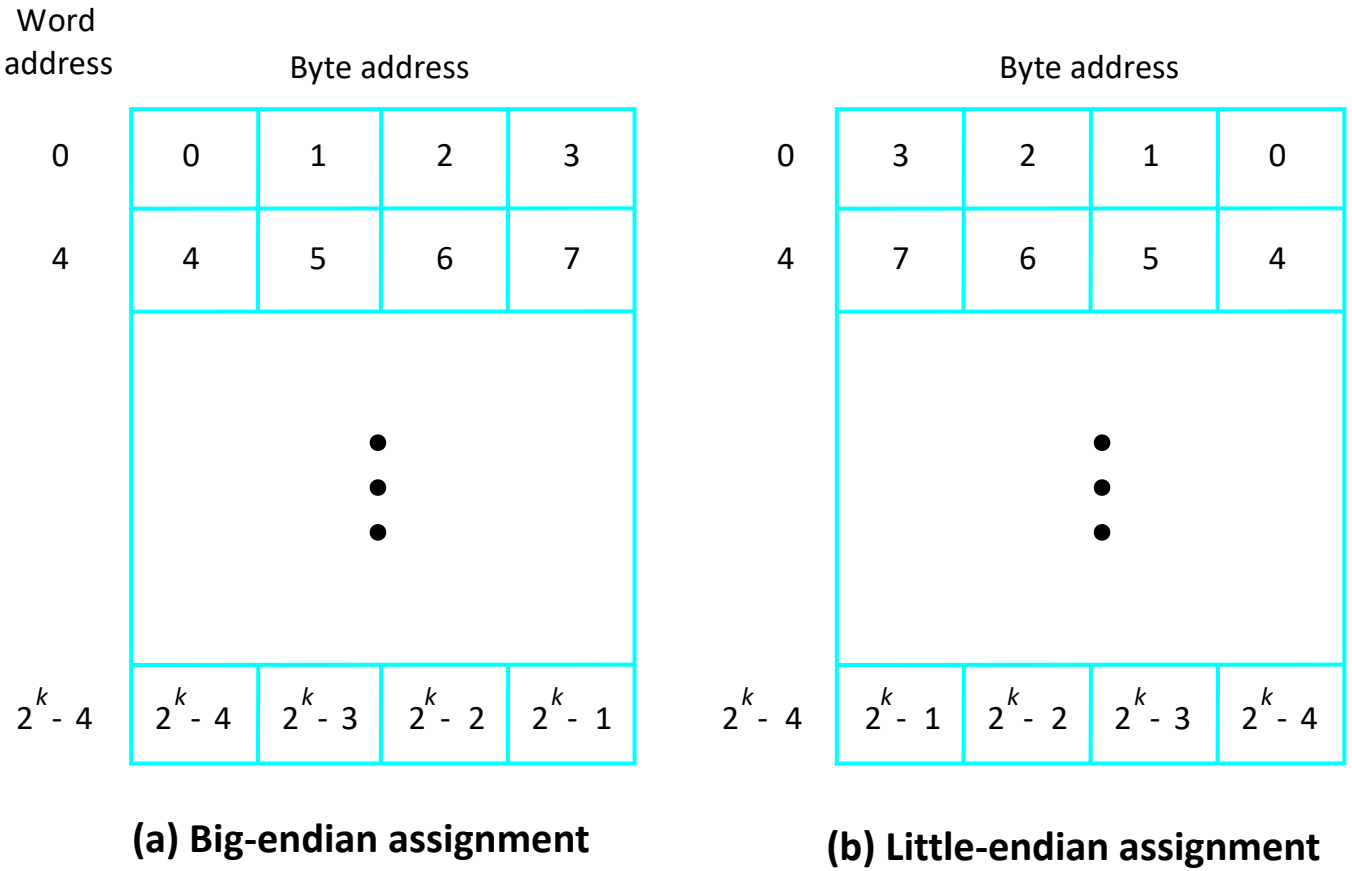
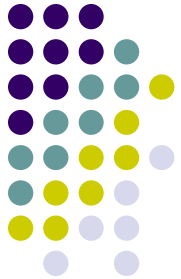


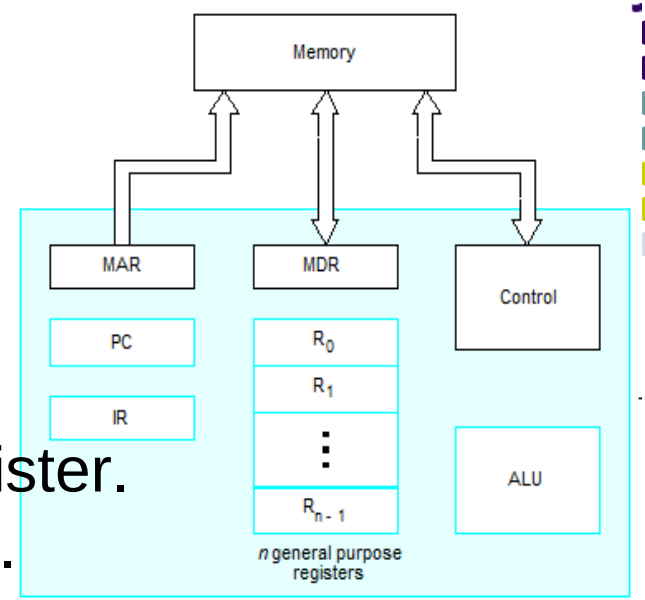
Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

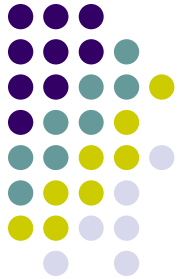


- Ordering of bytes: Little endian and Big endian schemes
- **Word alignment**
  - Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4, 6, 8, …. bytes
    - 32-bit word: word addresses: 0, 4, 8, 12, 16, …. bytes
    - 64-bit word: word addresses: 0, 8,16, 24, 32, …. bytes
- Access numbers, characters, and character strings

Memory Operation



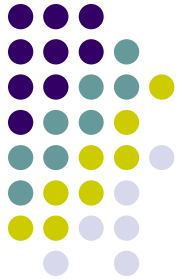
- **LOAD** (or Read or Fetch)
  - Copy content from memory to a Register. The memory content doesn't change.
  - CPU places the sought address in MAR register, then places the $\overline{RD}$ control signal to the memory chip, then waits, until it receives the desired data into the MDR register
- **STORE** (or Write)
  - Overwrite the content in memory
  - CPU Places the Address and Data in MAR and MDR registers, sends the WR control signal to the memory chip. Upon completion, the memory chip sends back MFC (Memory Function Complete) signal.



Instruction and Instruction Sequencing

“Must-Perform” Operations for a computer:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers



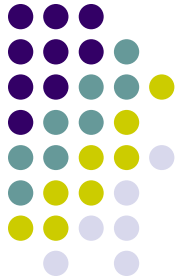
Computer instructions must be capable of performing 4 types of operations:

i. Data transfer/movement between memory and processor registers.
    E.g., memory read, memory write

ii. Arithmetic and logic operations:
    E.g., addition, subtraction, comparison between two numbers.

iii. Program sequencing and flow of control:
    Branch instructions

iv. Input/output transfers to transfer data to and from the real world.



Examples of different types of instructions in assembly language notation:

- Data transfers between processor and memory:

  Move A, B (B = A).
  Move A, R1 (R1 = A).

- Arithmetic and logic operation:

  Add A, B, C (C = A + B)

- Sequencing:

  Jump Label (Jump to the subroutine which starts at Label).

- Input/output data transfer:

  Input PORT, R5 (Read from i/o port “PORT” to register R5).

Register Transfer Notation

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- R0, R1, R2, ... => always indicates registers
- Any other symbol => indicates memory location
  - Example: X, Y, Z, A, B, M, LOC, LOCA, LOCB
- **Contents of a location** are denoted by placing **square brackets** around the name of the location (**R1←[LOC]**, **R3 ←[R1]+[R2]**)
- Register Transfer Notation (RTN)

Assembly Language Notation

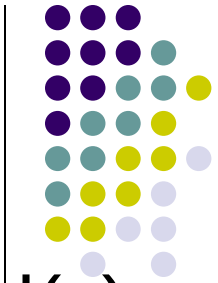
- Represent machine instructions and programs.

Opcode *source_operand* *destination_operand*

OP src_op dest_op

- MOV LOC, R1 === equivalent ==> **R1←[LOC]**
- ADD R1, R2, R3 =equivalent=> **R3 ←[R1]+[R2]**

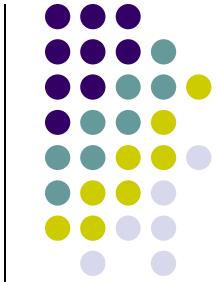
CPU Organization: Internal Storage Architecture:



- Controls how its instructions use the operand(s).
- The type of internal storage in a processor is the most basic differentiation.

i. Single Accumulator (AC) CPU Organization

- One operand is implicitly the Accumulator Register.
- The Accumulator is both an implicit input operand and a result.
- Accumulator (AC) has to be saved to memory quite often.



ii. Register-Memory CPU Organization:

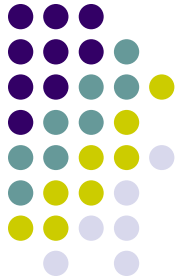
• One input operand is a register, one is in memory and the result goes to a register.

•It can access memory as part of any instruction.

iii. Register-Register/Load-Store CPU Organization:

• All operands are registers.

•It can access memory only with load and store instructions.



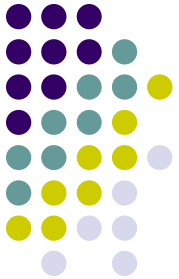
iv. Stack CPU Organization:

•No Registers, but CPU-internal stack memory holds operands and result are always in the stack.

•A Top of stack register (TOS) points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand and TOS is updated to point to the result.

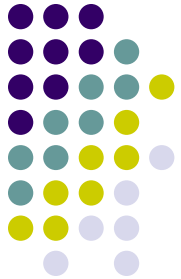
•All operands are implicit.

Instruction Formats



- Three-Address Instructions
  - ADD R2, R3, R1 R1 ← R2 + R3
- Two-Address Instructions
  - ADD R2, R1 R1 ← R1 + R2
- One-Address Instructions (usually for Single Accumulator CPU organization): AC register is always an implicit operand
  - ADD M (AC ← AC+M[AR] (AC is the Accumulator Register))
- Zero-Address Instructions (Usually for Stack CPU organization) No explicit operands, both operands are implicit
  - ADD (TOS ← TOS + (TOS – 1); TOS means Top of Stack)
- RISC Instructions: An instruction can have 3~4 registers! Memory Access is restricted Only to LOAD and STORE instructions

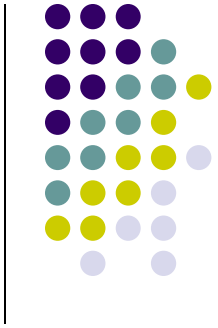
Instruction Formats



Example: Evaluate X ← (A+B) * (C+D)

- **Three-Address Format**
  1. ADD A, B, R1 ; R1 ← M[A] + M[B]
  2. ADD C, D, R2 ; R2 ← M[C] + M[D]
  3. MUL R1, R2, X ; M[X] ← R1 * R2

Instruction Formats



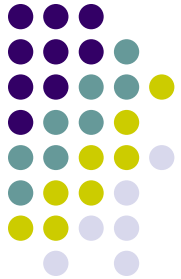
Example: Evaluate X←(A+B) ∗ (C+D)

- **Two-Address instruction format**

1.	MOV	A, R1	; R1 ← M[A]
2.	ADD	B, R1	; R1 ← R1 + M[B]
3.	MOV	C, R2	; R2 ← M[C]
4.	ADD	D, R2	; R2 ← R2 + M[D]
5.	MUL	R2, R1	; R1 ← R1 ∗ R2
6.	MOV	R1, X	; M[X] ← R1

Why not instructions like ADD A,B to make like B ← A+B
Because => both operands can't be memory locations, at least one must be register. Besides the content of B should not be overwritten (the programmer knows nothing about this side effect!)

Instruction Formats***

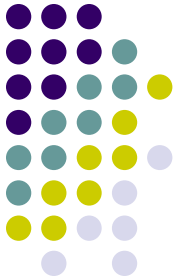


Example: Evaluate X ← (A+B) * (C+D)

- **One-Address Instruction Format**

1. LOAD A ; AC ← M[A]
2. ADD B ; AC ← AC + M[B]
3. STORE T ; M[T] ← AC
4. LOAD C ; AC ← M[C]
5. ADD D ; AC ← AC + M[D]
6. MUL T ; AC ← AC * M[T]
7. STORE X ; M[X] ← AC

Instruction Formats

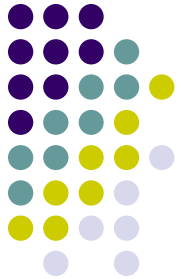


Example: Evaluate X = (A+B) ∗ (C+D)

- **Zero-Address Instruction Format**
  (must use stack processor organization.
  TOS means Top of Stack)

1.	PUSH A	; TOS ← A
2.	PUSH B	; TOS ← B
3.	ADD	; TOS ← (A + B)
4.	PUSH C	; TOS ← C
5.	PUSH D	; TOS ← D
6.	ADD	; TOS ← (C + D)
7.	MUL	; TOS ←(C+D)∗(A+B)
8.	POP X	; M[X] ← TOS

Instruction Formats

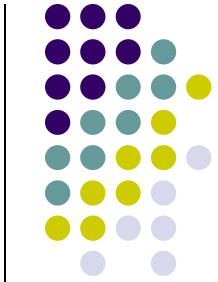


Example: Evaluate X = (A+B) ∗ (C+D)

- **RISC Instruction Format** (i).RISC can use 3 registers in a single instruction; (ii).Only the LOAD and STORE instructions can access memory

1. LOAD A, R1 ; R1 ← M[A]
2. LOAD B, R2 ; R2 ← M[B]
3. LOAD C, R3 ; R3 ← M[C]
4. LOAD D, R4 ; R4 ← M[D]
5. ADD R1, R2, R1 ; R1 ← R1 + R2
6. ADD R3, R4, R3 ; R3 ← R3 + R4
7. MUL R1, R3, R1 ; R1 ← R1 ∗ R3
8. STORE R1, X ; M[X] ← R1

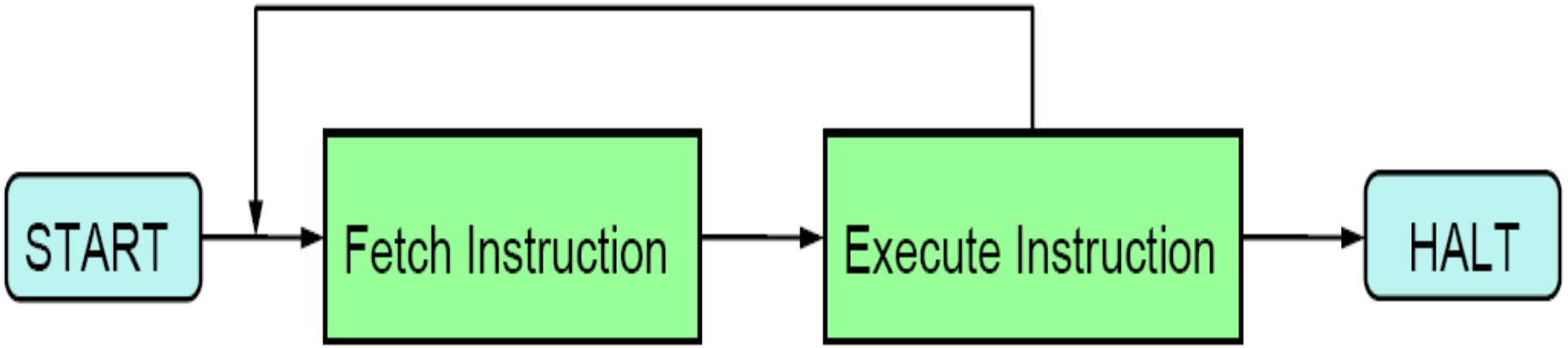
Using Registers



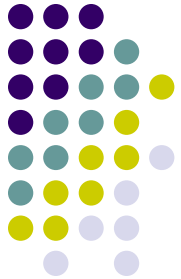
- Registers are faster
- Shorter instructions
  - The number of registers is smaller (e.g. 32 registers need 5 bits)
- Potential speedup
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution

- Basic instruction cycle



Straight-line sequencing



- The processor control circuits use the information in the program counter (PC) to fetch and execute instructions, one at a time, in the order of increasing addresses.
- This is called straight- line sequencing.

Instruction Execution and Straight-Line Sequencing

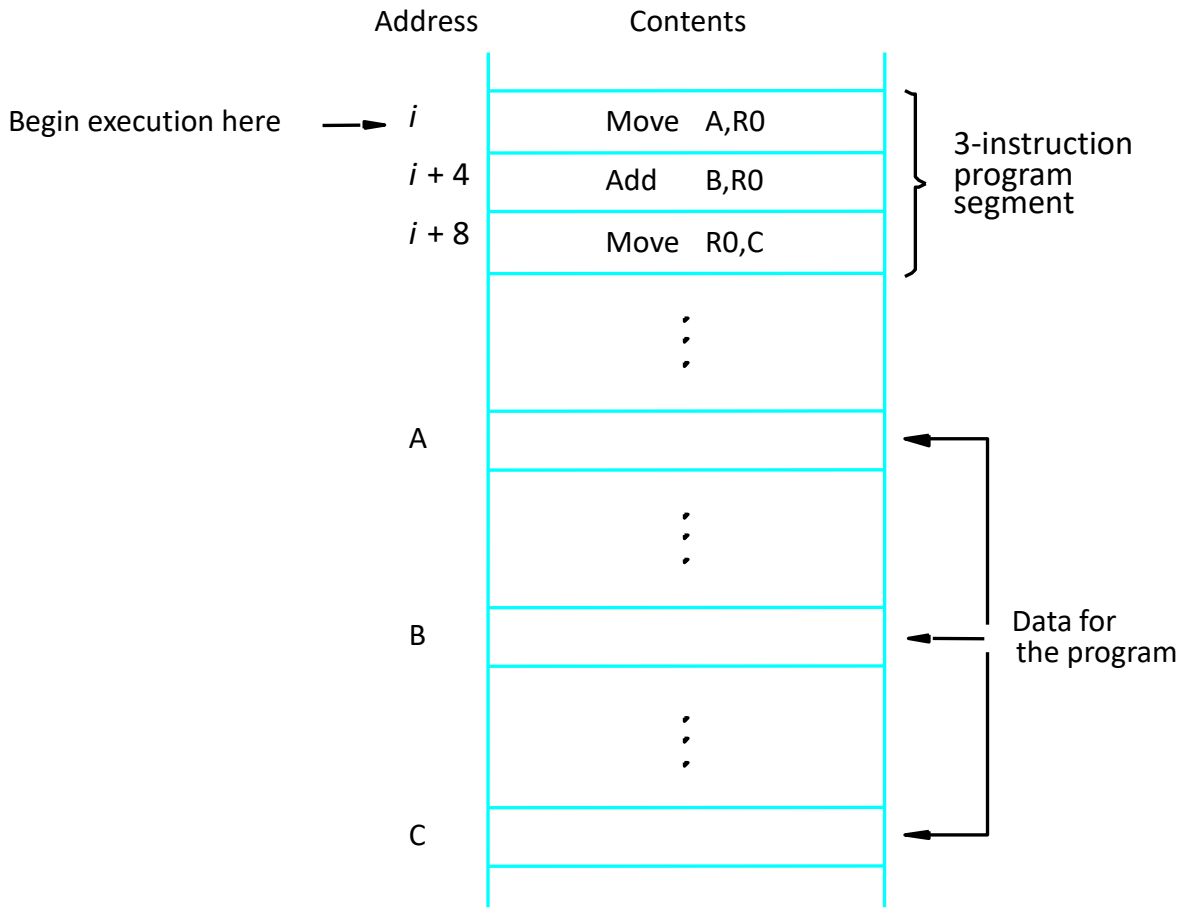


Figure 2.8. A program for C ← [A] + [B].

Assumptions:
- One memory operand per instruction
- **32-bit word length**
- Memory is byte addressable
- Each instruction fits in ONE word (Full memory address can be directly specified in a single-word instruction
(though this is not realistic!!!)

Two-phase procedure
-Instruction fetch
-Instruction execute

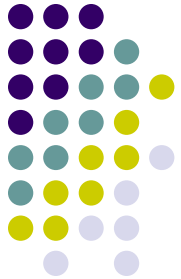
Page 43

Fetch/Execute cycle

- Execution of an instruction takes place in two phases:
  - Instruction fetch.
  - Instruction execute.

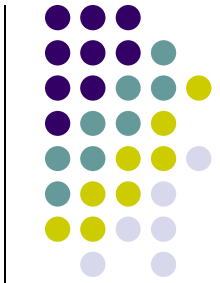
- Instruction fetch:
  - Fetch the instruction from the memory location whose address is in the Program Counter (PC).
  - Place the instruction in the Instruction Register (IR).
- Instruction execute:
  - Instruction in the IR is examined (decoded) to determine which operation is to be performed.
  - Fetch the operands from the memory or registers.
  - Execute the operation.
  - Store the results in the destination location.
- Basic fetch/execute cycle repeats indefinitely.

Branching



- Branch instructions load a new value 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.

Conditional Branching



- 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.

Branching

Adding An Array Of Numbers without Using any Loop (straight line program)

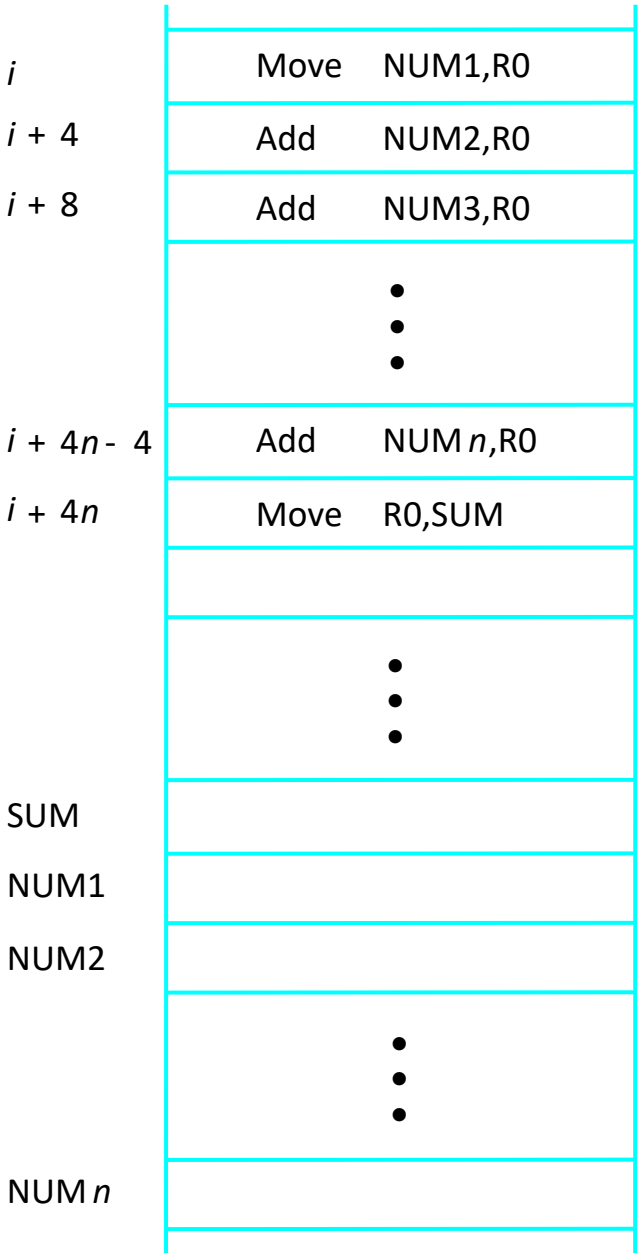


Figure 2.9. A straight-line program for adding *n* numbers.

Branching

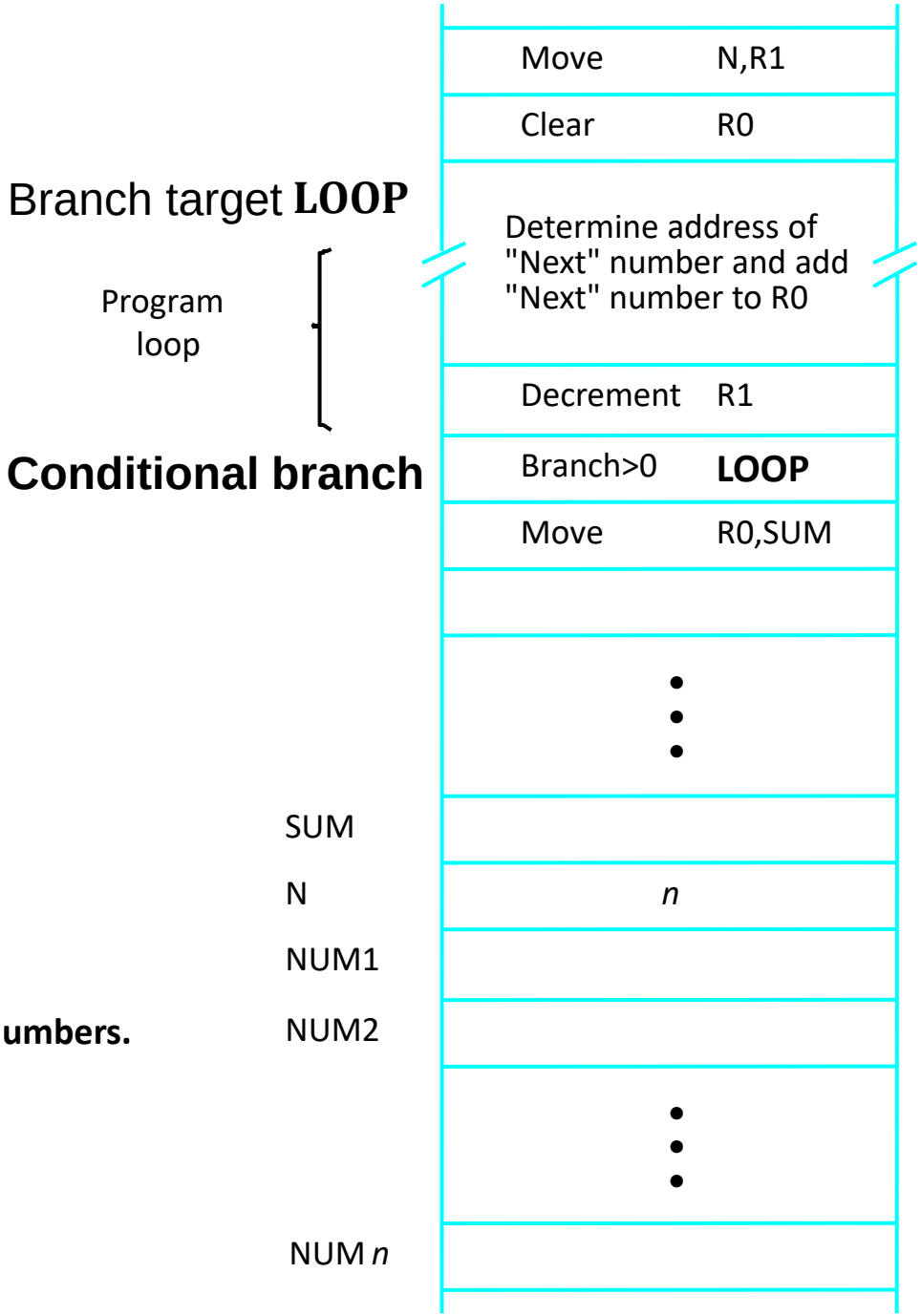
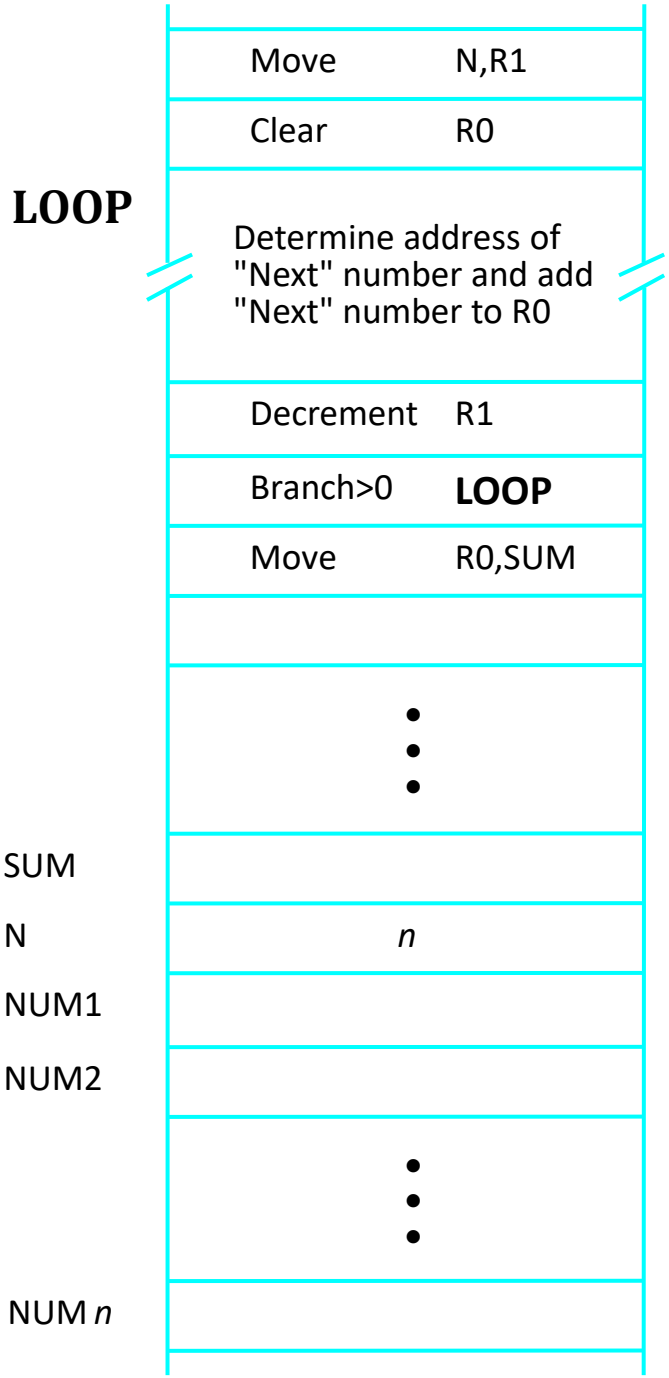


Figure 2.10. Using a loop to add *n* numbers.

Book Examples (Fig. 2.12): Indirect Addressing to Compute the Array Sum

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	

Figure 2.12 Use of indirect addressing in the program of Figure 2.10.



Book Examples (Fig. 2.12): Indirect Addressing to Compute the Array Sum (Fig. 2.10)

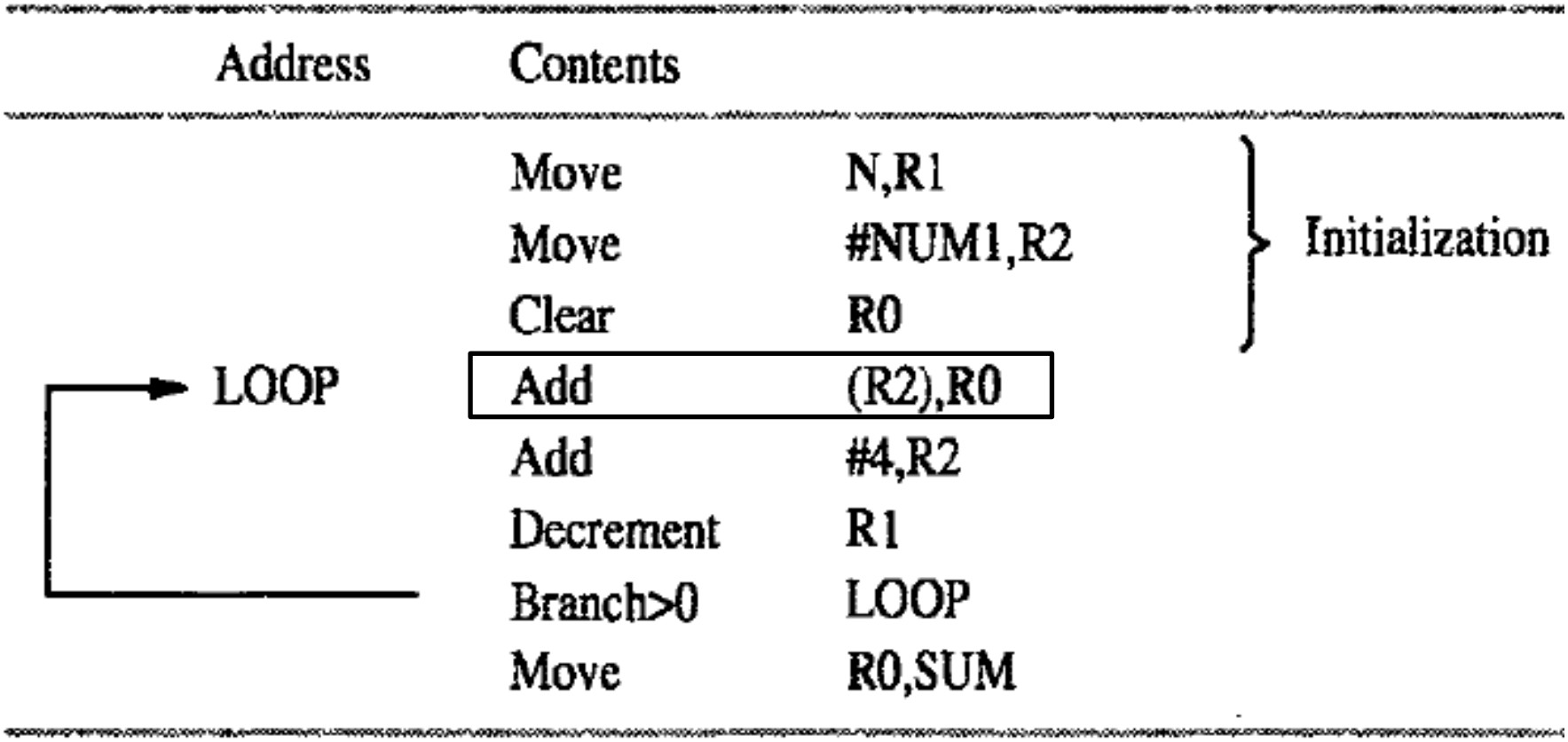
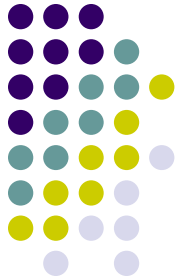


Figure 2.12 Use of indirect addressing in the program of Figure 2.10.

Condition Codes / Status flags



- Condition code flags (Status flags) of the CPU
- Condition code flags are the **bits of a special register (i.e., status register)** within the processor. They are affected by the most recent ALU operations
- **N (negative) or S (sign) flag** (i.e., bit of the status register) is Set to 1 if the result of most recent arithmetic operation is negative
  - Is used by some instructions, such as: **Branch<0 LOOP**
- **Z (zero) flag:** is **set** if the result of the most recent arithmetic operation is ZERO
  - Used by some instructions, like: **Branch==0 LABEL**
- **C (carry) flag** is set if a carry out from most recent operation
- **V (Overflow flag)** is set if Overflow occurs in most recent op.
- Different instructions affect different flags

Example: How Condition Codes or Status Flags Set/Reset

- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0

A = -16; B=20
So, A – B = -36

16 = 0001 0000

-16 = 2's complement of 16
= **1111 0000**

Subtract B ➔ Add (–B) = Add 2's complement of B
= 1110 1100

A: **1 1 1 1 0 0 0 0**

+(–B): **1 1 1 0 1 1 0 0**

1 1 1 0 1 1 1 0 0

C = 1

Z = 0

S = 1

V = 0

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]

Addressing modes

- Different ways in which the address of an operand is specified in an instruction is referred to as addressing modes.
- Register mode
  - Operand is the contents of a processor register.
  - Address of the register (its Name) is given in the instruction.
  - E.g. *Clear R1* *or* *Move R1, R2*
- Absolute mode
  - Operand is in a memory location.
  - Address of the memory location is given explicitly in the instruction.
  - E.g. *Clear A* *or* *Move LOC, R2*
  - Also called as "Direct mode" in some assembly languages
- Register and absolute modes can be used to represent ***variables***

Addressing modes (contd..)

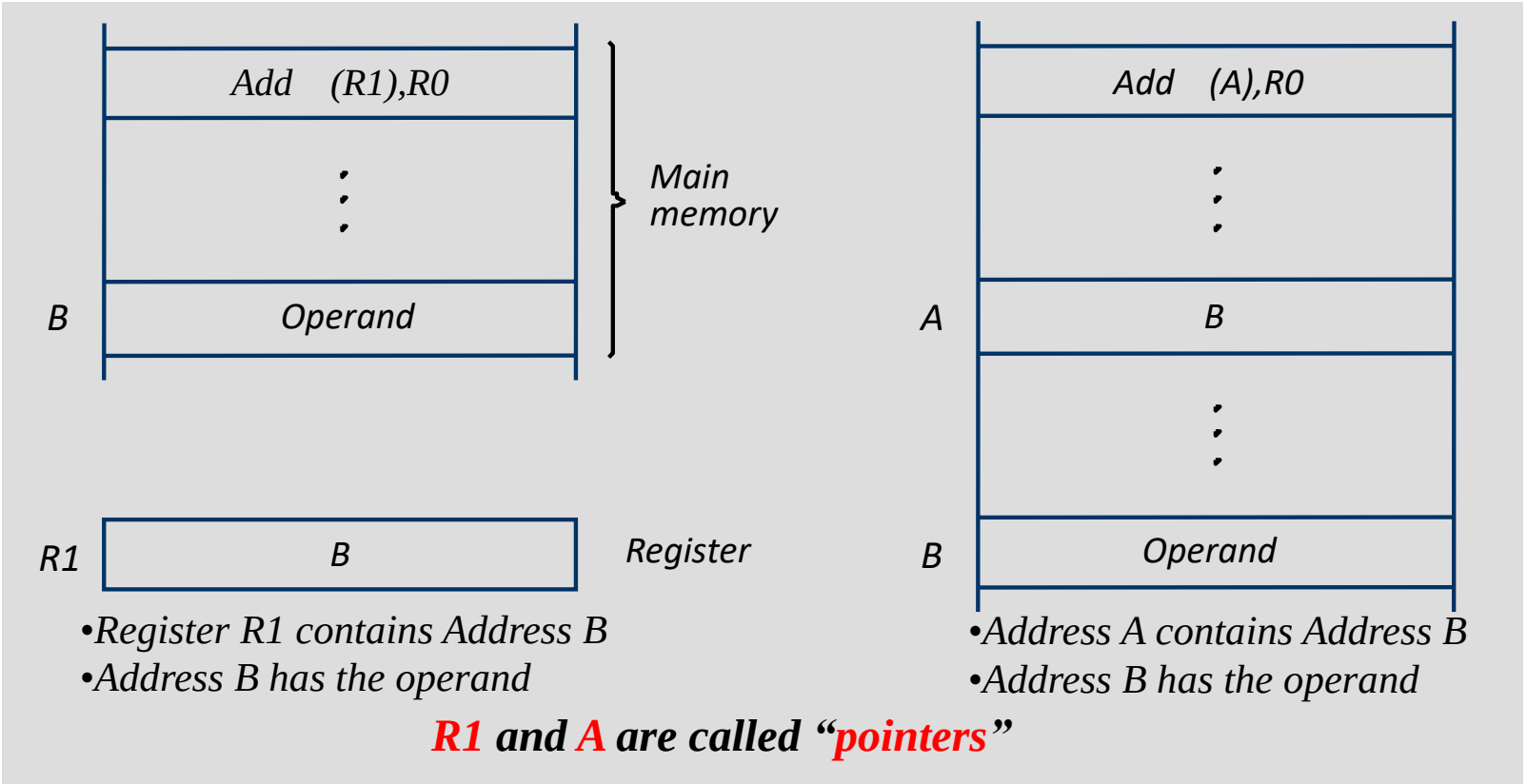
- Immediate mode
  - **Operand** is given explicitly in the instruction.
  - E.g. *Move #200, R0*
  - Can be used to represent ***constants***.
- Register, Absolute and Immediate modes contained either the address of the operand or the operand itself.
- Some instructions provide information from which the memory address of the operand can be determined
  - That is, they provide the "Effective Address" of the operand.
  - They do not provide the operand or the address of the operand explicitly.
- ***Different ways in which "Effective Address" of the operand can be generated.***

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

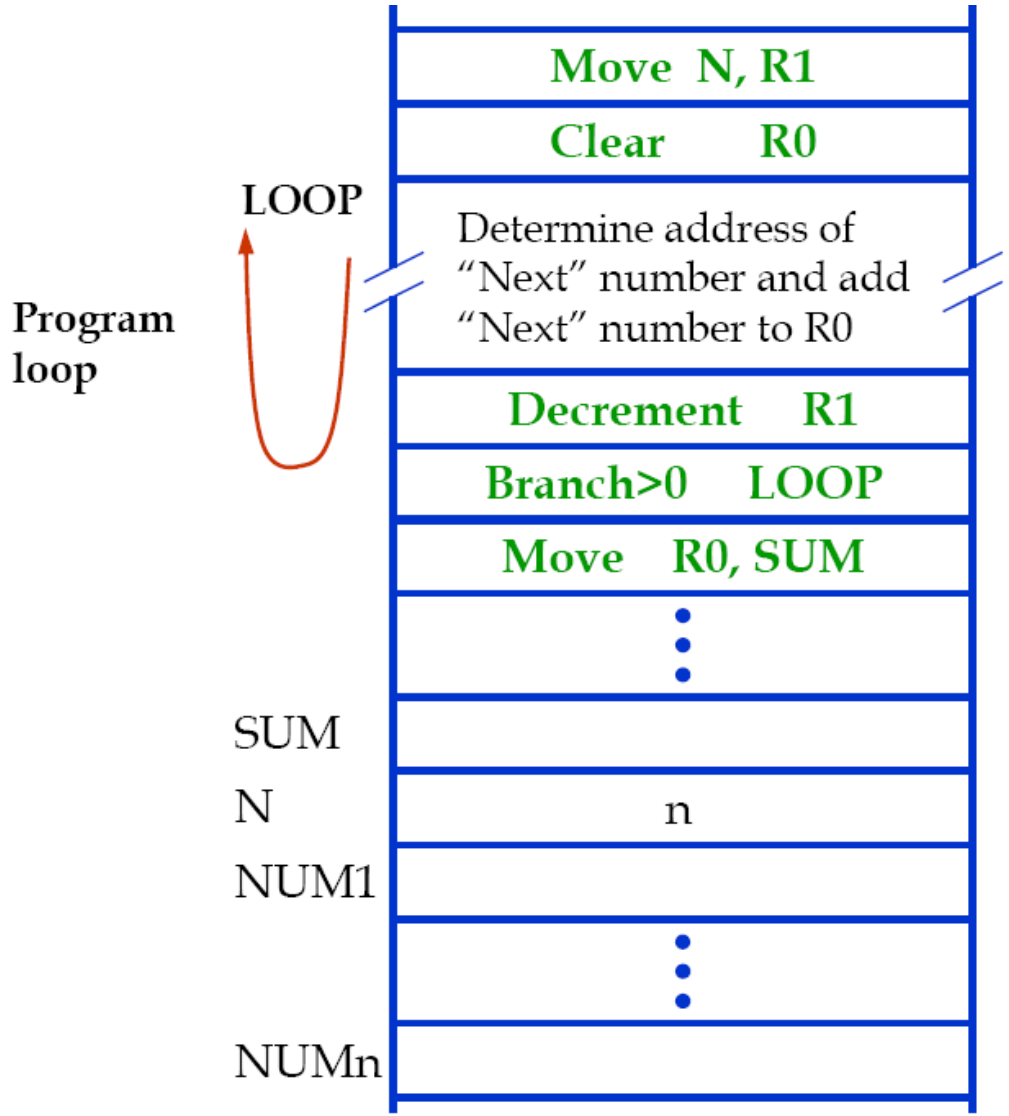
Addressing modes (contd..)

Effective Address of the operand is the contents of a register or a memory location whose address appears in the instruction.



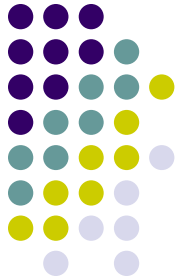
This is called as "Indirect Mode"

Using Indirect Addressing in a Program



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

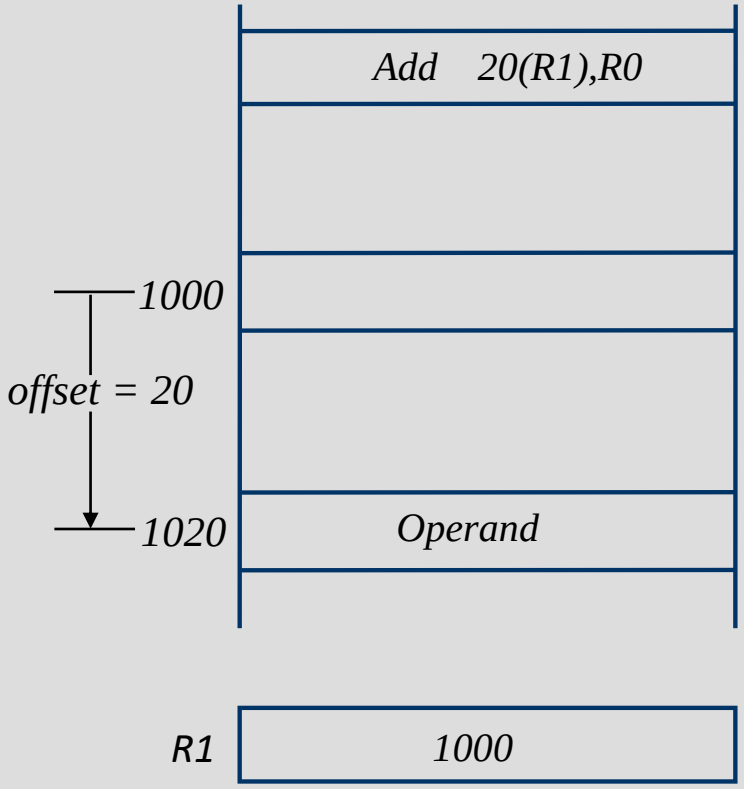
- Index mode: the effective address of the operand is generated by adding a constant value to the contents of a register
  - The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general purpose registers in the processor.
  - It is referred to as an *index register*

Indexing and Arrays

- 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

Addressing modes (contd..)

Effective Address of the operand is generated by adding a constant value to the contents of the register



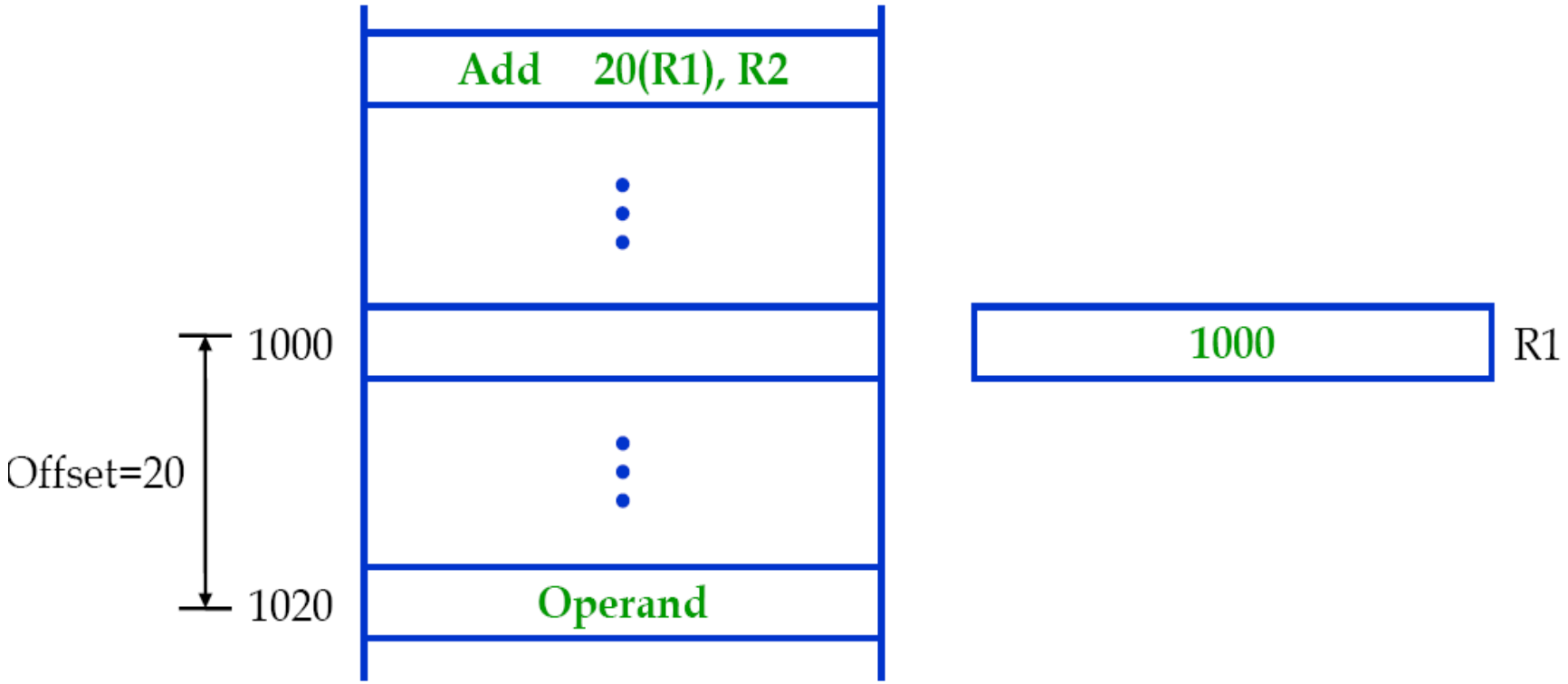
- *Operand is at address 1020*
- *Register R1 contains 1000*
- *Offset 20 is added to the contents of R1 to generate the address 1020*
- *Contents of R1 do not change in the process of generating the address*
- *R1 is called as an "index register"*

What address would be generated by Add 1000(R1), R0 if R1 had 20?

This is the "Indexing Mode"

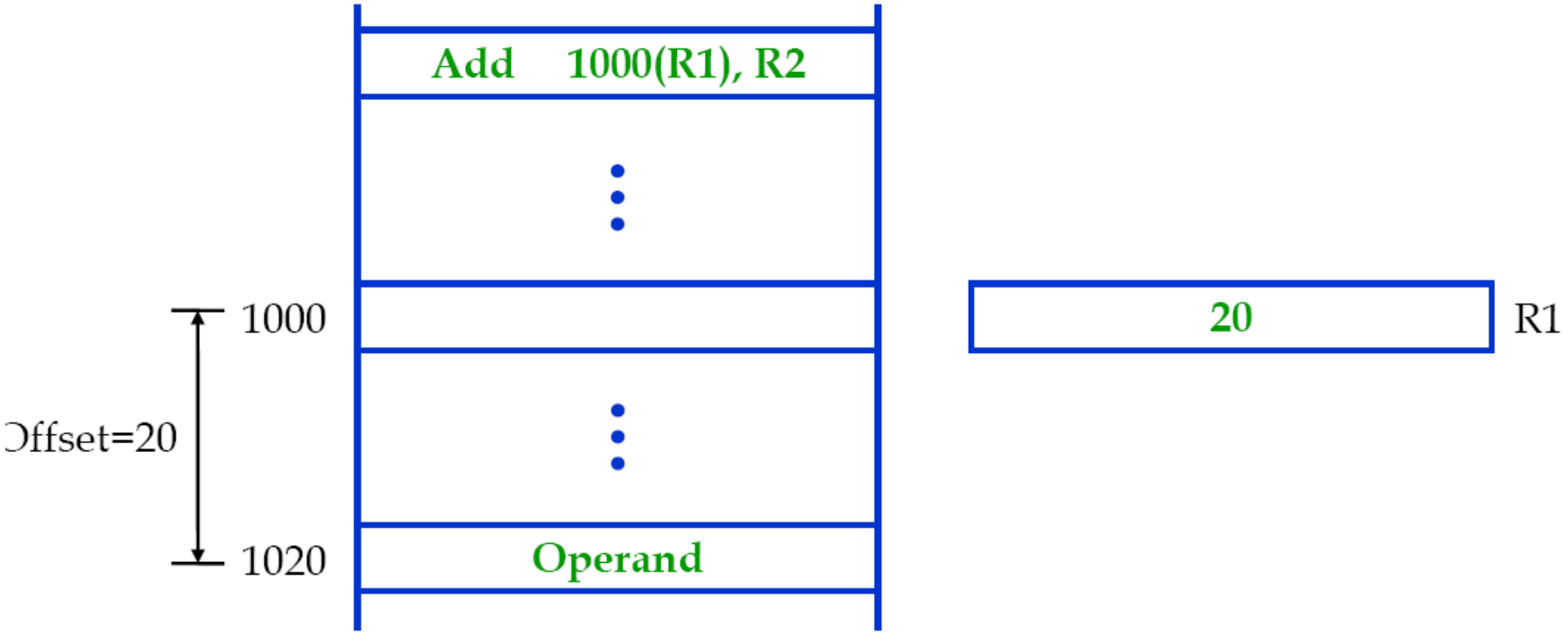
Indexed Addressing

Offset is given as a constant

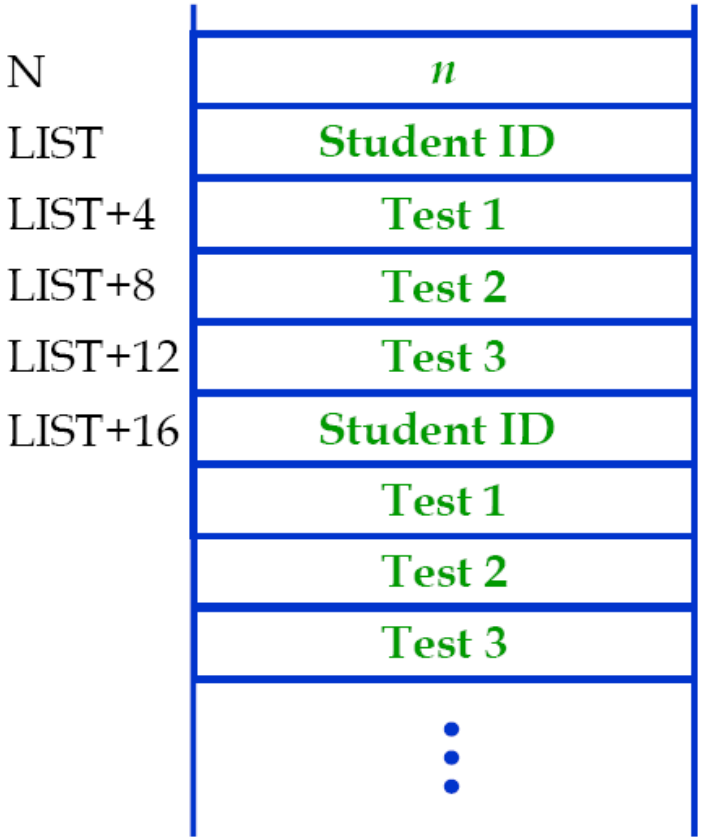


Indexed Addressing

Offset is in the index register



An Example for Indexed Addressing

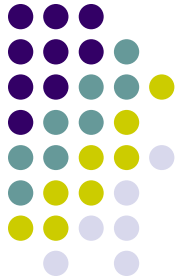



```
        Move        #LIST, R0
        Clear       R1
        Clear       R2
        Clear       R3
        Move        N, R4
LOOP    Add         4(R0), R1
        Add         8(R0), R2
        Add         12(R0), R3
        Add         #16, R0
        Decrement   R4
        Branch>0    LOOP
    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

Relative mode



- Effective Address of the operand is generated by adding a constant value to the contents of the Program Counter (PC).
- Variation of the Indexing Mode, where the index register is the PC instead of a general purpose register.
- When the instruction is being executed, the PC holds the address of the next instruction in the program.
- Useful for specifying target addresses in branch instructions.
- Addressed location is “relative” to the PC, this is called “Relative Mode”

Addressing Modes (contd..)

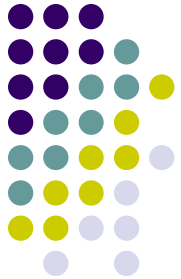
- **Relative mode:**
- The Instruction Branch > 0 Loop
- Suppose that the loop starts at address 1000, and the branch instruction at address 1012.
- The PC value now is 1016.
- To branch to location Loop (1000), the offset value is 1000 – 1016 = -16
- When the assembler processes such instruction, it computes the required offset value, and generates the corresponding machine instruction using the addressing mode:

-16(PC)

Addressing Modes (contd..)

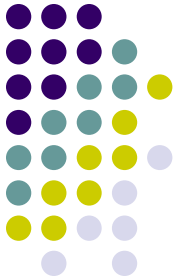
- Autoincrement mode:
  - Effective address of the operand is the contents of a register specified in the instruction.
  - **After** accessing the operand, the contents of this register are automatically incremented to point to the next consecutive memory location.
  - ***(R1)+***
- Autodecrement mode
  - Effective address of the operand is the contents of a register specified in the instruction.
  - **Before** accessing the operand, the contents of this register are automatically decremented to point to the previous consecutive memory location.
  - ***-(R1)***
- Autoincrement and Autodecrement modes are useful for implementing "Last-In-First-Out" data structures.

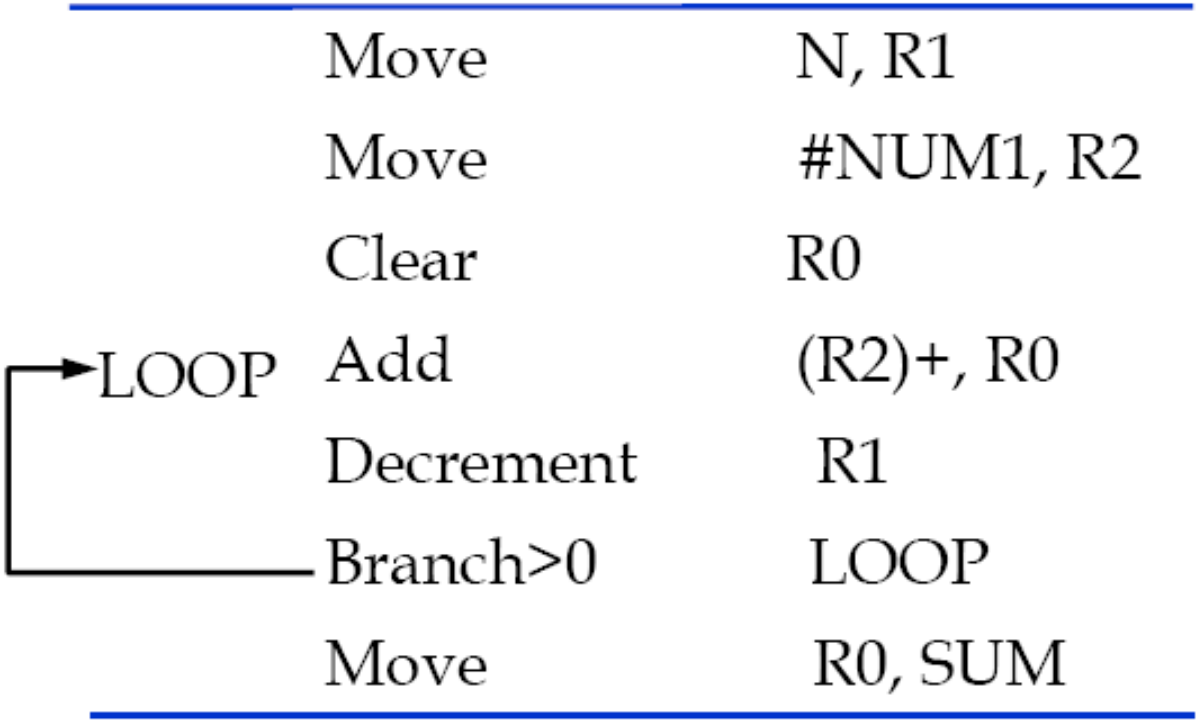
Addressing modes (contd..)



- Implicitly the increment and decrement amounts are 1.
  - This would allow us to access individual bytes in a byte addressable memory.
- Recall that the information is stored and retrieved one word at a time.
  - In most computers, increment and decrement amounts are equal to the word size in bytes.
- E.g., if the word size is 4 bytes (32 bits):
  - Autoincrement increments the contents by 4.
  - Autodecrement decrements the contents by 4.

An Example of Autoincrement Addressing





Book Examples (Fig. 2.33): Computing Dot Product of two vectors ... (1)

2.11.1 Vector Dot Product Program

The first example is a numerical application that is an extension of the loop program of Figure 2.16 for adding numbers. In calculations that involve vectors and matrices, it is often necessary to compute the dot product of two vectors. Let A and B be two vectors of length *n*. Their dot product is defined as

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

Book Examples (Fig. 2.33): Computing Dot Product of two vectors ... (2)

```
        Move        #AVEC,R1        R1 points to vector A.
        Move        #BVEC,R2        R2 points to vector B.
        Move        N,R3            R3 serves as a counter.
        Clear       R0              R0 accumulates the dot product.
LOOP    Move        (R1)+,R4        Compute the product of
        Multiply    (R2)+,R4          next components.
        Add         R4,R0           Add to previous sum.
        Decrement   R3              Decrement the counter.
        Branch>0    LOOP            Loop again if not done.
        Move        R0,DOTPROD      Store dot product in memory.
```

Figure 2.33 A program for computing the dot product of two vectors.