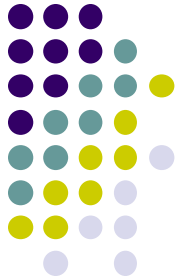
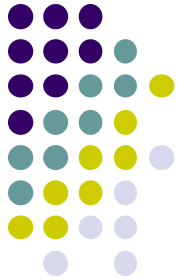
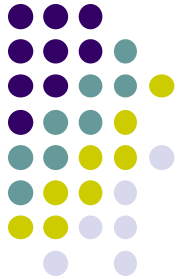
Instruction Set Architecture



Memory Locations, Addresses, and Operations





- Machine instructions and program execution
- Addressing methods for accessing register and memory operands

Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.

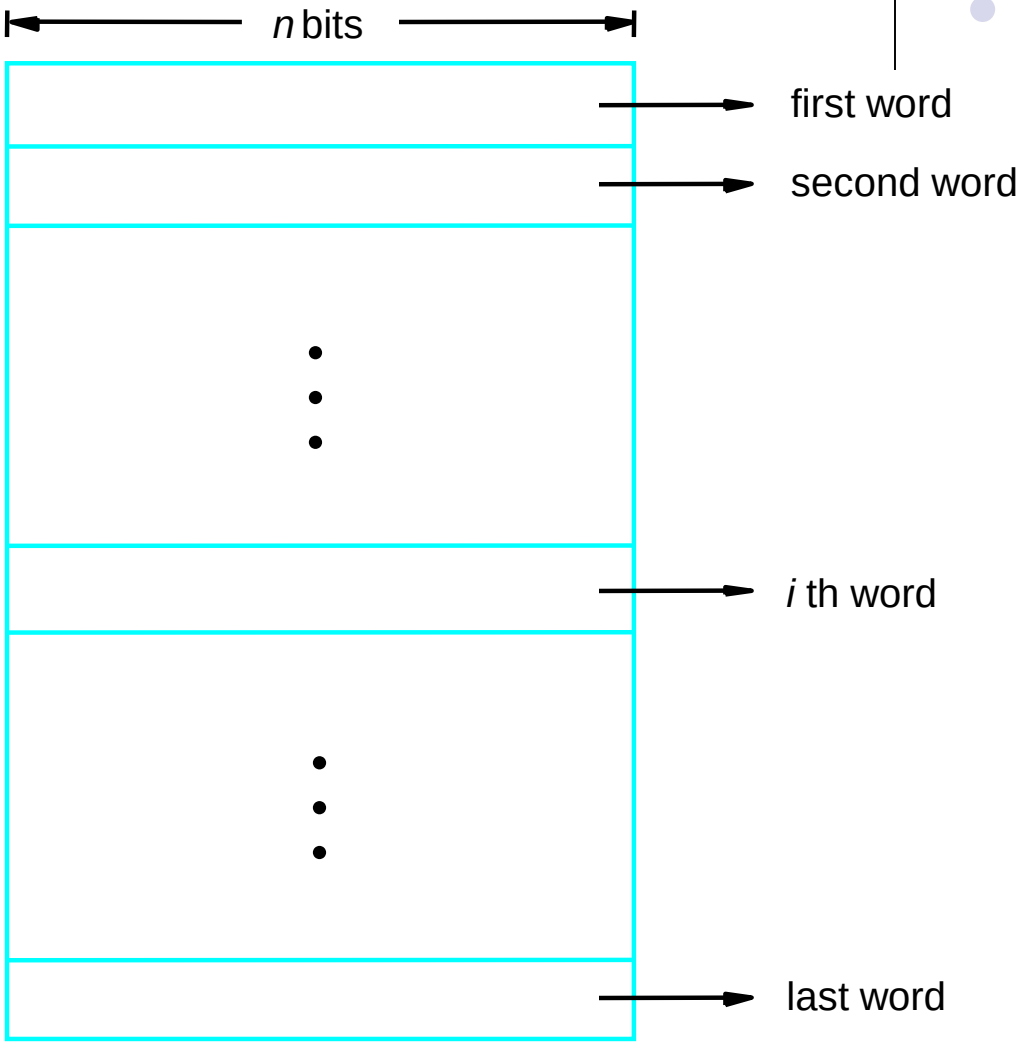
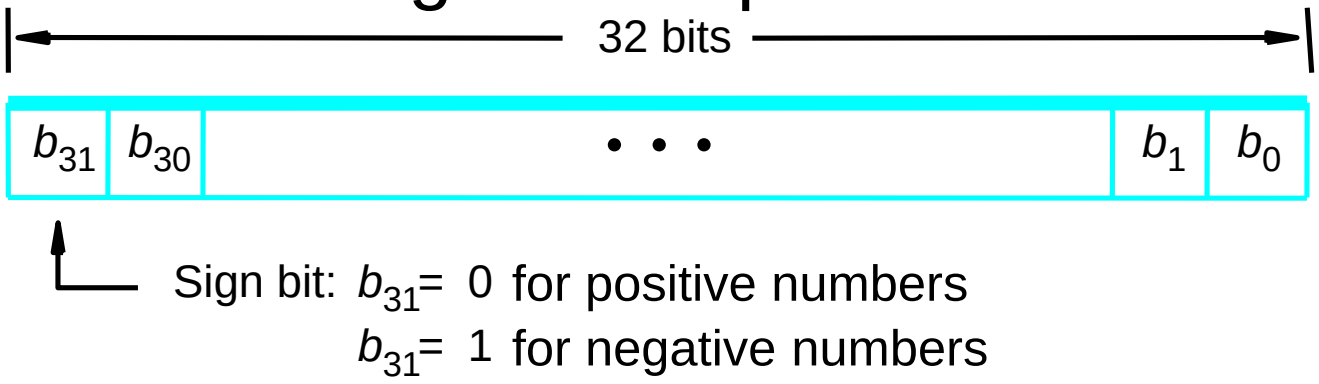


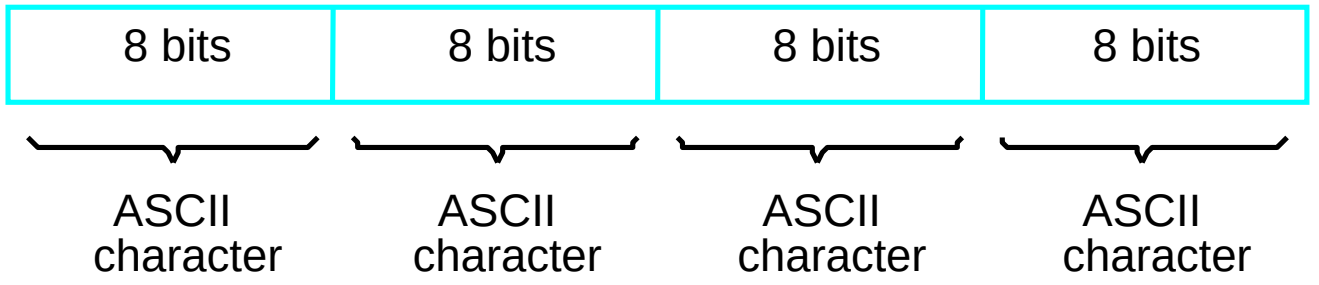
Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example

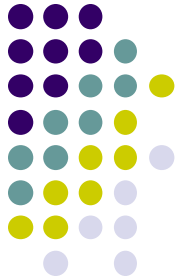


(a) A signed integer



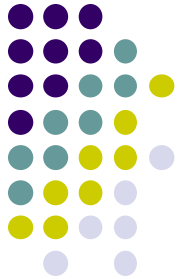
(b) Four characters

Memory Location, Addresses, and Operation



- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A *k*-bit address memory has 2^k memory locations, namely 0 – 2^k-1, called memory space.
- 24-bit memory: 2^{24} = 16,777,216 = 16M (1M=2^{20})
- 32-bit memory: 2^{32} = 4G (1G=2^{30})
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}

Byte Addressability



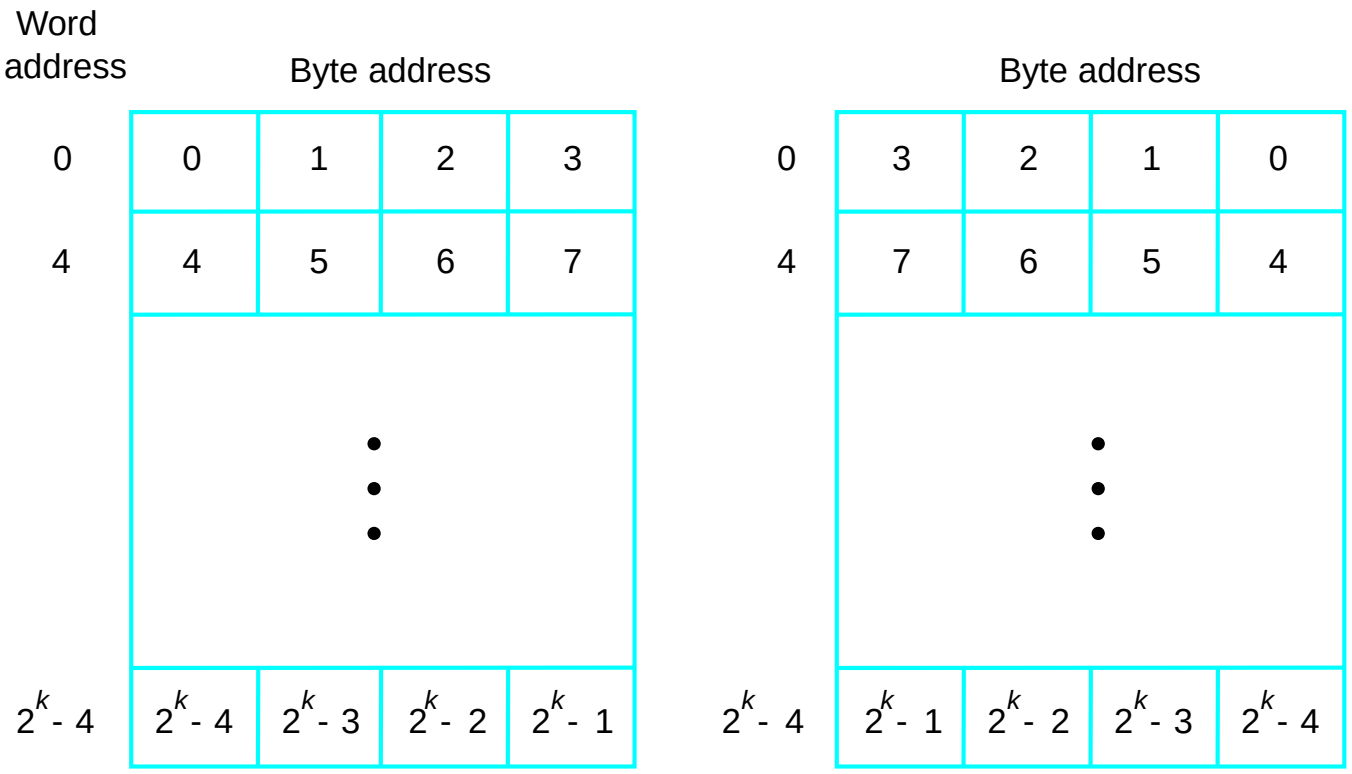
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits,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

Big-Endian and Little-Endian Assignments

two ways that byte addresses can be assigned across words

Big-Endian: lower byte addresses are used for the most significant bytes of the word

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

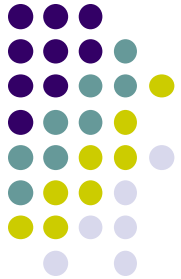


(a) Big-endian assignment

(b) Little-endian assignment

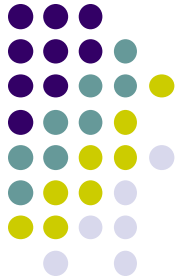
Figure 2.7. Byte and word addressing.

Word alignment



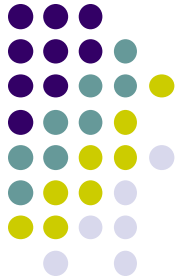
- Word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word.
  - 16-bit word: word addresses: 0, 2, 4,….
  - 32-bit word: word addresses: 0, 4, 8,….
  - 64-bit word: word addresses: 0, 8,16,….
- If words begin at arbitrary byte address, words are said to have *unaligned* addresses.

Accessing Numbers and Characters



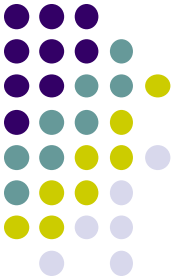
- A number usually occupies one word, and can be accessed in the memory by specifying its word address.
- Individual characters can be accessed by their byte address.

Memory Operation

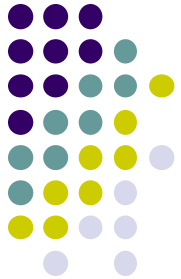


- Load (or Read or Fetch)
  - Copy the content. The memory content doesn’t change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

Instruction and Instruction Sequencing

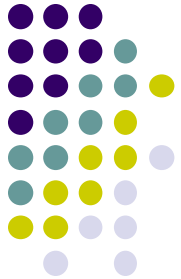


“Must-Perform” Operations



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

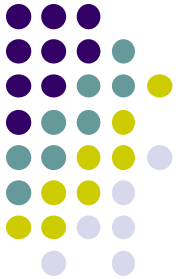
Register Transfer Notation



- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location
  - e.g. R1←[LOC]

    R3 ←[R1]+[R2]
- Register Transfer Notation (RTN)

Assembly Language Notation

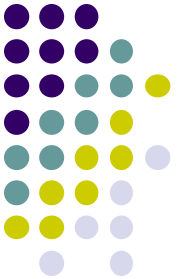


- Represent machine instructions and programs.

Move R1,LOC ➔ R1←[LOC]

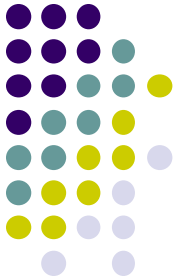
Add R3, R1, R2 ➔ R3 ←[R1]+[R2]

RISC and CISC Instruction Set
RISC



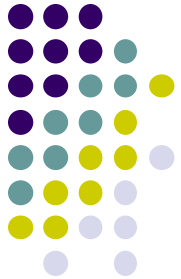
- most important characteristics that distinguish different computers is the nature of their instructions
- higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers
- various operations needed to process a sequence of instructions are performed in “pipelined” fashion
- number of different types of instructions may be included in the instruction set of a computer.
- Such computers are called *Reduced Instruction Set Computers* (RISC).

CISC



- An altenative to RISC is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations.
- computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

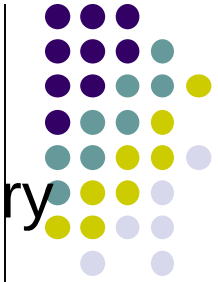
Two key characteristics of RISC instruction sets are



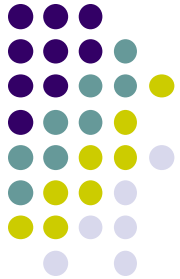
- Each instruction fits in a single word.
- A load/store architecture is used, in which

  – 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.



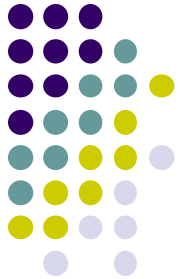
- At the start of execution of a program, all instructions and data used in the program are stored in the memory of a computer.
- Processor registers do not contain valid operands at that time.
- If operands are expected to be in processor registers before they can be used by an instruction, then it is necessary to first bring these operands into the registers.
- 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
- Load processor_register, memory_location



C = A + B

C ← [A] + [B]

- Load R2, A
- Load R3, B
- Add R4, R2, R3
- Store R4, C



- Three-Address Instructions

Add destination, source1, source2

- The Store instruction is of the form

Store source, destination

Instruction Execution and Straight-Line Sequencing

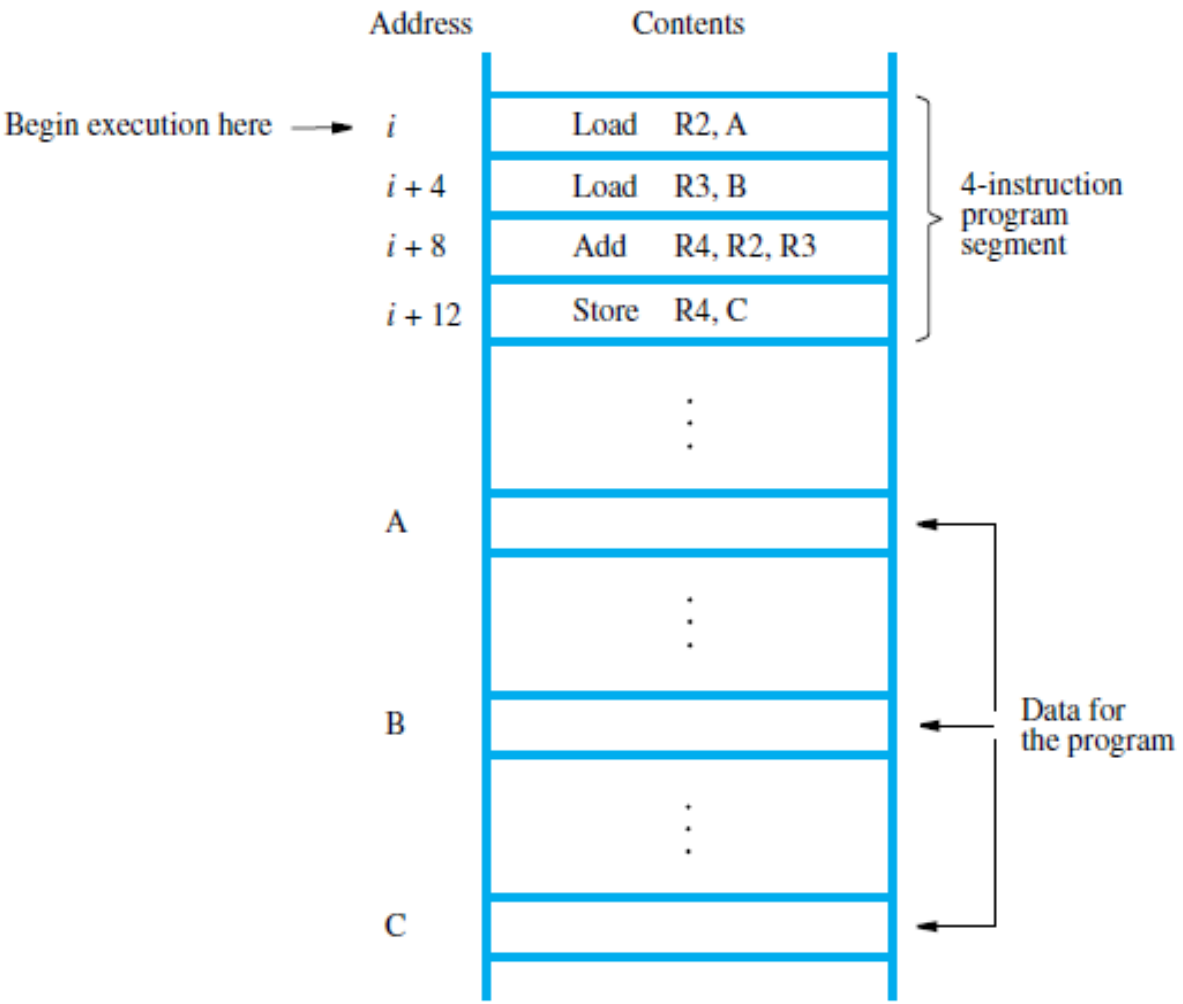
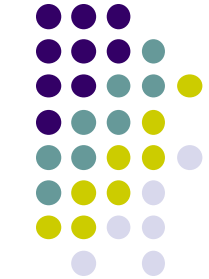


Figure 2.4 A program for C ← [A] + [B].

Branching



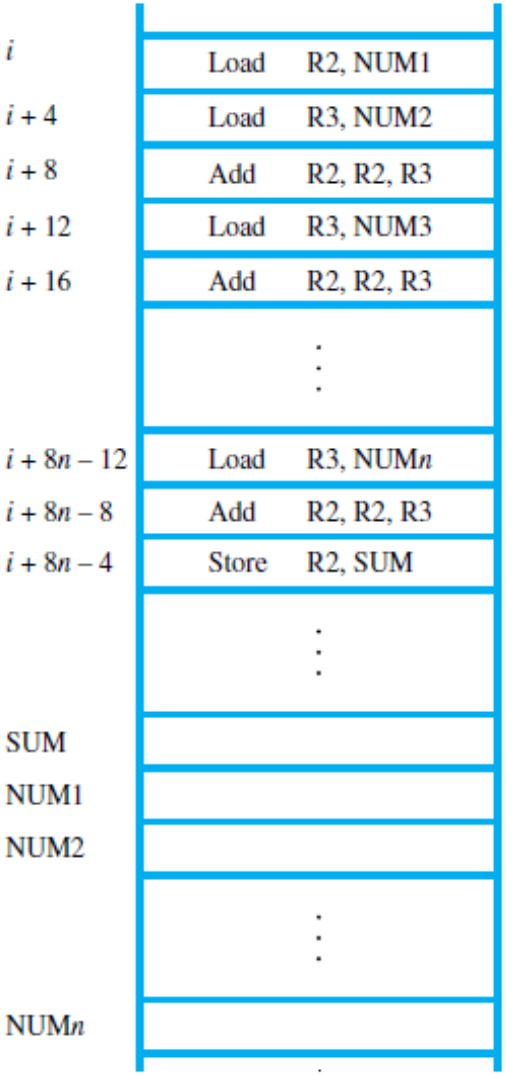


Figure 2.5 A program for adding *n* numbers.

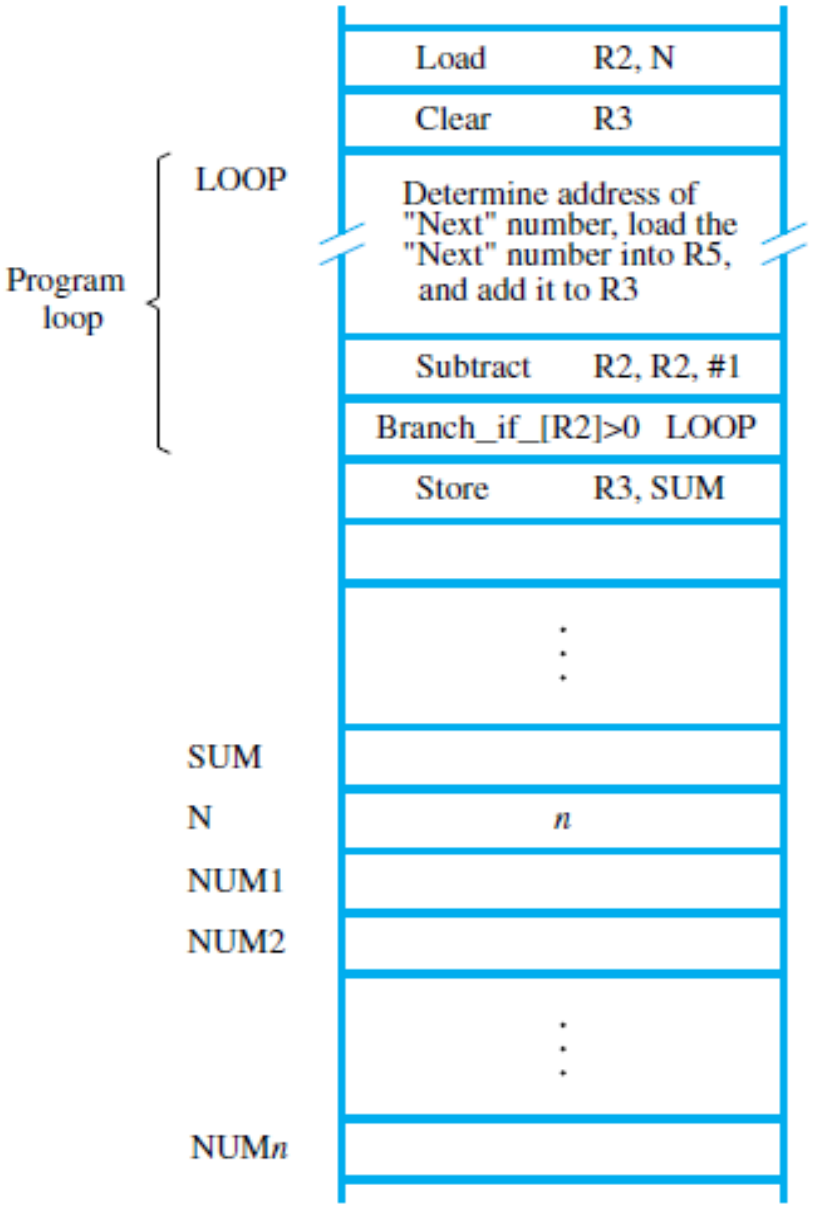
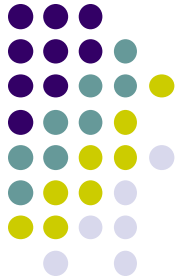
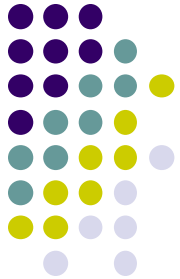


Figure 2.6 Using a loop to add *n* numbers.



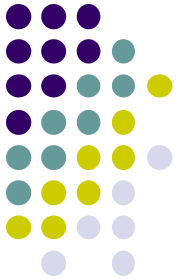
Branch_if_[R4]>[R5] LOOP
may be written in generic assembly language as
Branch_greater_than R4, R5, LOOP
or using an actual mnemonic as
BGT R4, R5, LOOP

Generating Memory Addresses

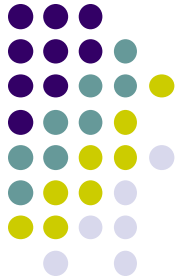


- The purpose of the instruction block starting at LOOP is to add successive numbers from the list during each pass through the loop
- must refer to a different address during each pass
- memory operand address cannot be given directly in a single Load instruction in the loop.

Addressing Modes

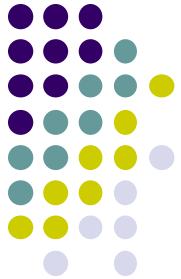


Generating Memory Addresses



- How to specify the address of branch target?
- Can we give the memory operand address directly in a single Add instruction in the loop?
- Use a register to hold the address of NUM1; then increment by 4 on each pass through the loop.

Implementation of Variables and Constants

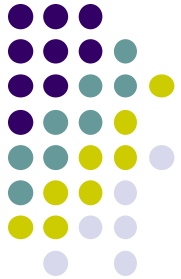


- *Register mode*—The operand is the contents of a processor register; the name of the register is given in the instruction.

e.g. Add R4, R2, R3

- *Absolute mode*—The operand is in a memory location; the address of this location is given explicitly in the instruction

e.g. Load R2, NUM1



- *Immediate mode*—The operand is given explicitly in the instruction.

e.g. Add R4, R6, $200_{immediate}$

Add R4, R6, #200

Indirection and Pointers

- *Indirect mode*—The effective address of the operand is the contents of a register that is specified in the instruction
- e.g. Load R2, (R5)

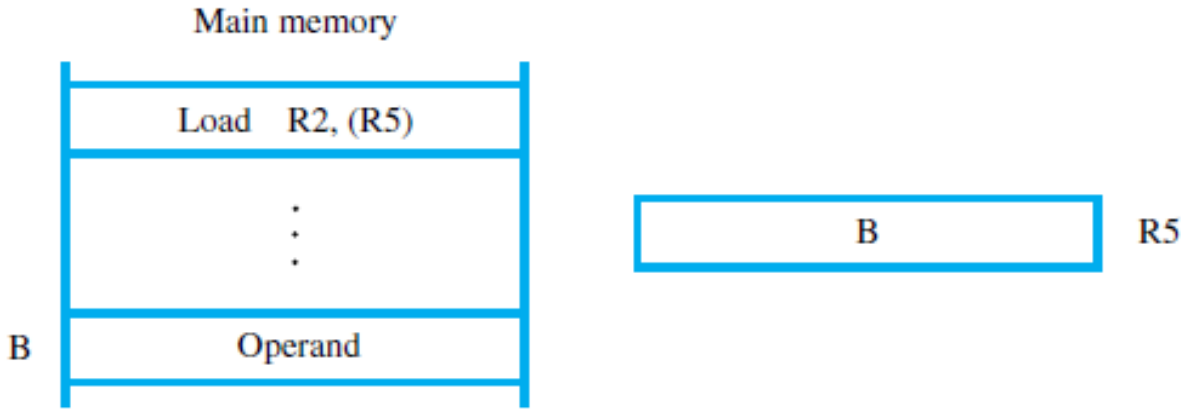
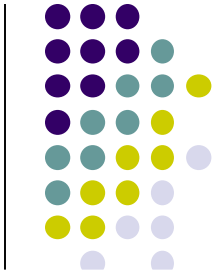


Figure 2.7 Register indirect addressing.



```
        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.
```

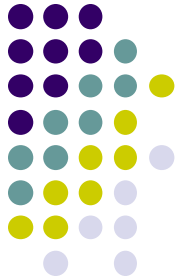
Figure 2.8 Use of indirect addressing in the program of Figure 2.6.

Another Example

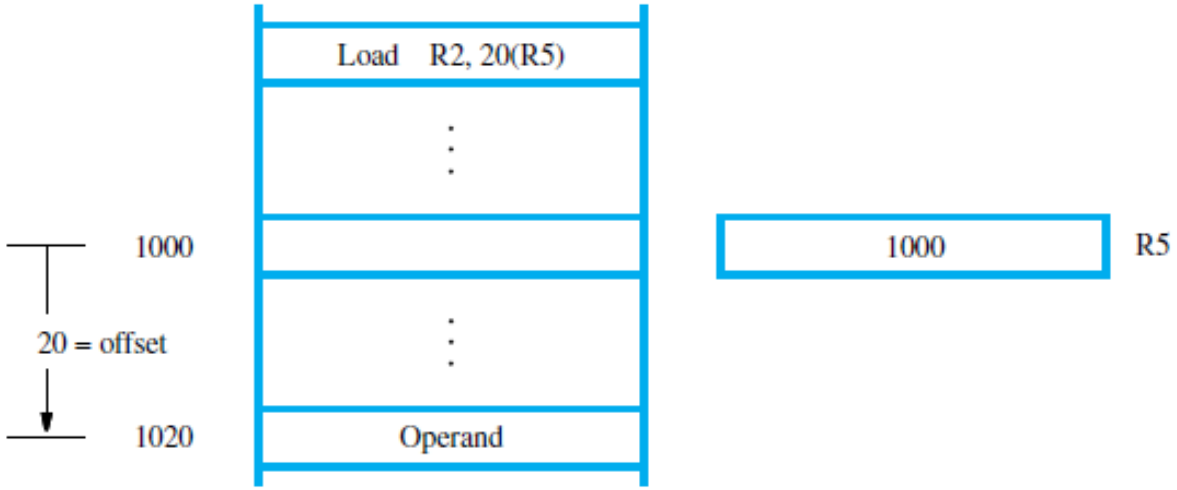
```
A = *B;
```

```
Load    R2, B
Load    R3, (R2)
Store   R3, A
```

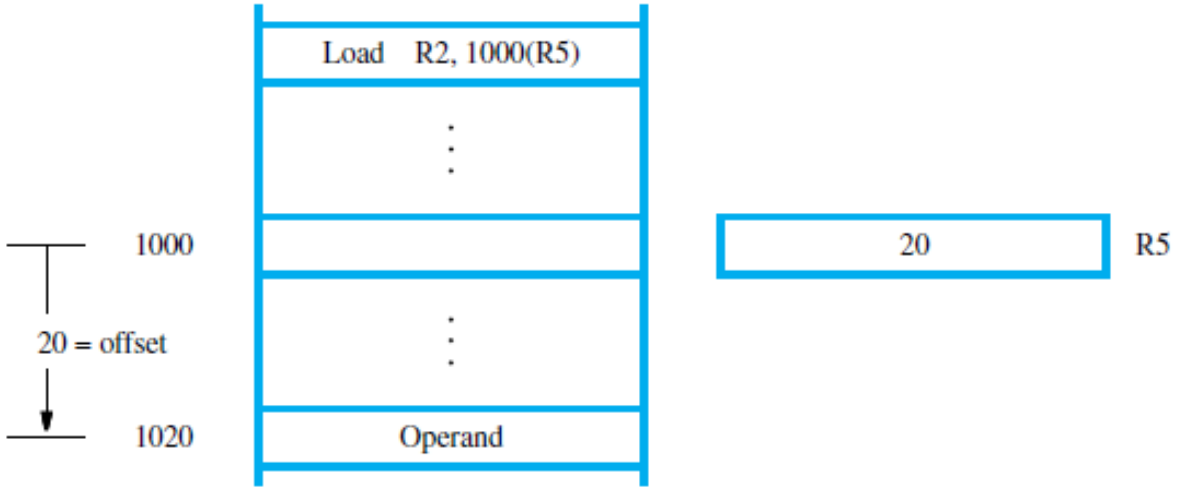
Indexing and Arrays



- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register
- $X(R_i)$: $EA = X + [R_i]$
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.



(a) Offset is given as a constant



(b) Offset is in the index register

Example

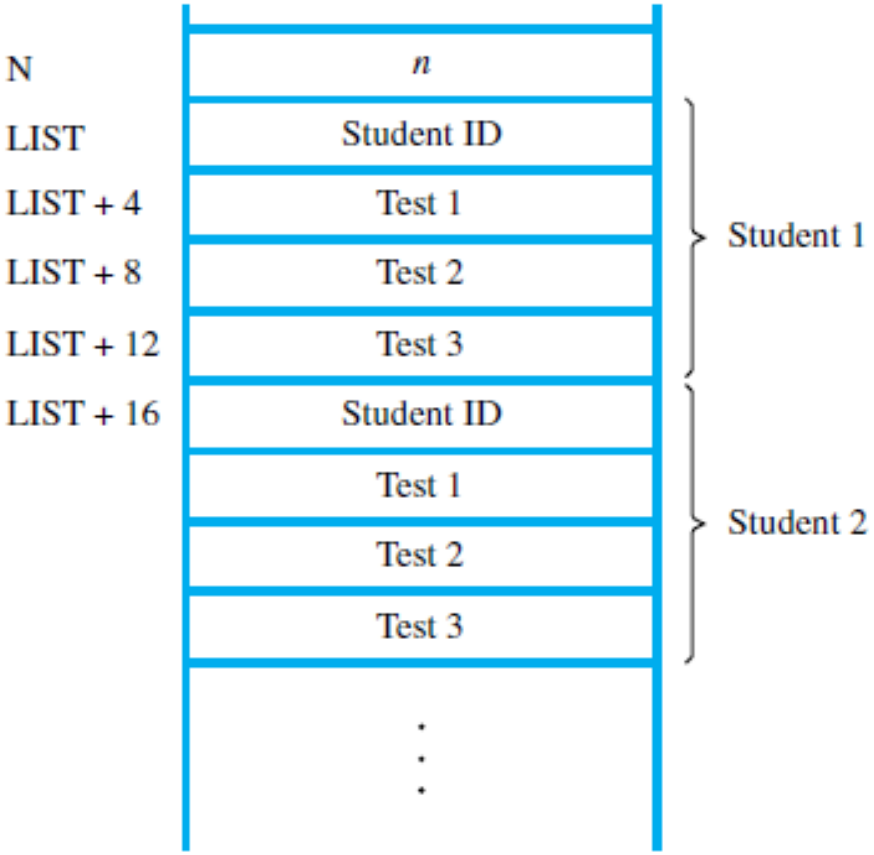
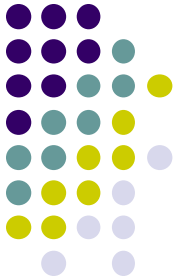


Figure 2.10 A list of students' marks.

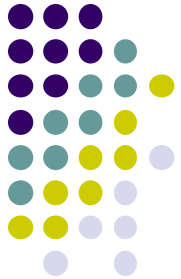
Example Contd…



	Move	R2, #LIST	Get the address LIST.
	Clear	R3	
	Clear	R4	
	Clear	R5	
	Load	R6, N	Load the value *n*.
LOOP:	Load	R7, 4(R2)	Add the mark for next student's
	Add	R3, R3, R7	Test 1 to the partial sum.
	Load	R7, 8(R2)	Add the mark for that student's
	Add	R4, R4, R7	Test 2 to the partial sum.
	Load	R7, 12(R2)	Add the mark for that student's
	Add	R5, R5, R7	Test 3 to the partial sum.
	Add	R2, R2, #16	Increment the pointer.
	Subtract	R6, R6, #1	Decrement the counter.
	Branch_if_[R6]>0	LOOP	Branch back if not finished.
	Store	R3, SUM1	Store the total for Test 1.
	Store	R4, SUM2	Store the total for Test 2.
	Store	R5, SUM3	Store the total for Test 3.

Figure 2.11 Indexed addressing used in accessing test scores in the list in Figure 2.10.

Indexing and Arrays



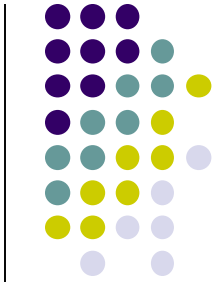
- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
- Several variations:
  (R_i, R_j): $EA = [R_i] + [R_j]$
  $X(R_i, R_j)$: $EA = X + [R_i] + [R_j]$

Table 2.1 RISC-type addressing modes.

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

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

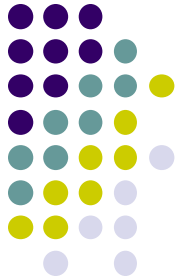
RISC style



RISC style is characterized by

- Simple addressing modes
- All instructions fit in a single word
- Fewer instructions in the instruction set, as a consequence of simple addressing modes
- Arithmetic and logic operations that can be performed only on operands in processor registers
- Load/store architecture does not allow direct transfers from one memory location to another; such transfers must take place via a processor register
- Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining
- Programs that tend to be larger in size, because more, but simpler instructions are needed to perform complex tasks

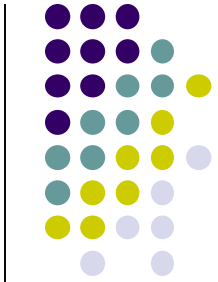
CISC style



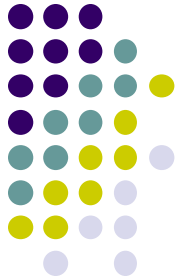
CISC style is characterized by:

- More complex addressing modes
- More complex instructions, where an instruction may span multiple words
- Many instructions that implement complex tasks
- Arithmetic and logic operations that can be performed on memory operands as well as operands in processor registers
- Transfers from one memory location to another by using a single Move instruction
- Programs that tend to be smaller in size, because fewer, but more complex instructions are needed to perform complex tasks

CISC Instruction Sets



- CISC instruction sets are not constrained to the *load/store architecture*, in which arithmetic and logic operations can be performed only on operands that are in processor registers.
- Instructions do not necessarily have to fit into a single word.
- Some instructions may occupy a single word, but others may span multiple words



- Most arithmetic and logic instructions use the *two-address* format

  Operation destination, source

- An Add instruction of this type is
- Add B, A
- B ← [A] + [B]

- Consider again the task of adding two numbers
- C = A + B
- where all three operands may be in memory locations
- Cannot be done with a single two-address instruction.
- Can be performed by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is
- Move C, B
- which performs the operation C←[B], leaving the contents of location B unchanged.
- The operation C←[A] + [B] can now be performed by the two-instruction sequence
- Move C, B
- Add C, A

- In some CISC processors one operand may be in the memory but the other must be in a register.
- In this case, the instruction sequence for the required task would be
- Move R*i*, A
- Add R*i*, B
- Move C, R*i*
- The general form of the Move instruction is
- Move destination, source

where both the source and destination may be either a memory location or a processor register

Additional Addressing Modes

- ***Autoincrement mode***—The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next operand in memory

  *(*R*i)*+

- Computers that have the Autoincrement mode automatically increment the contents of the register by a value that corresponds to the size of the accessed operand.
- Increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

- *Autodecrement mode*—The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand

$$-(Ri)$$

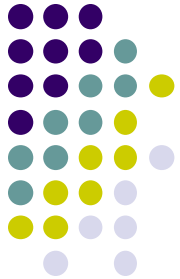
Subtract SP, #4

Move (SP), NEWITEM

to push a new item on the stack, we can use just one instruction

Move −(SP), NEWITEM

Relative Addressing

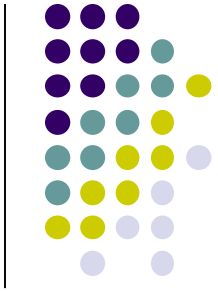


- Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
- X(PC) – note that X is a signed number
- Branch>0 LOOP
- This location is computed by specifying it as an offset from the current value of PC.
- Branch target may be either before or after the branch instruction, the offset is given as a singed number

Condition Codes

- Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero
- Maintain the information about these results for use by subsequent conditional branch instructions
- Accomplished by recording the required information in individual bits, often called ***condition code*** *flags*
- These flags are usually grouped together in a special processor register called the ***condition code register*** **or** ***status register*****.**
- Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed

Commonly used flags



N (negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z (zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
C (carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

e.g. Branch>0 LOOP

This instruction causes a branch if neither N nor Z is 1, that is, if the result produced by the Subtract instruction is neither negative nor equal to zero

```
        Move        R2, N          Load the size of the list.
        Clear       R3             Initialize sum to 0.
        Move        R4, #NUM1      Load address of the first number.
LOOP:   Add         R3, (R4)+      Add the next number to sum.
        Subtract    R2, #1         Decrement the counter.
        Branch>0    LOOP           Loop back if not finished.
        Move        SUM, R3        Store the final sum.
```

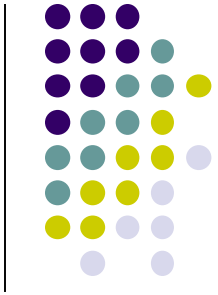
Figure 2.26 A CISC version of the program of Figure 2.8.

Example Programs
Vector Dot Product Program

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

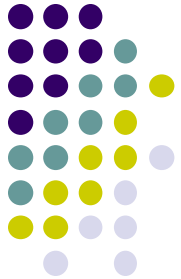
	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Load	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Load	R6, (R2)	Get next element of vector A.
	Load	R7, (R3)	Get next element of vector B.
	Multiply	R8, R6, R7	Compute the product of next pair.
	Add	R5, R5, R8	Add to previous sum.
	Add	R2, R2, #4	Increment pointer to vector A.
	Add	R3, R3, #4	Increment pointer to vector B.
	Subtract	R4, R4, #1	Decrement the counter.
	Branch_if_[R4]>0	LOOP	Loop again if not done.
	Store	R5, DOTPROD	Store dot product in memory.

Figure 2.27 A RISC-style program for computing the dot product of two vectors.



	Move	R2, #AVEC	R2 points to vector A.
	Move	R3, #BVEC	R3 points to vector B.
	Move	R4, N	R4 serves as a counter.
	Clear	R5	R5 accumulates the dot product.
LOOP:	Move	R6, (R2)+	Compute the product of
	Multiply	R6, (R3)+	next components.
	Add	R5, R6	Add to previous sum.
	Subtract	R4, #1	Decrement the counter.
	Branch>0	LOOP	Loop again if not done.
	Move	DOTPROD, R5	Store dot product in memory.

Figure 2.28 A CISC-style program for computing the dot product of two vectors.



- Go through examples given in Section 2.12 and 2.15 of the Reference Book

Reference

Carl Hamacher, ZvonkoVranesic and SafwatZaky, “Computer Organization and Embedded Systems”, Sixth edition, McGraw Hill Publication, 2012.