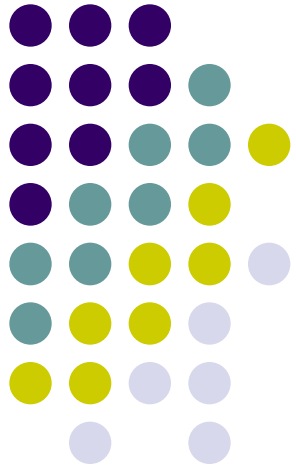
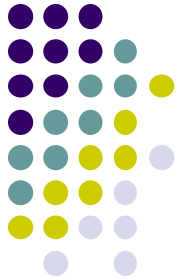
Chapter 2
Machine Instruction

陈俊颖

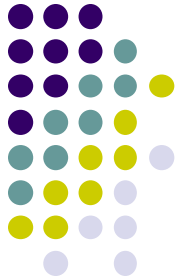


Contents



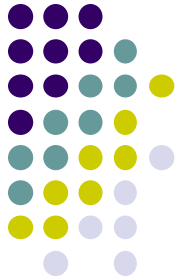
- 2.1~2.3 Instruction and Instruction Sequencing
- E.4 Instruction Formats
- 2.4 Addressing Modes
- 2.6 Stacks
- 2.7 Subroutines
- 2.8 Additional Instructions
- 2.10 CISC Instruction Sets
- 2.11 RISC and CISC Styles

2.1~2.3 Instruction and Instruction Sequencing



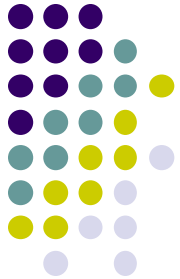
- Register Transfer Notation
- Assembly-Language Notation
- RISC and CISC Instruction Sets
- Introduction to RISC Instruction Sets
- Instruction Execution and Straight-Line Sequencing
- Branching

2.1~2.3 Instruction and Instruction Sequencing



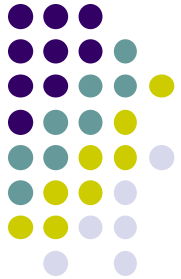
- Four Types of Instructions
  - Data transfers between the memory and the processor registers
  - Arithmetic and logic operations on data
  - Program sequencing and control
  - I/O transfers

Register Transfer Notation



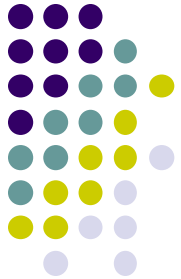
- Register transfer notation is used to describe hardware-level data transfers and operations
  - Processor register: R0, R5
  - I/O register: DATAIN, OUTSTATUS
  - Memory location: LOC, PLACE, A, VAR2
- Use […] to denote contents of a location
  Use ← to denote transfer to a destination

Register Transfer Notation



- Example1: R2 ← [LOC]
  - Transfer from LOC in memory to register R2
  - Right-hand expression always denotes a value, left-hand side always names a location
- Example2: R4 ← [R2] + [R3]
  - Add the contents of registers R2 and R3, place the sum in register R4

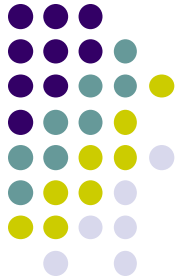
Assembly-Language Notation



- RTN shows data transfers and arithmetic
- Another notation needed to represent machine instructions & programs using them
- Assembly language is used for this purpose
- For the two preceding examples using RTN, the assembly-language instructions are:

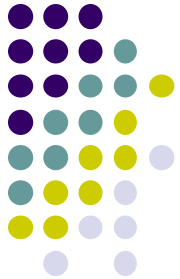
```
Load    R2, LOC
Add     R4, R2, R3
```

Assembly-Language Notation



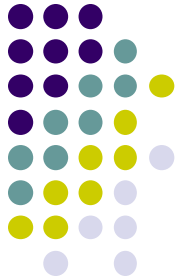
- An *instruction* specifies the desired operation and the operands that are involved
- Examples in this chapter will use English words for the operations (e.g., Load, Store, and Add)
- Commercial processors use mnemonics, usually abbreviations (e.g., LD, ST, and ADD)
- Mnemonics differ from processor to processor

RISC and CISC Instruction Sets



- Nature of instructions distinguishes computer
- Two fundamentally different approaches
  - Reduced Instruction Set Computers (RISC) have one-word instructions and require arithmetic operands to be in registers
    - Small set of instructions (typically 32)
    - Simple instructions, each executes in one clock cycle
    - Effective use of pipelining
    - Example: ARM

RISC and CISC Instruction Sets

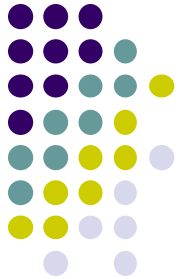


- Nature of instructions distinguishes computer
- Two fundamentally different approaches
  - Complex Instruction Set Computers (CISC) have multi-word instructions and allow operands directly from memory
    - Many instructions (several hundreds)
    - An instruction takes many cycles to execute
    - Example: Intel Pentium

RISC Instruction Sets

- Two key characteristics of RISC instruction sets
  - Each instruction occupies a single word
  - A load/store architecture is used, meaning:
    - Only Load and Store instructions are used to access memory operands
    - Operands for arithmetic/logic instructions must be in registers, or one of them may be given explicitly in instruction word

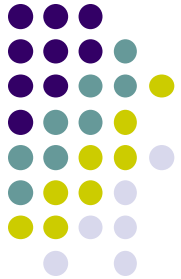
RISC Instruction Sets



- Instructions/data are stored in the memory
- Because RISC requires register operands, data transfers are required before arithmetic
- The Load instruction is used for this purpose:

  Load *procr_register*, *mem_location*
- Addressing mode specifies memory location; different modes are discussed later

RISC Instruction Sets



- Consider high-level language statement:

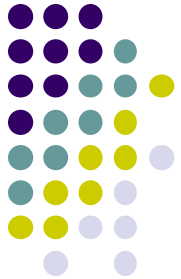
$$C = A + B$$

  - A, B, and C correspond to memory locations
- RTN specification with these symbolic names:

$$C \leftarrow [A] + [B]$$

  - Steps: fetch contents of locations A and B, compute sum, and transfer result to location C

RISC Instruction Sets

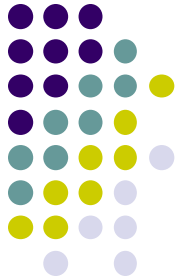


- Sequence of simple RISC instructions for task:

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

  - Load instruction transfers data to register
  - Store instruction transfers data to the memory
  - Destination differs with same operand order

Instruction Execution and Straight-Line Sequencing



- Example:

  C = A + B

  C ← [A] + [B]

- Assume that
  - The word length is 32 bits
  - The memory is byte-addressable
  - A desired memory address can be directly specified in Load and Store instructions.

Instruction Execution and Straight-Line Sequencing

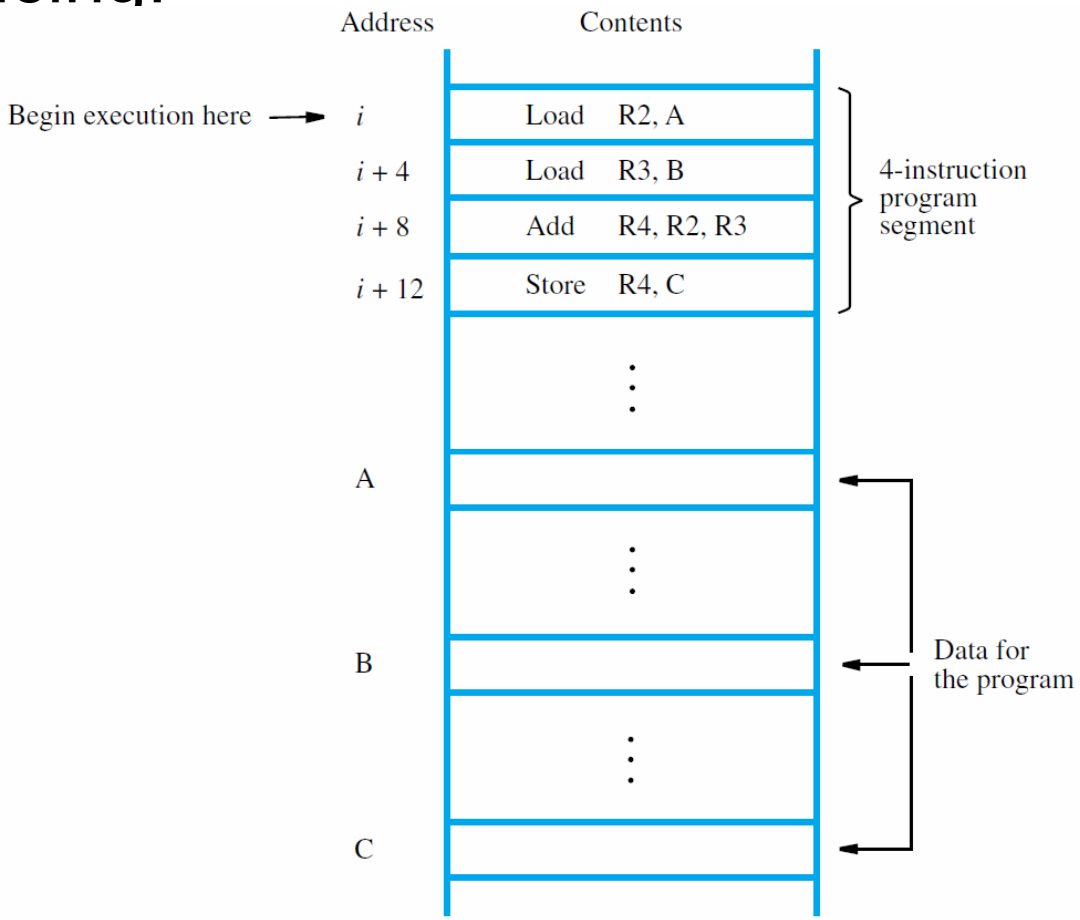
- Straight-line sequencing:

  [PC] = i

  [PC] = i + 4

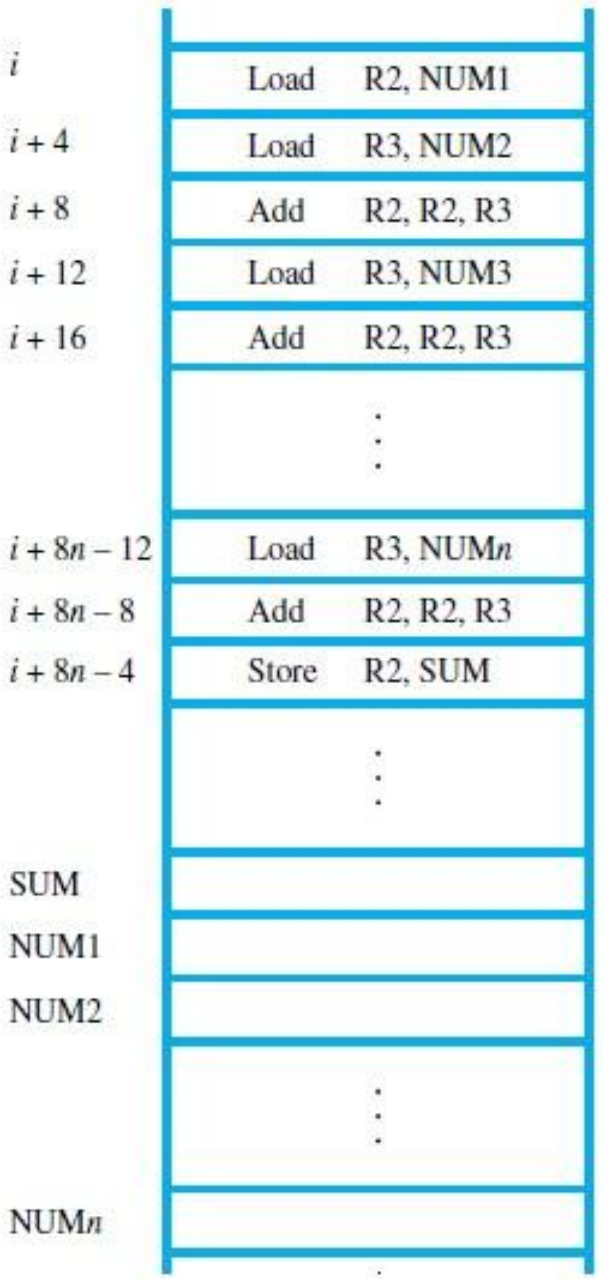
  …

- Instruction Fetch
- Instruction Execute



Branching

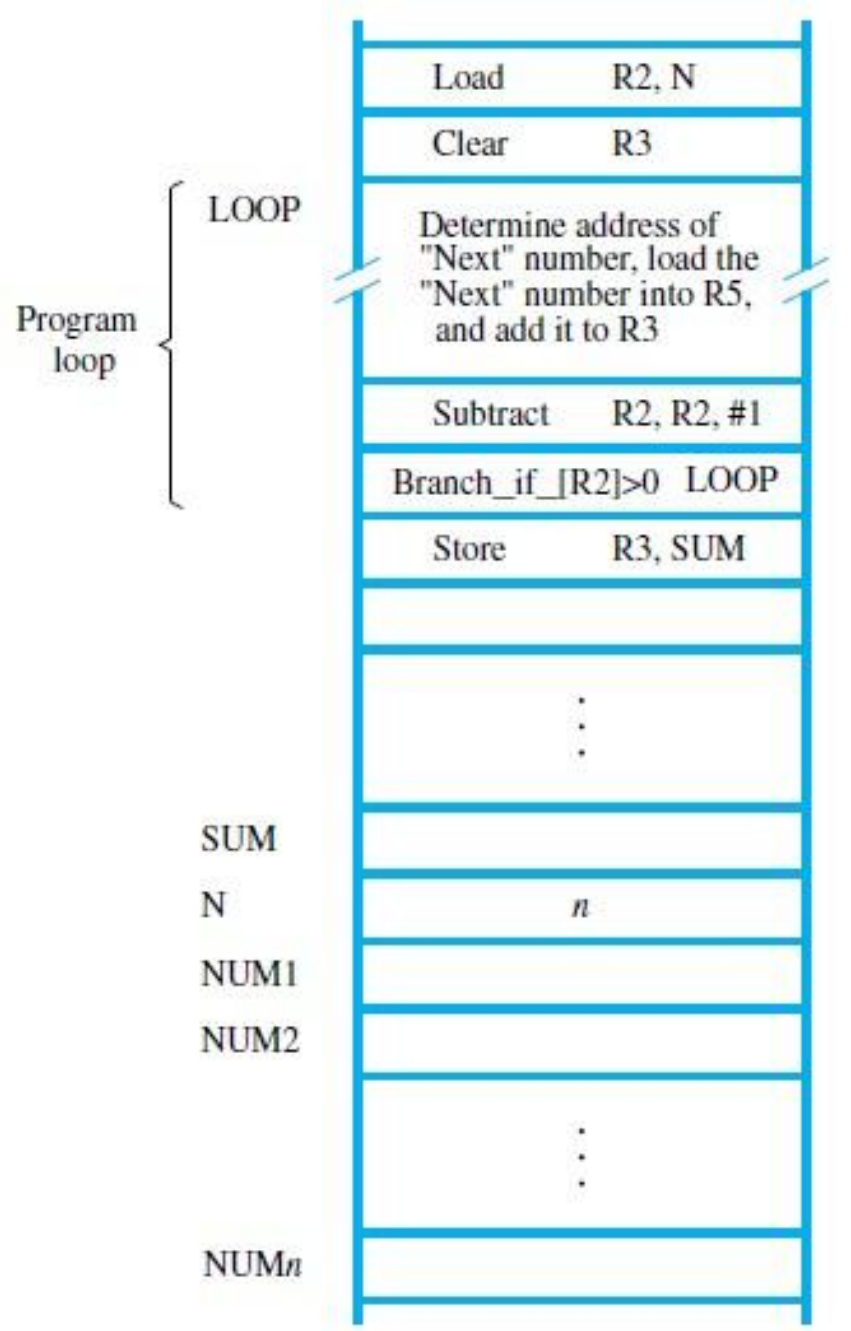
- Adding a list of n numbers
  - Separate Load and Add Instructions



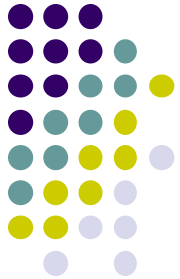
Branching

- Adding a list of n numbers
  - A program loop
  - Branch Instructions
    - Load a new address into the PC
    - Branch Target
  - Conditional Branch
    - Compare the contents of two registers

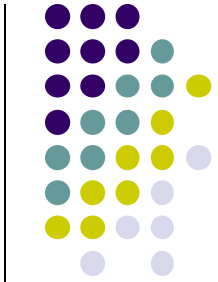
      Branch_if_[R4]>[R5] Loop
    - Condition Codes



E.4 Instruction Formats



- What's Machine Instruction?
  - The instructions which specify the actions that must be performed by the processor circuitry to carry out the desired tasks.
  - Usually represented by assembly codes
- What's Instruction Set?
  - The collection of different machine instructions that the processor can execute.



- Elements of a Machine Instruction
  - Operation code (Opcode)
    - Specify the operation to be performed (e.g., ADD, I/O), expressed as a binary code.
  - Source operand reference
    - Operands required for the instruction are specified.
  - Result operand reference
    - Where should the result of the operation be placed?
    - Source and result operands can be in one of three areas
      - Main or virtual memory
      - Processor register
      - I/O device

- Elements of a Machine Instruction
  - Next instruction reference
    - How / Where is the next instruction to be found?
      - In most cases, this is not explicitly stated in the instruction.
      - Next instruction is the one that logically follows the current one in the program (sequential / linear progression through the program).

- Instruction Representation
  - Within the computer, each instruction is represented by a sequence of binary bits.
  - The instruction is divided into two fields
    - Operation code field
      - Specify the operation to be performed
    - Address field

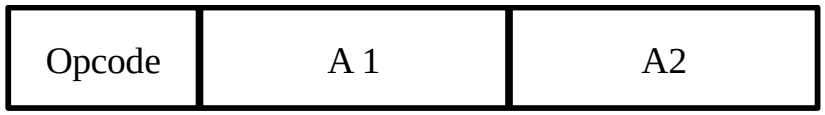
- Instruction Design Criteria
  - Short instructions are better than long ones
  - Sufficient room in the instruction format to express all the operations desired
  - Number of bits in the address field

- Instruction Address Field Formats
  - Zero-address Instruction

  - One-address Instruction

    - OP [A] → A
    - [AC] OP [A] → AC
      - Typically, it is understood implicitly that a second operand is in the accumulator of the processor.

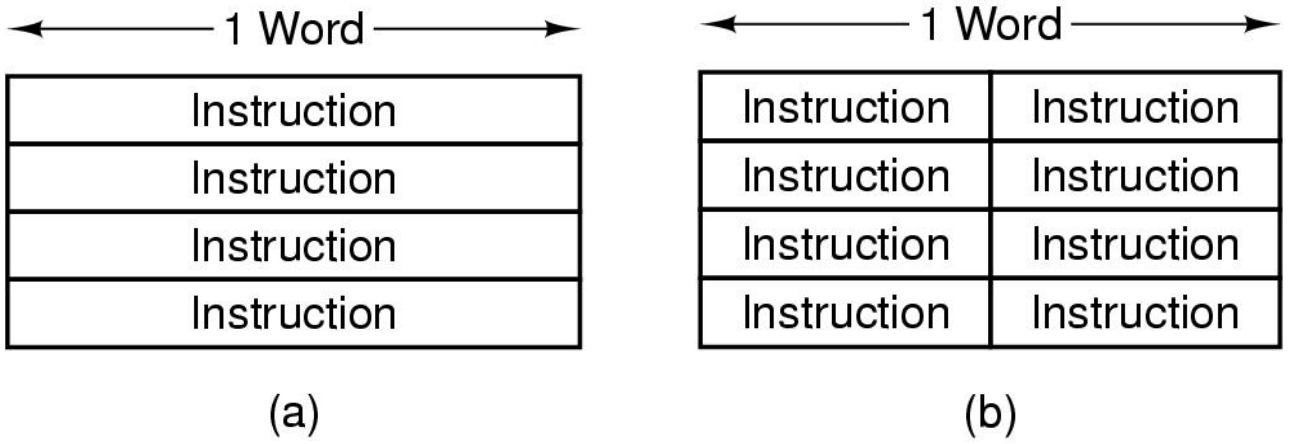
- Instruction Address Field Formats
  - Two-address Instruction



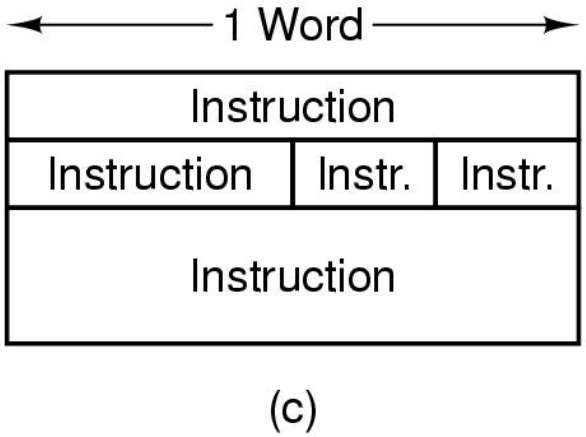
    - [A1] OP [A2] → A1
      - A1: Destination operand address
      - A2: Source operand address
  - Three-address Instruction

    - [A2] OP [A3] → A1

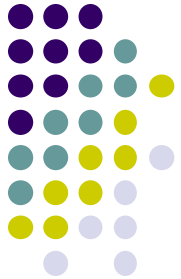
- Instruction Length
  - Fixed length: All instructions have the same length



  - Variable length: Instructions may be many different lengths

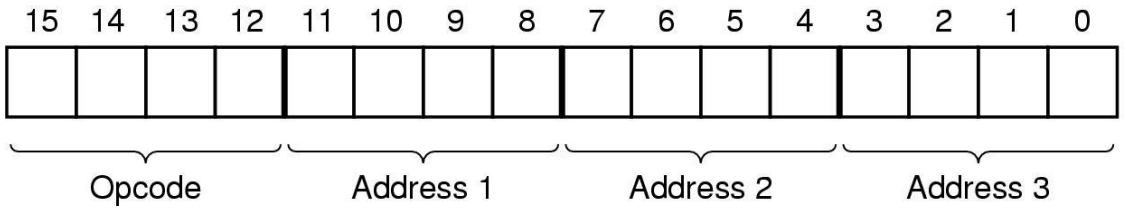


- Instruction Length
  - Methods to reduce instruction length
    - If the operand is to be used several times, it can be moved into a register. (Note: If an operand is to be used only once, putting it in a register is not worth it.)
    - Specify one or more operands implicitly.



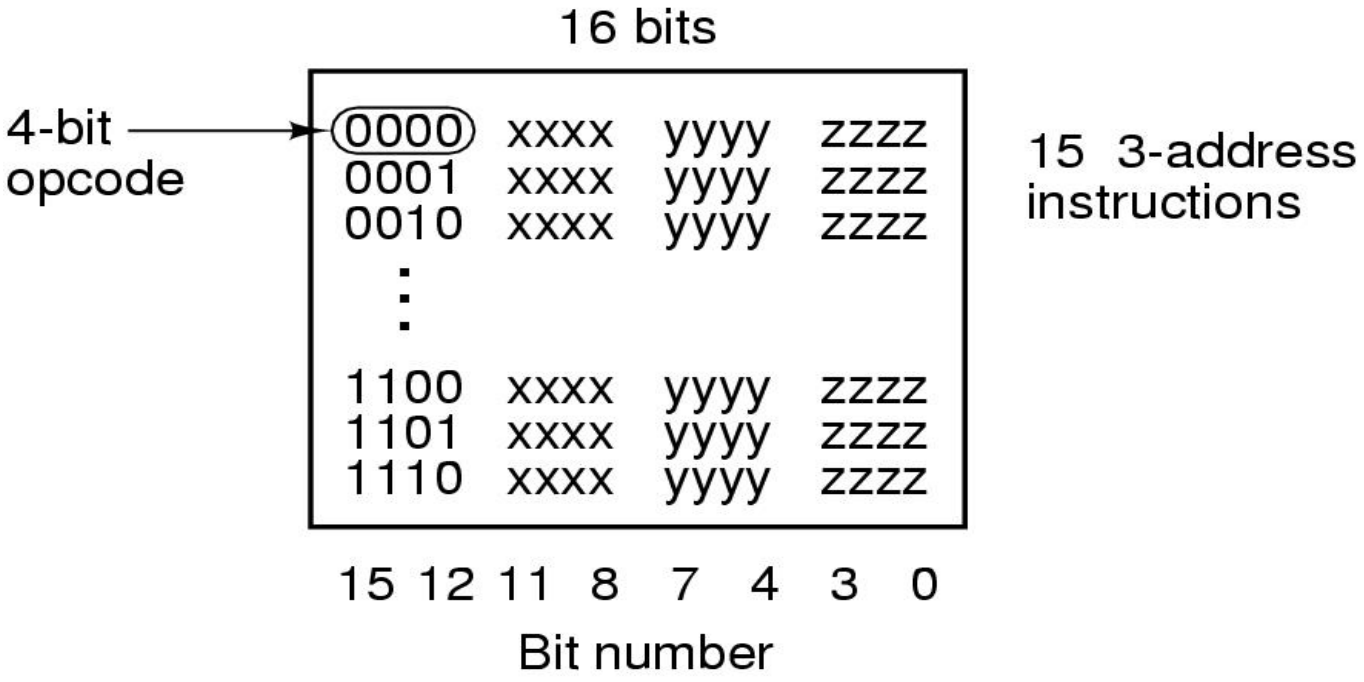
- Opcode Format
  - Fixed-length Opcode
    - The Opcode length is fixed, but the instruction length is variable.
    - Suppose that, k bits opcode, n bits operand address
      - Allows for 2^k different operations
      - Allows for 2^n addressable memory cells

- Opcode Format
  - Variable-length Opcode (Expanding Opcode)
    - Usually, the instruction length is fixed, and the length of Opcode and operand address is limited each other.
    - Example: Instruction length is 16-bit, operand address is 4-bit.
      - This might be reasonable on a machine that has 16 registers on which all arithmetic operations take place.
      - One design would be a 4-bit opcode and three addresses in each instruction, giving 16 three-address instructions.

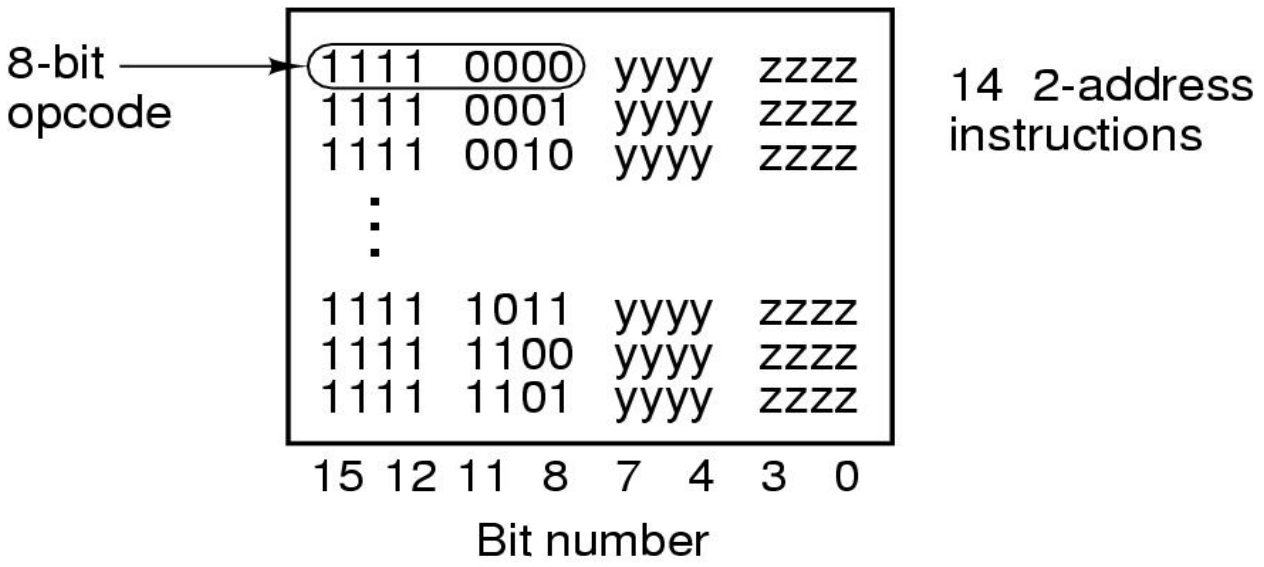


- Opcode Format
  - Variable-length Opcode (Expanding Opcode)
    - Example: Instruction length is 16-bit, operand address is 4-bit. (ctd.)
      - Suppose the designers need
        - 15 three-address instructions
        - 14 two-address instructions
        - 31 one-address instructions
        - 16 zero-address instructions
      - How should we design the instruction format?

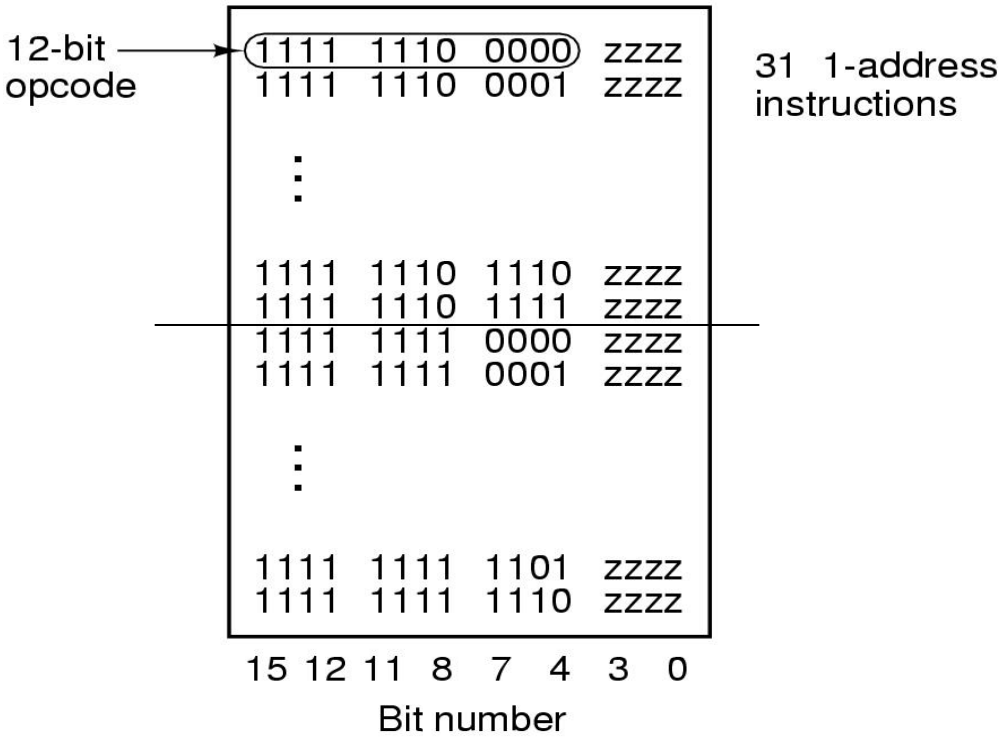
- Opcode Format
  - Variable-length Opcode (Expanding Opcode)
    - Example: Instruction length is 16-bit, operand address is 4-bit. (ctd.)
      - 15 three-address instructions
        - 4-bit opcode 0000 ~ 1110 (15~12 bit)



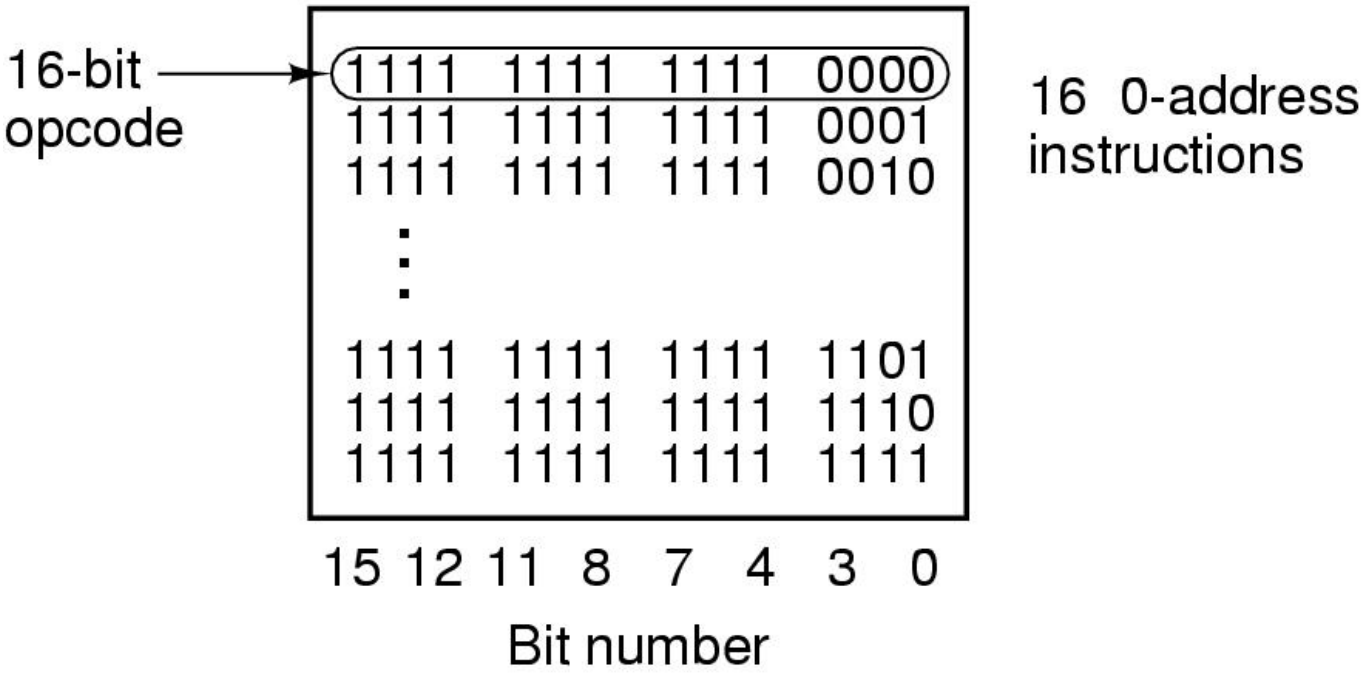
- Opcode Format
  - Variable-length Opcode (Expanding Opcode)
    - Example: Instruction length is 16-bit, operand address is 4-bit. (ctd.)
      - 14 two-address instructions
        - 8-bit opcode 11110000 ~ 11111101 (15~8 bit)



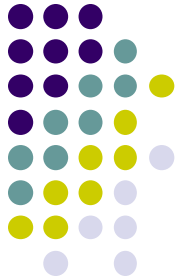
- Opcode Format
  - Variable-length Opcode (Expanding Opcode)
    - Example: Instruction length is 16-bit, operand address is 4-bit. (ctd.)
      - 31 one-address instructions
        - 12-bit opcode 1111 1111**0** 0000 ~ 1111 111**1** **1110** (15~4 bit)



- Opcode Format
  - Variable-length Opcode (Expanding Opcode)
    - Example: Instruction length is 16-bit, operand address is 4-bit. (ctd.)
      - 16 zero-address instructions
        - 16-bit opcode 111111111111 0000 ~ 111111111111 1111



2.4 Addressing Modes



- Addressing Modes
  - How to specify where an operand for an instruction is located?
  - How the bits of an address field in an instruction are interpreted?
  - What is Addressing Modes?
    - The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

- Typical Addressing Modes

RISC-type addressing modes.

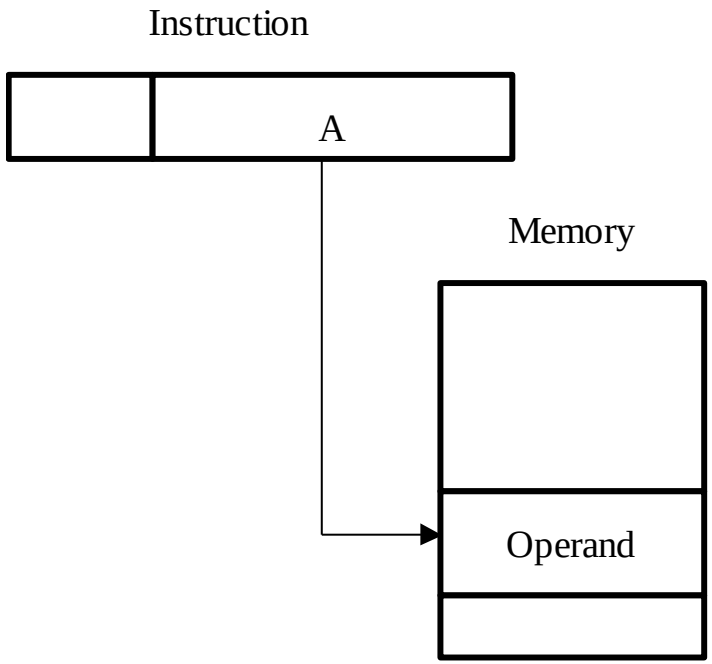
Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute	LOC	EA = LOC
Register indirect	(R*i*)	EA = [R*i*]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]

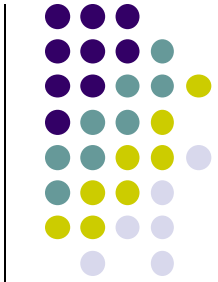
EA = effective address
Value = a signed number
X = index value

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

  - Example: Add R4, R6, #200
  - Usage
    - Define and use constants
    - Set initial values of variables
  - Advantage
    - No memory reference other than the instruction fetch is required to obtain the operand.
  - Disadvantage
    - Only a constant can be supplied this way.
    - The size of the number is limited by the size of the address field.

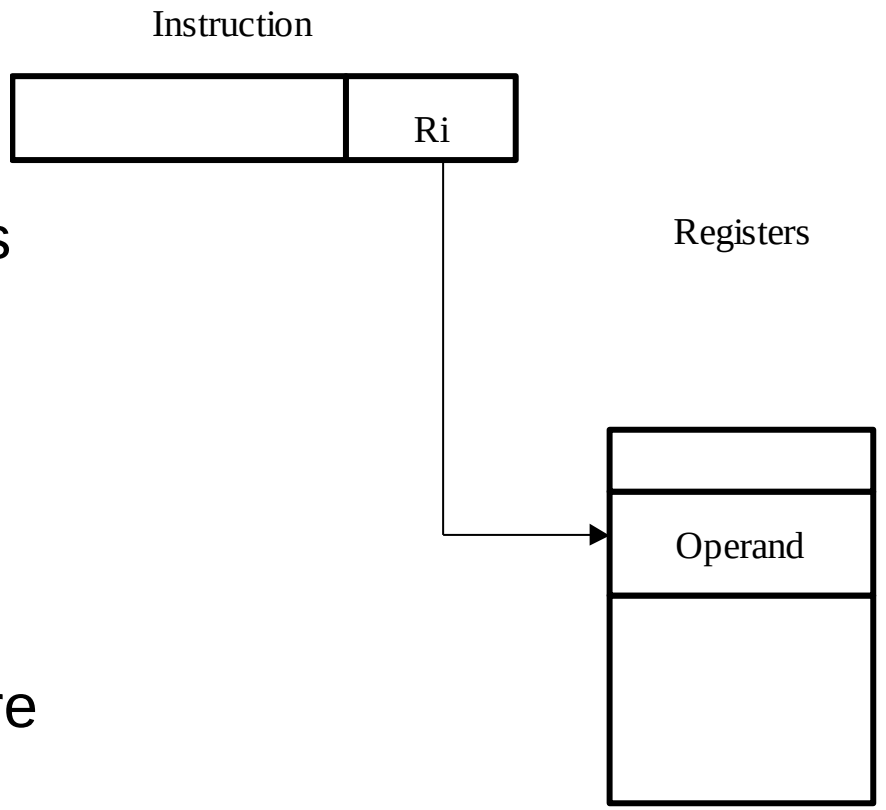
- Absolute mode
  - The operand is in a memory location; the address of this location is given explicitly in the instruction.
  - Example: Load R2, NUM1
  - EA=A
    - EA: Actual (Effective) address of the location containing the referenced operand
    - A: contents of an address field in the instruction
  - Note
    - In a system without virtual memory, the effective address will be either a main memory address or a register address.
    - In a virtual memory system, the effective address is a virtual address or a register address.



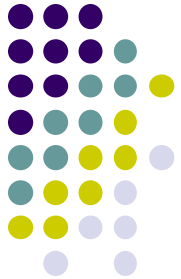


- Absolute mode (ctd.)
  - Usage
    - Access global variables whose address is known at compile time.
  - Advantage
    - Only one memory reference and no special calculation.
  - Disadvantage
    - The instruction will always access exactly the same memory location.
    - It provides only a limited address space.

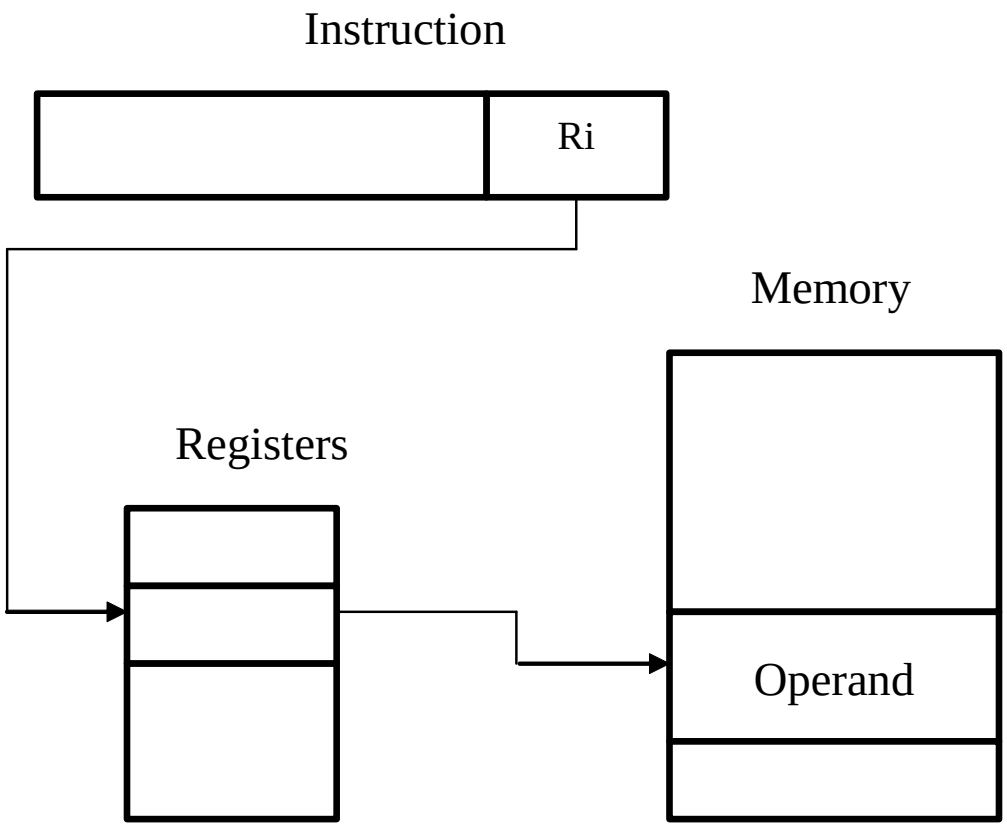
- Register mode
  - The operand is the contents of a processor register; the address of the register is given in the instruction.
  - EA = Ri
    - Ri: contents of an address field in the instruction that refers to a register.
  - Example: Add R4, R2, R3
  - Usage
    - Access variables which are accessed most often.



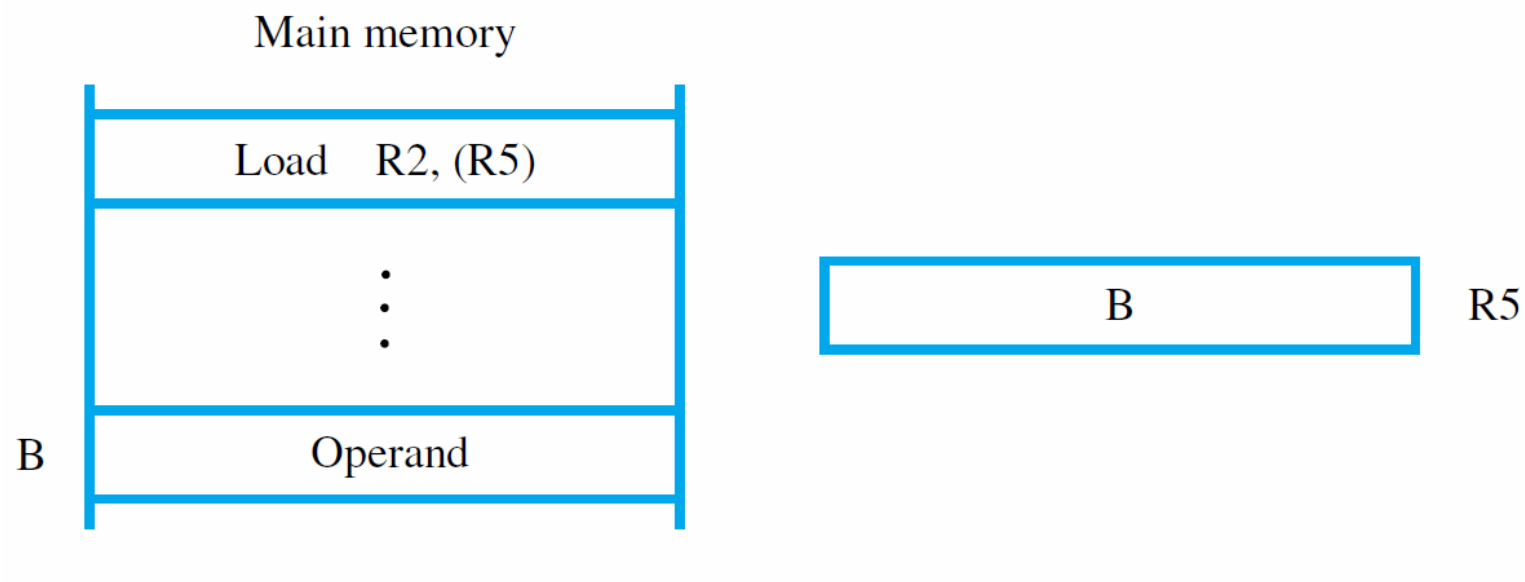
- Register mode (ctd.)
  - Advantage
    - Only a small address field is needed in an instruction.
    - No memory reference references are required.
  - Disadvantage
    - The address space is very limited.



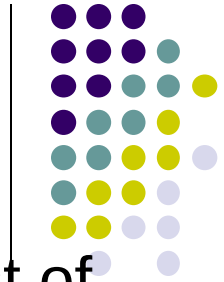
- Register Indirect mode
  - The effective address of the operand is the contents of a register whose address appears in the instruction.
  - EA = [Ri]



- Register Indirect mode (ctd.)
  - Example: Load R2, (R5)



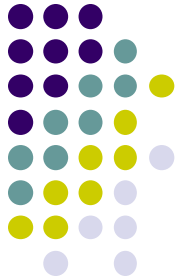
- Register Indirect mode (ctd.)
  - Example: Using indirect addressing to access a list of n numbers



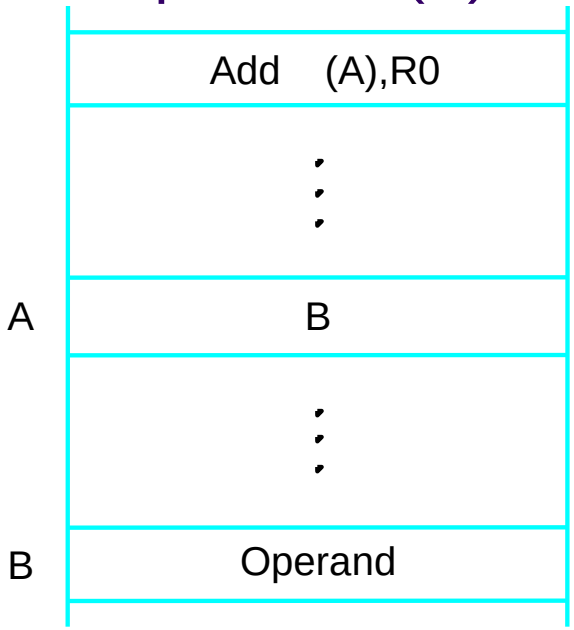
```
        Load        R2, N          Load the size of the list.
        Clear       R3             Initialize sum to 0.
        Move        R4, #NUM1      Get address of the first number.
LOOP:   Load        R5, (R4)       Get the next number.
        Add         R3, R3, R5     Add this number to sum.
        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.
```

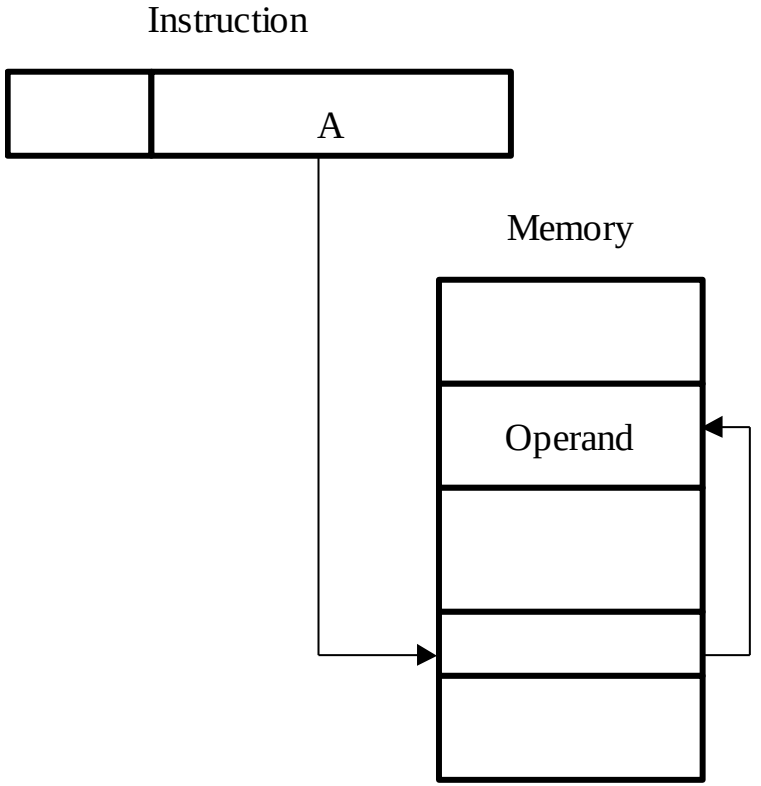
Figure 2.8 Use of indirect addressing in the program of Figure 2.6

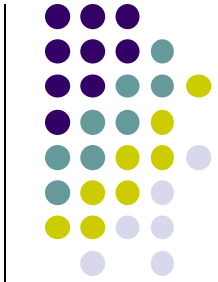
- Register Indirect mode (ctd.)
  - Advantage
    - It can reference memory without paying the price of having a full memory address in the instruction.
    - Reduce the memory access times.



- Indirect mode
  - Indirect addressing through a memory location is also possible, but it is found only in CISC-style processors.
  - The effective address of the operand is the contents of a memory location whose address appears in the instruction.
  - EA = [A]
  - Example: Add (A), R0

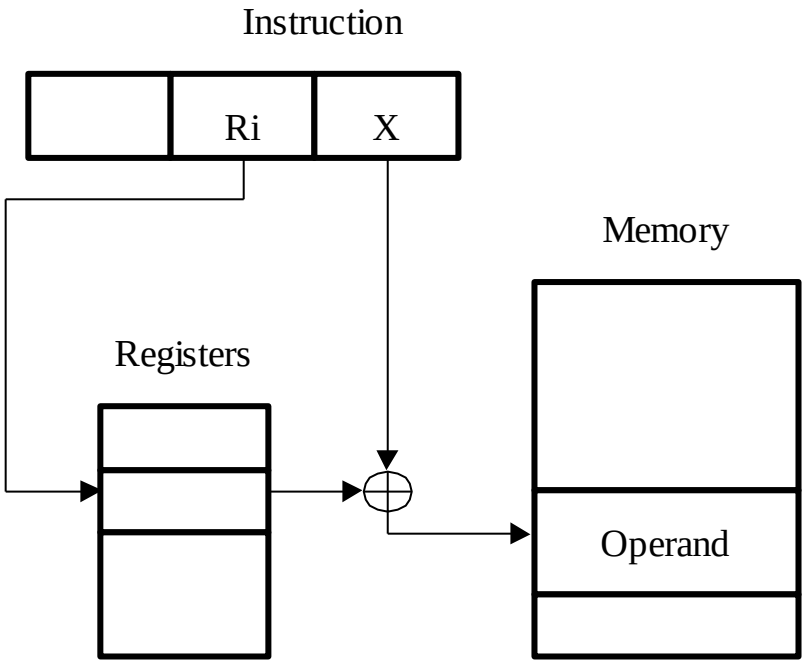






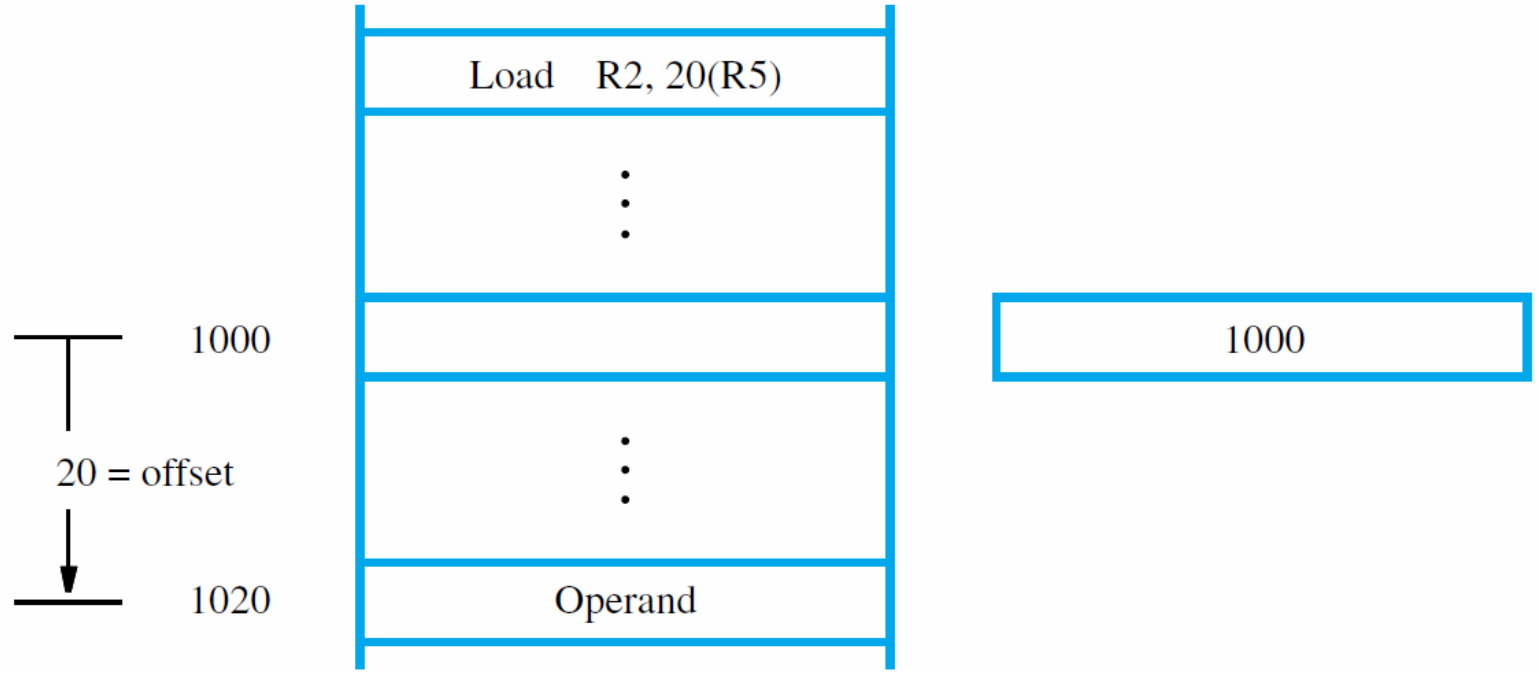
- Indirect mode (ctd.)
  - Advantage
    - The address space is very large.
    - If memory word length is N, then 2^N address space
  - Disadvantage
    - Instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.
  - Multilevel indirect addressing
    - EA = […[A]…]
    - One bit of a full-word address is an indirect flag (I).
      - If the I bit is 0, then the word contains the EA.
      - If the I bit is 1, then another level of indirection is invoked.

- Indexed mode
  - X(Ri)
  - The effective address of the operand is generated by adding a constant value to the contents of a register (index register).
  - EA = X + [Ri]
    - X (offset): the constant value contained in the instruction



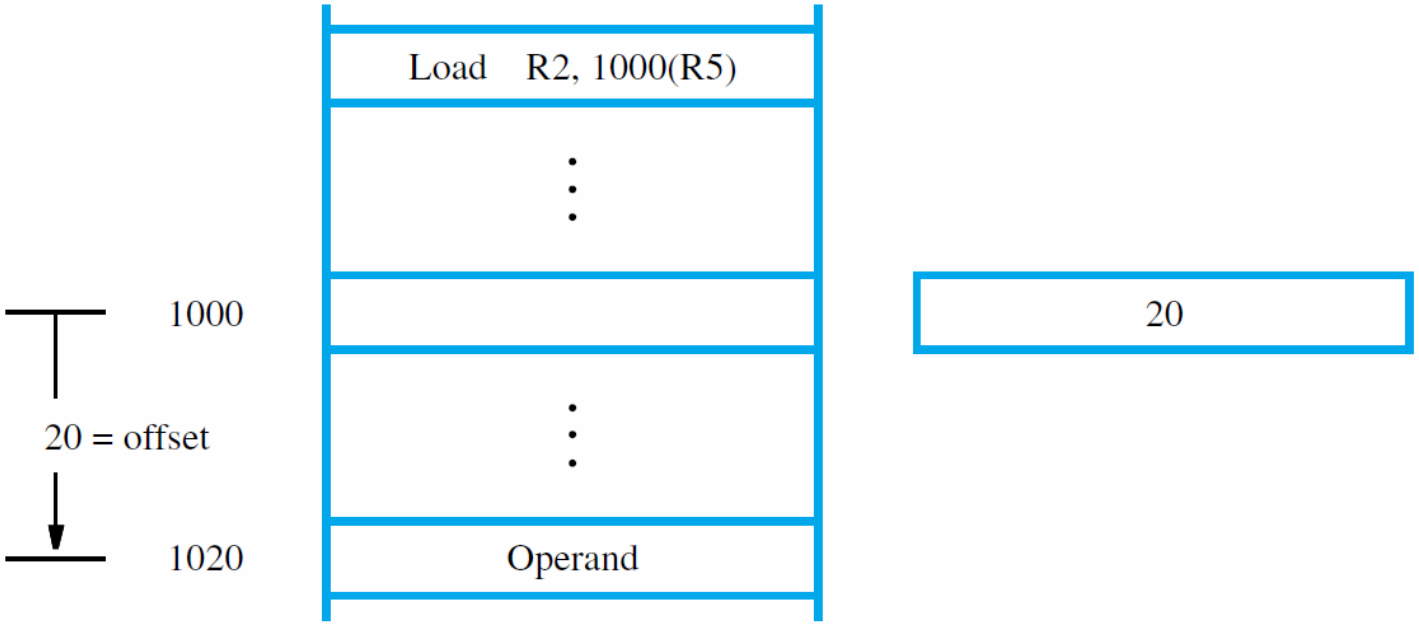
  - Note
    - The contents of the index register are not changed in the process of generating the effective address.

- Indexed mode (ctd.)
  - Two ways of using the index mode
    - Offset is given as a constant
      - Example



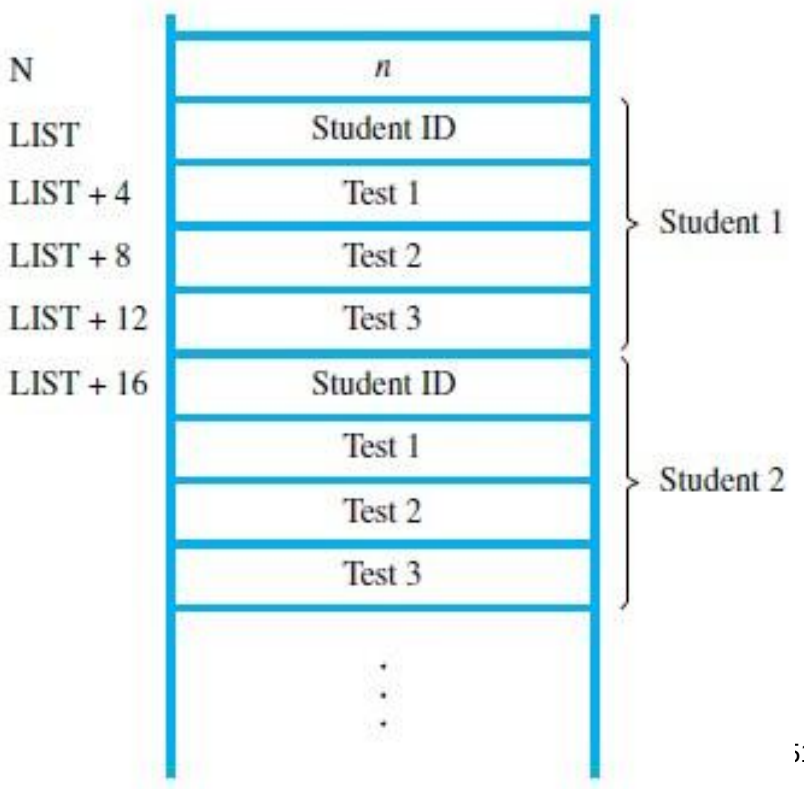
(a) Offset is given as a constant

- Indexed mode (ctd.)
  - Two ways of using the index mode
    - Offset is in the index register. This form requires an offset field in the instruction large enough to hold an address.
      - Example

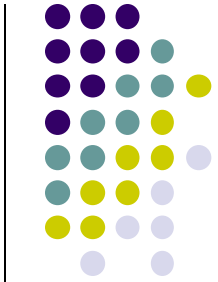


(b) Offset is in the index register

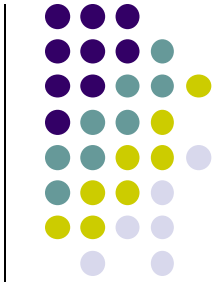
- Indexed mode (ctd.)
  - Usage
    - Facilitate access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
    - Example: N rows and four columns array
      - The memory is byte addressable and the word length is 32 bits



- Indexed mode (ctd.)
  - Usage
    - Example

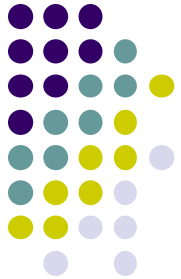


```
        Move            R2, #LIST       Get the address LIST.
        Clear           R3
        Clear           R4
        Clear           R5
        Load            R6, N           Load the value n.
LOOP:   Load            R7, 4(R2)       Add the mark for next student's
        Add             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.
```

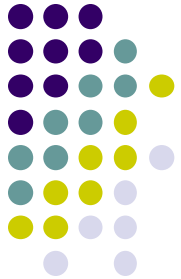


- Indexed Addressing (ctd.)
  - Variations of indexed addressing mode
    - Base with index
      - (Ri, Rj)
      - A second register (base register) is used to contain the offset X.
      - EA = [Ri] + [Rj]
    - Base with index and offset
      - X(Ri, Rj)
      - Use index register, base register and a constant.
      - EA = [Ri] + [Rj] + X

2.6 Stacks



- A stack is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only.
- The structure is sometimes referred to as a pushdown stack or last-in–first-out (LIFO) stack.
- Push: Place a new item on the stack
- Pop: Remove the top item from the stack
- In modern computers, a stack is implemented by using a portion of the main memory.
- Programmer can create a stack in the memory
- There is often a special processor stack as well



- Processor stack
  - Processor has stack pointer (SP) register that points to top of the processor stack.
  - Assume a byte-addressable memory with a 32-bit word length
  - Push operation involves two instructions:

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

  - Pop operation also involves two instructions:

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

- Processor stack

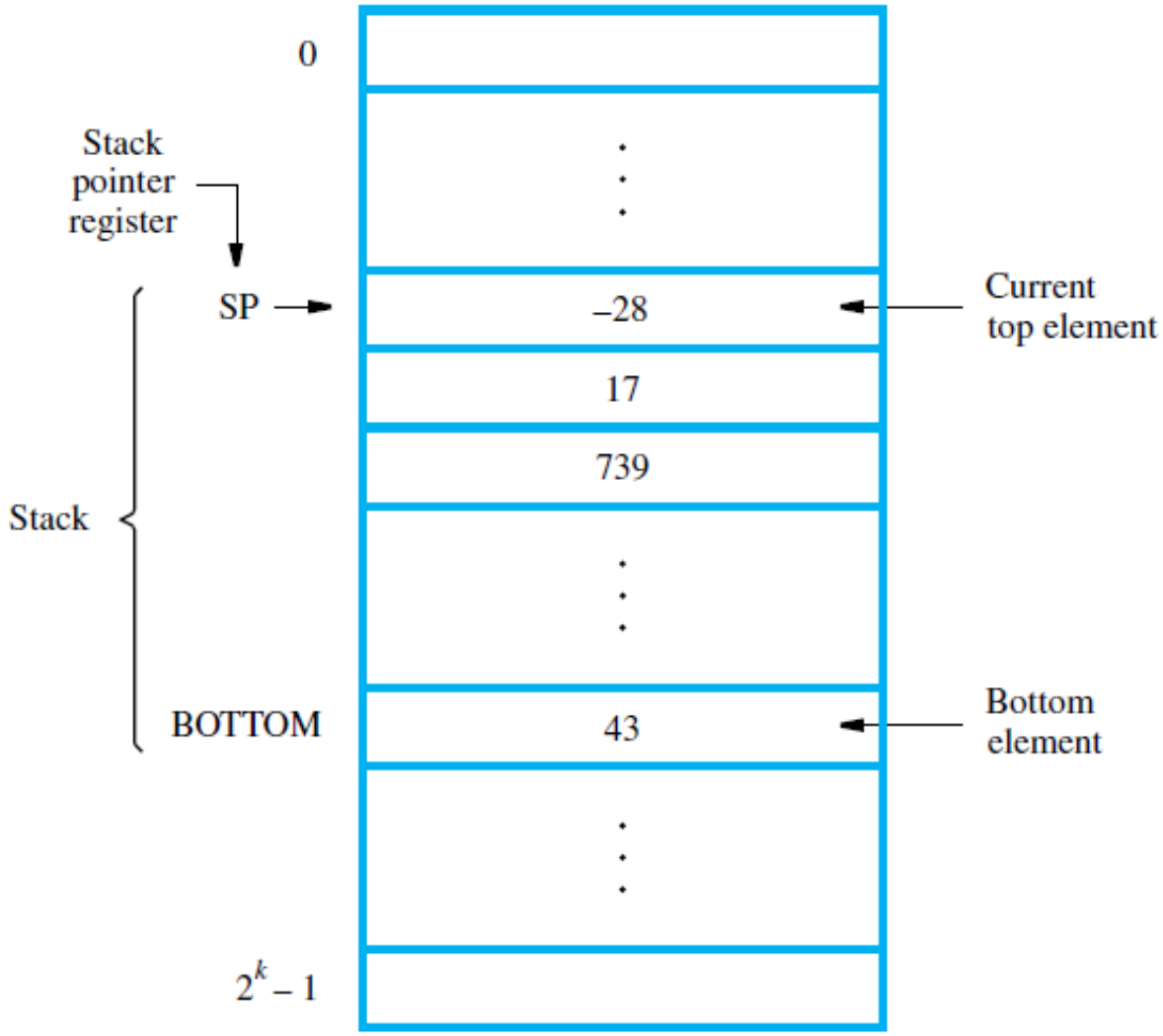


Figure 2.14 A stack of words in the memory.

- Processor stack

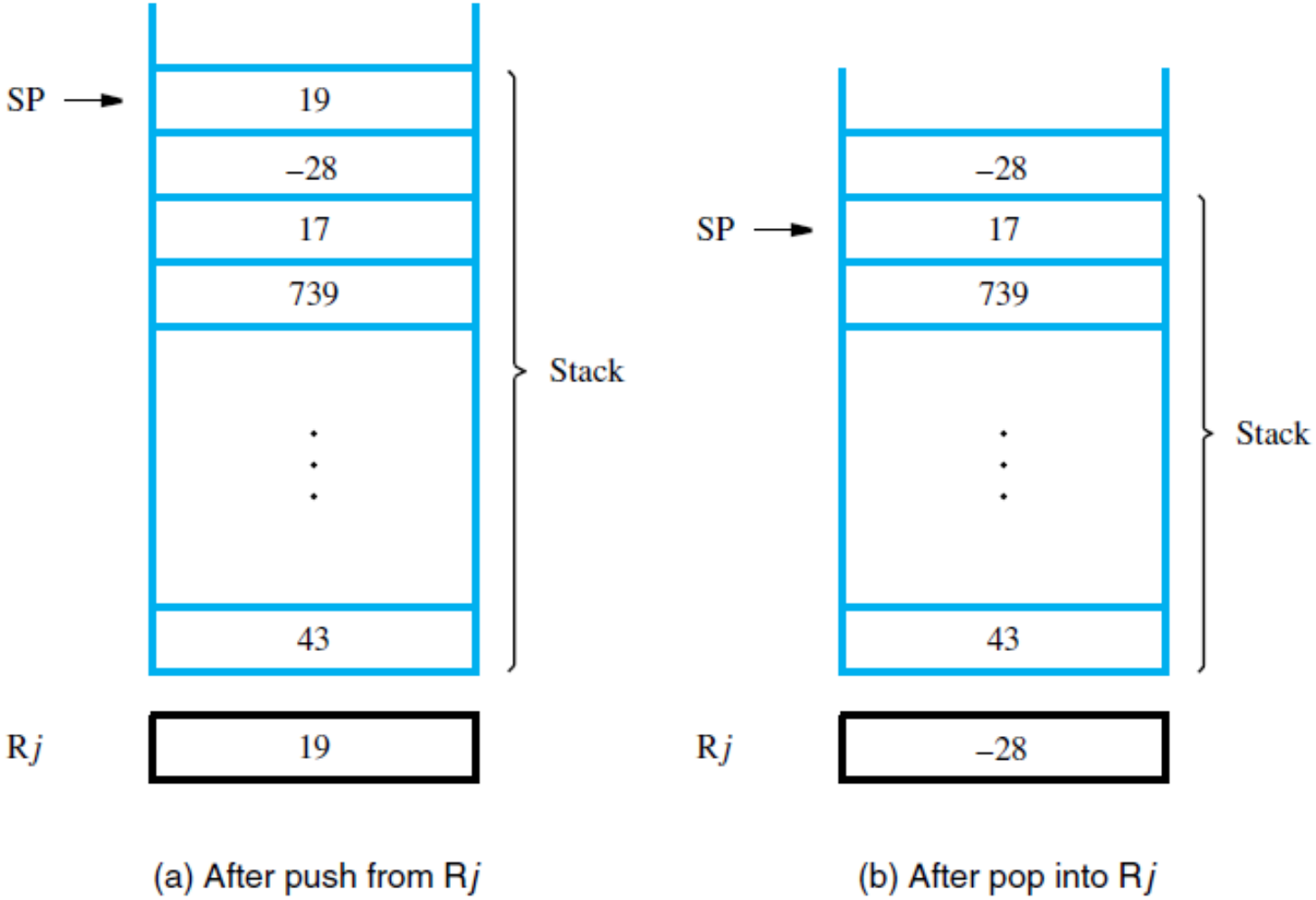
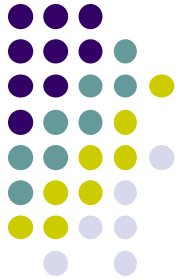


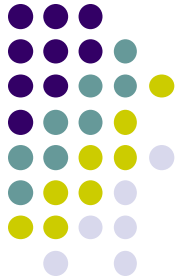
Figure 2.15 Effect of stack operations on the stack in Figure 2.14.

2.7 Subroutines

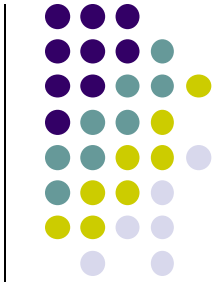


- In a given program, a particular task may be executed many times using different data
- Examples: mathematical function, list sorting
- Implement the task in one block of instructions
- This is called a subroutine
- Rather than reproduce entire subroutine block in each part of program, use a subroutine call
- Special type of branch with Call instruction

2.7 Subroutines

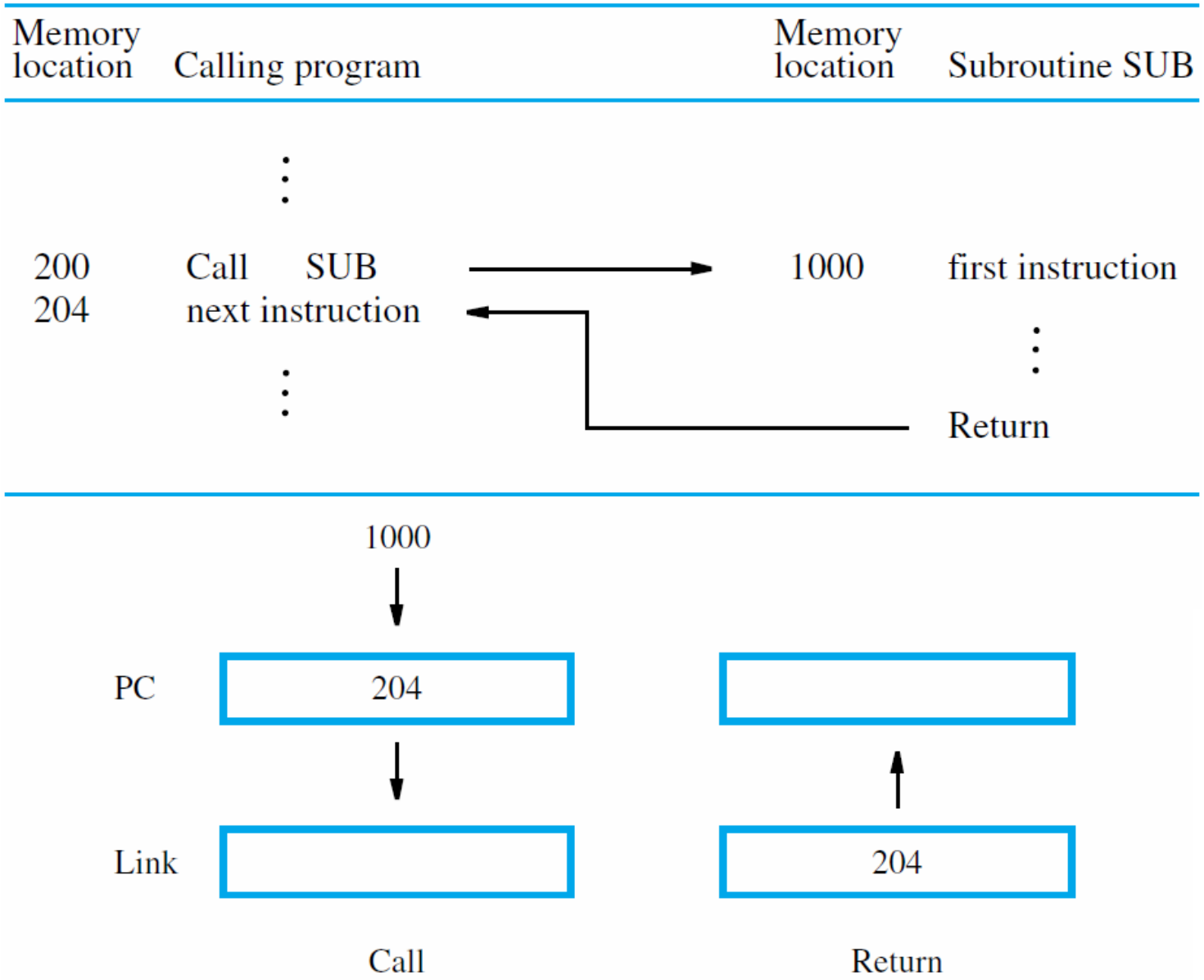


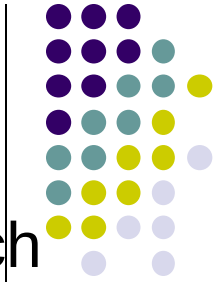
- Branching to same block of instructions saves space in memory, but must branch back
- The subroutine must return to calling program after executing last instruction in subroutine
- This branch is done with a Return instruction



- Subroutine Linkage
  - Subroutine can be called from different places
  - How can return be done to correct place?
  - This is the issue of subroutine linkage
  - During execution of Call instruction, PC updated to point to instruction after Call
  - Save this address for Return instruction to use
  - Simplest method: place address in link register
  - Call instruction performs two operations: store updated PC contents in link register, then branch to target (subroutine) address
  - Return just branches to address in link register

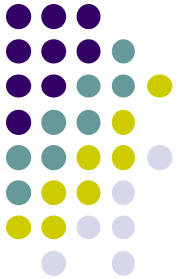
- Subroutine Linkage





- Subroutine Nesting & The Processor Stack
  - We can permit one subroutine to call another, which results in subroutine nesting.
  - Link register contents after first subroutine call are overwritten after second subroutine call.
  - First subroutine should save link register on the processor stack before second call.
  - After return from second subroutine, first subroutine restores link register.
  - Subroutine nesting can be carried out to any depth.
  - Return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

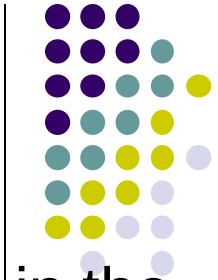
2.8 Additional Instructions



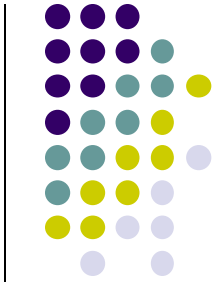
- Logic Instructions
  - AND, OR, and NOT operations on single bits are basic building blocks of digital circuits.
  - Similar operations in software on multiple bits.
  - Using RISC-style instructions, all operands are in registers or specified as immediate values:

```
Or      R4, R2, R3
And     R5, R6, #0xFF
```

    - 16-bit immediate is zero-extended to 32 bits



- Logic Instructions
  - Suppose that four ASCII characters are contained in the 32-bit register R2. In some task, we wish to determine if the rightmost character is Z. The ASCII code for Z is 01011010, which is expressed in hexadecimal notation as 5A.
    - And R2, R2, #0xFF
    - Move R3, #0x5A
    - Branch_if_[R2]=[R3] FOUNDZ

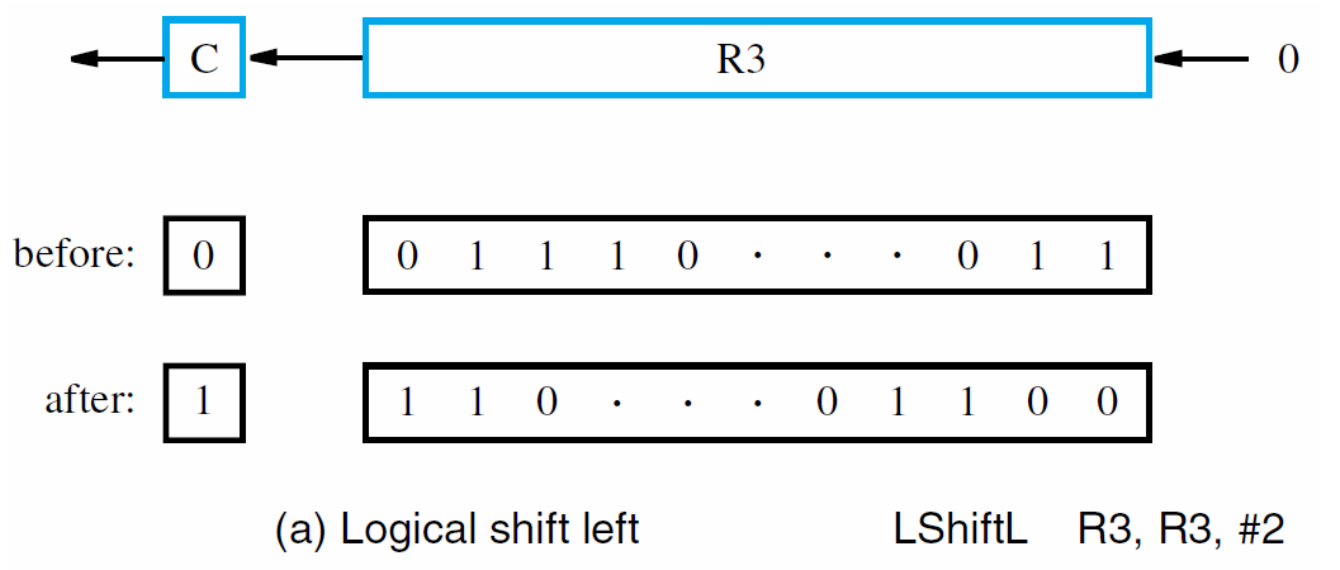


- Shift and Rotate Instructions
  - Shifting binary value left/right = mult/div by 2
  - Logical Shifts
    - For general operands, we use a logical shift.
    - Logic Shifting Left (LShiftL)

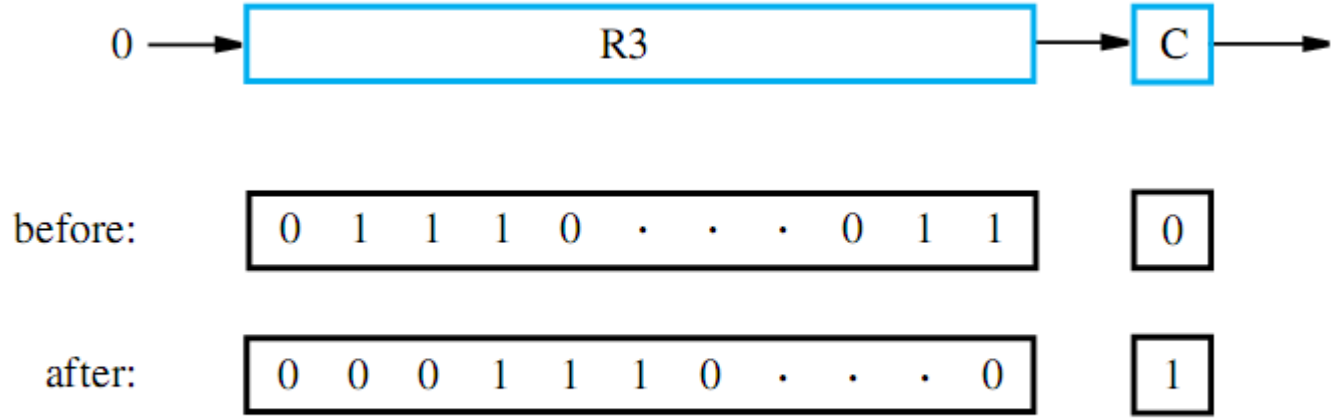
      LShiftL Ri, Rj, count

      Shift the contents of register R*j* left by a number of bit positions given by the *count* operand, and places the result in register R*i*, without changing the contents of R*j*.
    - Logic Shifting Right (LShiftR)

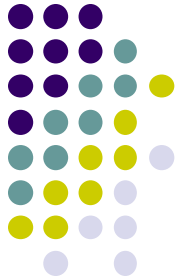
- Shift and Rotate Instructions



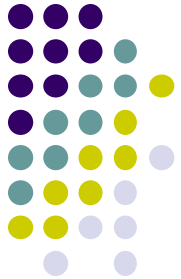
(a) Logical shift left LShiftL R3, R3, #2



(b) Logical shift right LShiftR R3, R3, #2



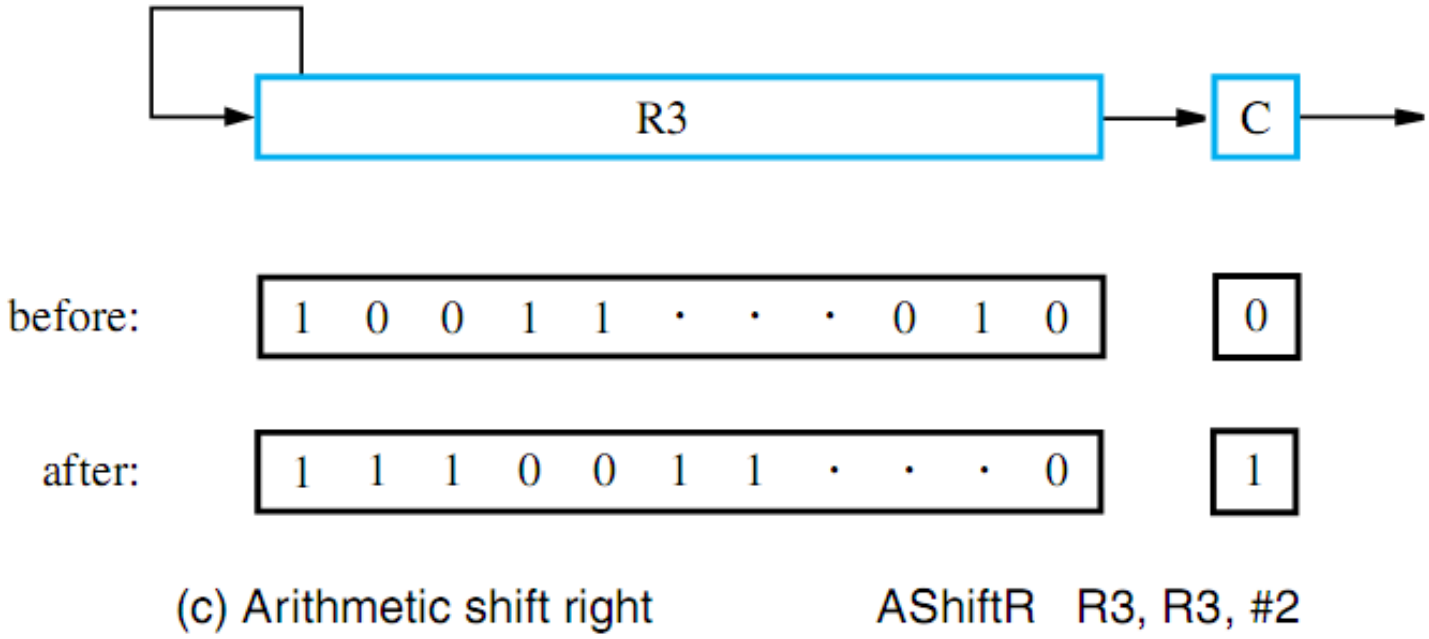
- Shift and Rotate Instructions
  - Digit-Packing Example
    - Illustrate shift, logic, byte-access instructions
    - Memory has two binary-coded decimal (BCD) digits
    - Pointer set to 1st byte for index-mode access to load 1st digit, which is shifted to upper bits
    - Upper bits of 2nd digit are cleared by ANDing
    - ORing combines 2nd digit with shifted 1st digit for result of two packed digits in a single byte
    - 32-bit registers, but only 8 lowest bits relevant



- Shift and Rotate Instructions
  - Digit-Packing Example

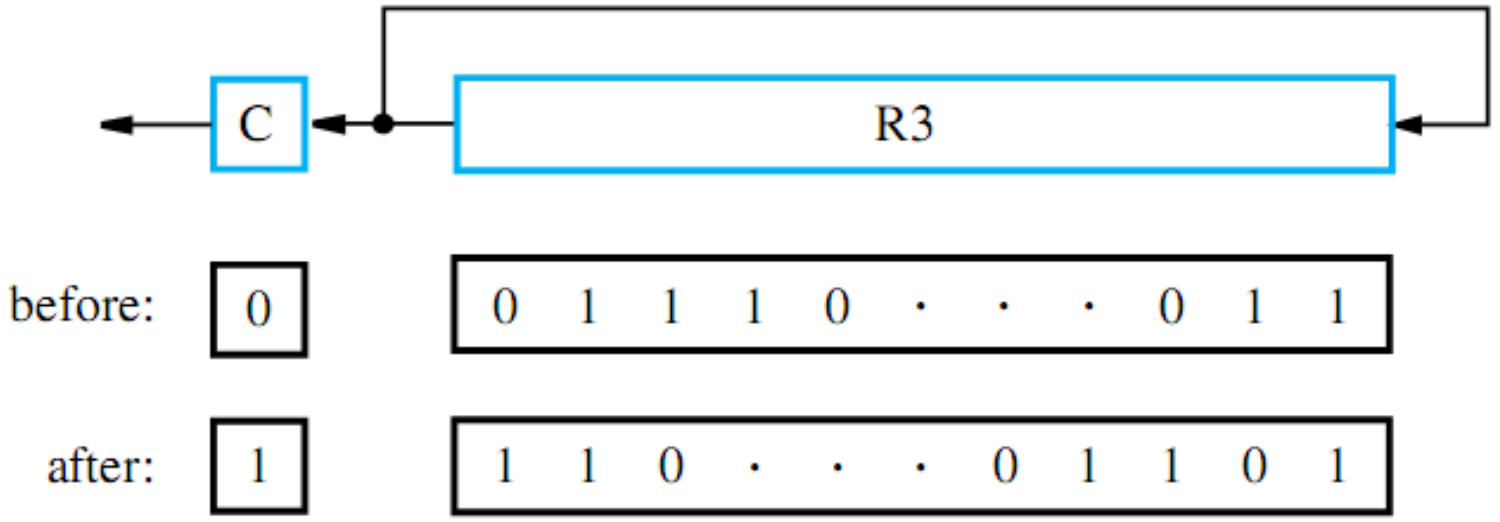
```
Move        R2, #LOC      R2 points to data.
LoadByte    R3, (R2)      Load first byte into R3.
LShiftL     R3, R3, #4    Shift left by 4 bit positions.
Add         R2, R2, #1    Increment the pointer.
LoadByte    R4, (R2)      Load second byte into R4.
And         R4, R4, #0xF  Clear high-order bits to zero.
Or          R3, R3, R4    Concatenate the BCD digits.
StoreByte   R3, PACKED    Store the result.
```

- Shift and Rotate Instructions
  - Arithmetic shifts
    - In an arithmetic shift, the bit pattern being shifted is interpreted as a signed number.
    - On a right shift the sign bit must be repeated as the fill-in bit for the vacated position as a requirement of the 2's-complement representation for numbers.



(c) Arithmetic shift right AShiftR R3, R3, #2

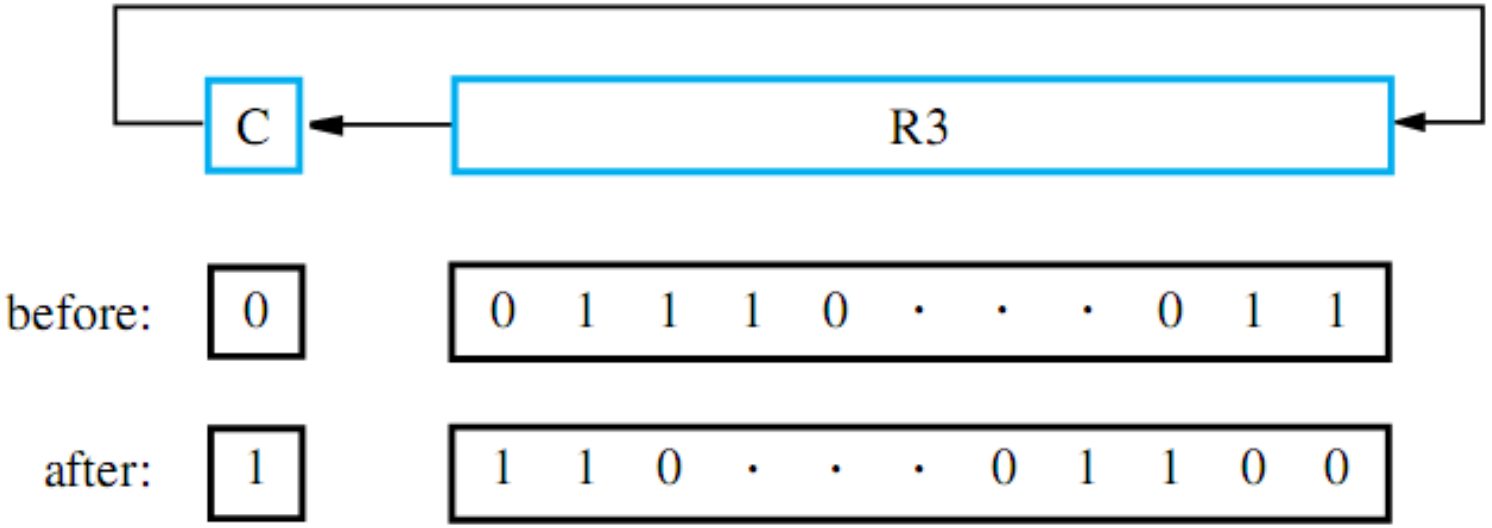
- Shift and Rotate Instructions
  - Rotate operations
    - Rotate left without carry



(a) Rotate left without carry

RotateL R3, R3, #2

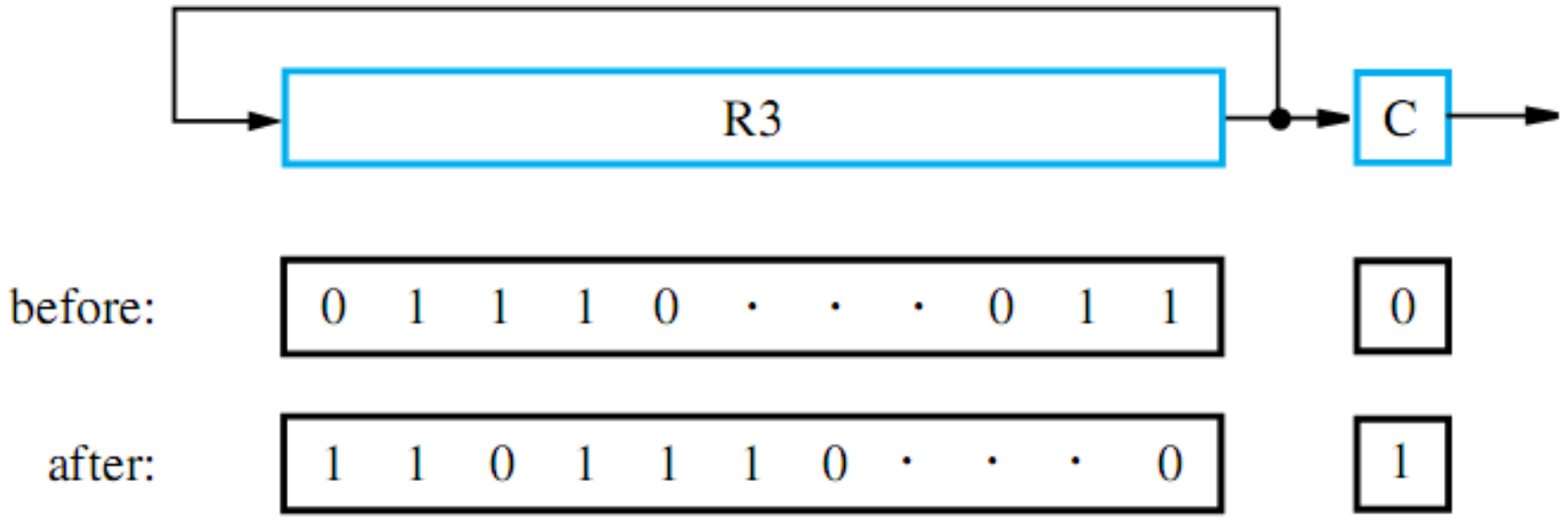
- Shift and Rotate Instructions
  - Rotate operations
    - Rotate left with carry



(b) Rotate left with carry

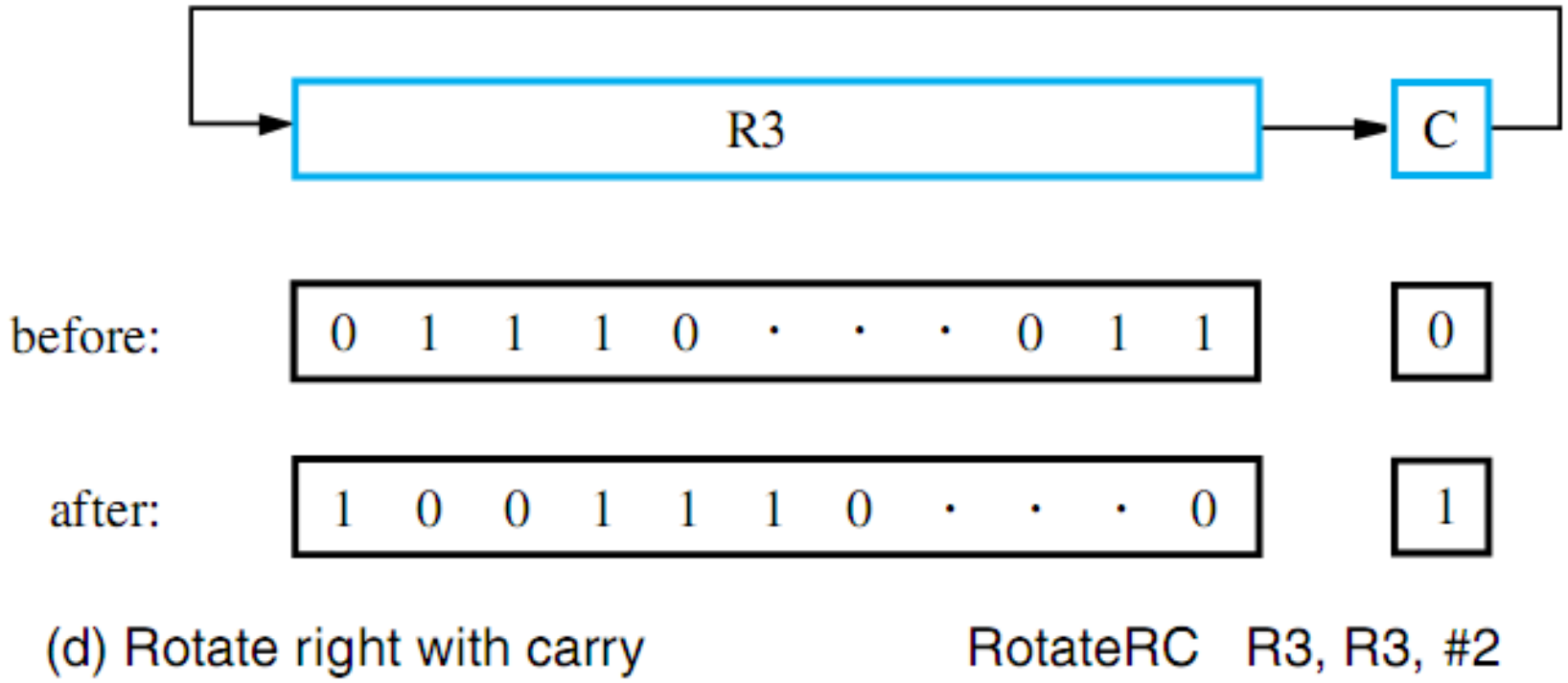
RotateLC R3, R3, #2

- Shift and Rotate Instructions
  - Rotate operations
    - Rotate right without carry

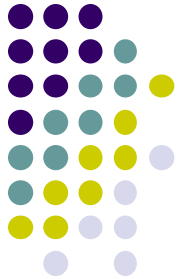


(c) Rotate right without carry RotateR R3, R3, #2

- Shift and Rotate Instructions
  - Rotate operations
    - Rotate right with carry

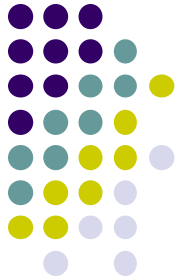


(d) Rotate right with carry RotateRC R3, R3, #2



- Multiplication and Division Instructions
  - Signed integer multiplication of *n*-bit numbers produces a product with as many as 2*n* bits
  - Processor truncates product to fit in a register:
    Multiply R*k*, R*i*, R*j* (R*k* ← [R*i*] × [R*j*])
  - For general case, 2 registers (R*k* and R(*k*+*1*)) may hold result
  - Integer division produces quotient as result:
    Divide R*k*, R*i*, R*j* (R*k* ← [R*i*] / [R*j*])
  - Remainder is discarded or placed in a register R(*k*+*1*)

2.10 CISC Instruction Sets



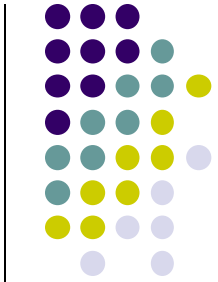
- CISC instruction sets are not constrained to the *load/store architecture*, in which arithmetic and logic operations can be performed only on operands that are in processor registers.
- CISC instructions do not necessarily have to fit into a single word. Some instructions may occupy a single word, but others may span multiple words.
- Most arithmetic and logic instructions use the *two-address format*.
  - Operation destination, source
  - E.g., Add B, A
  - performs the operation B ← [A] + [B] on memory operands.

- The Move instruction includes the functionality of the Load and Store instructions.
  - Move destination, source
  - Example C = A + B (all three operands may be in memory)
    - 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.
    - Move R*i*, A
    - Add R*i*, B
    - Move C, R*i*

Autoincrement and Autodecrement Mode

- Autoincrement mode
  - The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
  - (Ri)+
  - EA = [Ri] Increment Ri
  - Useful for adjusting pointers in loop body:

    Add SUM, (R*i*)+
    MoveByte (R*j*)+, R*k*
  - Increment by 4 for words, and by 1 for bytes



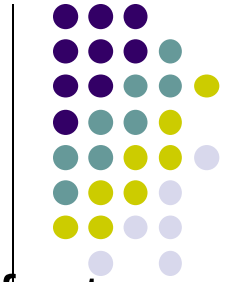
Autoincrement and Autodecrement Mode

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)
- Decrement Ri EA = [Ri]
- Use autoinc. & autodec. for stack operations:

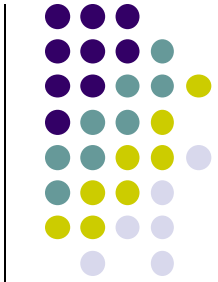
  Move –(SP), NEWITEM (push)
  Move ITEM, (SP)+ (pop)

- Relative Mode
  - The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.
  - EA = [PC] + X
    - X is a signed number
  - Usage
    - Access data operand
    - Specify the target address in branch instructions
    - Example: Branch > 0 Loop
      - The branch target location can be computed by specifying it as an offset from the current value of the program counter.



- Condition Codes
  - Processor can maintain information on results to affect subsequent conditional branches
  - Results from arithmetic/comparison & Move
  - Condition code flags in a status register:

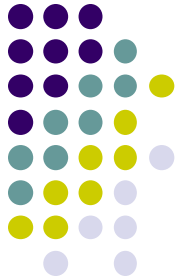
N	(negative)	1 if result negative, else 0
Z	(zero)	1 if result zero, else 0
V	(overflow)	1 if overflow occurs, else 0
C	(carry)	1 if carry-out occurs, else 0



- Branches Using Condition Codes
  - CISC branches check condition code flags
  - For example, decrementing a register causes N and Z flags to be cleared if result is *not* zero
  - A branch to check logic condition $N + Z = 0$:

    Branch>0 LOOP
  - Other branches test conditions for $<$, $=$, $\neq$, $\leq$, $\geq$
  - Also Branch_if_overflow and Branch_if_carry
  - Consider CISC-style list-summing program

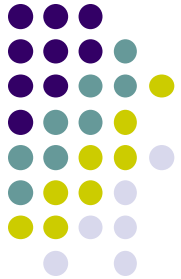
- Branches Using Condition Codes



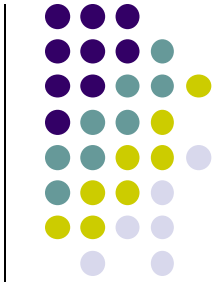
```
        Move      R2, N        Load the size of the list.
        Clear     R3           Initialize sum to 0.
        Move      R4, #NUM1    Load address of the first number.
LOOP:   Add       R3, (R4)+    Add the next number to sum.
        Subtract  R2, #1       Decrement the counter.
        Branch>0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

2.11 RISC and CISC Styles

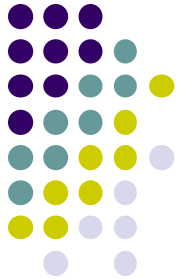
- RISC characteristics include:
  - simple addressing modes
  - all instructions fitting in a single word
  - fewer total instructions
  - arithmetic/logic operations on registers
  - load/store architecture for data transfers
  - more instructions executed per program
- Simpler instructions make it easier to design faster hardware (e.g., use of pipelining)



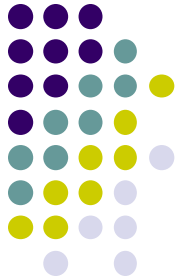
- CISC characteristics include:
  - more complex addressing modes
  - instructions spanning more than one word
  - more instructions for complex tasks
  - arithmetic/logic operations on memory
  - memory-to-memory data transfers
  - fewer instructions executed per program
- Complexity makes it somewhat more difficult to design fast hardware, but still possible



- Summary
  - Instruction and instruction sequencing
    - Assembly-language Notation
    - RISC instruction sets
    - Instruction Execution
      - Straight-Line Sequencing and Branching
  - Instruction Formats
    - Instruction Representation
    - Common Instruction Address Field Formats
      - Zero-, One-, Two-, and Three-address Instruction
    - Opcode Format (Expanding Opcode)



- Summary (ctd.)
  - Addressing Modes
    - The different ways in which the location of an operand is specified in an instruction.
    - Typical RISC Addressing Modes
  - Stack and Subroutine
  - Additional instructions
    - Logical, Shift & Rotate, Multiplication and Division Instructions
  - CISC instruction sets
    - Autoincrement, Autodecrement and Relative mode
    - Condition Codes
  - RISC vs. CISC styles



Homework

- 2.4, 2.9
- Assume that a computer's instruction length is 16-bit, and its operand address is 6-bit. Suppose the designers need two-address instructions, one-address instructions and zero-address instructions. How should we design the instruction format? And specify the numbers of each type of instruction can be designed.
- Assume that a computer's word-length is 16-bit, the capacity of main memory is 64K words. If it employs single-word-length one-address instructions and contains 64 instructions. Design an instruction format with addressing modes of direct, indirect, index, and relative.