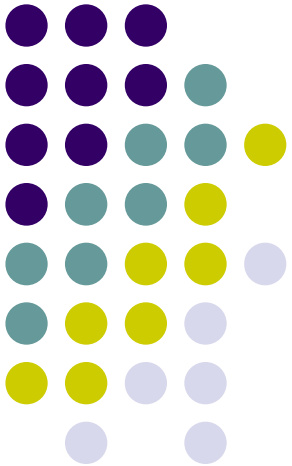
Computer Organization

18CS34

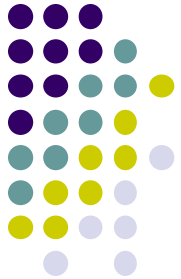
Dr. Sudhamani M J

Asst. Professor

Department of CSE



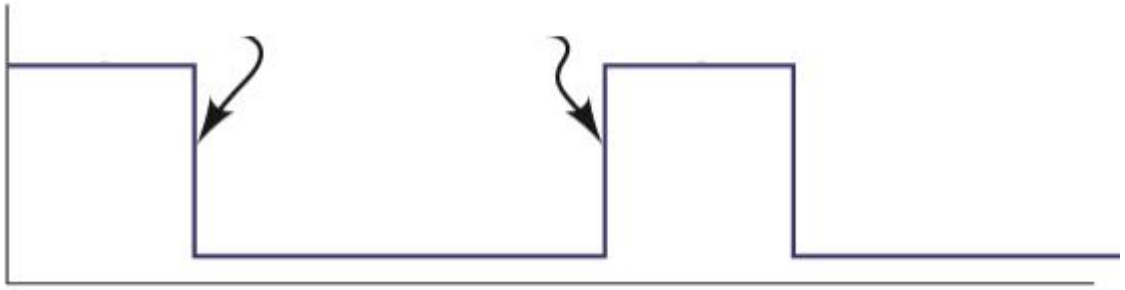
Introduction



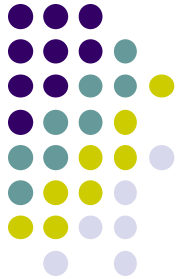
- **Assessment Questions**
  - What are the two numeric digits used to represent states in a digital system?
  - What are the two terms used to represent the two logic levels?
  - What is the abbreviation for binary digit?

Introduction

- **Digital Signals**
  - The transition between the two states is called an edge.
  - At dawn, when the signal proceeds from HIGH to LOW, it is considered a falling edge, or negative edge.



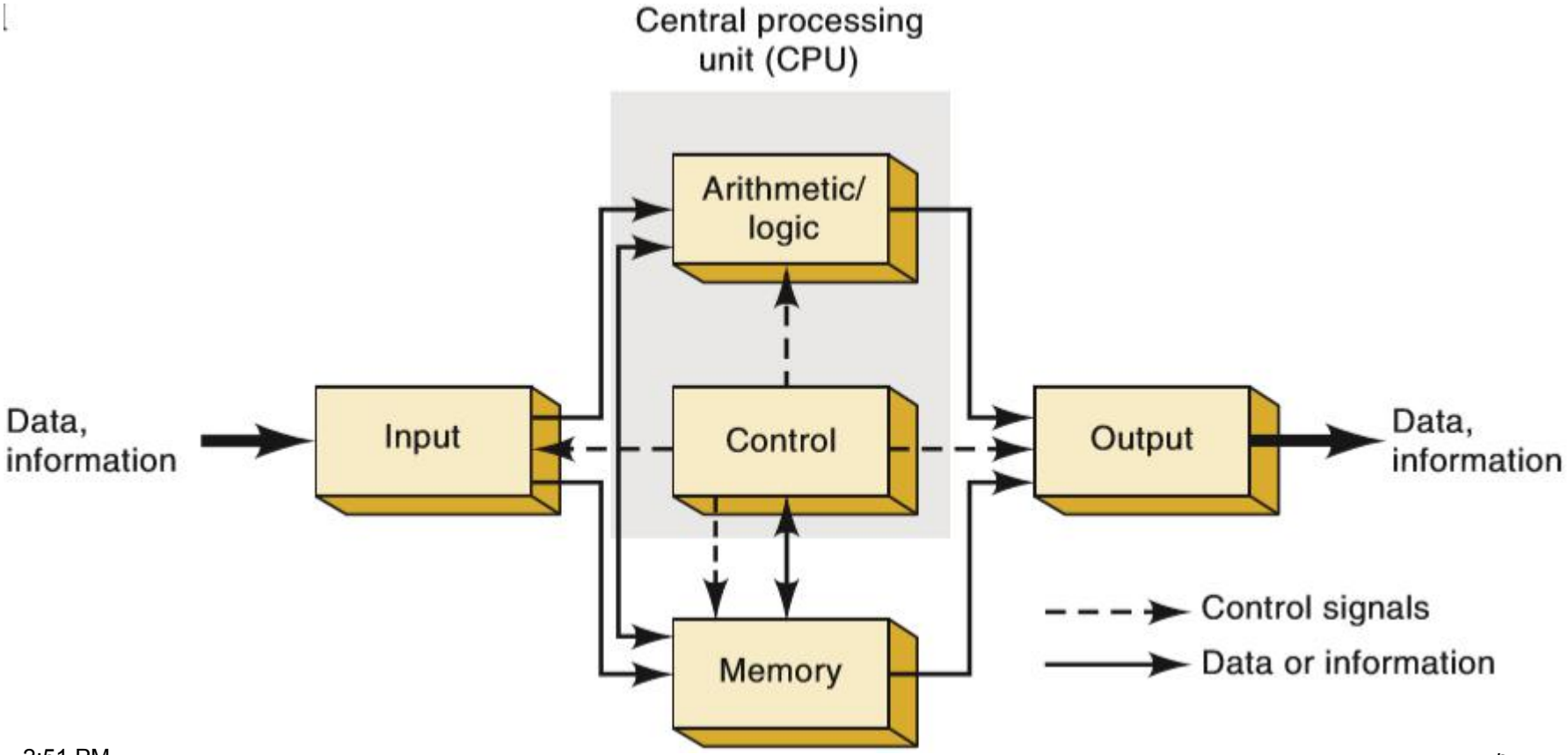
Introduction



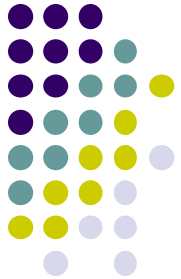
- **Advantages of Digital Techniques**
  - Digital systems are generally easier to design
  - Information storage is easy
  - Accuracy and precision are easier to maintain throughout the system
  - Operations can be programmed
  - Digital circuits are less affected by noise
  - More digital circuitry can be fabricated on IC chips

Introduction

- **Digital Computers**

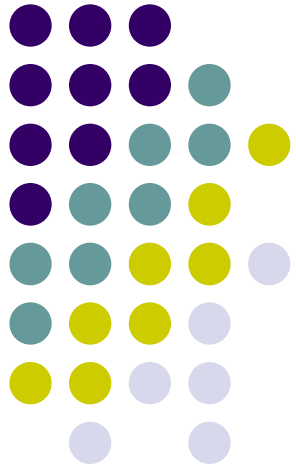


Introduction

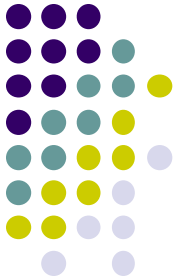


- Digital Computers
  - Major Parts of a Computer
    - Input unit
    - Output unit
    - Memory unit
    - Arithmetic/logic unit
    - Control unit

Module-1. Basic Structure of Computers, Machine Instructions and Programs

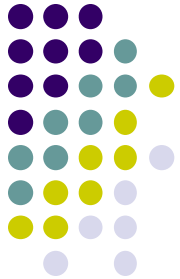


Referred Books



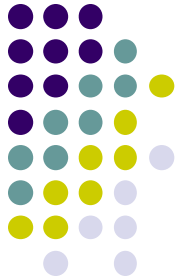
- **Text Books:**
  - Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.
  - Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian : Computer Organization and Embedded Systems, 6th Edition, Tata McGraw Hill, 2012.
- **Reference Books:**
  - William Stallings: Computer Organization & Architecture, 9th Edition, Pearson, 2015.

What You Will Learn



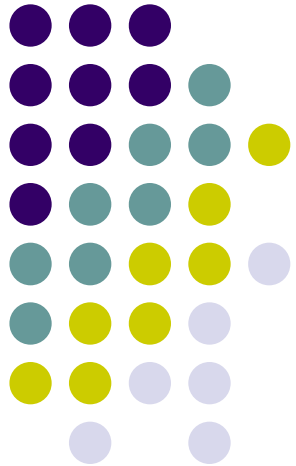
- How programs are translated into the machine language
  - And how the hardware executes them
- The hardware/software interface
- What determines program performance
  - And how it can be improved
- How hardware designers improve performance

Understanding Performance



- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast I/O operations are executed

Functional Units



Functional Units

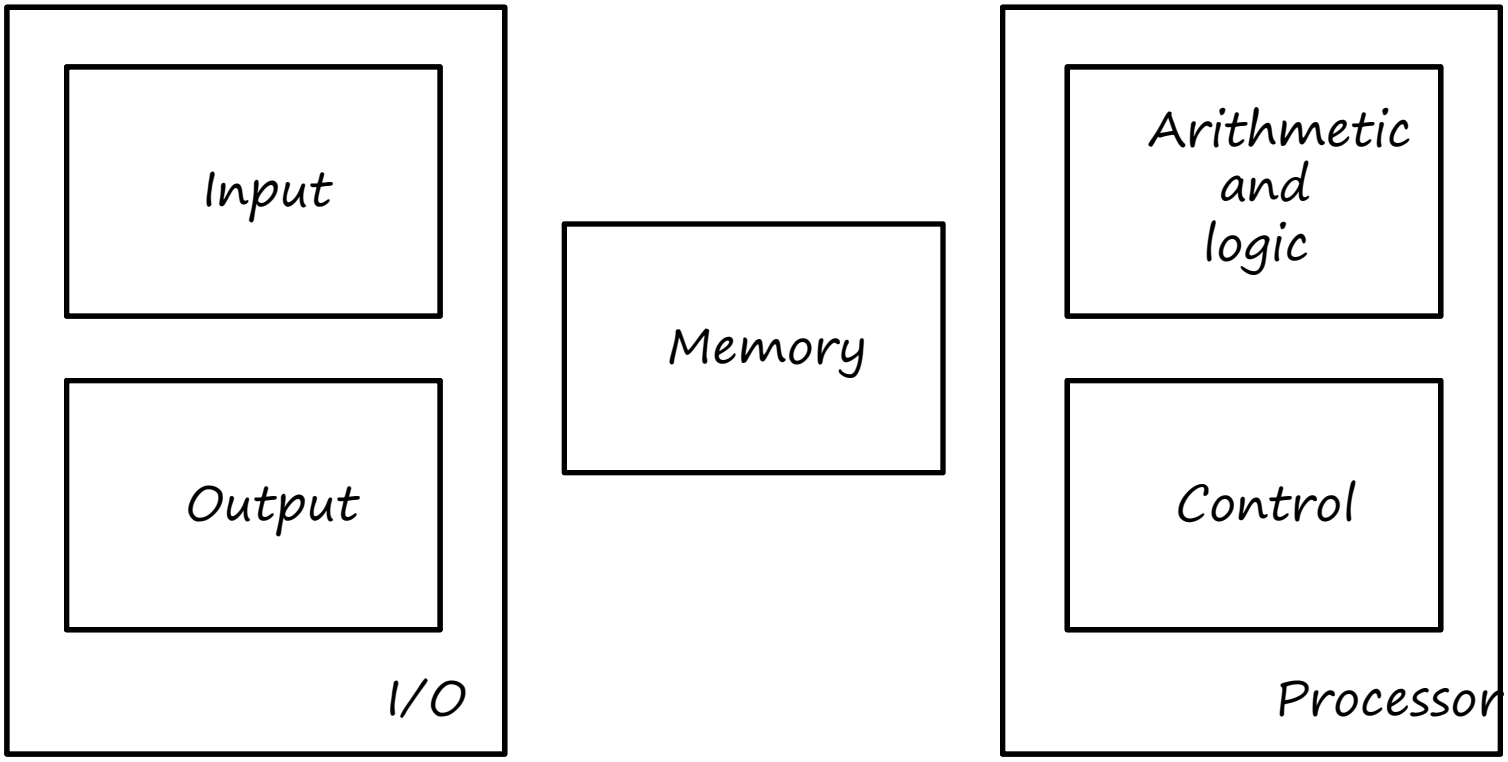
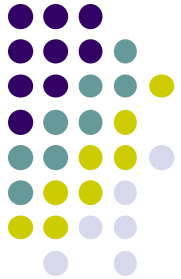


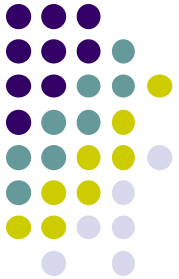
Figure 1.1. Basic functional units of a computer.

Information Handled by a Computer



- Instructions/machine instructions
  - Govern the transfer of information within a computer as well as between the computer and its I/O devices
  - Specify the arithmetic and logic operations to be performed
  - Program
- Data
  - Used as operands by the instructions
  - Source program
- Encoded in binary code – 0 and 1

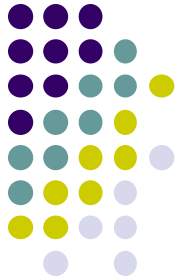
Memory Unit



- Store programs and data
- Two classes of storage

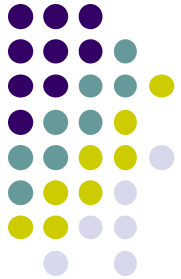
- Primary storage
  - Fast
  - Programs must be stored in memory while they are being executed
  - Large number of semiconductor storage cells
  - Processed in words
  - Address
  - RAM and memory access time
  - Memory hierarchy – cache, main memory
- Secondary storage – larger and cheaper

Arithmetic and Logic Unit (ALU)



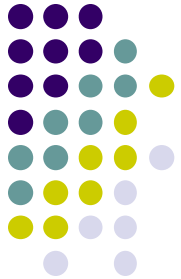
- Most computer operations are executed in ALU of the processor.
  - Load the operands into memory
  - bring them to the processor
  - perform operation in ALU
  - store the result back to memory or retain in the processor.
- Registers
- Fast control of ALU

Control Unit



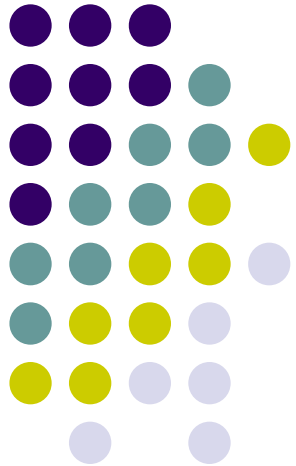
- All computer operations are controlled by the control unit.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Control unit is usually distributed throughout the machine instead of standing alone.

The operations of a computer

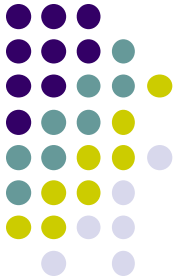


- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched under program control into an arithmetic and logic unit, where it is processed.
- Processed information leaves the computer through an output unit.
- All activities in the computer are directed by the control unit.

Basic Operational Concepts

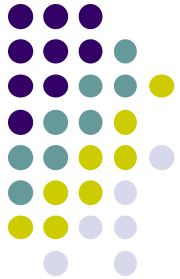


Review



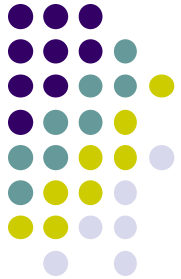
- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

A Typical Instruction



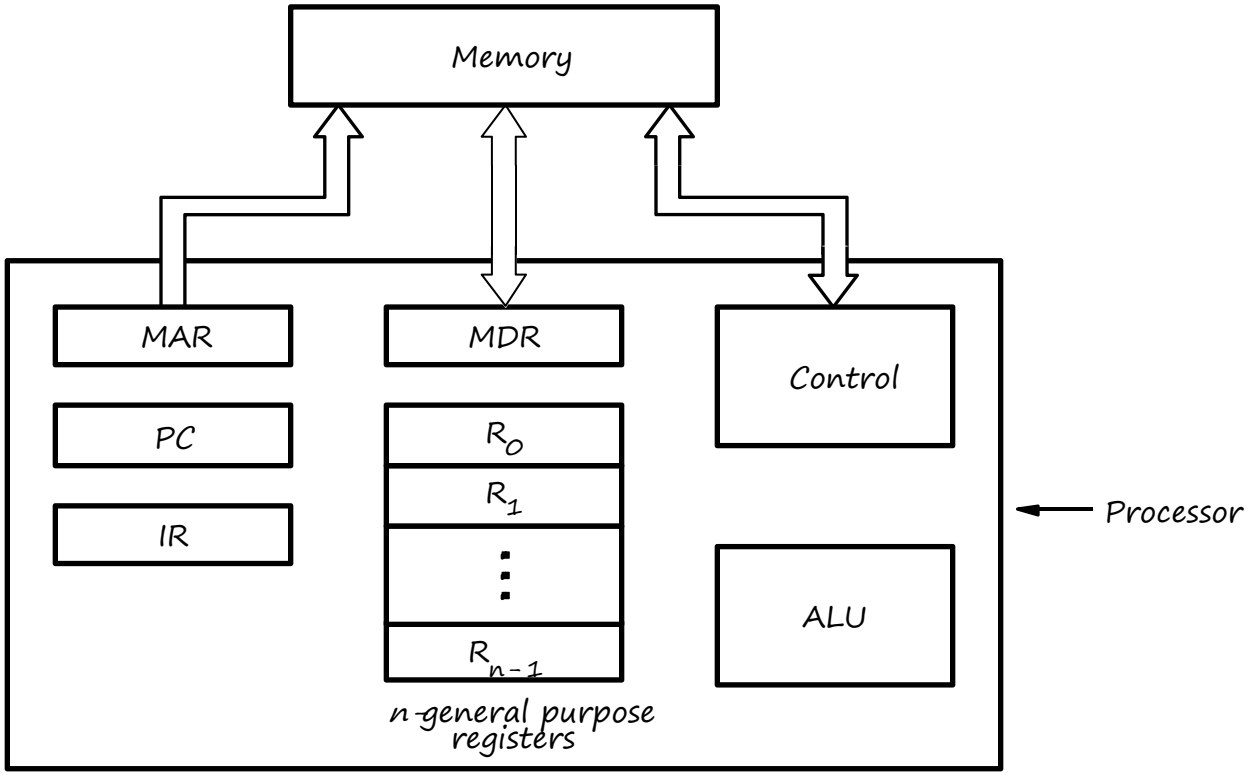
- Add LOCA, R0
- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

Separate Memory Access and ALU Operation



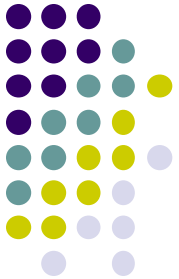
- Load LOCA, R1
- Add R1, R0
- Inc R5
- Sub 5,R2

Connection Between the Processor and the Memory



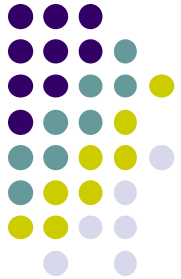
Connections between the processor and the memory.

Registers



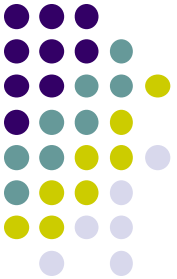
- Instruction register (IR)
- Program counter (PC)
- General-purpose register (R_0 – R_{n-1})
- Memory address register (MAR)
- Memory data register (MDR)

Typical Operating Steps



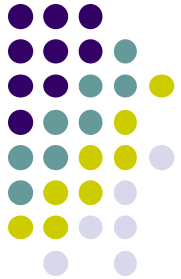
- Programs reside in the memory through input devices
- PC is set to point to the first instruction
- The contents of PC are transferred to MAR
- A Read signal is sent to the memory
- The first instruction is read out and loaded into MDR
- The contents of MDR are transferred to IR
- Decode and execute the instruction

Typical Operating Steps (Cont')



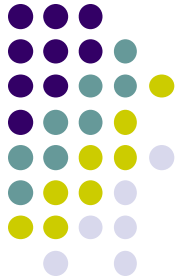
- Get operands for ALU
  - General-purpose register
  - Memory (address to MAR – Read – MDR to ALU)
- Perform operation in ALU
- Store the result back
  - To general-purpose register
  - To memory (address to MAR, result to MDR – Write)
- During the execution, PC is incremented to the next instruction

Interrupt



- Normal execution of programs may be preempted if some device requires urgent servicing.
- The normal execution of the current program must be interrupted – the device raises an *interrupt* signal.
- Interrupt-service routine
- Current system information backup and restore (PC, general-purpose registers, control information, specific information)

Bus Structures



- There are many ways to connect different parts inside a computer together.
- A group of lines that serves as a connecting path for several devices is called a *bus*.
- Address/data/control

Bus Structure

- Single-bus

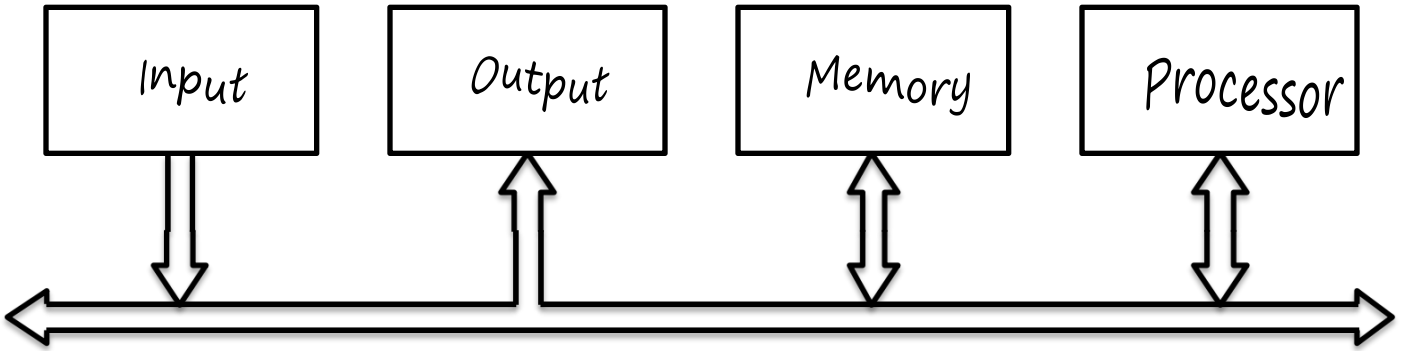
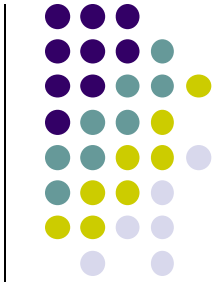


Figure 1.3. Single-bus structure.

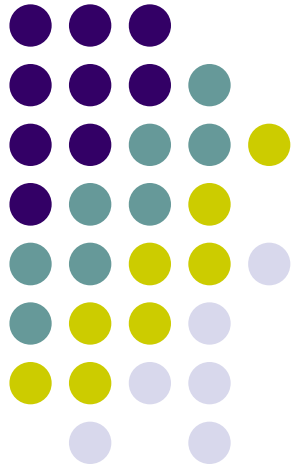
- Multiple Buses

Speed Issue

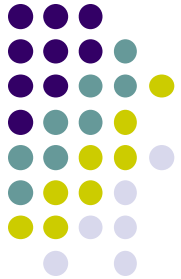


- Different devices have different transfer/operate speed.
- If the speed of bus is bounded by the slowest device connected to it, the efficiency will be very low.
- How to solve this?
- A common approach – use buffers.

Performance



Performance



- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
  - Hardware design
  - Instruction set
  - Compiler

Performance

- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions.

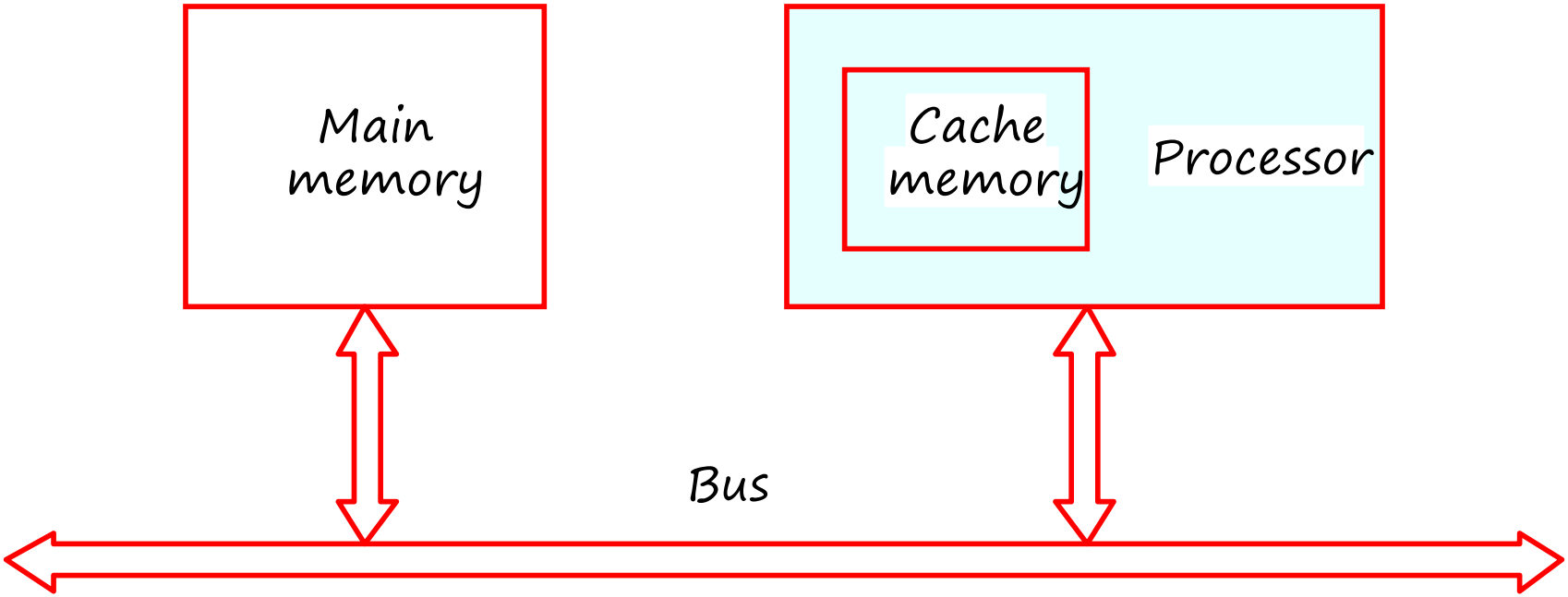
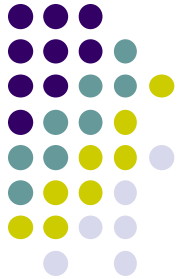


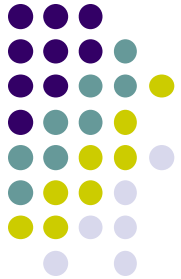
Figure 1.5.The processor cache.

Performance

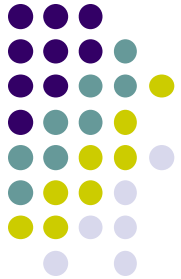


- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip.
  - Speed
  - Cost
  - Memory management

Processor Clock



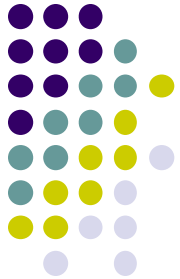
- Clock, clock cycle (P), and clock rate (R=1/P)
- The execution of each instruction is divided into several steps (Basic Steps), each of which completes in one clock cycle.
- Hertz – cycles per second



- What is cycles per instruction?

ex : Integer multiplication/division (32/64-bit) 7 cycles

Basic Performance Equation



- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate

$$T = \frac{N \times S}{R}$$

- How to improve T?
- Reduce N and S, Increase R, but these affect one another

Performance Measurement

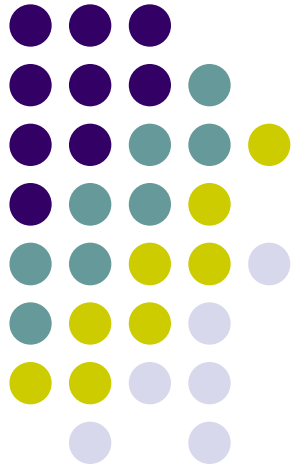
- Measure computer performance using benchmark programs.
- System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
- Reference computer

$$SPEC\ \ rating = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

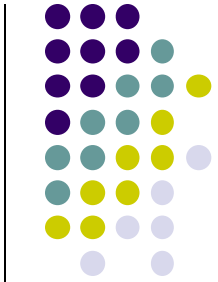
$$SPEC\ \ rating = \left(\prod_{i=1}^{n} SPEC_i\right)^{\frac{1}{n}}$$

- n is the number of program in the suite

Machine Instructions and Programs

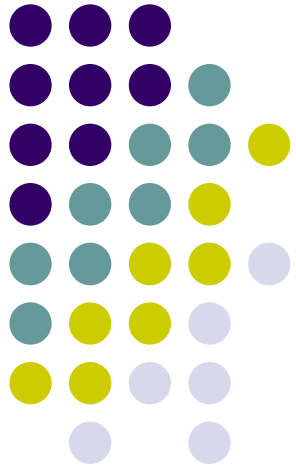


Objectives



- Machine instructions and program execution, including branching and subroutine call and return operations.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

Memory Locations, Addresses, and Operations



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.

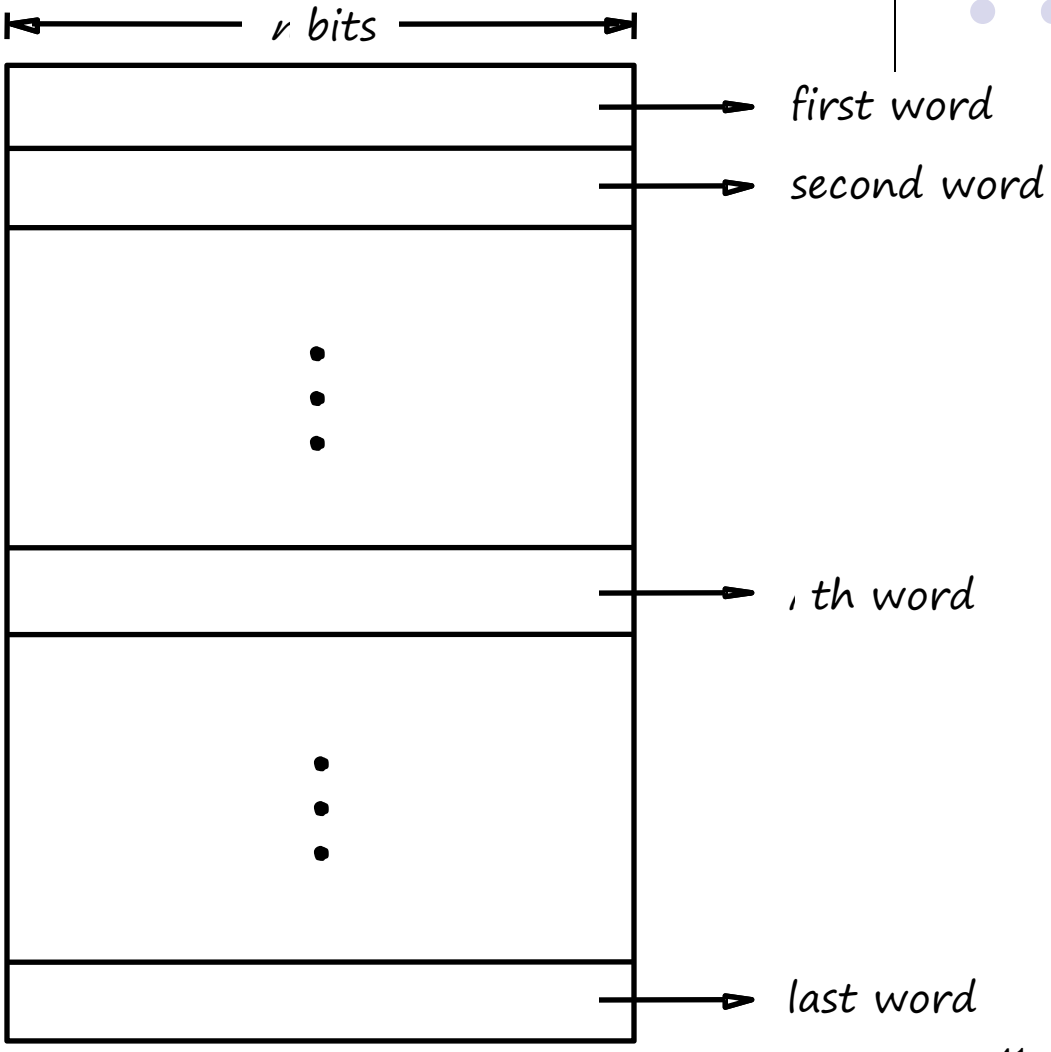
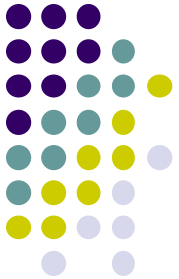
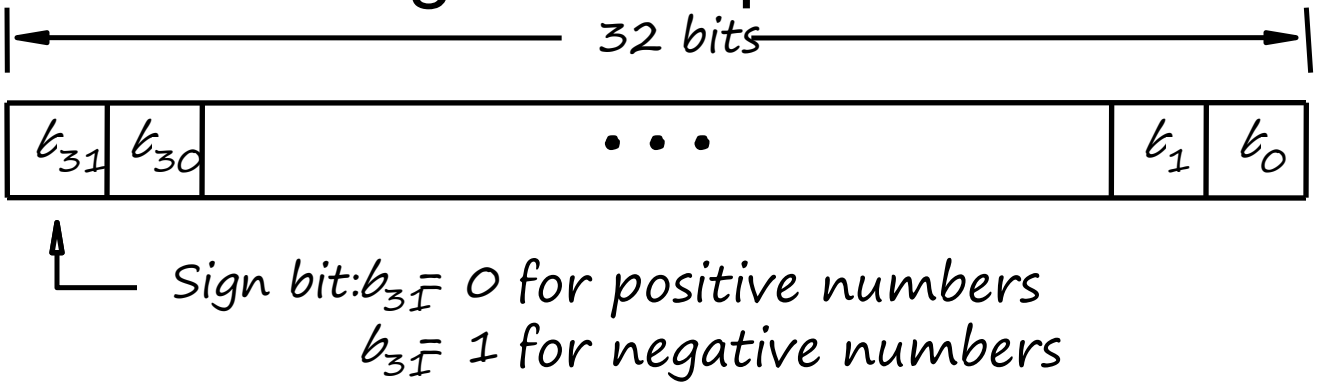


Fig: Memory words.

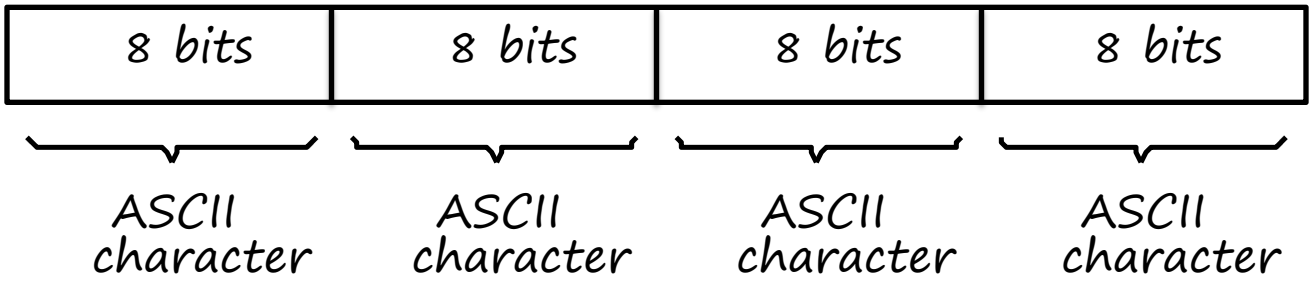
Memory Location, Addresses, and Operation



- 32-bit word length example

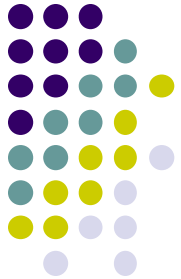


(a) A signed integer



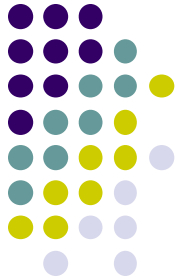
(b) Four characters

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

Note : 1K(kilo)=2^{10} , 2^{30} = 1G , 1T(tera)=2^{40}



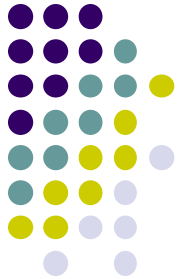
	0
	1

	00
	01
	10
	11

	000
	001
	010
	011
	100
	101
	110
	111

n bits address = 0 to 2^{n-1} locations

Memory Location, Addresses, and Operation

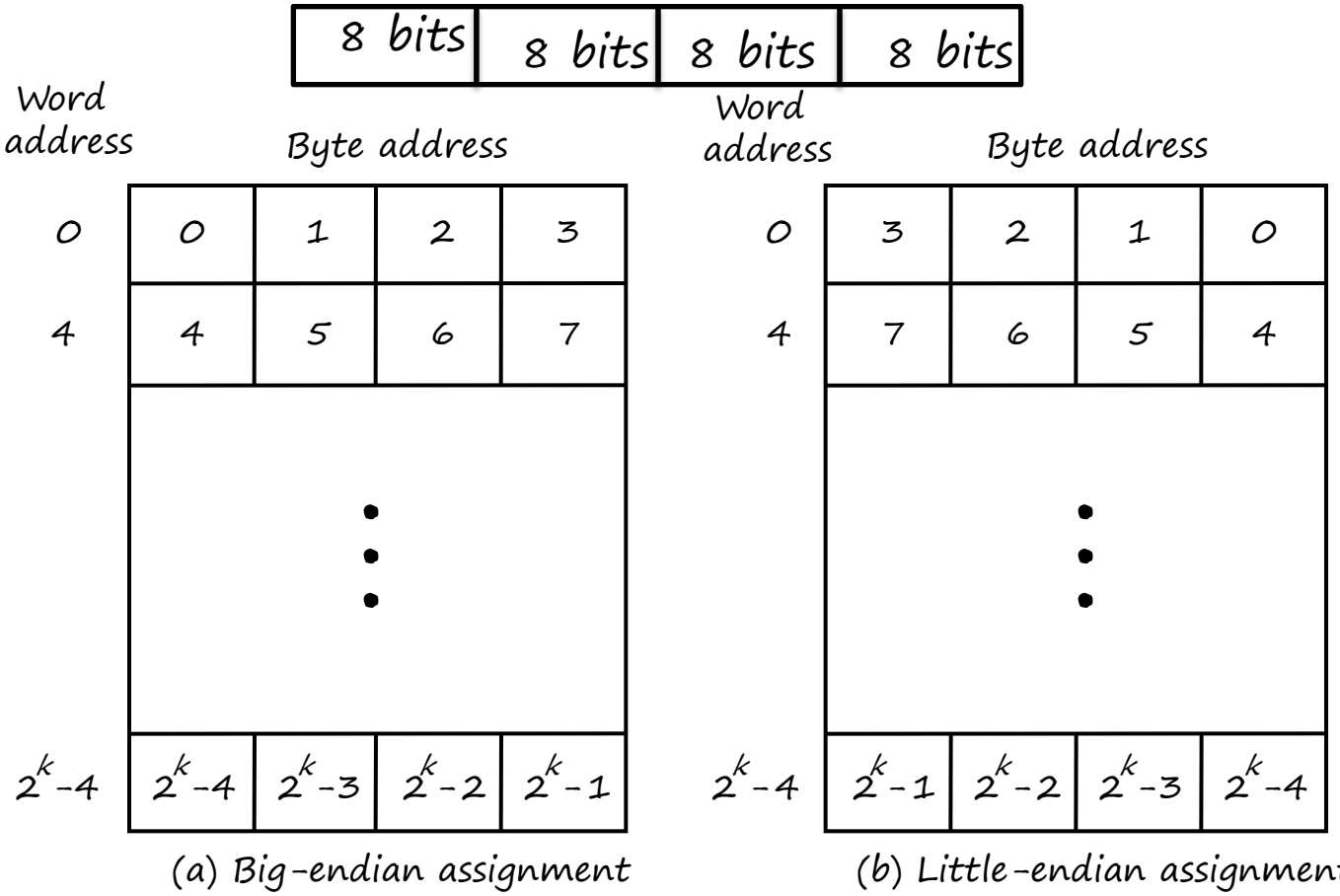


- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits,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

Big-Endian and Little-Endian Assignments

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

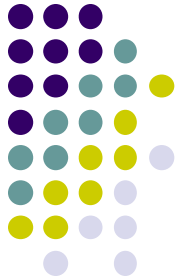
Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word



(a) Big-endian assignment (b) Little-endian assignment

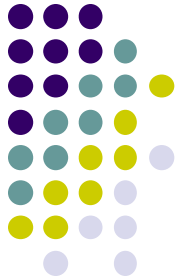
Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation



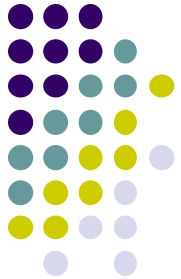
- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….
- Access numbers, characters, and character strings

Memory Operation

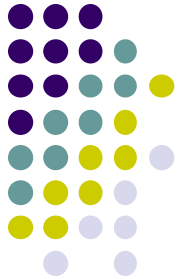


- Load (or Read or Fetch)
  - Copy the content. The memory content doesn't change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

Instruction and Instruction Sequencing

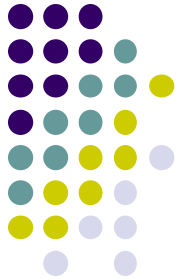


“Must-Perform” Operations



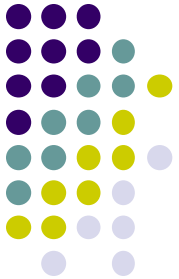
- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation



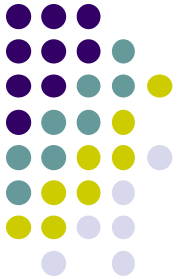
- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation



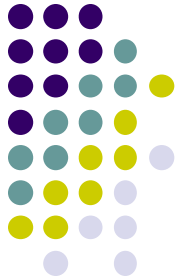
- Represent machine instructions and programs.
- Move LOC, R1 ; R1←[LOC]
- Add R1, R2, R3 ; R3 ←[R1]+[R2]

CPU Organization



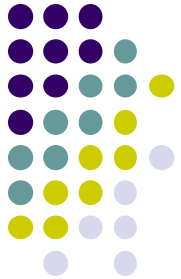
- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats



- Three-Address Instructions
  - ADD R2, R3, R1 ;R1 ← [R2] + [R3]
- Two-Address Instructions
  - ADD R2, R1 ;R1 ← [R1] + [R2]
- One-Address Instructions
  - ADD y ;AC ← [AC] + M[y]
- Zero-Address Instructions
  - ADD ;TOS ← [TOS] + [TOS – 1]

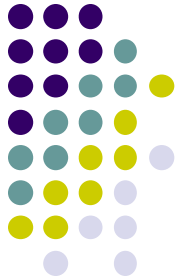
Instruction Formats



Example: Evaluate (A+B) * (C+D)

- Three-Address
  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

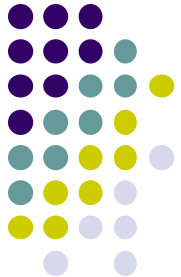


Example: Evaluate (A+B) * (C+D)

- Two-Address

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]

Instruction Formats

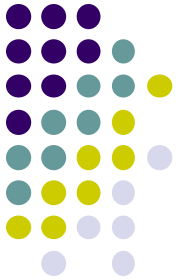


Example: Evaluate (A+B) * (C+D)

- One-Address

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

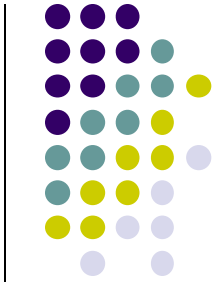


Example: Evaluate (A+B) * (C+D)

- Zero-Address

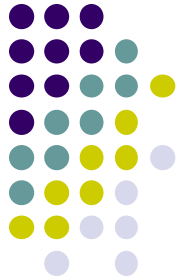
```
1. PUSH   A      ; TOS ← [A]
2. PUSH   B      ; TOS ← [B]
3. ADD           ; TOS ← [A + B]
4. PUSH   C      ; TOS ← [C]
5. PUSH   D      ; TOS ← [D]
6. ADD           ; TOS ← [C + D]
7. MUL           ; TOS ← [C+D]*[A+B]
8. POP    X      ; M[X] ← [TOS]
```

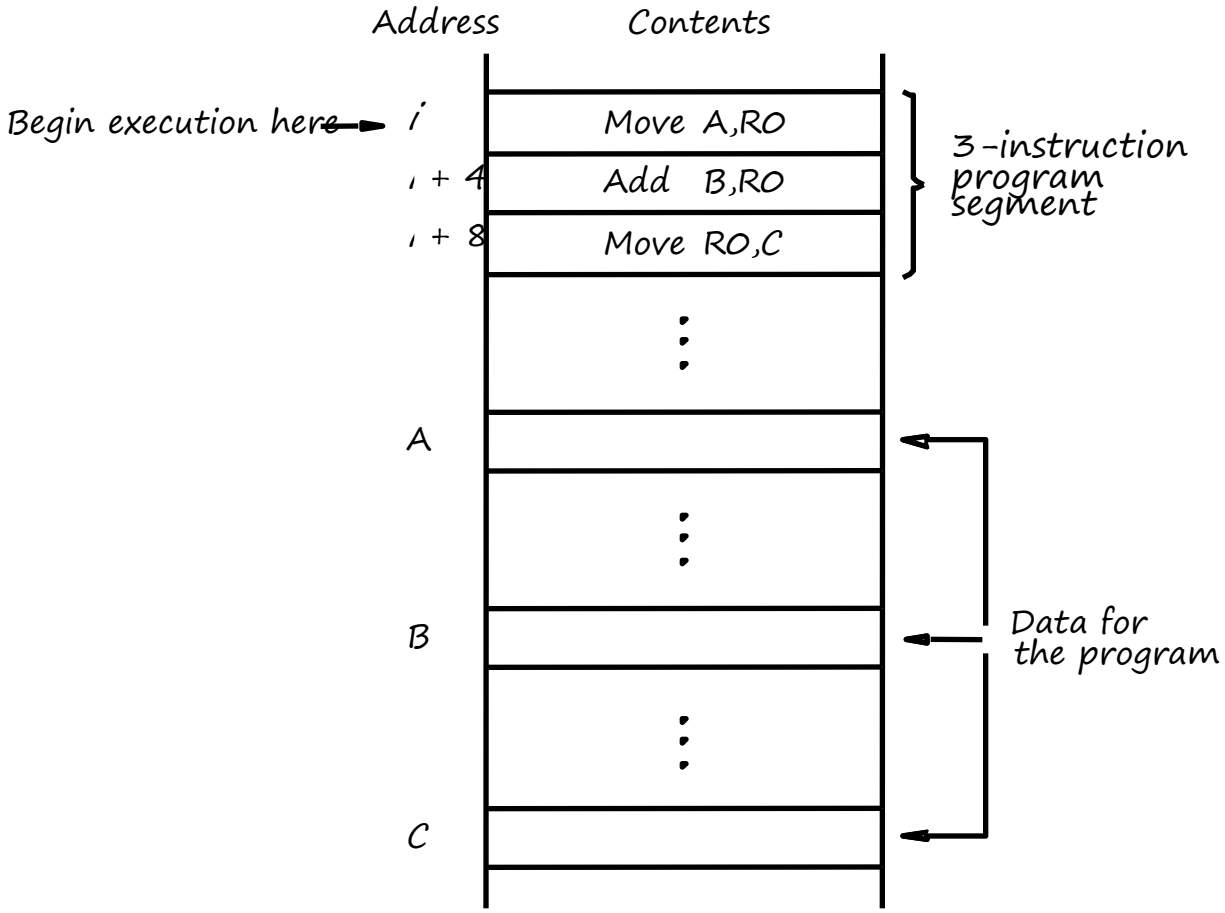
Using Registers



- 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 and Straight-Line Sequencing





Assumptions:
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure
-Instruction fetch
-Instruction execute

Figure 2.8. A program for $C \leftarrow [A] + [B]$.

Branching

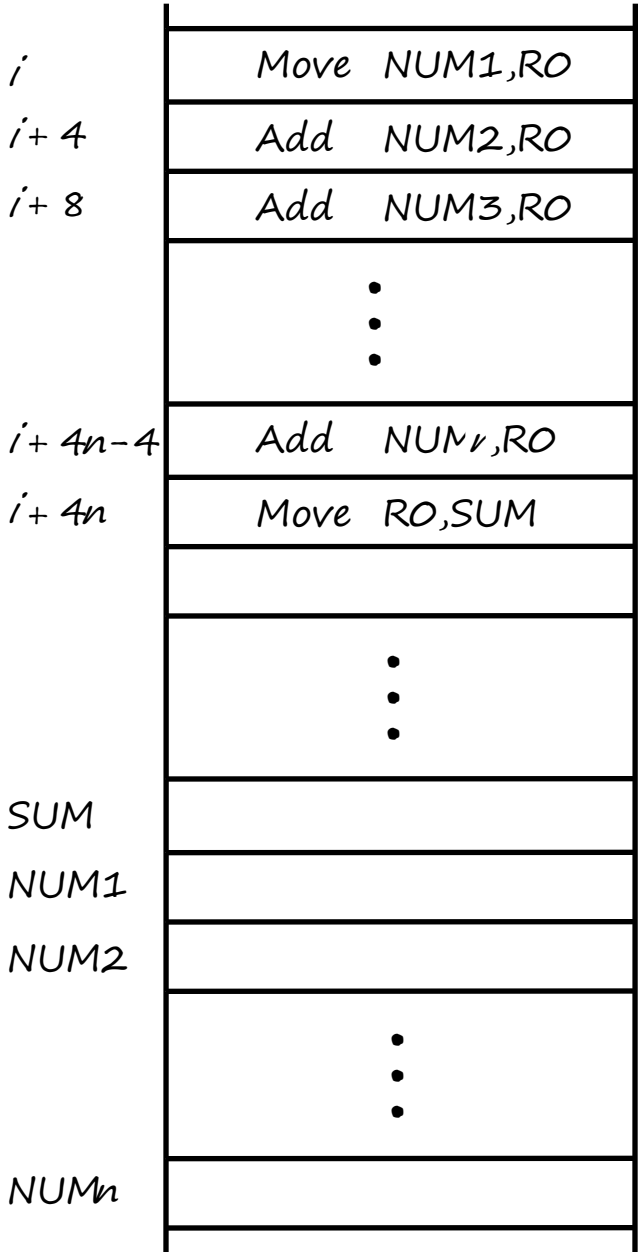


Figure 2.9. A straight-line program for adding n numbers.

Branching

Branch target

Conditional branch

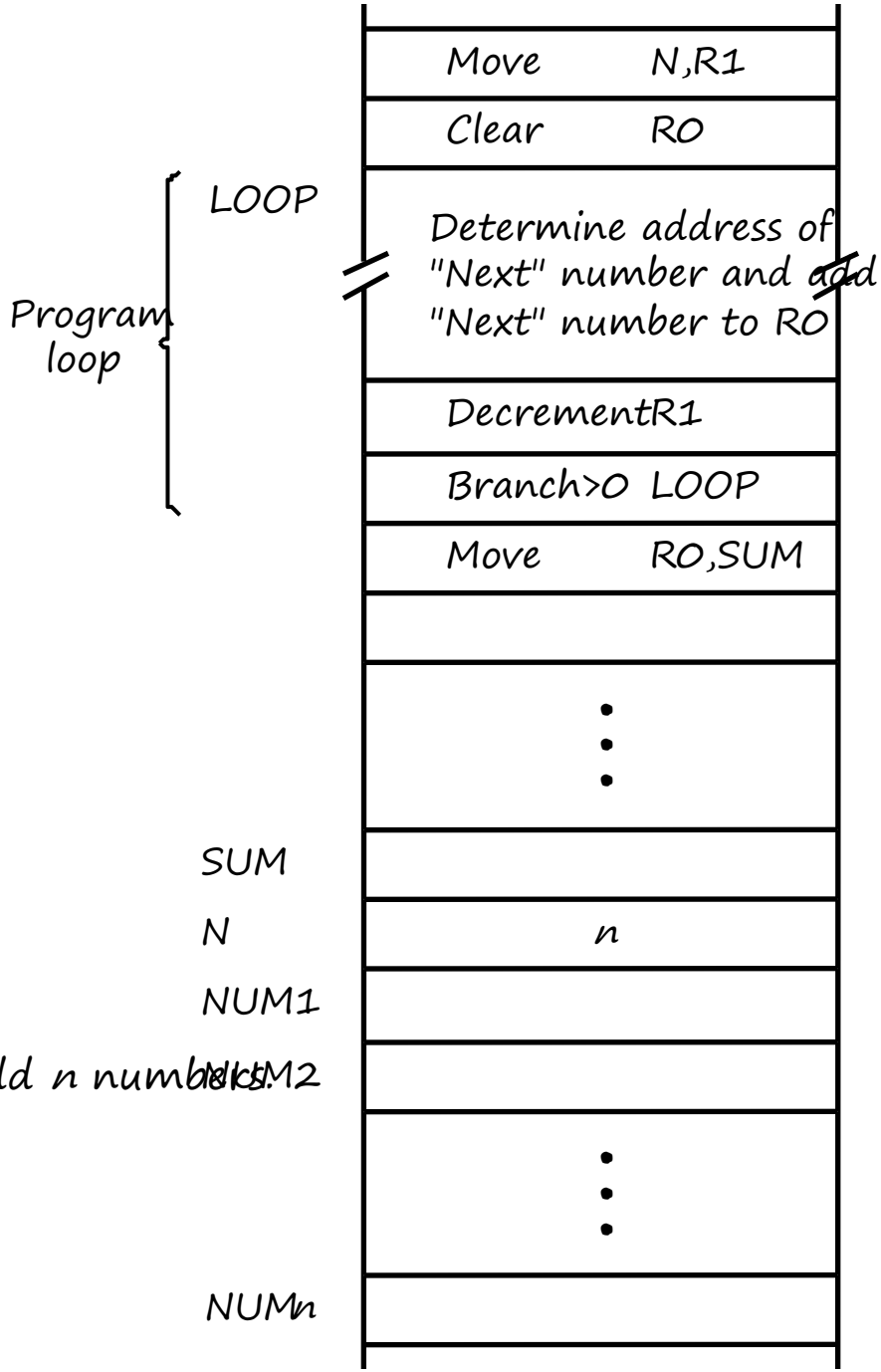
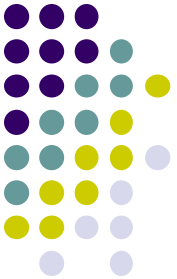


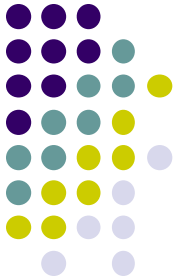
Figure 2.10. Using a loop to add n numbers.

Condition Codes

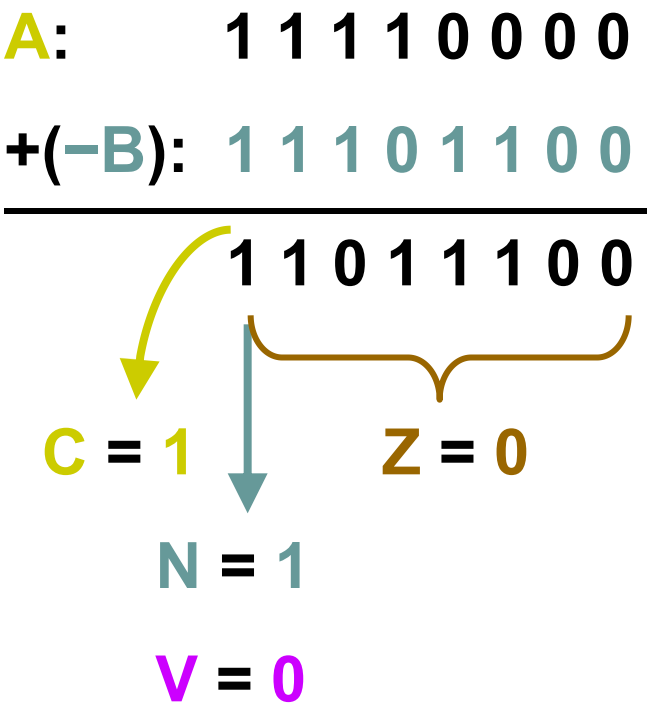


- Condition code flags (bits)
- Condition code register / status register
  - N (negative)
  - Z (zero)
  - V (overflow)
  - C (carry)
- Different instructions affect different flags

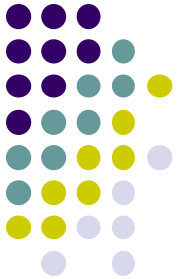
Conditional Branch Instructions



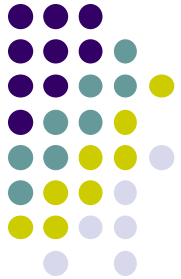
- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



Addressing Modes



Generating Memory Addresses



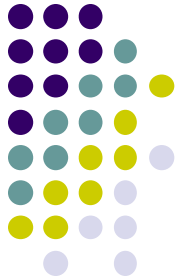
- How to specify the address of branch target?
- Can we give the memory operand address directly in a single Add instruction in the loop?
- Use a register to hold the address of NUM1; then increment by 4 on each pass through the loop.

Addressing Modes

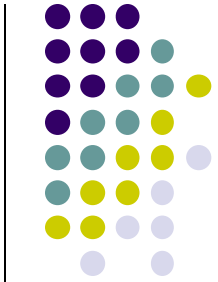
- The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri) (LOC)	EA = [Ri] 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]

Effective Address (EA)



- In the addressing modes that follow, the instruction does not give the operand or its address explicitly. Instead, it provides information from which an *effective address* (EA) can be derived by the processor when the instruction is executed.
- The effective address is then used to access the operand.



- immediate addressing

ex: Add #10,R1

Mov #23H, A

Mov #$23,LOC

- Register Addressing

ex: Add R4,R6

Sub R7,R8

- Direct addressing (absolute addressing)

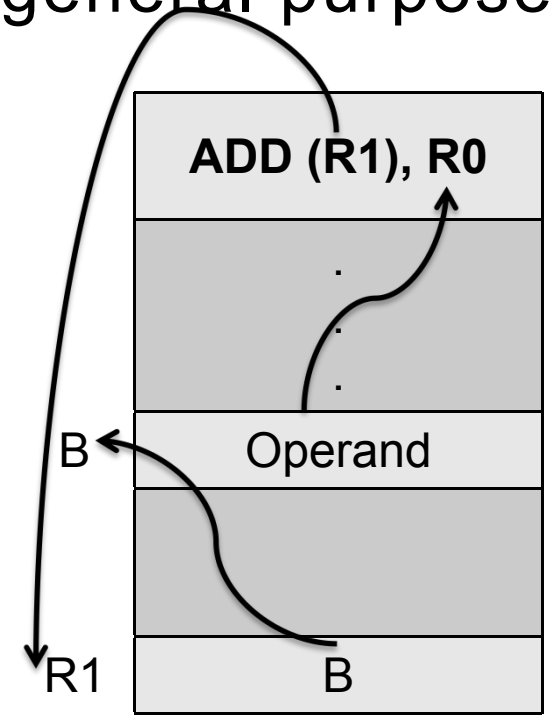
Mov LOC,R1

ADD R0,SUM

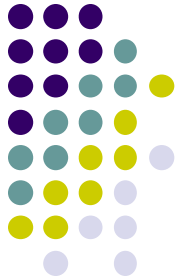
Addressing Modes
Indirect Addressing

- Indirect Addressing
  - Indirection and Pointer
  - Indirect addressing through a general purpose register.
  - Indicate the register (e.g. R1) that holds the address of the variable (e.g. B) that holds the operand

    ADD (R1), R0
  - The register or memory location that contain the address of an operand is called a pointer

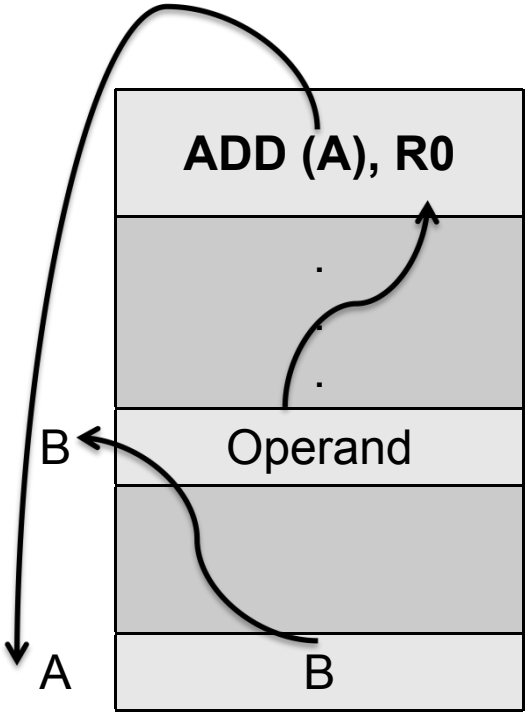


Addressing Modes
Indirect Addressing



- Indirect Addressing
  - Indirect addressing through a memory addressing.
  - Indicate the memory variable (e.g. A)that holds the address of the variable (e.g. B) that holds the operand

    ADD (A), R0

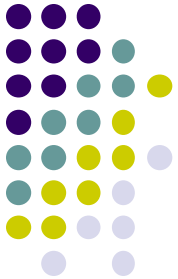


Indirect Addressing Example

- Addition of N numbers

```
1.              Move N,R1           ; N = Numbers to add
2.              Move  #NUM1,R2      ; R2= Address of 1st no.
3.              Clear R0            ; R0 = 00
4.    Loop :    Add (R2), R0        ; R0 = [NUM1] + [R0]
5.              Add #4, R2           ; R2= To point to the  next
                                    ; number
6.              Decrement R1        ; R1 = [R1] -1
7.              Branch>0 Loop       ; Check if R1>0 or not if
                                              ; yes go to Loop
8.               Move R0, SUM       ; SUM= Sum of all no.
```

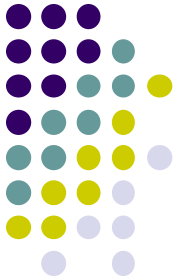
Example



- Addition of N numbers

```
1.              Move N,R1           ; N = 5
2.              Move #NUM1,R2       ; R2= 10000H
3.              Clear R0            ; R0 = 00
4.   Loop :     Add (R2), R0        ; R0 = 10 + 00 = 10
5.              Add #4, R2          ; R2 = 10004H
6.              Decrement R1        ; R1 = 4
7.              Branch>0 Loop       ; Check if R1>0 if
                                    ; yes go to Loop
8.               Move R0, SUM       ; SUM=
```

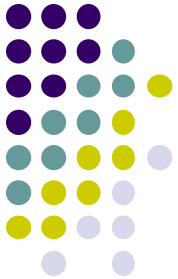
Example



- Addition of N numbers

```
1.              Move N,R1           ; N = 5
2.              Move #NUM1,R2       ; R2= 10000H
3.              Clear R0            ; R0 = 00
4.   Loop :     Add (R2), R0        ; R0 = 20 + 10 = 30
5.              Add #4, R2          ; R2 = 10008H
6.              Decrement R1        ; R1 = 3
7.              Branch>0 Loop       ; Check if R1>0 if
                                    ; yes go to Loop
8.               Move R0, SUM       ; SUM=
```

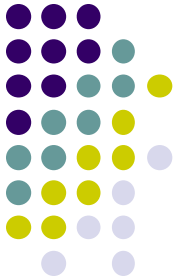
Example



- Addition of N numbers

```
1.              Move N,R1           ; N = 5
2.              Move #NUM1,R2       ; R2= 10000H
3.              Clear R0            ; R0 = 00
4.   Loop :     Add (R2), R0        ; R0 = 30 + 30 = 60
5.              Add #4, R2          ; R2 = 1000CH
6.              Decrement R1        ; R1 = 2
7.              Branch>0 Loop       ; Check if R1>0 if
                                    ; yes go to Loop
8.               Move R0, SUM       ; SUM=
```

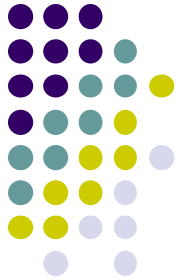
Example



- Addition of N numbers

```
1.              Move N,R1           ; N = 5
2.              Move #NUM1,R2       ; R2= 10000H
3.              Clear R0            ; R0 = 00
4.   Loop :     Add (R2), R0        ; R0 = 40 + 60 = 100
5.              Add #4, R2          ; R2 = 10010H
6.              Decrement R1        ; R1 = 1
7.              Branch>0 Loop       ; Check if R1>0 if
                                    ; yes go to Loop
8.               Move R0, SUM       ; SUM=
```

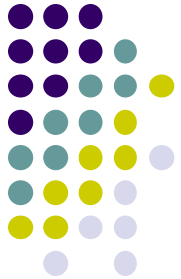
Example



- Addition of N numbers

```
1.              Move N,R1           ; N = 5
2.              Move #NUM1,R2       ; R2= 10000H
3.              Clear R0            ; R0 = 00
4.   Loop :     Add (R2), R0        ; R0 = 50 + 100 = 150
5.              Add #4, R2          ; R2 = 10014H
6.              Decrement R1        ; R1 = 0
7.              Branch>0 Loop       ; Check if R1>0 if
                                    ; yes go to Loop
8.               Move R0, SUM       ; SUM =
```

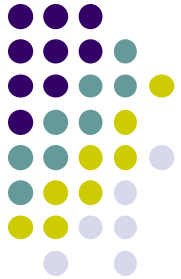
Example



- Addition of N numbers

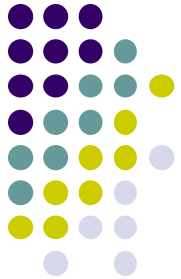
```
1.              Move N,R1           ; N = 5
2.              Move #NUM1,R2       ; R2= 10000H
3.              Clear R0            ; R0 = 00
4.   Loop :     Add (R2), R0        ; R0 = 50 + 100 = 150
5.              Add #4, R2          ; R2 = 10014H
6.              Decrement R1        ; R1 = 0
7.              Branch>0 Loop       ; Check if R1>0 if
                                    ; yes go to Loop
8.               Move R0, SUM       ; SUM = 150
```

Addressing Modes
Indexing and Arrays



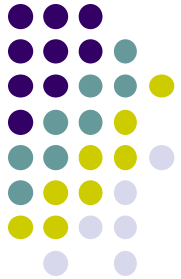
- Indexing and Array
- The EA of the operand is generated by adding a constant value to the contents of a register.
- X(Ri) ; EA= X + (Ri) X= Signed number
- X defined as offset or displacement

Addressing Modes
Indexing and Arrays



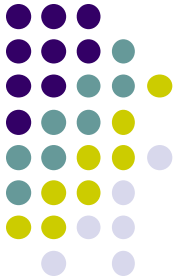
- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register
  - $X(R_i)$: $EA = X + [R_i]$
  - The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
  - If X is shorter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

Addressing Modes
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.
- 2D Array
  - (Ri, Rj) so EA = [Ri] + [Rj]
  - Rj is called the base register
- 3D Array
  - X(Ri, Rj) so EA = X + [Ri] + [Rj]

Addressing Modes
Indexing and Arrays



Address	Memory
	Add 20(R1), R2
	. . . .
10000H	
Offset=20 ↓	. . . .
10020H	Operand

R1	10000H

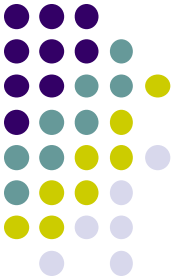
Offset is given as a Constant

Address	Memory
	Add 10000H(R1), R2
	. . . .
10000H	
Offset=20 ↓	. . . .
10020H	Operand

R1	20H

Offset is in the index register

Addressing Modes Indexing and Arrays

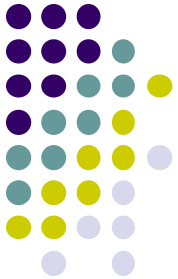


- Array
- E.g. List of students marks

Address	Memory	Comments
N	n	No. of students
LIST	Student ID1	Student 1
LIST+4	Test 1	
LIST+8	Test 2	
LIST+12	Test 3	
LIST+16	Student ID2	Student 2
LIST+20	Test 1	
LIST+24	Test 2	
LIST+28	Test 3	

- Indexed addressing used in accessing test marks from the list

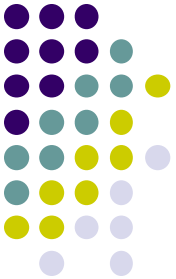
Addressing Modes Indexing and Arrays



- Program to find the sum of marks of all subjects of each students and store it in memory. Assume all data are 32-bit.

Address	Memory	Comments
N	n	No. of students
100 LIST	Student ID1	Student 1
104 LIST+4	Test 1 60	
108 LIST+8	Test 2 80	
112 LIST+12	Test 3 100	
116 LIST+16	Student ID2	Student 2
120 LIST+20	Test 1	
124 LIST+24	Test 2	
LIST+28	Test 3	
	. . . . .	
2000 SUM	Test Sum of Students ID1 240	
SUM+4	Test Sum of Students ID2	

Addressing Modes Indexing and Arrays



```
1.              Move #LIST, R0
2.              Clear R1
3.              Clear R2
4.              Move #SUM, R2
5.              Move N, R4
6.    Loop :    Add 4(R0), R1
7.              Add 8(R0), R1
8.              Add 12(R0),R1
9.              Move R1, (R2)
10.             Clear R1
11.             Add #16, R0
12.             Add #4, R2
13.             Decrement R4
14.             Branch>0 Loop
```

R0	116
R1	00
R2	2004
R3	
R4	2

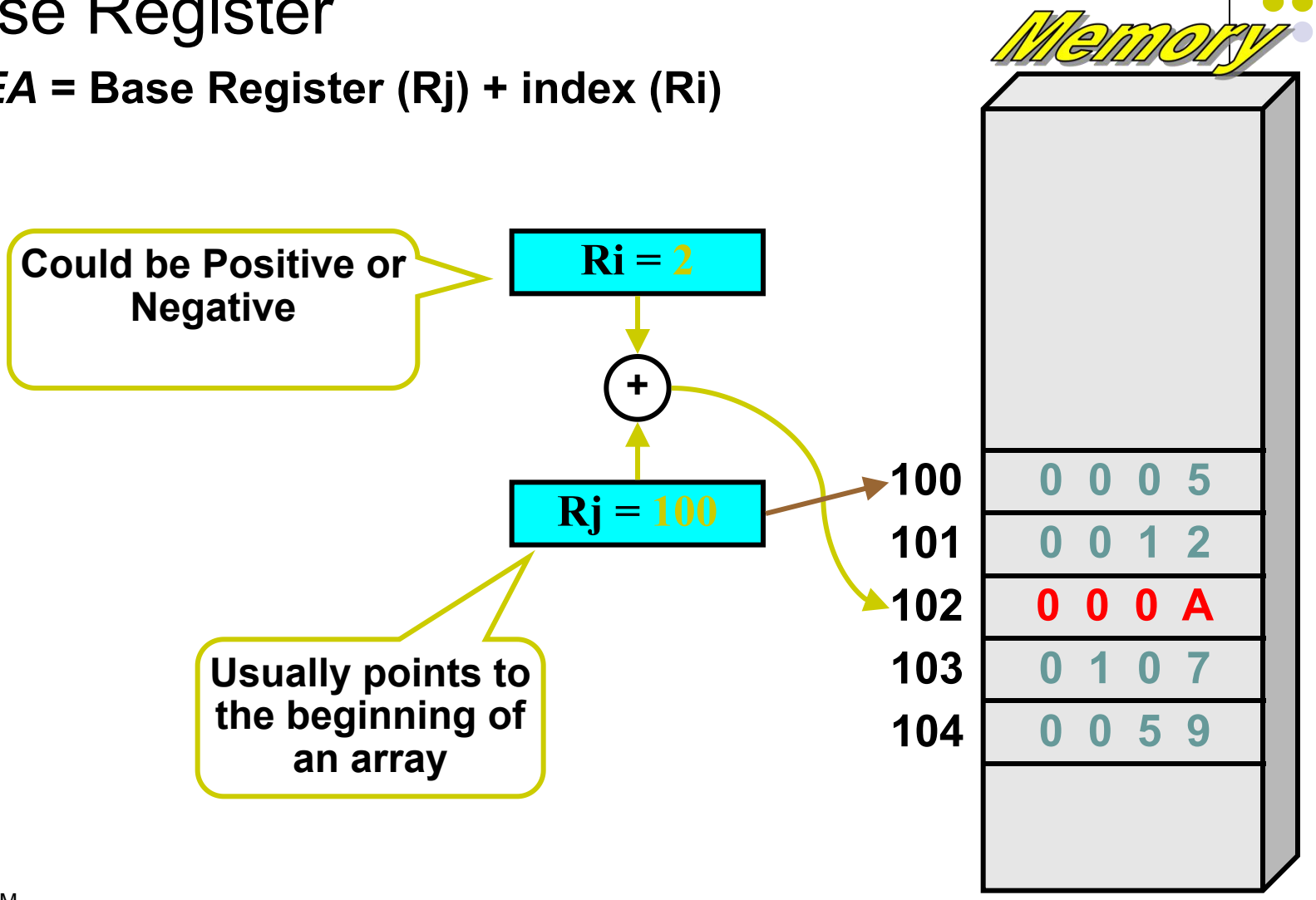
Addressing Modes

- The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

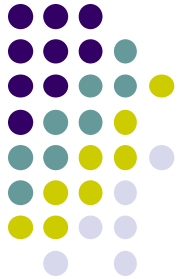
Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri) (LOC)	EA = [Ri] EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]
Base with index and offset	X(Ri,Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	–(Ri)	Decrement Ri; EA = [Ri]

Addressing Modes

- Base Register
  - ***EA*** **= Base Register (Rj) + index (Ri)**



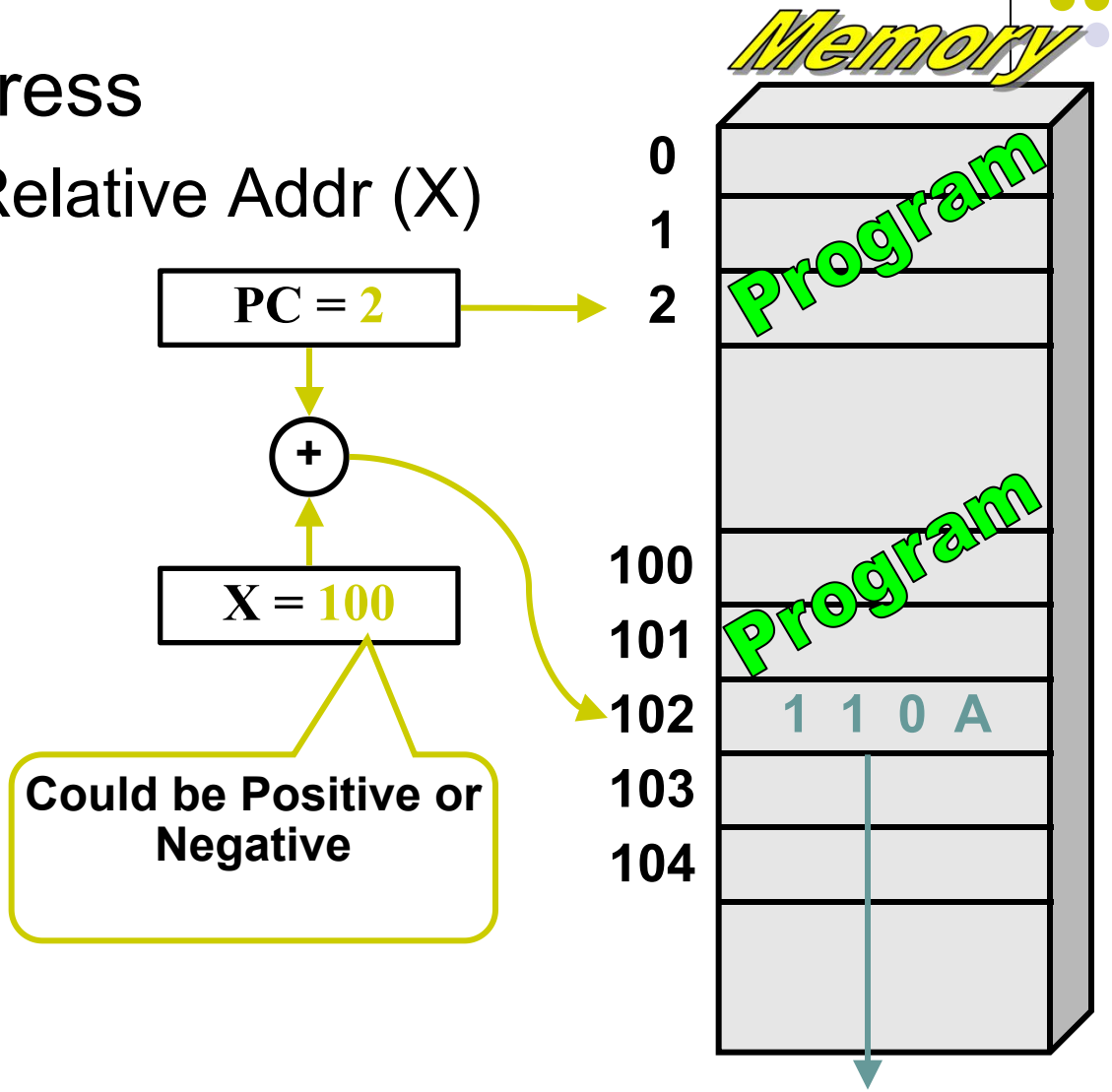
Addressing Modes
Relative Addressing



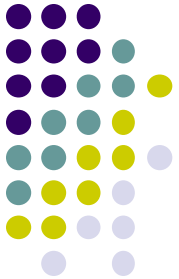
- Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
- X(PC) – where X is a signed number
- ex: Branch>0 LOOP
- This location is computed by specifying it as an offset from the current value of PC.
- Branch target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

Relative Addressing

- Relative Address
  - *EA* = PC + Relative Addr (X)



Addressing Modes
Additional Modes

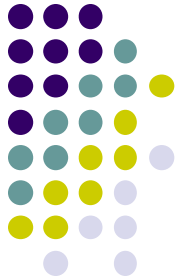


- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- $(R_i)+$. The increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.
- Autodecrement mode: $-(R_i)$ – decrement first

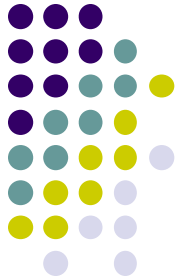
```
                Move        N,R1
                Move        #NUM1,R2      } Initialization
                Clear       R0
LOOP            Add         (R2)+,R0
                Decrement   R1
                Branch>0    LOOP
                Move        R0,SUM
```

Figure 2.16. The Autoincrement addressing mode

Assembly Language



Assembly Language

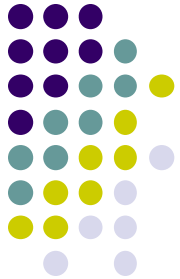


- Machine instructions are represented by patterns of 0s and 1s. So these patterns represented by symbolic names called "*mnemonics*"
- E.g. Load, Store, Add, Move, BR, BGTZ
- A complete set of such symbolic names and rules for their use constitutes a programming language, referred to as an *assembly language*.
- The set of rules for using the mnemonics and for specification of complete instructions and programs is called the *syntax* of the language.
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*.
- The assembler program is one of a collection of utility programs that are a part of the system software of a computer.

Assembly Language

- The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*.
- The assembly language for a given computer is not case sensitive.
- E.g. MOVE R1, SUM

Assembler Directives



- In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program.
- Assign numerical values to any names used in a program.
  - For e,g, name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as TWENTY EQU 20
- If the assembler is to produce an object program according to this arrangement, it has to know
  - How to interpret the names
  - Where to place the instructions in the memory
  - Where to place the data operands in the memory

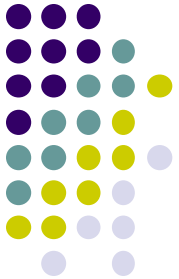
A
re

-

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that	START	MOVE	N,R1
generate		MOVE	#NUM1,R2
machine		CLR	R0
instructions	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

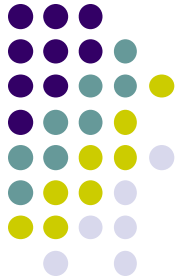
Assembly language representation for the program

Assembly and Execution of Programs



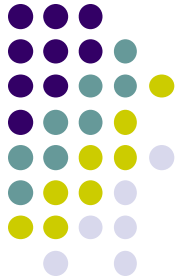
- A source program written in an assembly language must be assembled into a machine language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.
- A key part of the assembly process is determining the values that replace the names. Assembler keep track of Symbolic name and Label name, create table called symbol table.
- The symbol table created by scan the source program twice.
- A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter to the target instruction. The assembler computes this branch offset, which can be positive or negative, and puts it into the machine instruction.

Assembly and Execution of Programs

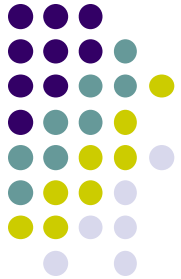


- The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk. The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called *a loader* must already be in the memory.
- Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will be stored.
- The assembler usually places this information in a header preceding the object code (Like start/end offset address).
- When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily.
- The assembler can only detect and report syntax errors. To help the user find other programming errors, the system software usually includes *a debugger* program.
- This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

Number Notation

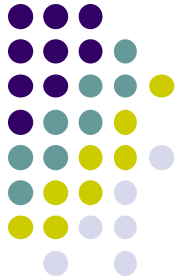


- Decimal Number
  - ADD #93,R1
- Binary Number
  - ADD #%0101110,R1
- Hexadecimal Number
  - ADD #$5D,R1



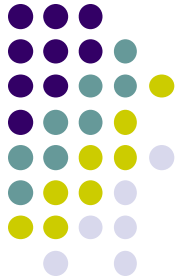
Basic Input/Output Operations

I/O



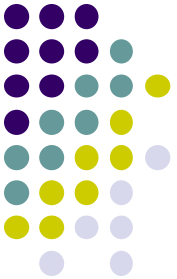
- The data on which the instructions operate are not necessarily already stored in memory.
- Data need to be transferred between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

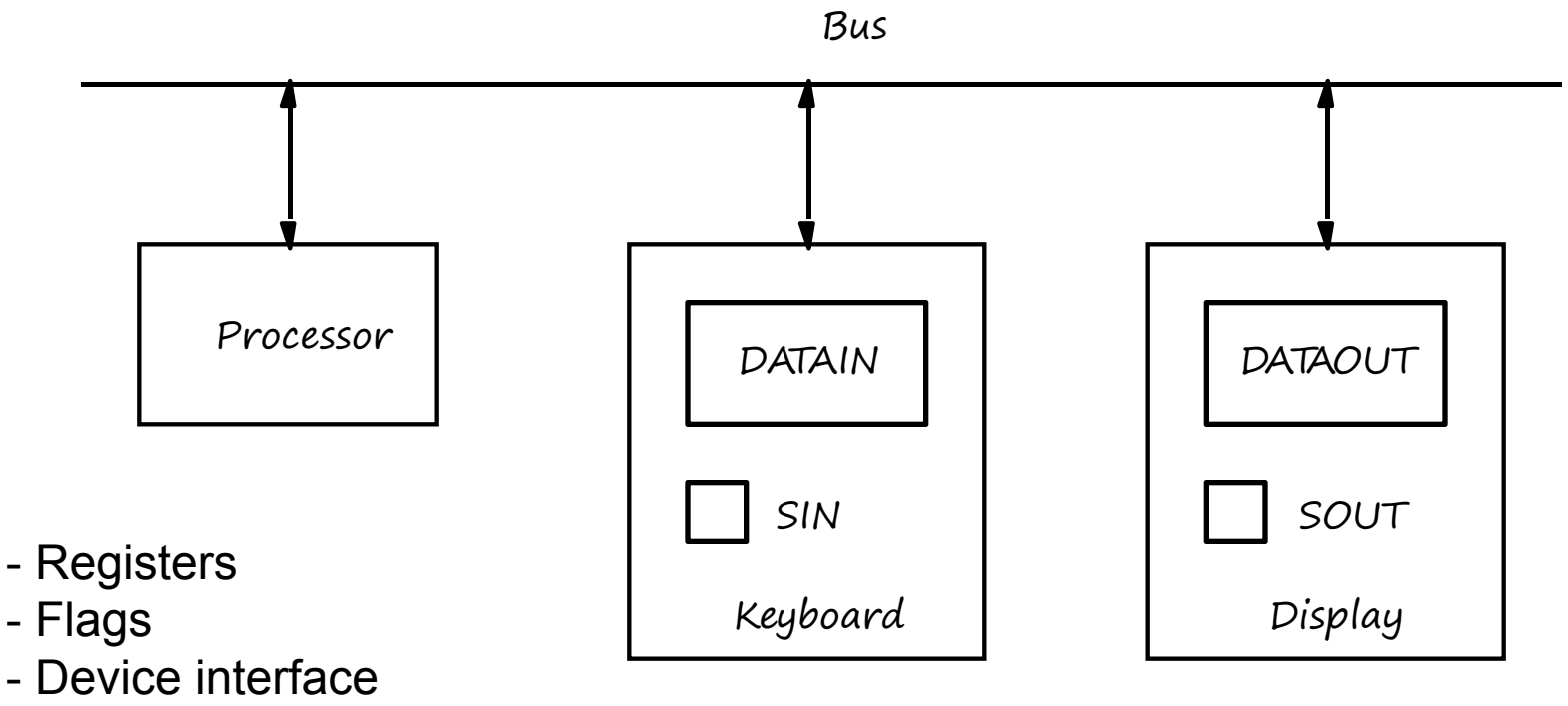
Program-Controlled I/O Example



- Read in character input from a keyboard and produce character output on a display screen.
  - Rate of data transfer (keyboard, display, processor)
  - Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.
  - A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way.

Program-Controlled I/O Example

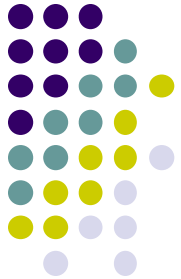




- Registers
- Flags
- Device interface

Figure 2.1 Bus connection for processor, keyboard, and display

Program-Controlled I/O Example

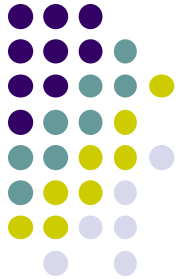


- Machine instructions that can check the state of the status flags and transfer data:

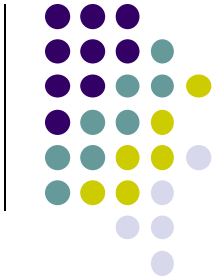
```
READWAIT  Branch to READWAIT if SIN = 0
          Input from DATAIN to R1

WRITEWAIT Branch to WRITEWAIT if SOUT = 0
          Output from R1 to DATAOUT
```

Program-Controlled I/O Example



- **Memory-Mapped I/O** – some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also uses device status registers.
  - E.g. Movebyte DATAIN,R1
  - Movebyte R1,DATAOUT
- READWAIT Testbit #3, INSTATUS
  Branch=0 READWAIT
  MoveByte DATAIN, R1
- WRITEWAIT Testbit #3, OUTSTATUS
  Branch=0 WRITEWAIT
  MoveByte R1, DATAOUT

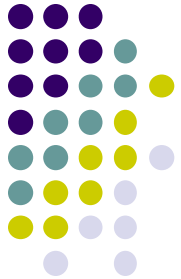


A program that reads a line of characters and displays it.

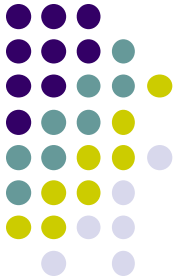
	Move	#LOC, R0	register Ro points to the memory address LOC
READ	TestBit	#3, INSTATUS	wait for the character to be entered in the
	Branch=0	READ	DATAIN buffer
	MoveByte	DATAIN, (R0)	transfer the character from DATAIN into the memory (this clears SIN to 0)
ECHO	TestBit	#3, OUTSTATUS	wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(R0), DATAOUT	move the character just read to DATAOUT(this clears SOUT to 0)
	Compare	#CR, (R0)+	check if the character just read is CR(carriage return) and also increment the pointer to store the next character
	Branch≠0	READ	if it is not CR then branch back to read the next character

STACKS AND QUEUES: -

Program-Controlled I/O Example

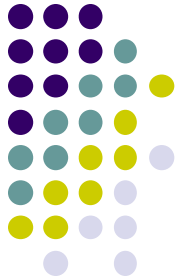


- Assumption – the initial state of SIN is 0 and the initial state of SOUT is 1.
- Any drawback of this mechanism in terms of efficiency?
  - Two wait loops→processor execution time is wasted
- Alternate solution?
  - Interrupt



Stacks

Stacks

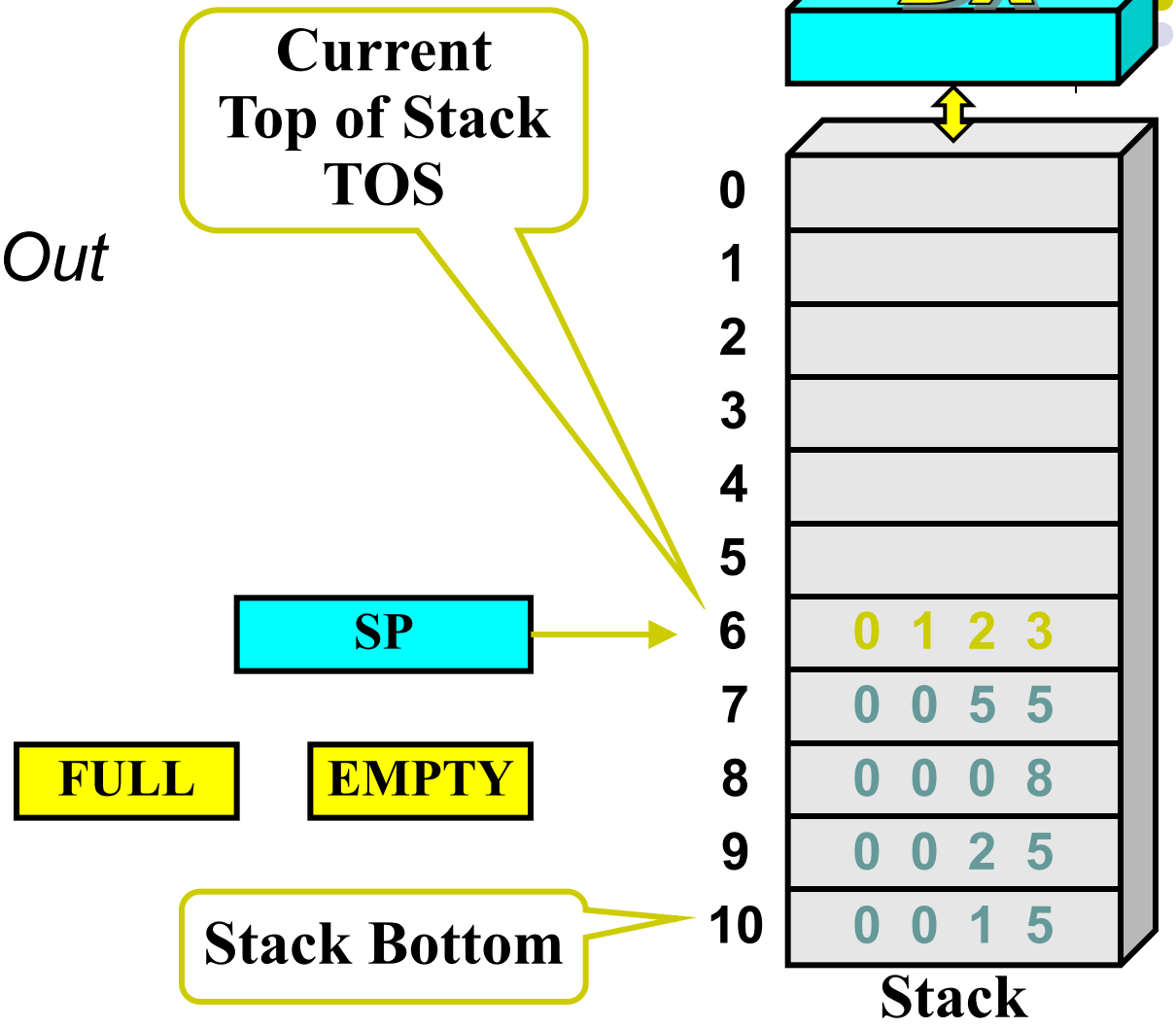


- A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack.
- *last-in–first-out* (LIFO) stack working.
- The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.
- The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time.

Stack Organization

- LIFO

  Last In First Out



Stack Organization

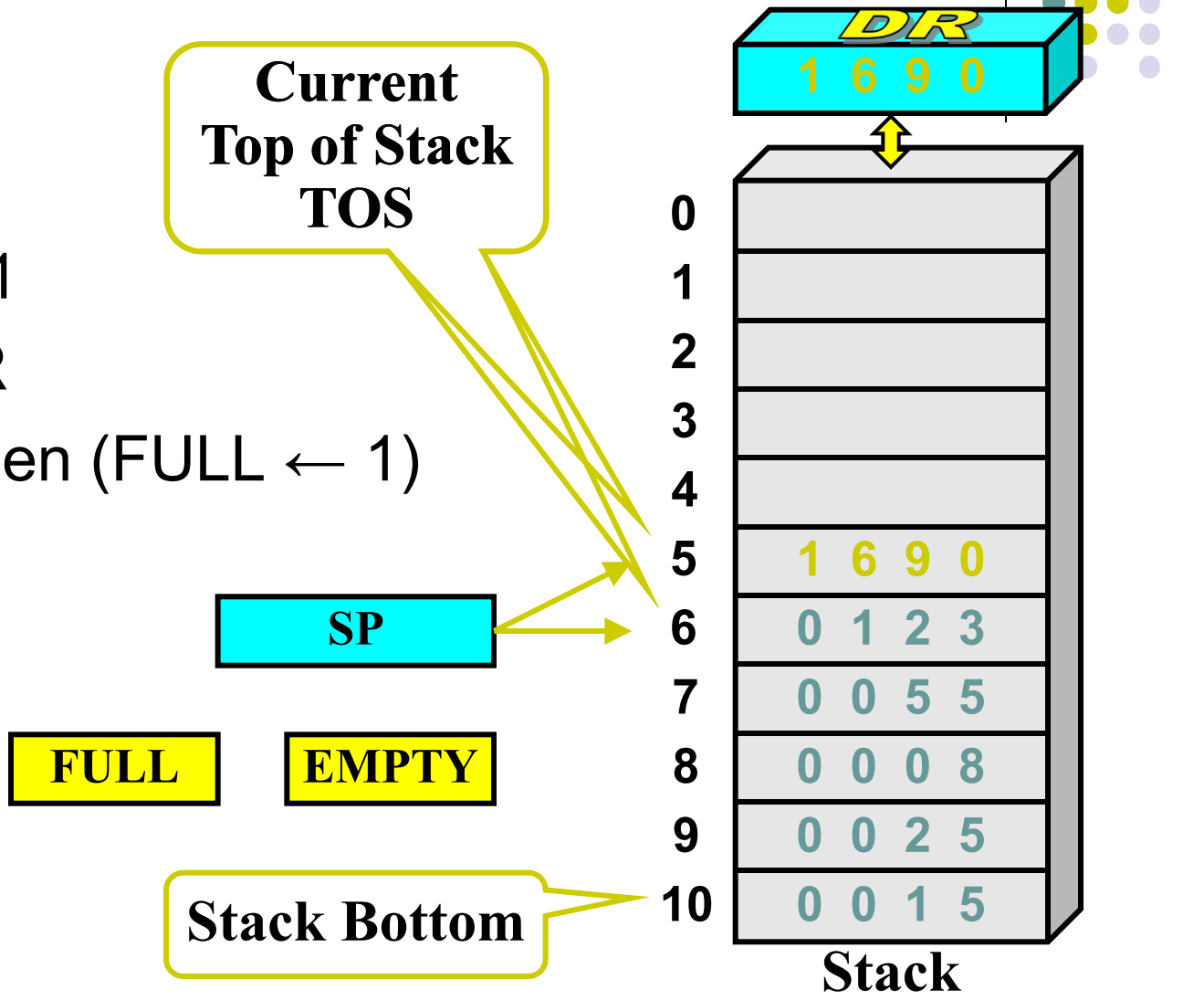
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

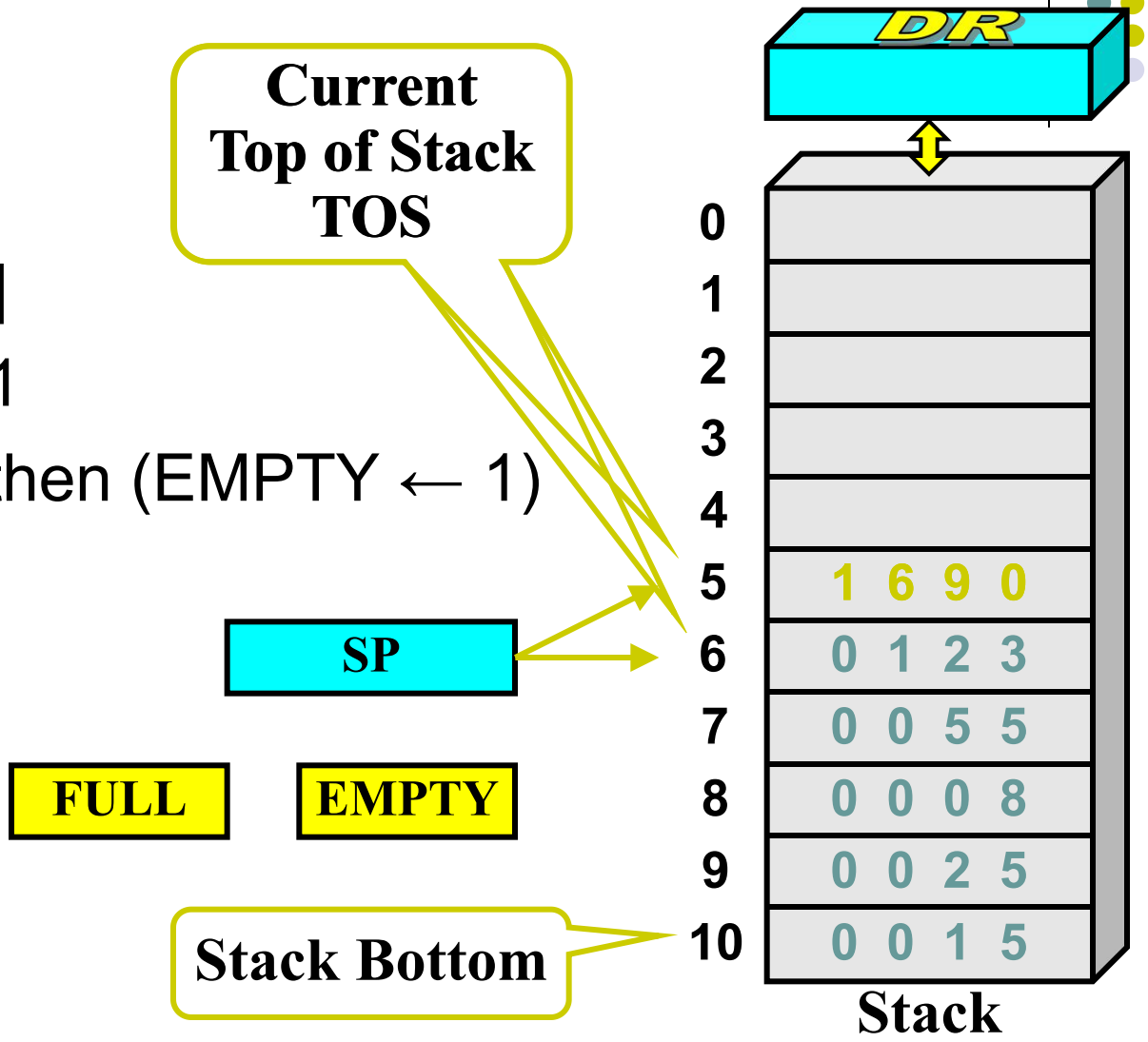
- POP

  DR ← M[SP]

  SP ← SP + 1

  If (SP = 11) then (EMPTY ← 1)

  FULL ← 0



Stack Organization

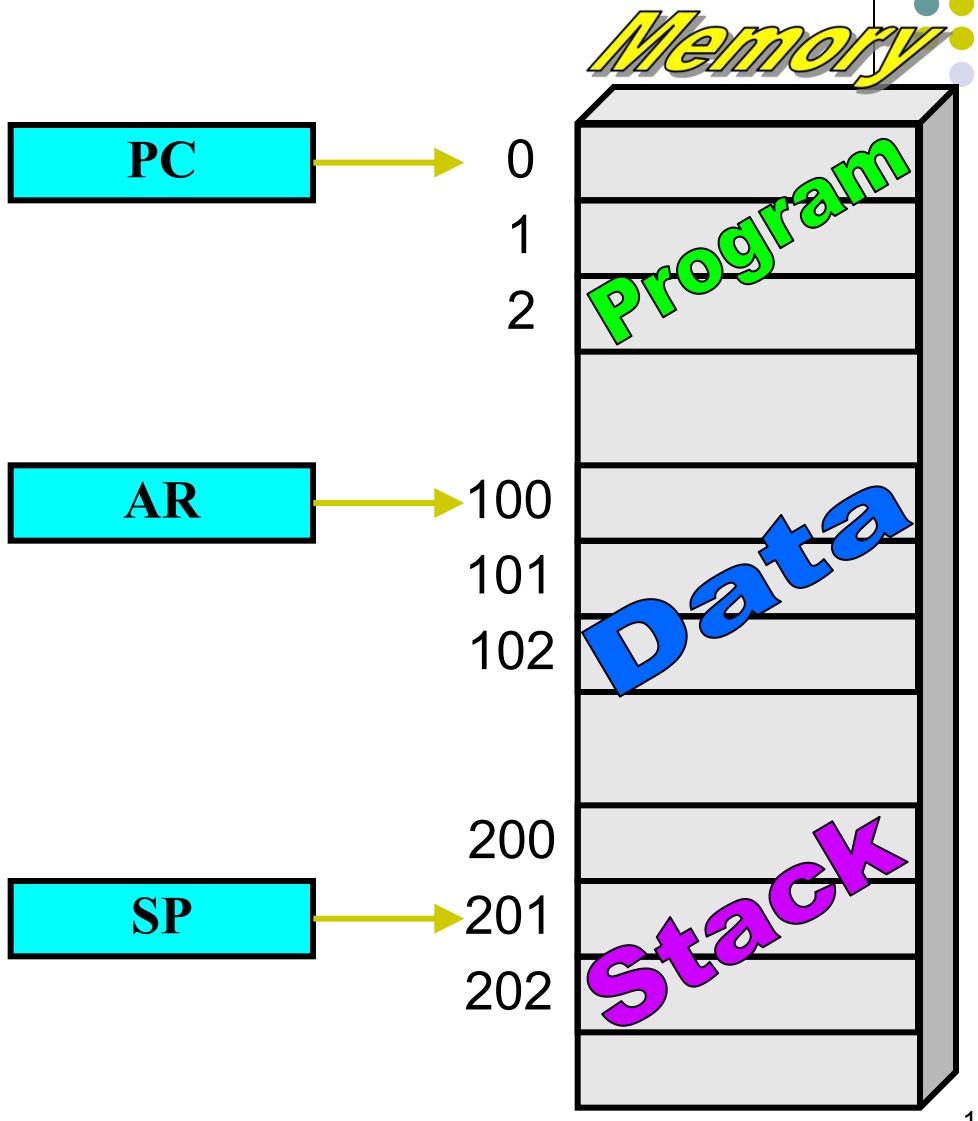
- Memory Stack
  - PUSH

    SP ← SP – 1

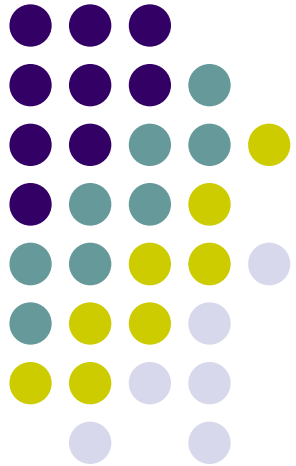
    M[SP] ← DR
  - POP

    DR ← M[SP]

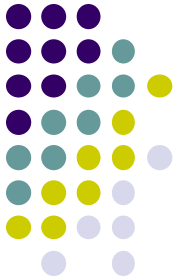
    SP ← SP + 1



Queue

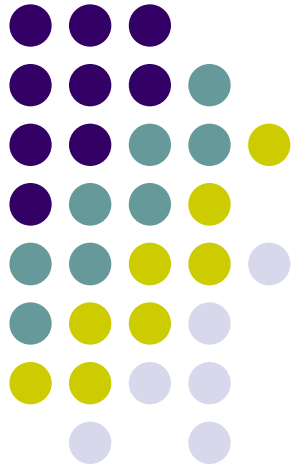


Queue

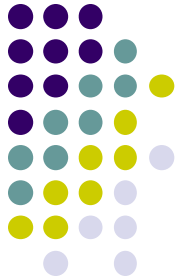


- FIFO basis
- Data are stored in and retrieved from a queue on a first-in–first-out (FIFO) basis.
- Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

Subroutines

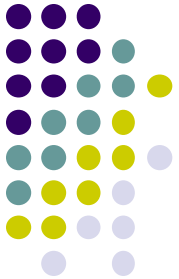


Subroutines



- In a given program, it is often necessary to perform a particular task many times on different data values. It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a *subroutine*.
- However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location.
- When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a Call instruction.
- After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it, and it does so by executing a Return instruction.

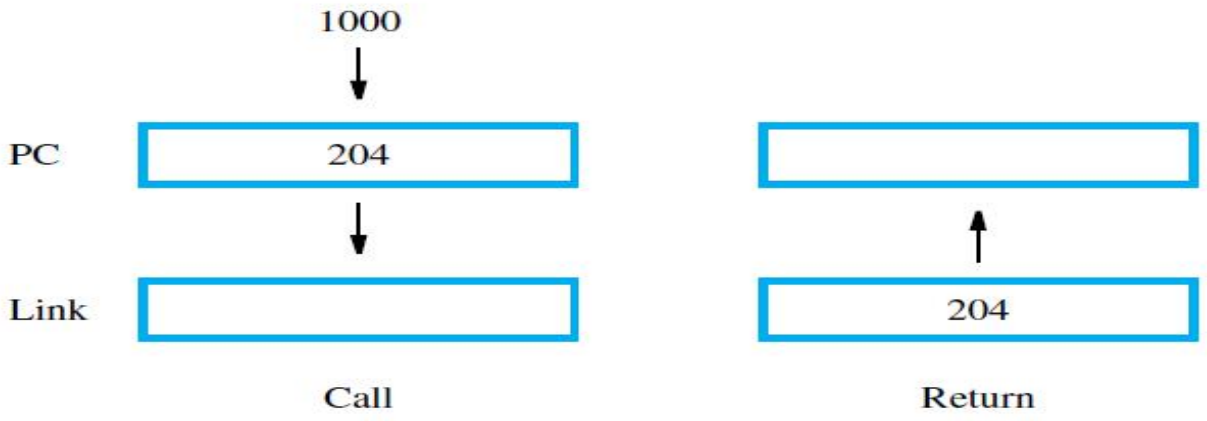
Subroutines



- Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed.
- Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.
- The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method.
- The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

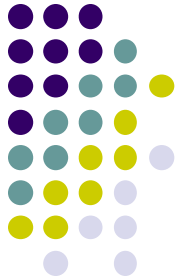
Subroutines

Memory location	Calling program	Memory location	Subroutine SUB
	⋮		
200	Call SUB	1000	first instruction
204	next instruction		⋮
	⋮		Return



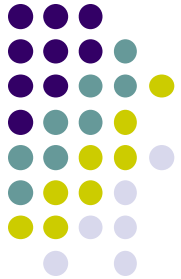
Subroutine linkage using a link register.

Subroutines



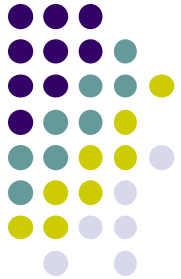
- The Call instruction is just a special branch instruction that performs the following operations:
  - Store the contents of the PC in the link register
  - Branch to the (PC update to) target address specified by the Call instruction
- The Return instruction is a special branch instruction that performs the operation
  - Branch to the (PC update to) address contained in the link register

Parameter Passing



- exchange of information between a calling program and a subroutine is referred to as *parameter passing*.
- Parameter passing may be accomplished in several ways.
  - ✓ registers,
  - ✓ memory locations, or
  - ✓ the processor stack

Program of subroutine Parameters passed through registers.



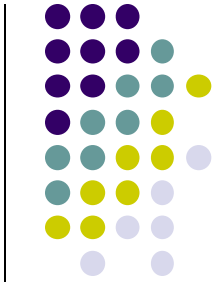
- Calling Program

1. Move N, R1
2. Move #NUM1,R2
3. Call LISTADD
4. Move R0,SUM

- Subroutine

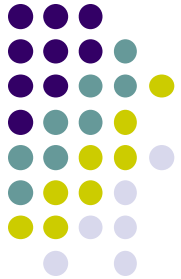
1. LISTADD: Clear R0
2. LOOP: Add (R2)+,R0
3. Decrement R1
4. Branch>0 LOOP
5. Return

Parameter Passing by Value and by Reference



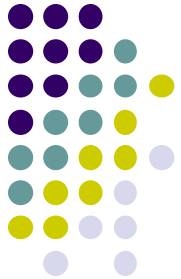
- Instead of passing the actual Value(s), the calling program passes the address of the Value(s). This technique is called *passing by reference*.
- The second parameter is *passed by value*, that is, the actual number of entries, is passed to the subroutine.

Subroutine Nesting and the usage of Stack



- A common programming practice, called *subroutine nesting*, is to have one subroutine call another.
- In this case, the return address of the second call is also stored in the link register, overwriting its previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.
- That is, 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 stack.

Program of subroutine Parameters passed on the stack.

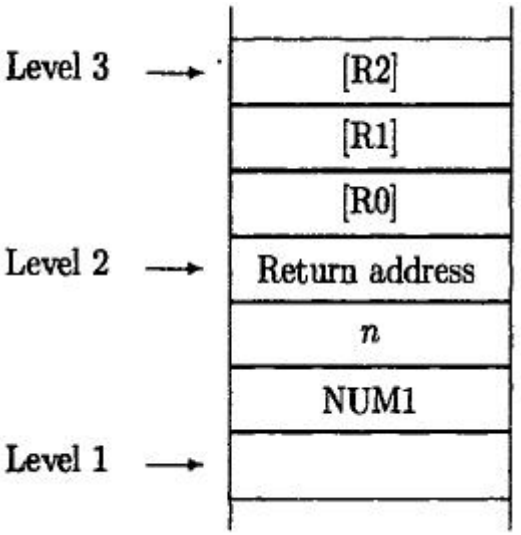


- MoveMultiple R0-R2, -(SP)
- MoveMultiple to store contents of register R0 through R2 on the stack

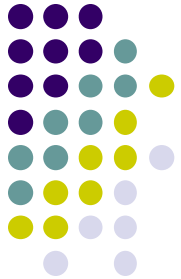
Write an assembly code to add N numbers in the list using subroutine. Show the contents of stack at different levels.

Assume top of stack is at level 1 below.

```
            Move        #NUM1,-(SP)     Push parameters onto stack.
            Move        N,-(SP)
            Call        LISTADD         Call subroutine
                                          (top of stack at level 2).
            Move        4(SP),SUM       Save result.
            Add         #8,SP           Restore top of stack
                                          (top of stack at level 1).
            :
LISTADD     MoveMultiple R0-R2,-(SP)    Save registers
                                          (top of stack at level 3).
            Move        16(SP),R1       Initialize counter to n.
            Move        20(SP),R2       Initialize pointer to the list.
            Clear       R0              Initialize sum to 0.
LOOP        Add         (R2)+,R0        Add entry from list.
            Decrement   R1
            Branch>0    LOOP
            Move        R0,20(SP)       Put result on the stack.
            MoveMultiple (SP)+,R0-R2    Restore registers.
            Return                      Return to calling program.
```



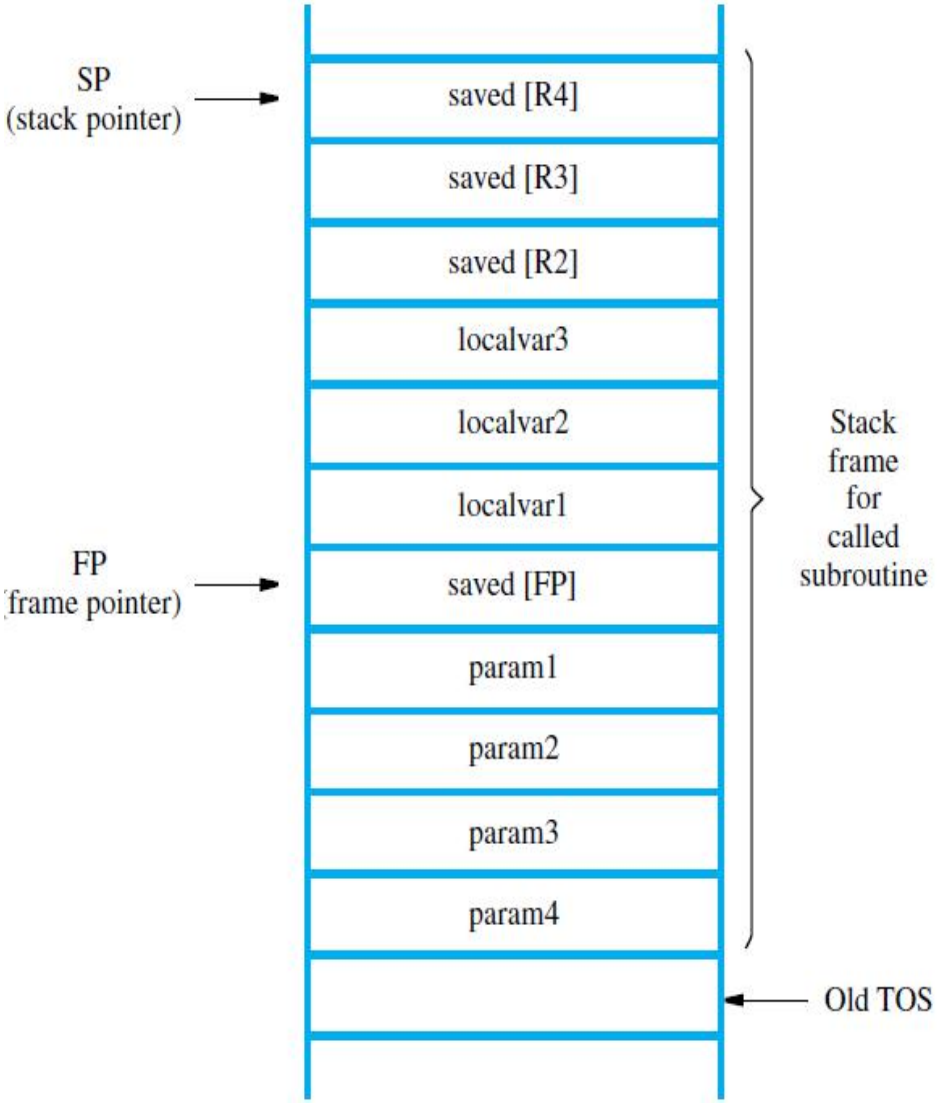
The Stack Frame



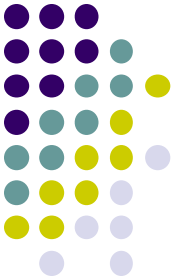
- If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack this area of stack is called Stack Frame.
- For e.g. during execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program.

The Stack Frame

- *Frame pointer* (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine.
- In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R2, R3, and R4 need to be saved because they will also be used within the subroutine.
- When nested subroutines are used, the stack frame of the calling subroutine would also include the return address, as we will see in the example that follows.

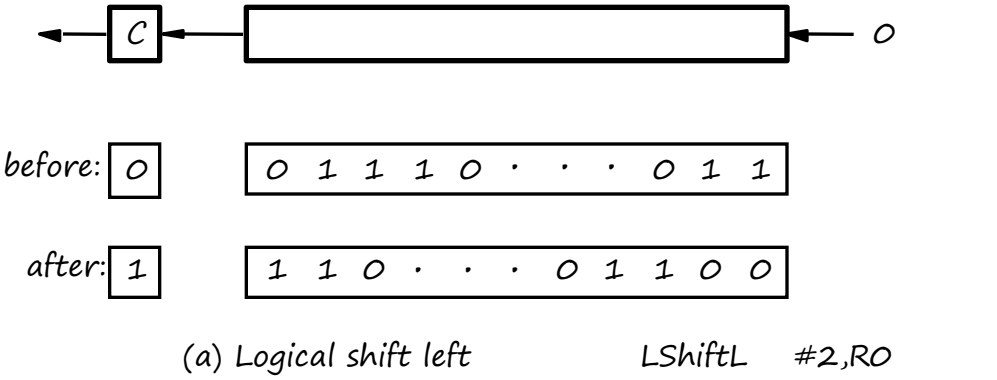


Additional Instructions

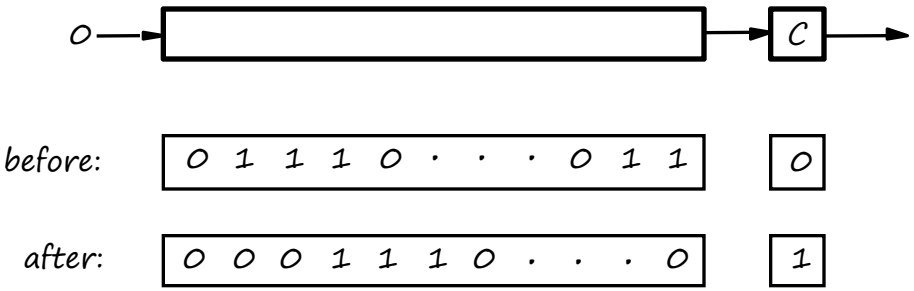


Logical Shifts

- Logical shift – shifting left (LShiftL) and shifting right (LShiftR)

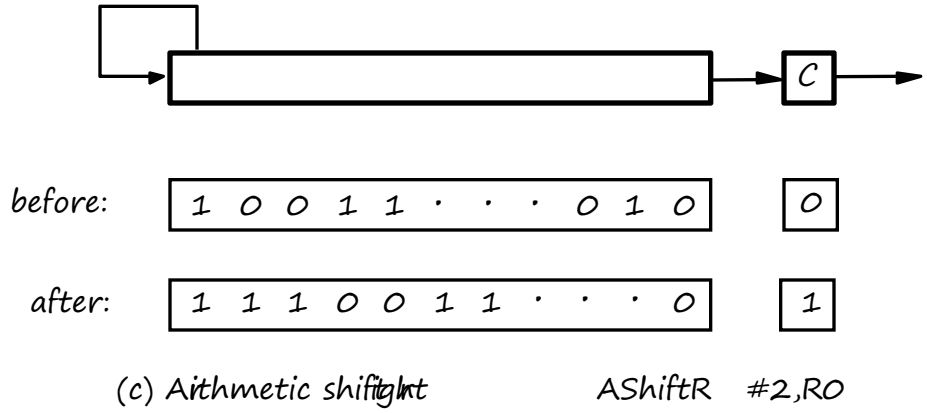


(a) Logical shift left LShiftL #2,R0



(b) Logical shift right LShiftR #2,R0

Arithmetic Shifts



(c) Arithmetic shift right AShiftR #2,R0

Rotate

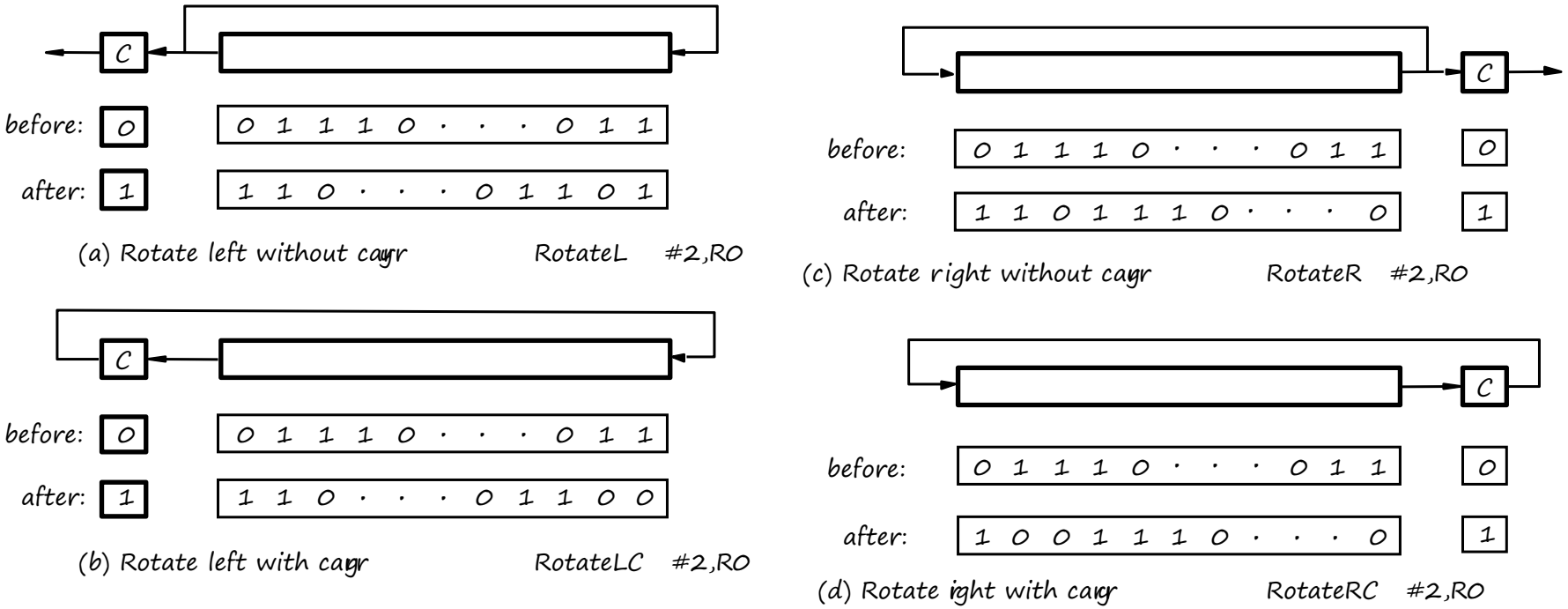
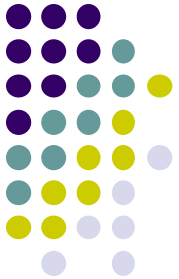


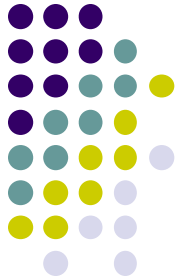
Figure 2.32. Rotate instructions.

Logic Instructions



- And R2, R3, R4
- And #Value, R4, R2
- And #$0FF, R2, R2,

Write an Assembly language instruction to find whether first Byte contains the character “Z” along with the memory representation for each Instruction.



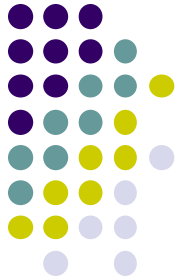
```
MSBCMP      AND   #$FF000000, R0
            CMP #$5A000000, R0
            BRANCH=0 SUCCESS
```

R0= 1010 1100 1111 1000 1101..... 1111 (32 bit data)

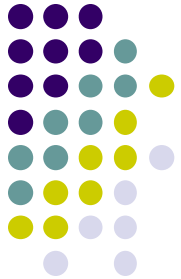
& 1111 1111 0000 0000 0000..... 0000 (FF000000)

R0=1010 1100 0000 0000 0000 ....0000 (MSB 8 bits retained)

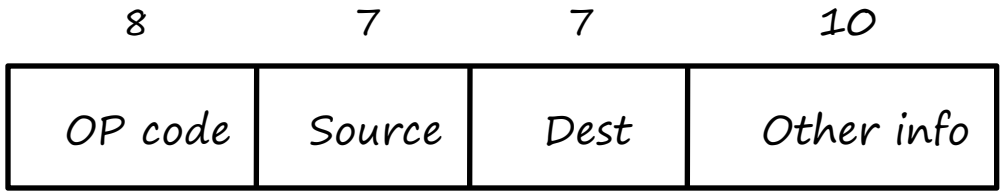
Encoding of Machine Instructions



Encoding of Machine Instructions

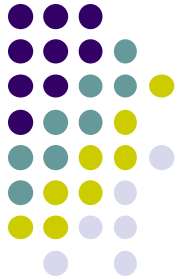


- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)
- In the previous section, an assumption was made that all instructions are one word in length.
- OP code: the type of operation to be performed and the type of operands used may be specified using an encoded binary pattern
- Suppose 32-bit word length, 8-bit OP code (how many instructions can we have?), 16 registers in total (how many bits?), 3-bit addressing mode indicator.
  - **Add R1, R2**
  - **Move 24(R0), R5**
  - **LshiftR #2, R0**
  - **Move #$3A, R1**

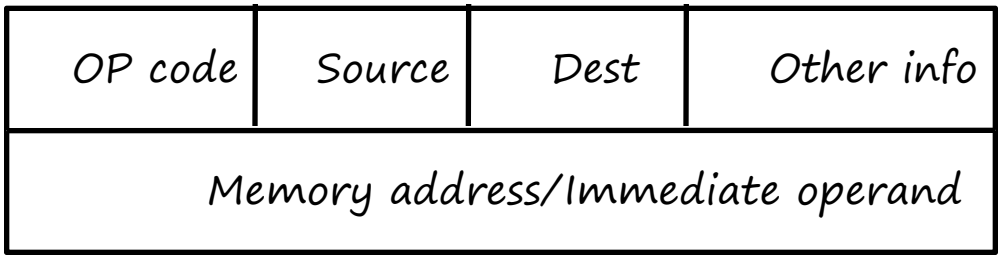


(a) One-word instruction

Encoding of Machine Instructions

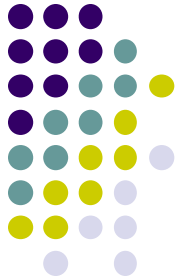


- What happens if we want to specify a memory operand using the Absolute addressing mode?
  - **Move R2, LOC**
  - **14-bit for LOC – insufficient**
- Solution – use two words



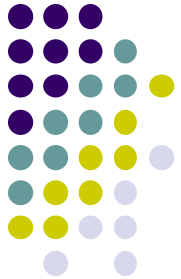
(b) Two-word instruction

Encoding of Machine Instructions



- Then what if an instruction in which two operands can be specified using the Absolute addressing mode?
  - **Move LOC1, LOC2**
- Solution – use two additional words
- This approach results in instructions of variable length. Complex instructions can be implemented, closely resembling operations in high-level programming languages – Complex Instruction Set Computer (CISC)

Encoding of Machine Instructions



- If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction.
- It is still possible to define a highly functional instruction set, which makes extensive use of the **processor registers**.
  - **Add R1, R2 ----- yes**
  - **Add LOC, R2 ----- no**
  - **Add (R3), R2 ----- yes**