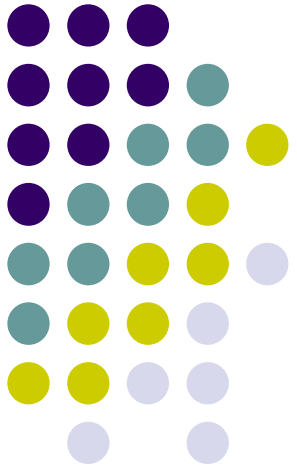
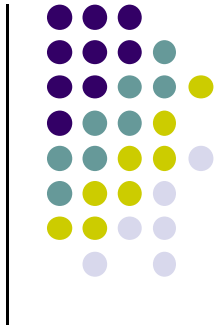
Computer Organization
19IS304 (CO)

INTRODUCTION



Computer Organization



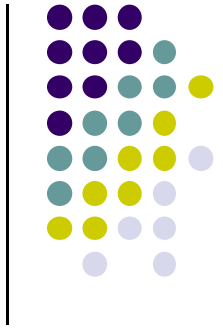
- Computer Organization mainly deals with Computer hardware and computer architecture.
- **Computer Hardware**

  Collection of physical parts of the computer.

- **Computer Architecture**

  Specifies the functionality of computer systems, instruction sets and the hardware units that implement these instructions.

Course Objectives



CO1: Understand the basic structure and organization of Computer Systems

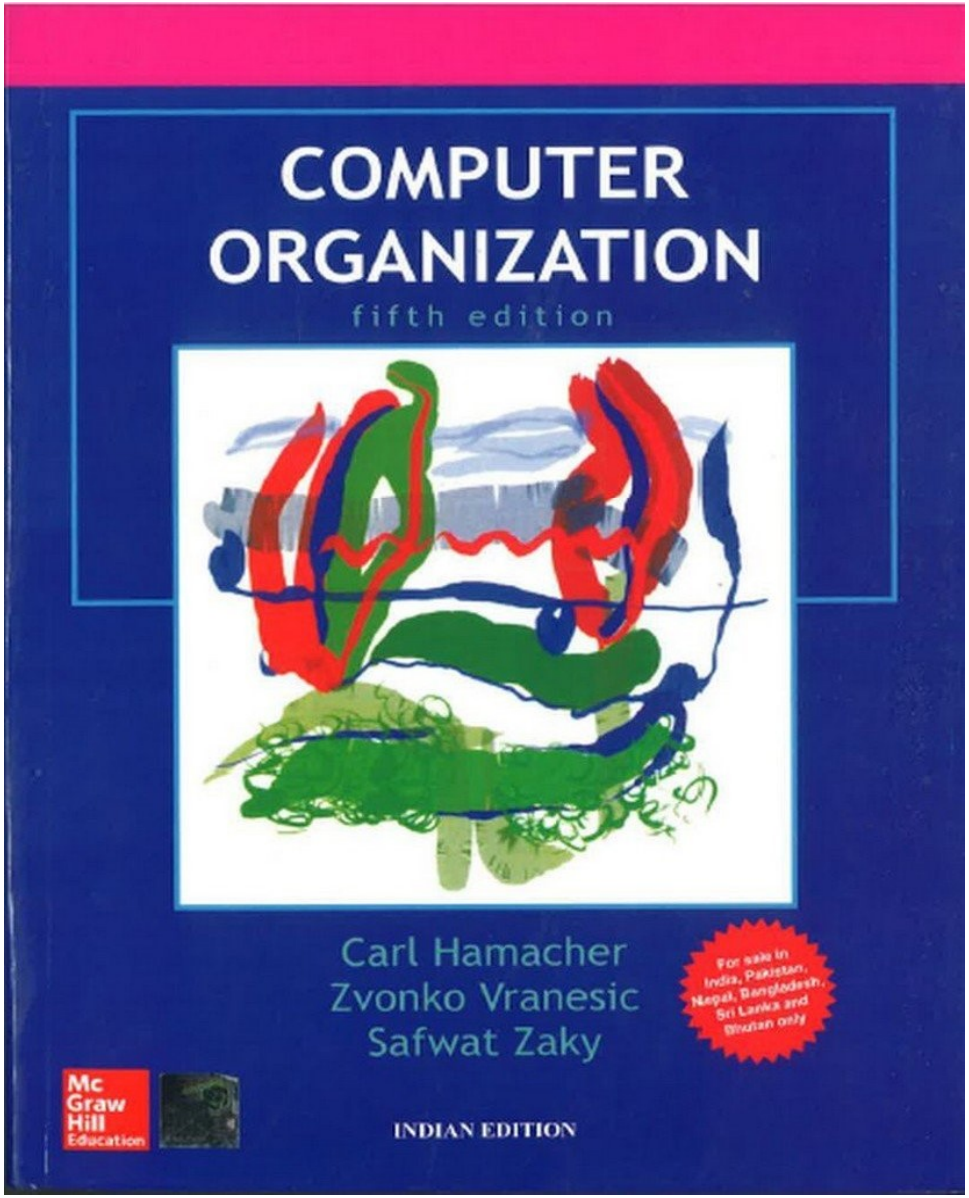
CO2: Comprehend the basic input/output operations in a computer system

CO3: Model the computer memory systems

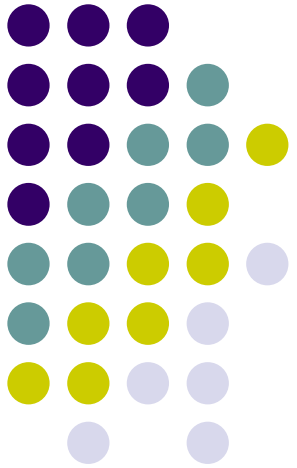
CO4: Identify and apply the circuits of arithmetic operations

CO5: Model the instruction processing which is implemented in a processor

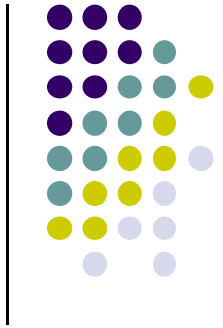
Text Book


COMPUTER
ORGANIZATION
fifth edition
Carl Hamacher
Zvonko Vranesic
Safwat Zaky
For sale in India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan only
Mc Graw Hill Education
INDIAN EDITION

Unit 1 : Chapter 1
Basic Structure of Computers



Computer Types



- **What is a Computer?**

  Computer is a fast electronic calculating machine that accepts digitized input information, processes it according to a list of internally stored instructions, and produces the resulting information.

- **Which are the different types of Computers?**
  - Personal Computers
  - Workstations
  - Enterprise Systems and Servers
  - Supercomputers

Personal Computer

- Widely used at schools, homes, business office etc.
- Usually inexpensive and small
- Two types:
  - **Desktop Computer** - Systems having a processing unit, storage unit, Visual Display, Keyboard, Audio Unit that can all be located on a desk of a home or an office.
  - **Portable Notebook Computer** - Compact version of a desktop computer with all the components packaged into a single unit, like a thin briefcase.

Workstations

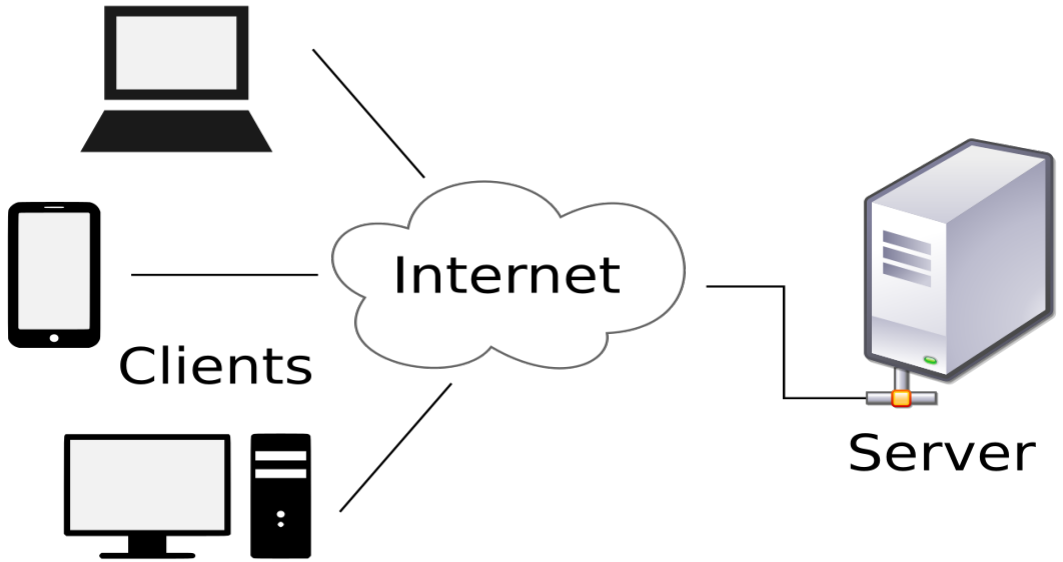
- A computer with high resolution graphic input and output capability which has the dimension of the desktop but with a higher computational power.
- Example: GPUs, System used in medical applications like Ultrasound, CT Scan etc.

Enterprise Systems and Servers

- **Enterprise Systems**
  - Also called mainframes.
  - Used for large business processing that has high computational power and storage capacity than workstations
  - Example: ATM Machine

Enterprise Systems and Servers

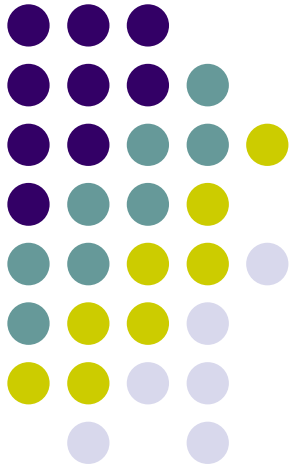
- **Servers**
  - These are the ones which can handle large volume of requests to handle data
  - Here, requests and responses, both are transferred through network.
  - Example: File Servers, DB Servers



Supercomputers

- Used for large scale numerical calculations required in applications such as weather forecasting and aircraft design and simulation.

Functional Units



Functional Units

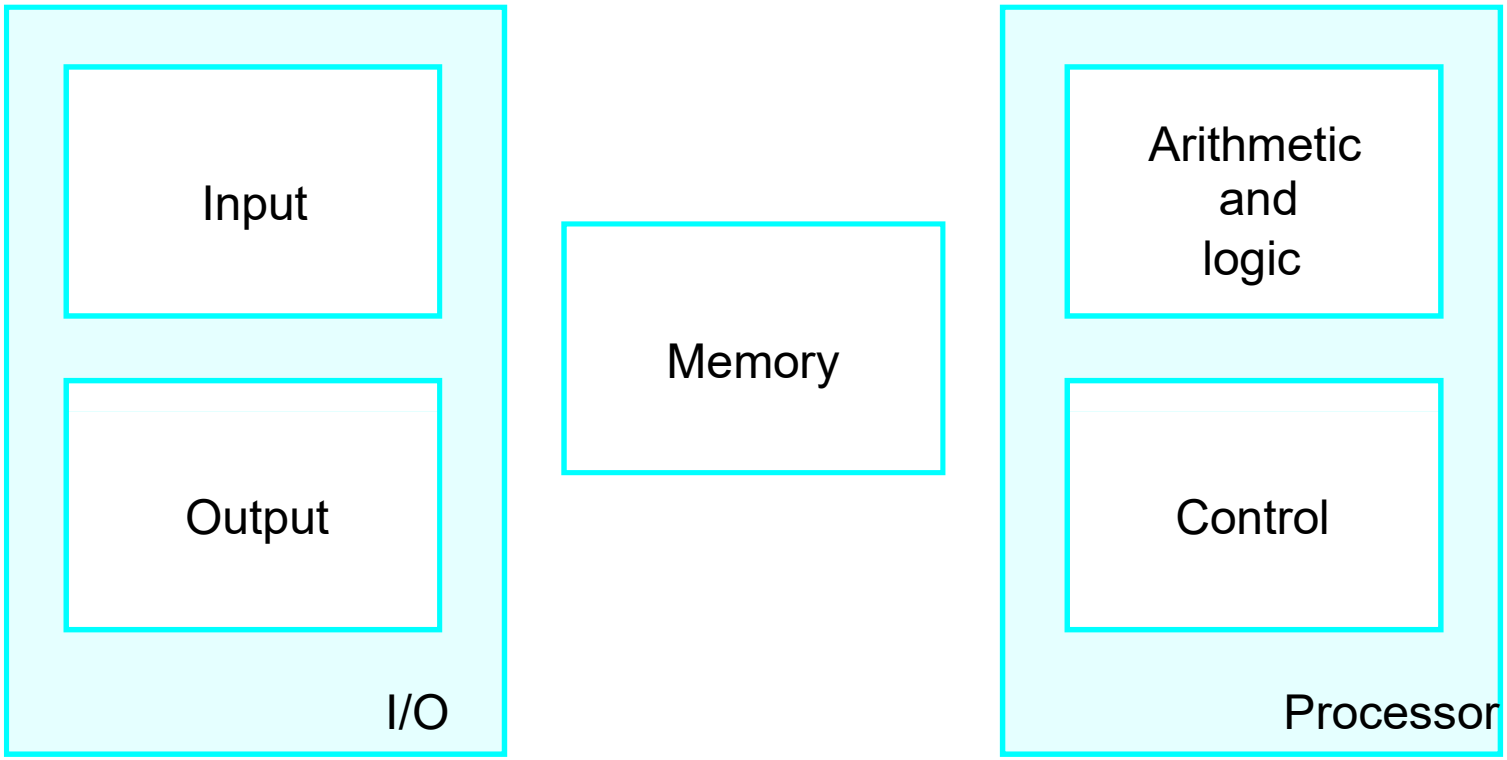
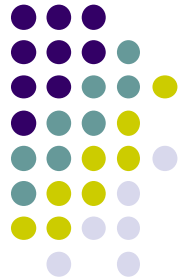


Figure 1.1. Basic functional units of a computer.

Functional Units



- A computer system has 5 functionally independent units for processing instructions
  1. Input Unit
  2. Memory Unit
  3. Arithmetic and Logic Unit (ALU)
  4. Output Unit
  5. Control Unit

Information handled by computer

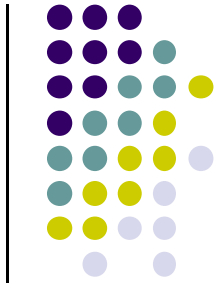
- The information handled by the computer is either an instruction or data.

- **Instructions :**
  - Instructions are commands that**:**
    1. Govern the transfer of information within a computer as well as between the computer and its I/O devices
    2. Specify the arithmetic and logic operations to be performed
  - Set of Instructions is called as a **Program**

- **Data:**
  - Data are numbers and encoded characters that are used as operands by the instructions

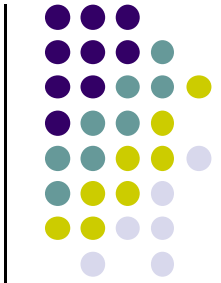
- All the information will be handled as bits(0 and 1)

Input Unit



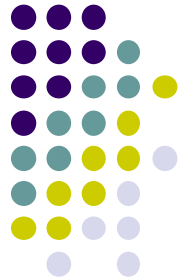
- **Input Unit**
  - Input unit accepts coded information from the humans, through input devices.
  - Example: Keyboard
  - Whenever a key is pressed, the corresponding letter or digit is automatically converted to its binary format and transmitted to memory or the processor over a cable.

Memory Unit



- **Memory Unit**
  - The function of the memory unit is to **store** programs and data.
  - There are two classes of storage: **Primary** and **Secondary**
- **Primary Memory**
  - Fast memory operating at electronic speeds.
  - Organized as an array of large number of storage cells. Each cell is capable of storing 1 bit of information.
  - Any program to be executed has to stored in memory.
  - The processing is done in groups of bits called **words**.
  - Each word is associated with unique address.
  - Word Length – Number of bits in each word (16 to 64 bits)
  - Example: RAM

Memory Unit



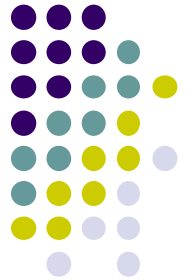
- **Primary Memory**
  - Time taken to access one word - **Memory Access Time**
  - Memory Hierarchy – Cache, Main Memory
  - Expensive

- **Secondary Memory**
  - Cheaper storage when large volume of data is to be stored.
  - It is used for storing that is not accessed frequently.
  - Example: CDs, Magnetic Tapes, USB Devices

Arithmetic and Logic Unit (ALU)

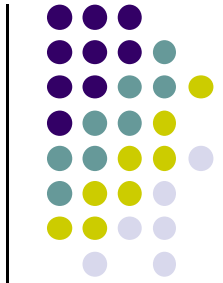
- **Arithmetic and Logic Unit**
  - Most of the computer operations is performed in the ALU
  - Any Arithmetic or Logical operation is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.
  - The operands are stored in the high speed storage elements called **Registers**.
  - Each register can store one word of data.
  - Access time to registers is the fastest.

Output Unit



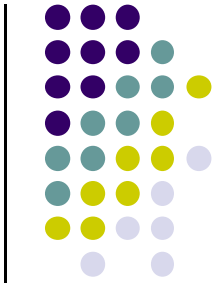
- **Output Unit**
  - Sends the processed results to the outside world.
  - Example: Monitor, Printer

Control Unit



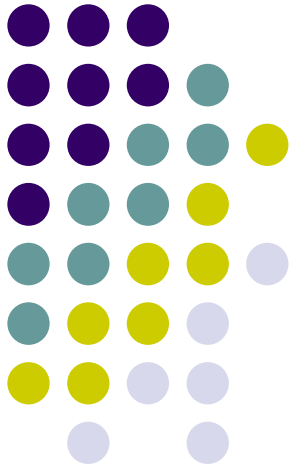
- **Control Unit**
  - The computer operations are coordinated by the control unit.
  - The timing signals that govern the I/O transfers are also generated by the control unit.

Operation of a Computer

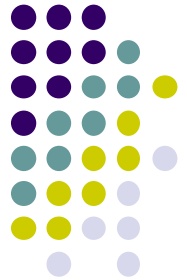


- **Summary**
  - Accept information in the form of programs and data through an input unit and store it in the memory.
  - Fetch the information stored in the memory, under program control, into ALU, where the information is processed.
  - Output the processed information through an output unit.
  - Control all activities inside the machine through a control unit.

Basic Operational Concepts

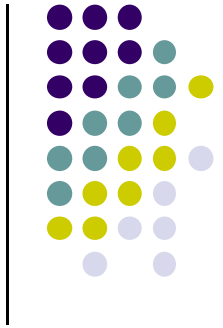


Basic Operational Concepts



- 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.

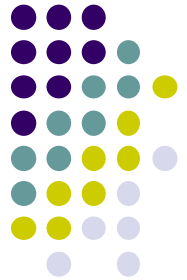
A Typical Instruction



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 ---- **Memory Access Operation**
- Add R1, R0 ---- **ALU Operation**

- **Whose contents will be overwritten?**

Connection Between the Processor and the Memory

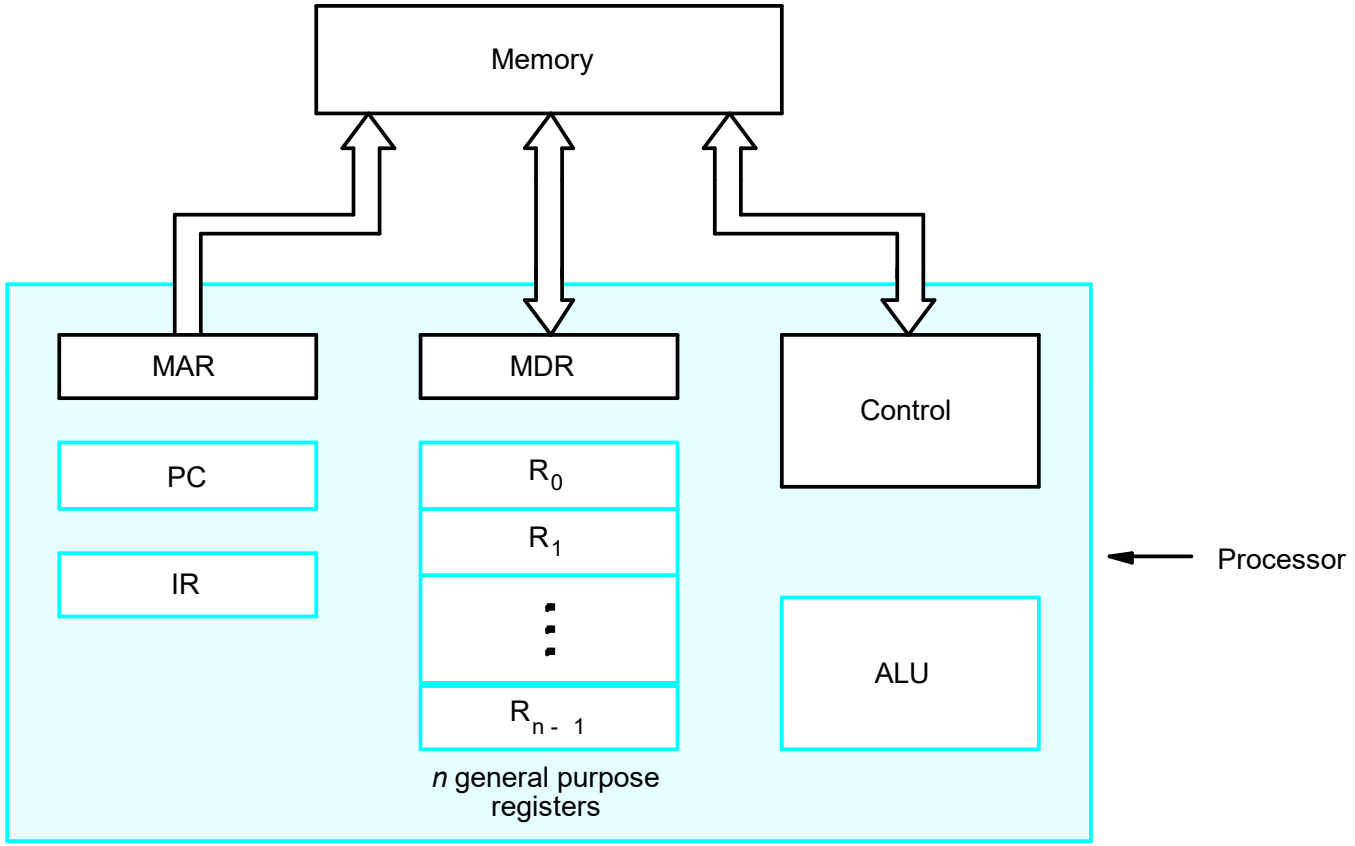
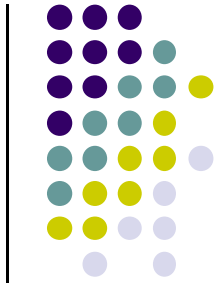


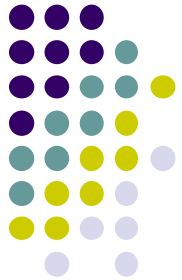
Figure 1.2. Connections between the processor and the memory.

Registers



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

Registers



- **Instruction register (IR)**

  Holds the instruction that is currently being executed.

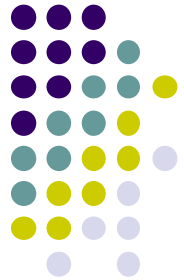
- **Program counter (PC)**

  Contains the address of the next instruction to be fetched and executed.

- **General-purpose register ($R_0 - R_{n-1}$)**

  Used for intermediate storage of data.

Registers



These registers facilitate the communication with the memory.

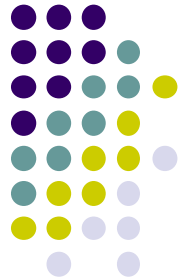
- **Memory Address Register (MAR)**

  MAR holds the address of the location to be accessed.

- **Memory Data Register (MDR)**

  MDR contains the data that is to be written into or read out from the addressed location.

Typical Operating Steps



- Programs reside in the memory through input devices.
- Execution starts when 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.
- Processor decodes and executes the instruction.

Typical Operating Steps – ALU Operation

- 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

Example – MOV LOCA, R0

1. Initially PC will be pointing to this instruction.
2. The contents of PC is transferred to MAR and Read control signal is sent.
3. The contents present at location pointed by MAR is read into MDR.
4. Contents of MDR is transferred to IR.
5. Meanwhile, PC is incremented to point to next instruction.
6. Instruction is decoded at IR.
7. Since source operand is memory location, IR sends the address of LOCA to MAR.
8. Control Unit issues Read Control Signal.

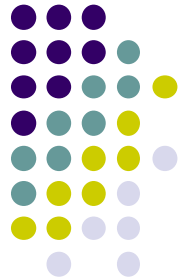
Example – MOV LOCA, R0

9. Contents present at address pointed by MAR is loaded into MDR.
10. Since destination operand is a general purpose register, content of MDR is directly transferred to the Register R0.

Example – MOV R0, LOCA

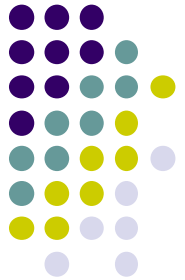
1. Initially PC will be pointing to this instruction.
2. The contents of PC is transferred to MAR and Read control signal is sent.
3. The contents present at location pointed by MAR is read into MDR.
4. Contents of MDR is transferred to IR.
5. Meanwhile, PC is incremented to point to next instruction.
6. Instruction is decoded at IR.
7. Since, the destination operand is a memory location, the address of LOCA is MAR.
8. Since source operand is a general purpose register R0, the data in R0 is moved to MDR.

Example – MOV R0, LOCA



9. Control Unit issues the Write Signal.
10. Contents of MDR is written into memory location pointed to by MAR.

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

Memory Read/Write Operation

- **Read Operation ---- Load LOCA, R0**
  - Place the address of operand LOCA on the **A**ddress Bus.
  - Send Read Control Signal on the **C**ontrol Bus.
  - On receiving the Read signal, Memory unit places the data at the specified address location onto the **D**ata Bus.

- **Write Operation ---- Store R0, LOCA**
  - Place the address of the location LOCA on the **A**ddress Bus.
  - Place the data to be written to memory on the **D**ata Bus.
  - Send Write Control Signal on the **C**ontrol Bus.

Bus Structure

- Single-bus

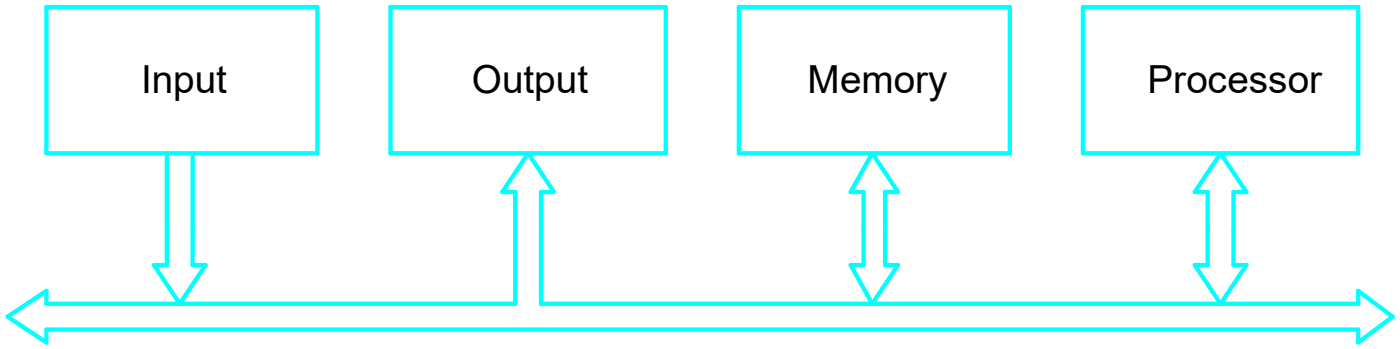
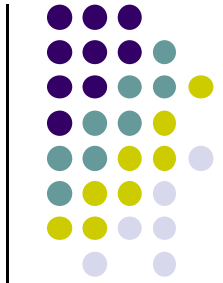


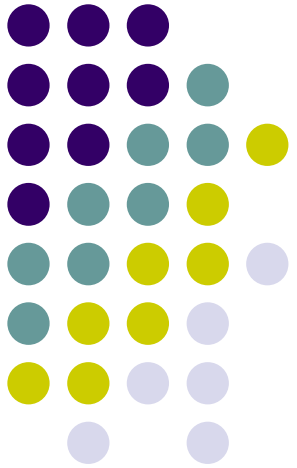
Figure 1.3. Single-bus structure.

Speed Issue

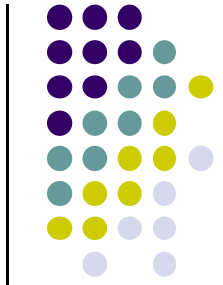


- Different devices have different transfer/operate speed.
- If the speed of bus is bounded by the slowest device connected to it, the efficiency will be very low.
- How to solve this?
- A common approach – use buffers.

Performance

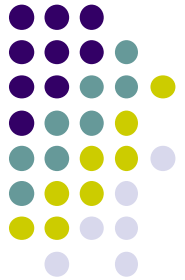


Performance



- 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.

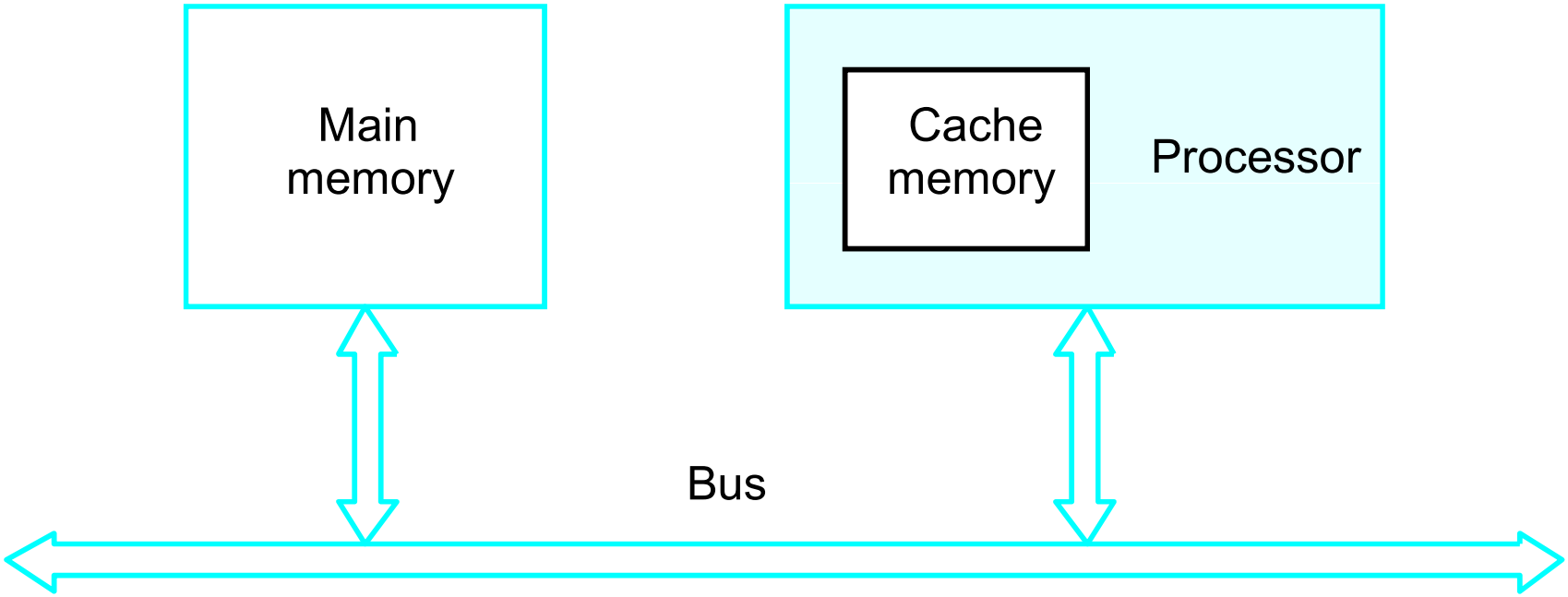
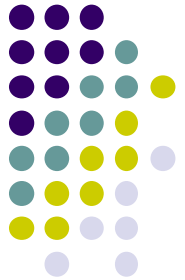


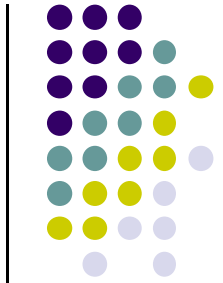
Figure 1.5. The processor cache.

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, and clock rate
- The execution of each instruction is divided into several steps, each of which completes in one clock cycle.
- Hertz – cycles per second

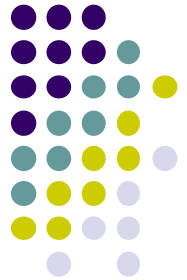
Basic Performance Equation

- 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 (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- Note: these are not independent to each other

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

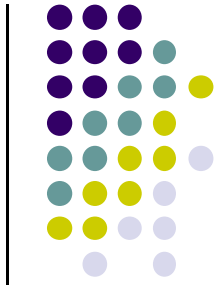
How to improve T?

Pipeline and Superscalar Operation



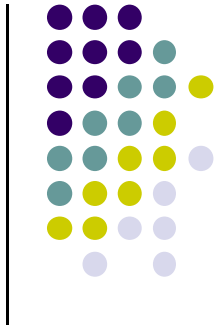
- Instructions are not necessarily executed one after another.
- The value of S doesn't have to be the number of clock cycles to execute one instruction.
- Pipelining – overlapping the execution of successive instructions.
- Add R1, R2, R3
- Superscalar operation – multiple instruction pipelines are implemented in the processor.
- Goal – reduce S (could become <1!)

Clock Rate



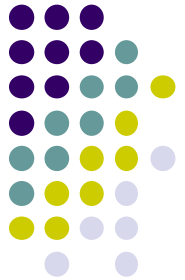
- Increase clock rate
  - Improve the integrated-circuit (IC) technology to make the circuits faster
  - Reduce the amount of processing done in one basic step (however, this may increase the number of basic steps needed)
- Increases in R that are entirely caused by improvements in IC technology affect all aspects of the processor's operation equally except the time to access the main memory.

CISC and RISC



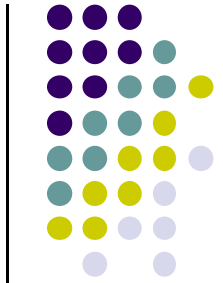
- Tradeoff between N and S
- A key consideration is the use of pipelining
  - S is close to 1 even though the number of basic steps per instruction may be considerably larger
  - It is much easier to implement efficient pipelining in processor with simple instruction sets
- Reduced Instruction Set Computers (RISC)
- Complex Instruction Set Computers (CISC)

Compiler



- A compiler translates a high-level language program into a sequence of machine instructions.
- To reduce N, we need a suitable machine instruction set and a compiler that makes good use of it.
- Goal – reduce N×S
- A compiler may not be designed for a specific processor; however, a high-quality compiler is usually designed for, and with, a specific processor.

Performance Measurement

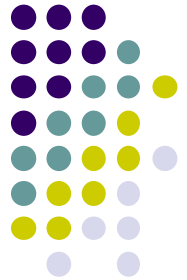


- T is difficult to compute.
- Measure computer performance using benchmark programs.
- System Performance Evaluation Corporation (SPEC) selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
- Compile and run (no simulation)
- Reference computer

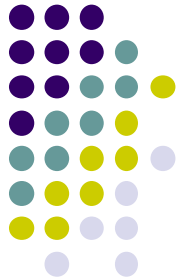
$$SPEC\ \ rating\ \ = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

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

Chapter 2. Machine Instructions and Programs

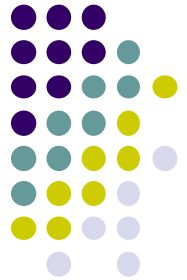


Objectives

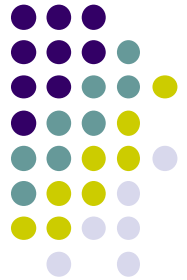


- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2's-complement system.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

Number, Arithmetic Operations, and Characters

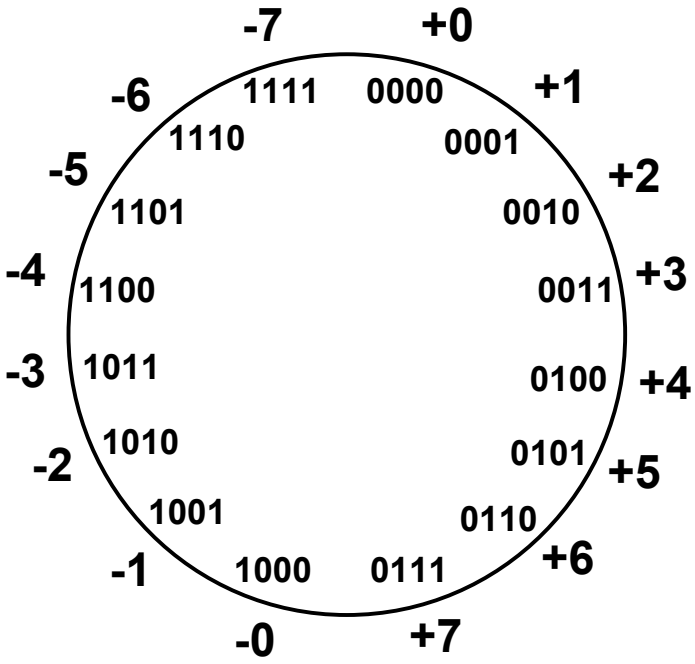


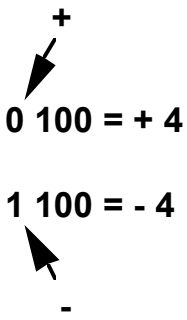
Signed Integer



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

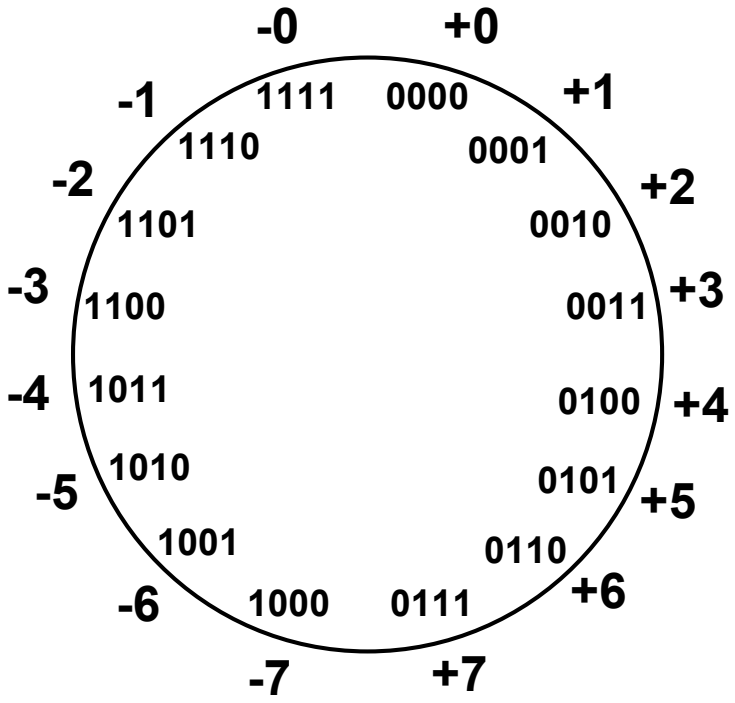
Sign and Magnitude Representation

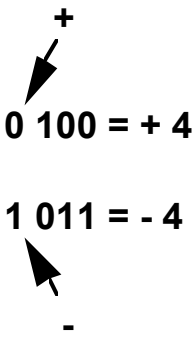




High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Number range for n bits = $+/-2^{n-1}$ -1
Two representations for 0

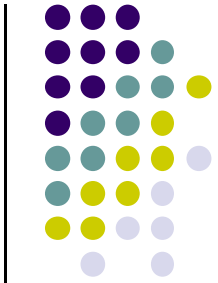
One's Complement Representation



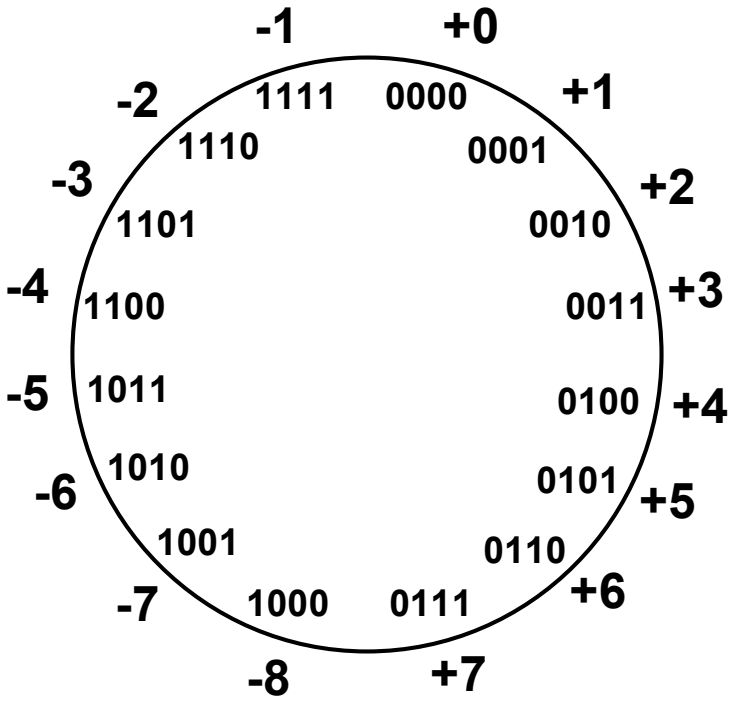


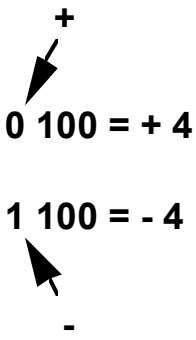
- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

Two's Complement Representation



like 1's comp except shifted one position clockwise





- Only one representation for 0
- One more negative number than positive number

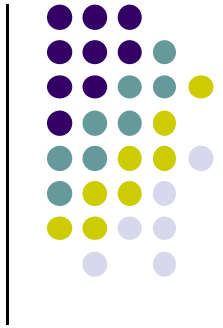
Binary, Signed-Integer Representations

Page 28

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

Figure 2.1. Binary, signed-integer representations.

Addition and Subtraction – 2's Complement



If carry-in to the high order bit = carry-out then ignore carry

4	0100	-4	1100
+ 3	0011	+ (-3)	1101
7	0111	-7	11001

if carry-in differs from carry-out then overflow

4	0100	-4	1100
- 3	1101	+ 3	0011
1	10001	-1	1111

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

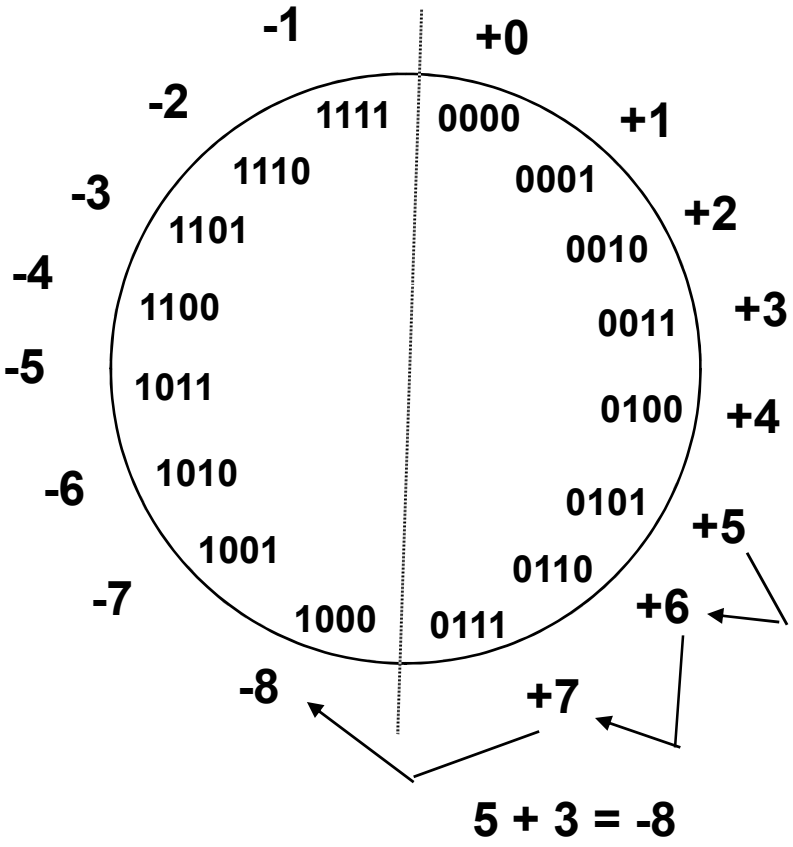
2's-Complement Add and Subtract Operations

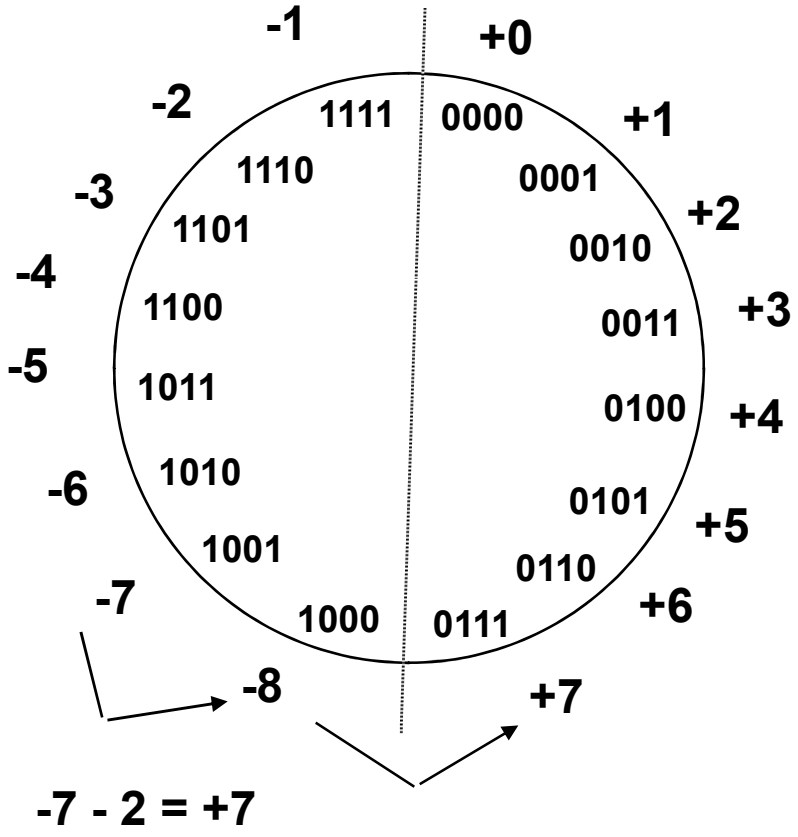
Page 31

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

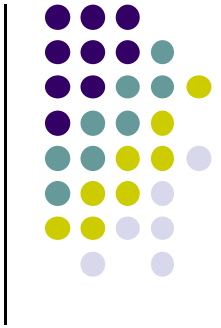
Figure 2.4. 2's-complement Add and Subtract operations.

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





Overflow Conditions



	0 1 1 1	
5	0 1 0 1	
3	0 0 1 1	
-8	1 0 0 0	

Overflow

	1 0 0 0	
-7	1 0 0 1	
-2	1 1 0 0	
7	1 0 1 1 1	

Overflow

	0 0 0 0	
5	0 1 0 1	
2	0 0 1 0	
7	0 1 1 1	

No overflow

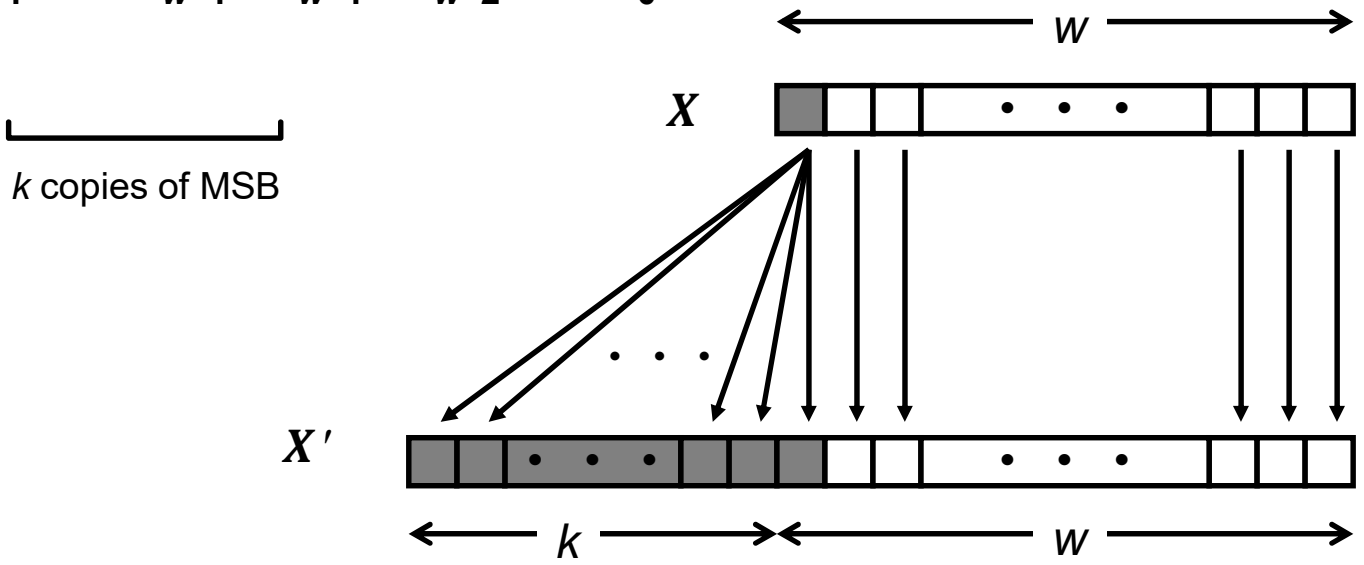
	1 1 1 1	
-3	1 1 0 1	
-5	1 0 1 1	
-8	1 1 0 0 0	

No overflow

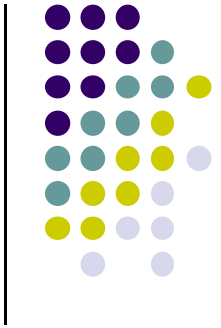
Overflow when carry-in to the high-order bit does not equal carry out

Sign Extension

- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of sign bit:
  - $X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0$



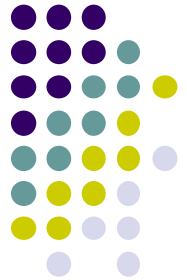
Sign Extension Example



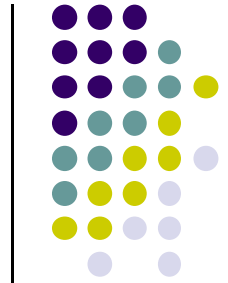
```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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.

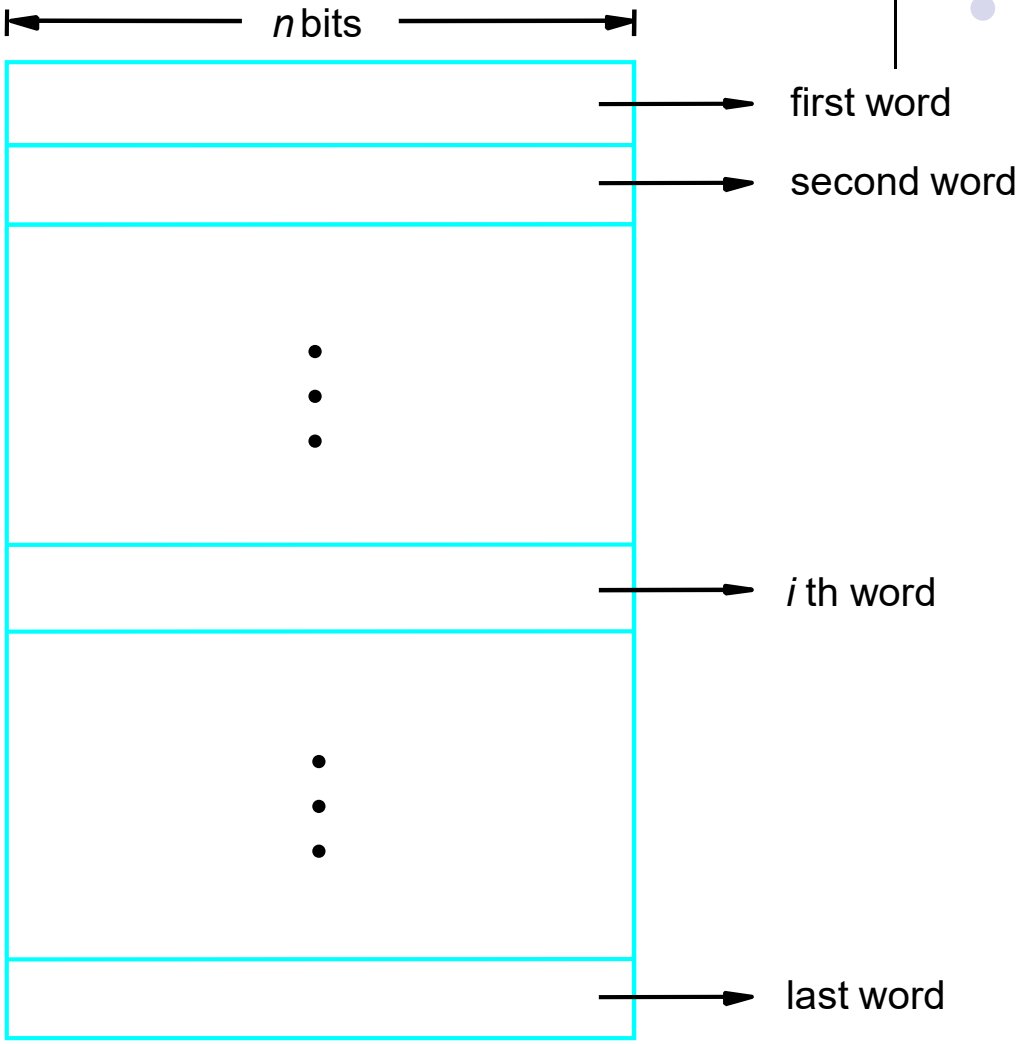
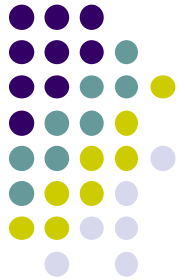
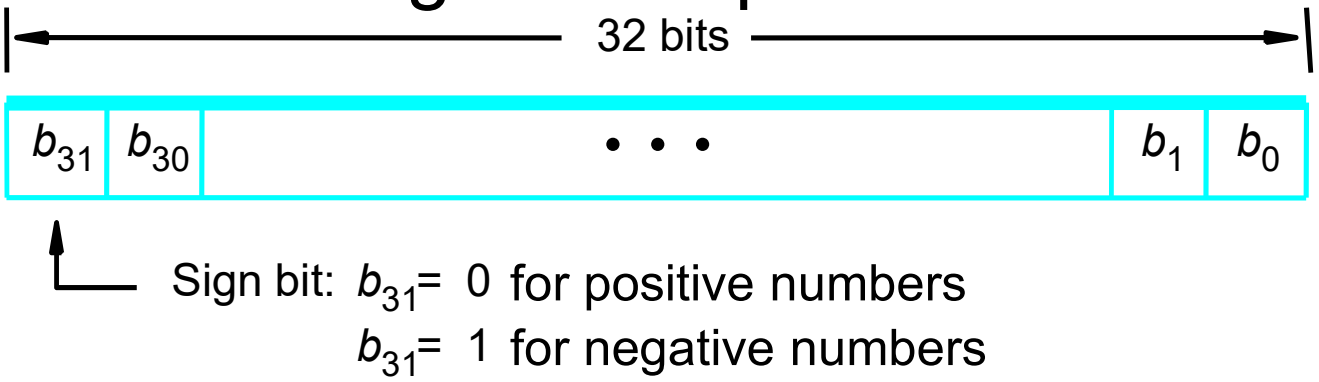


Figure 2.5. Memory words.

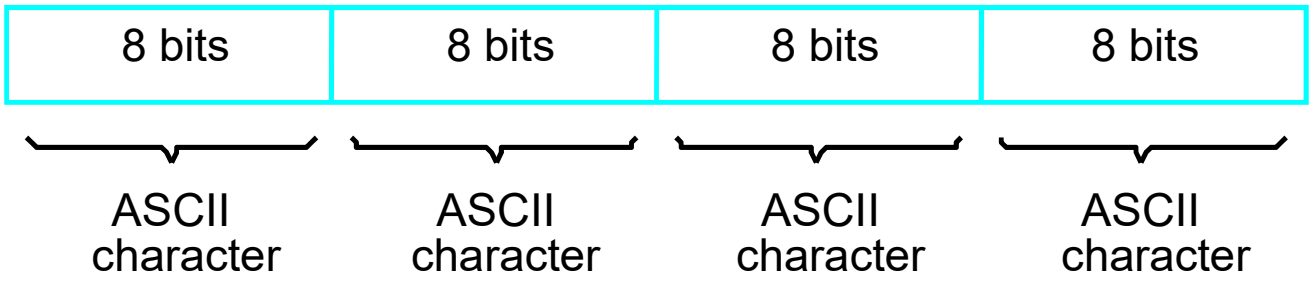
Memory Location, Addresses, and Operation



- 32-bit word length example



(a) A signed integer



(b) Four characters

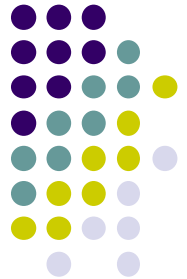
Memory Location, Addresses, and Operation

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

Memory Location, Addresses, and Operation

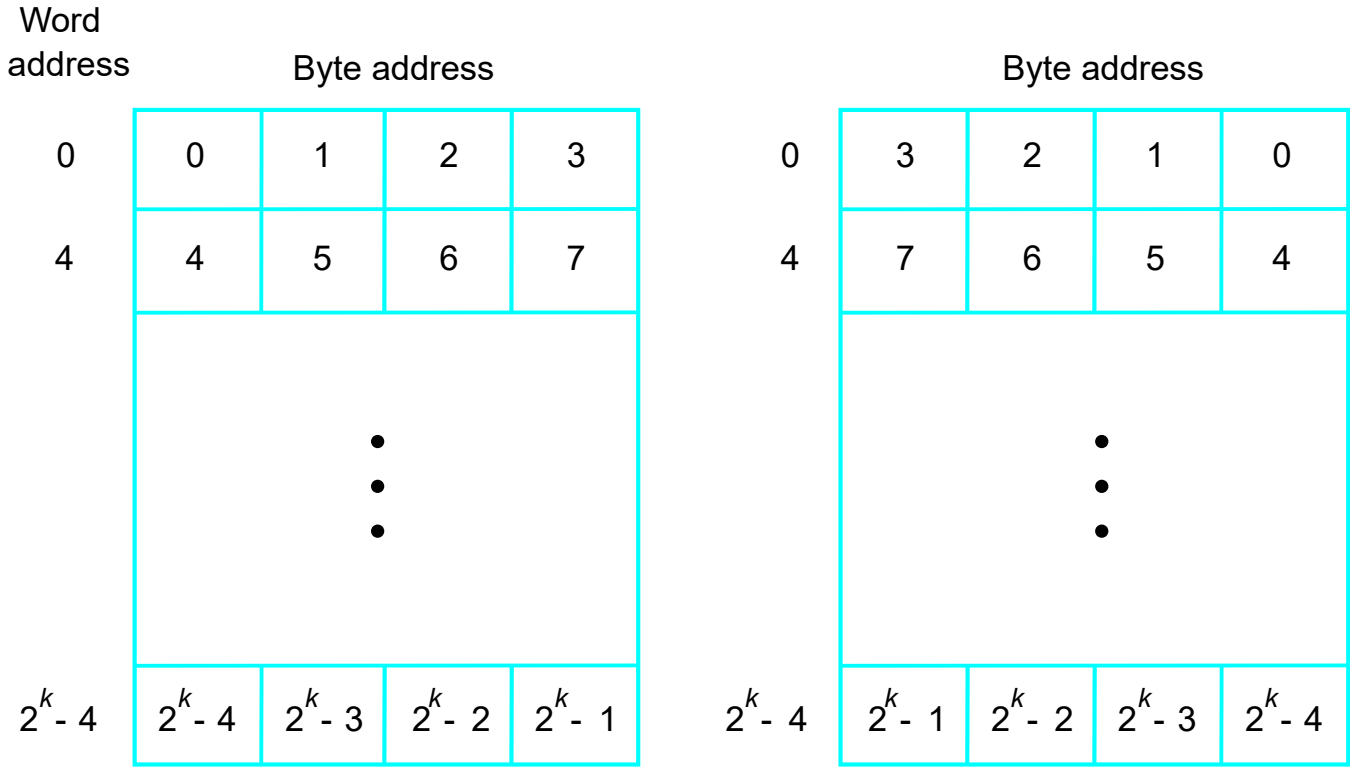
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits,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

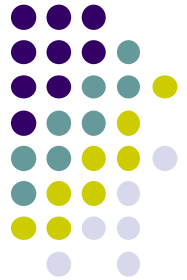
Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word



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

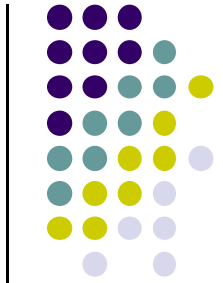
Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation



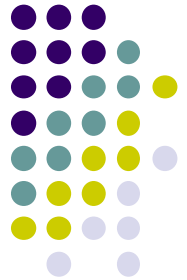
- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….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

Memory Operation

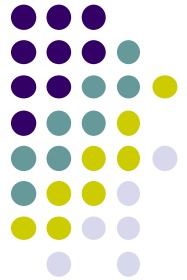


- Load (or Read or Fetch)
  - Copy the content. The memory content doesn't change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

Instruction and Instruction Sequencing



“Must-Perform” Operations

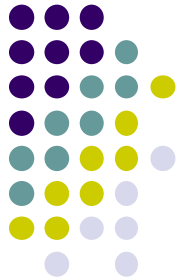


- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

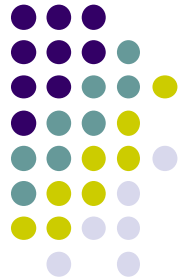
- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation



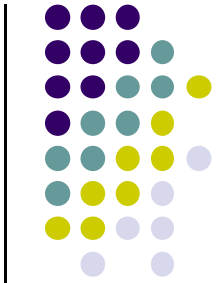
- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

CPU Organization



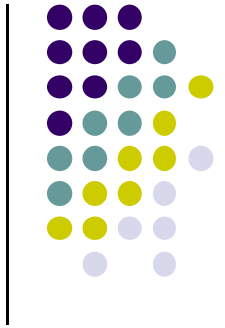
- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats



- Three-Address Instructions
  - ADD R1, R2, R3 R1 ← R2 + R3
- Two-Address Instructions
  - ADD R1, R2 R1 ← R1 + R2
- One-Address Instructions
  - ADD M AC ← AC + M[AR]
- Zero-Address Instructions
  - ADD TOS ← TOS + (TOS – 1)
- RISC Instructions
  - Lots of registers. Memory is restricted to Load & Store

Instruction Formats

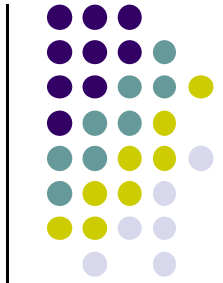


Example: Evaluate (A+B) * (C+D)

- Three-Address

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

Instruction Formats

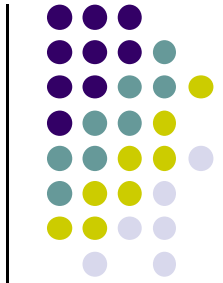


Example: Evaluate (A+B) * (C+D)

- Two-Address

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

Instruction Formats

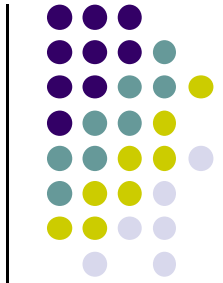


Example: Evaluate (A+B) * (C+D)

- 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

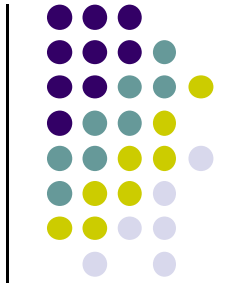
1.	PUSH A	; TOS ← A
2.	PUSH B	; TOS ← B
3.	ADD	; TOS ← (A + B)
4.	PUSH C	; TOS ← C
5.	PUSH D	; TOS ← D
6.	ADD	; TOS ← (C + D)
7.	MUL (C+D)*(A+B)	; TOS ←
8.	POP X	; M[X] ← TOS

Instruction Formats

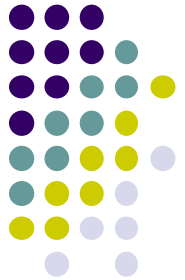
Example: Evaluate (A+B) * (C+D)

- RISC

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

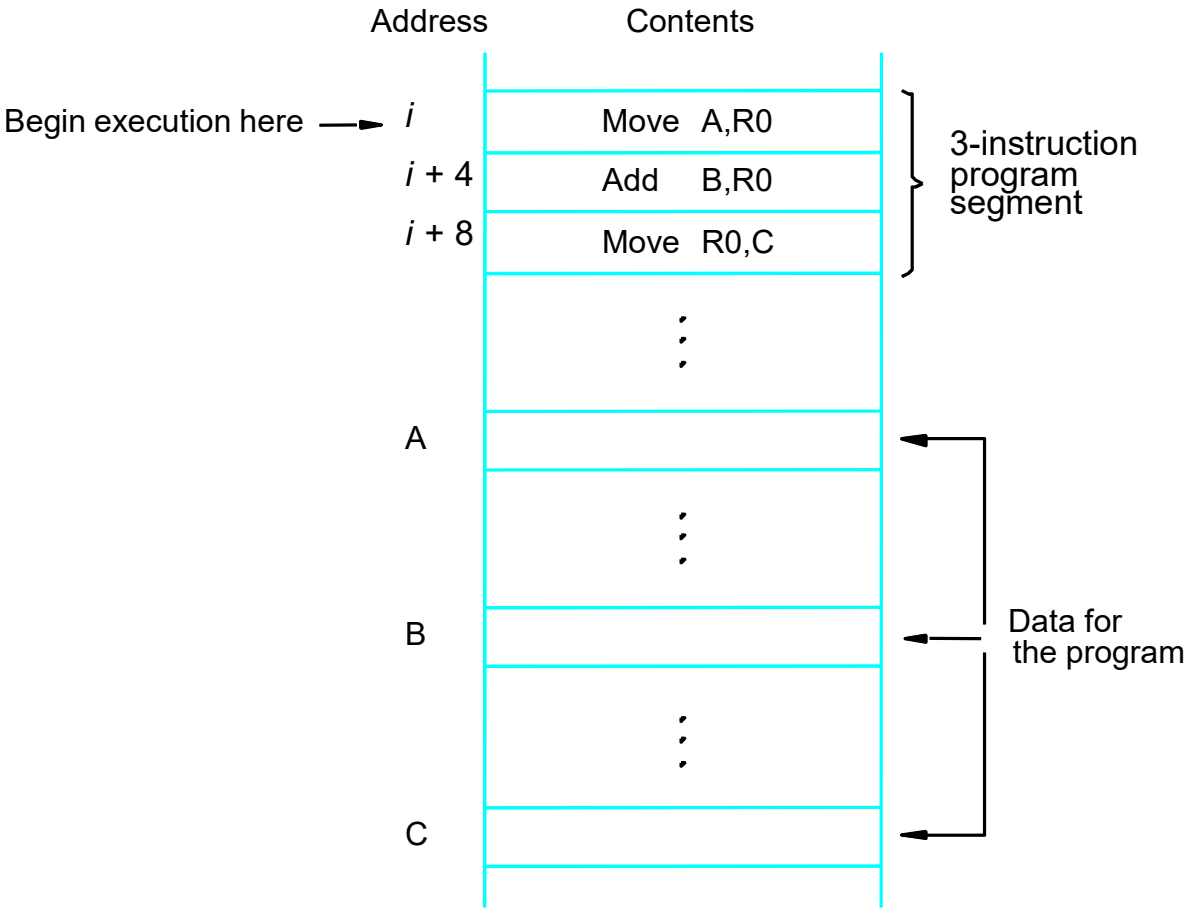


Using Registers



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

Instruction Execution 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 ← [A] + [B].

Branching

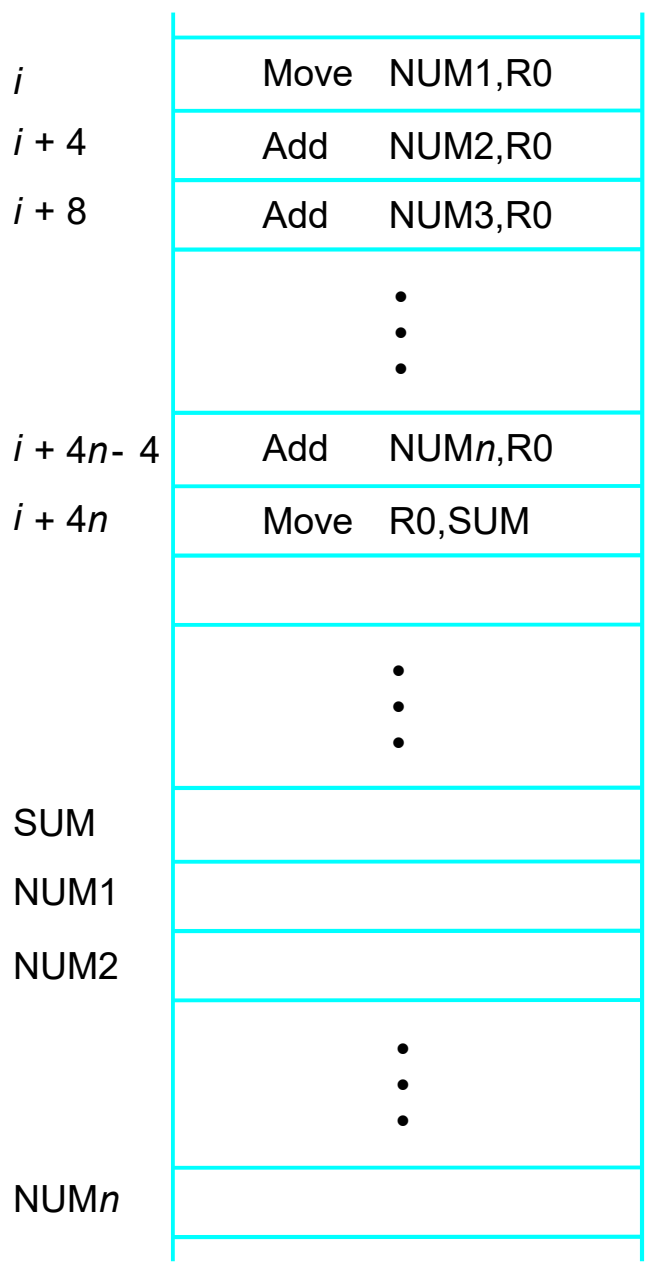


Figure 2.9. A straight-line program for adding *n* numbers.

Branching

Branch target

Conditional branch

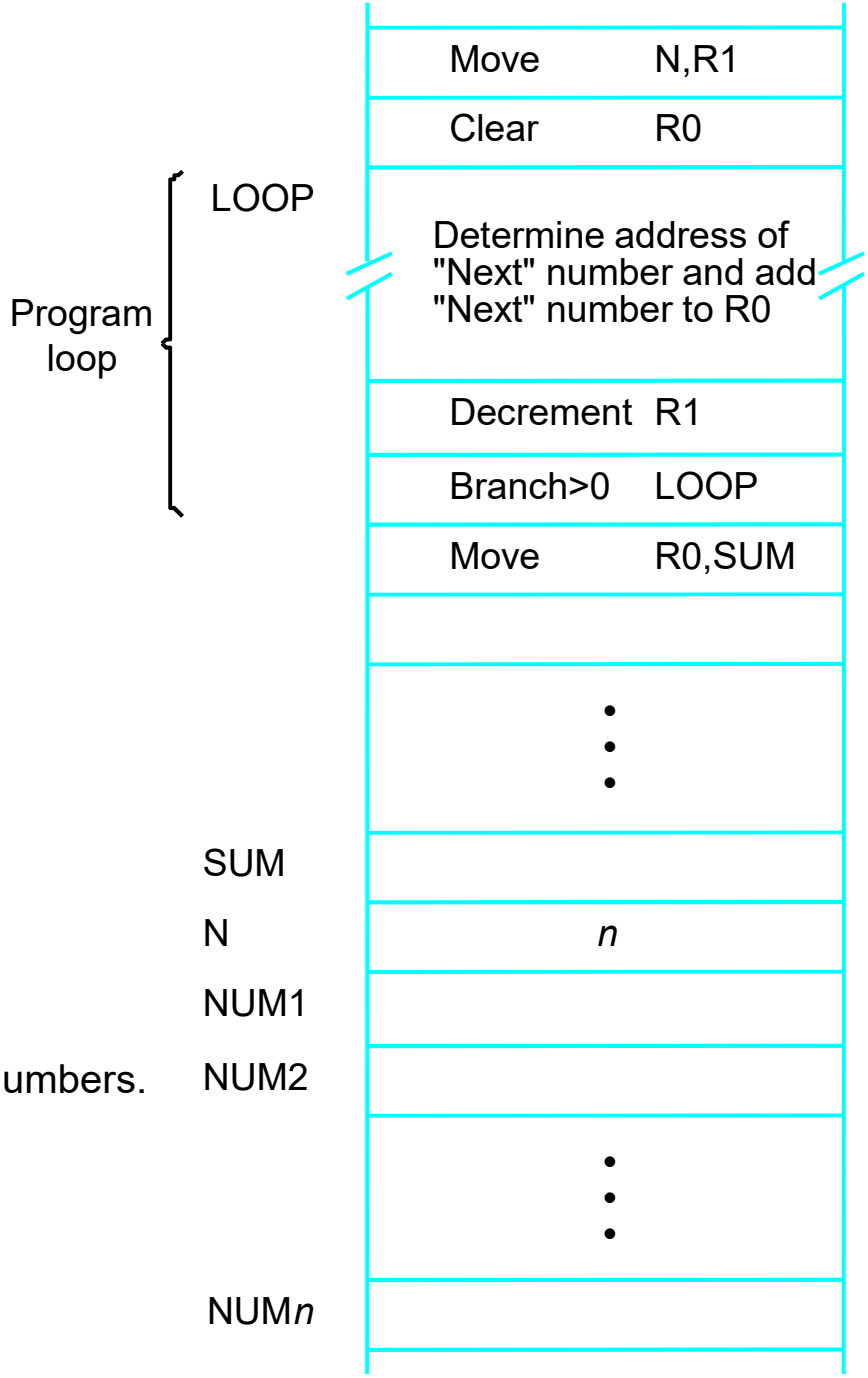
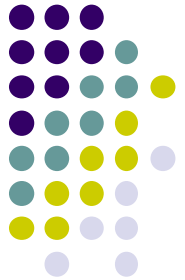


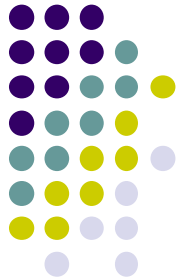
Figure 2.10. Using a loop to add *n* numbers.

Condition Codes

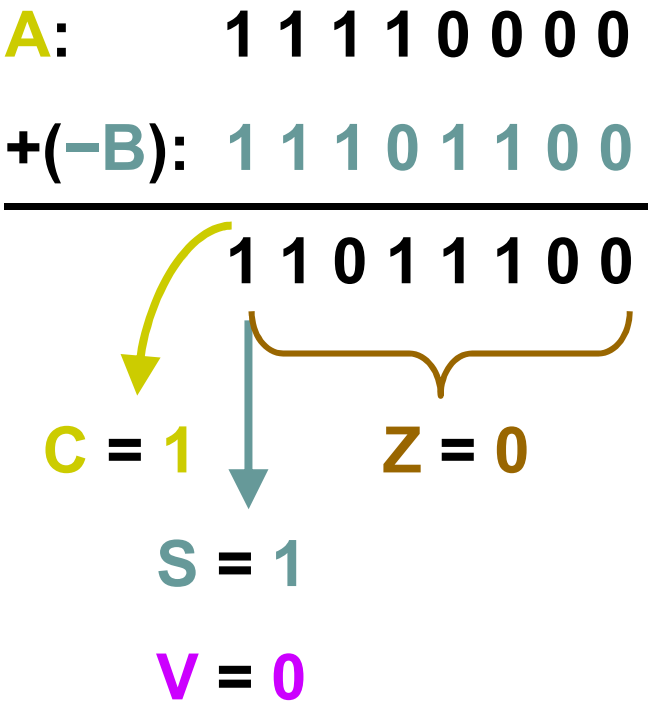


- Condition code flags
- Condition code register / status register
- N (negative)
- Z (zero)
- V (overflow)
- C (carry)
- Different instructions affect different flags

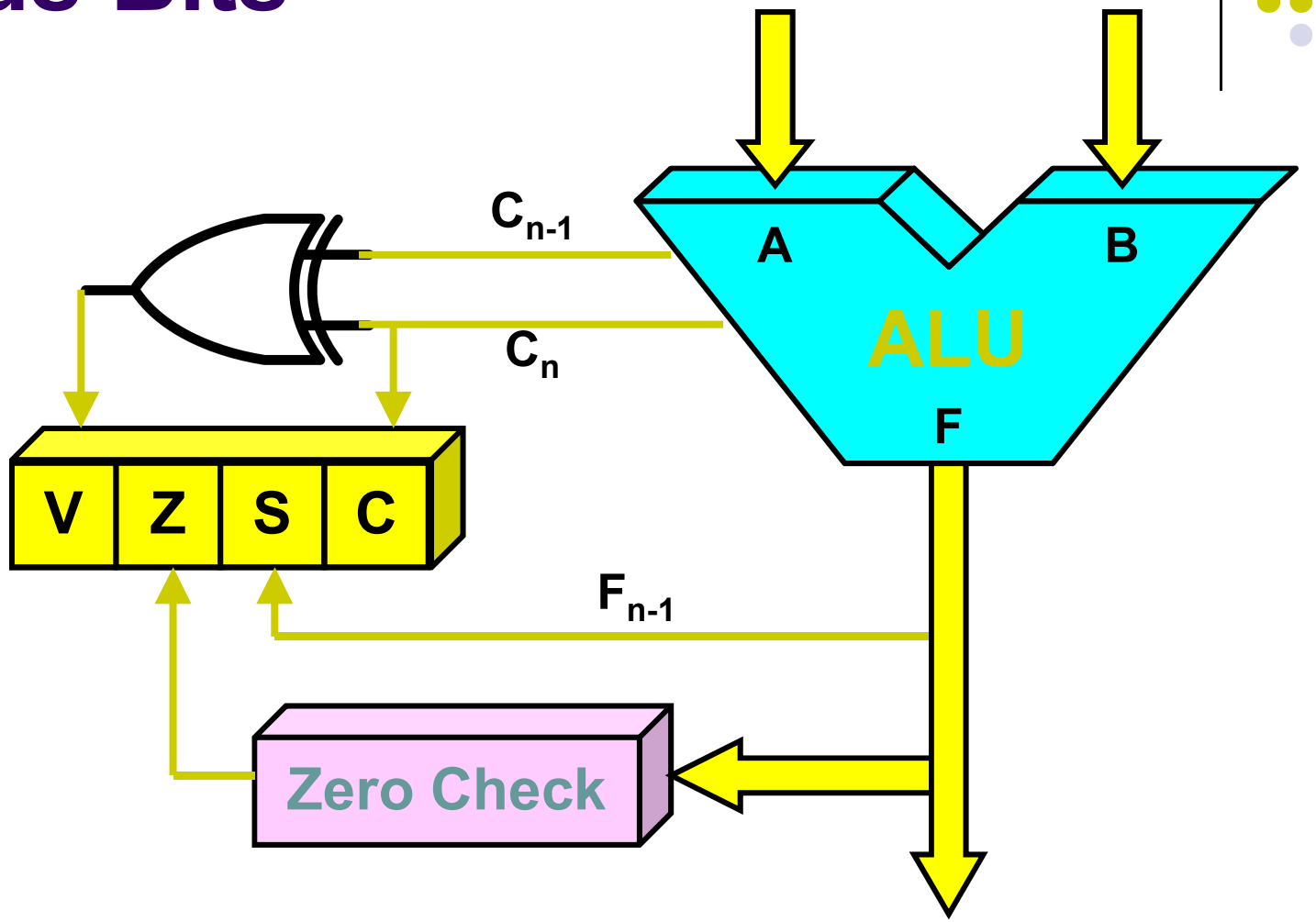
Conditional Branch Instructions



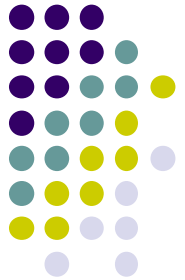
- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0



Status Bits



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.

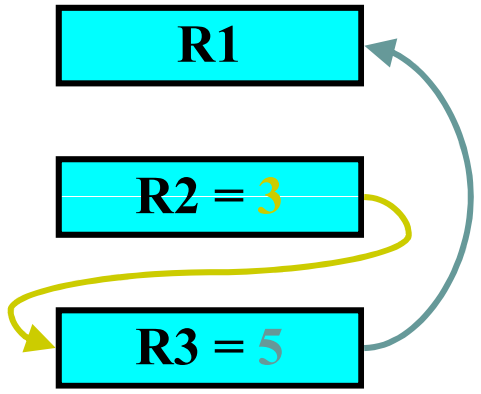
Addressing Modes

- Implied
  - AC is implied in “ADD M[AR]” in “One-Address” instr.
  - TOS is implied in “ADD” in “Zero-Address” instr.
- Immediate
  - The use of a constant in “MOV R1, 5”, i.e. R1 ← 5
- Register
  - Indicate which register holds the operand

Addressing Modes

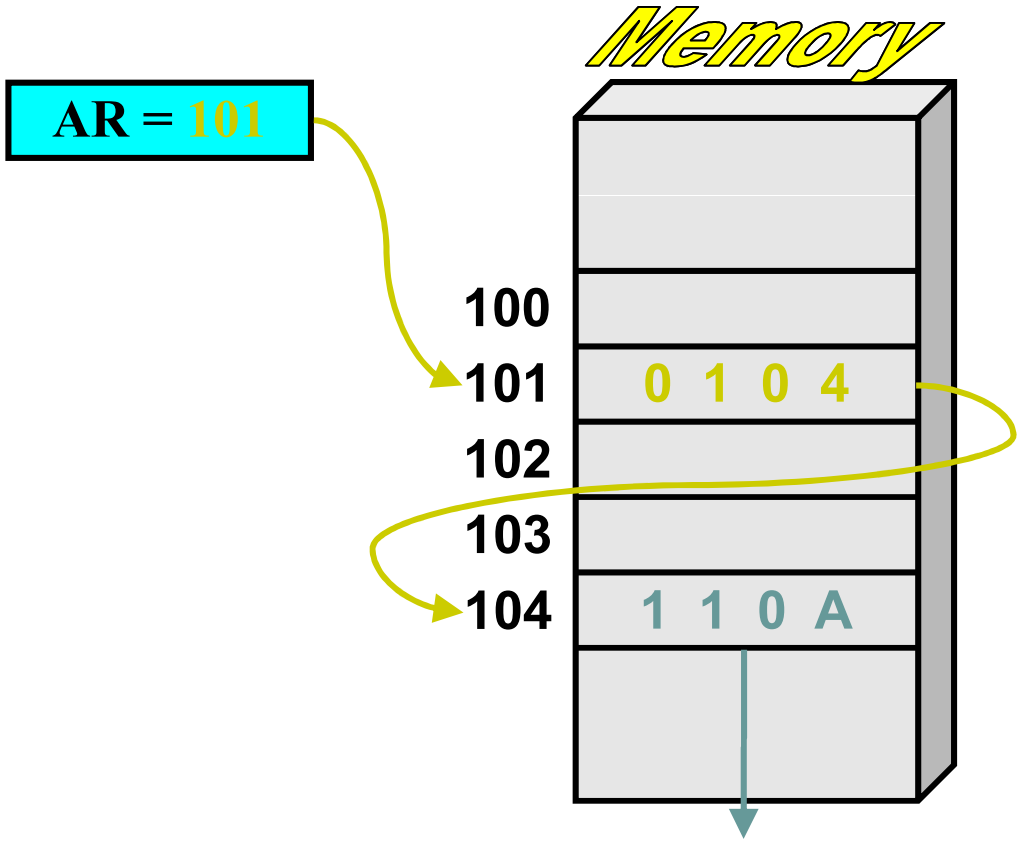
- Register Indirect
  - Indicate the register that holds the number of the register that holds the operand

    MOV R1, (R2)
- Autoincrement / Autodecrement
  - Access & update in 1 instr.
- Direct Address
  - Use the given address to access a memory location



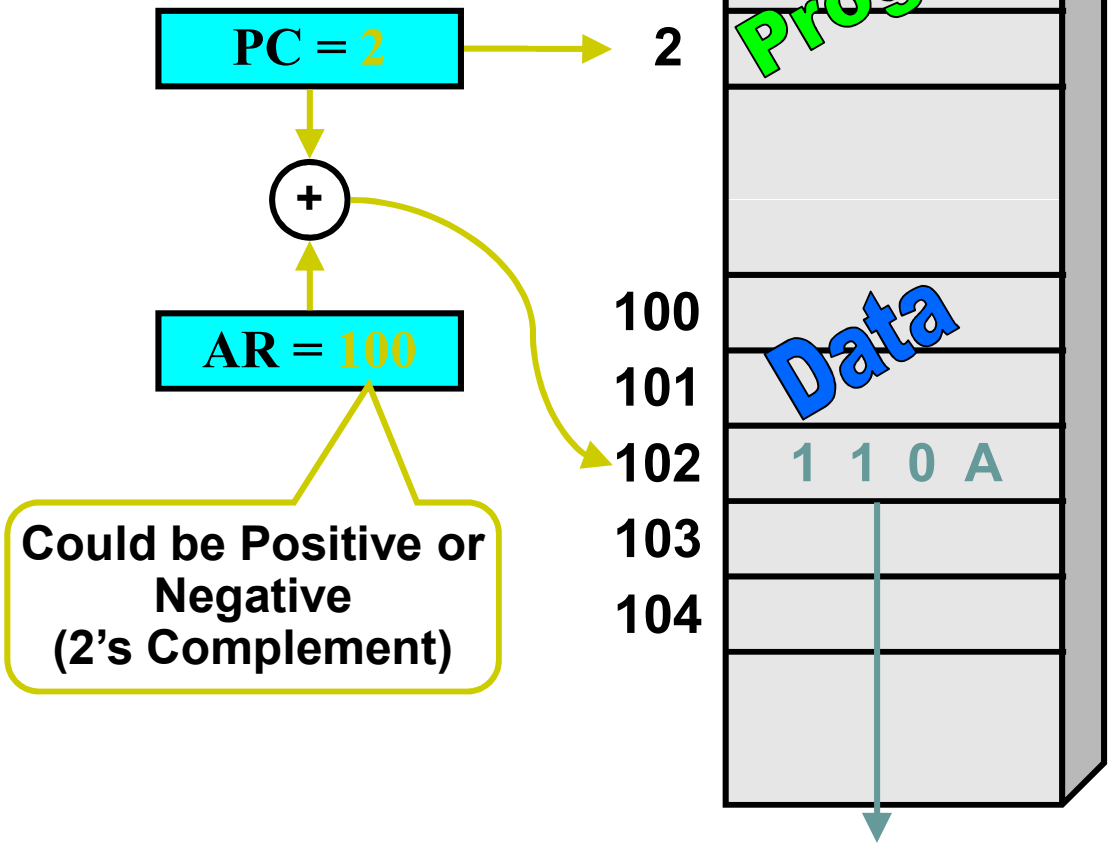
Addressing Modes

- Indirect Address
  - Indicate the memory location that holds the address of the memory location that holds the data



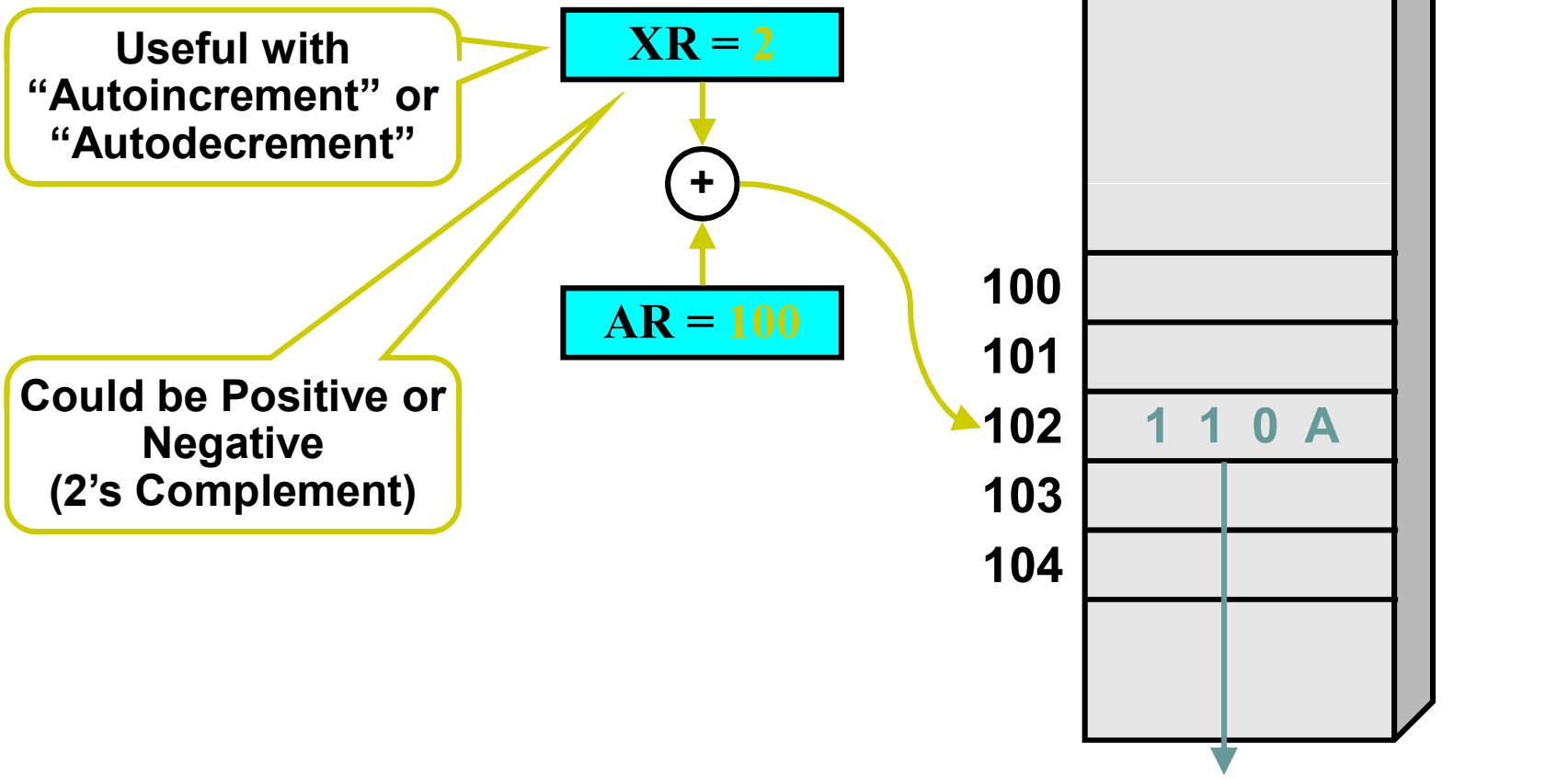
Addressing Modes

- Relative Address
  - *EA* = PC + Relative Addr



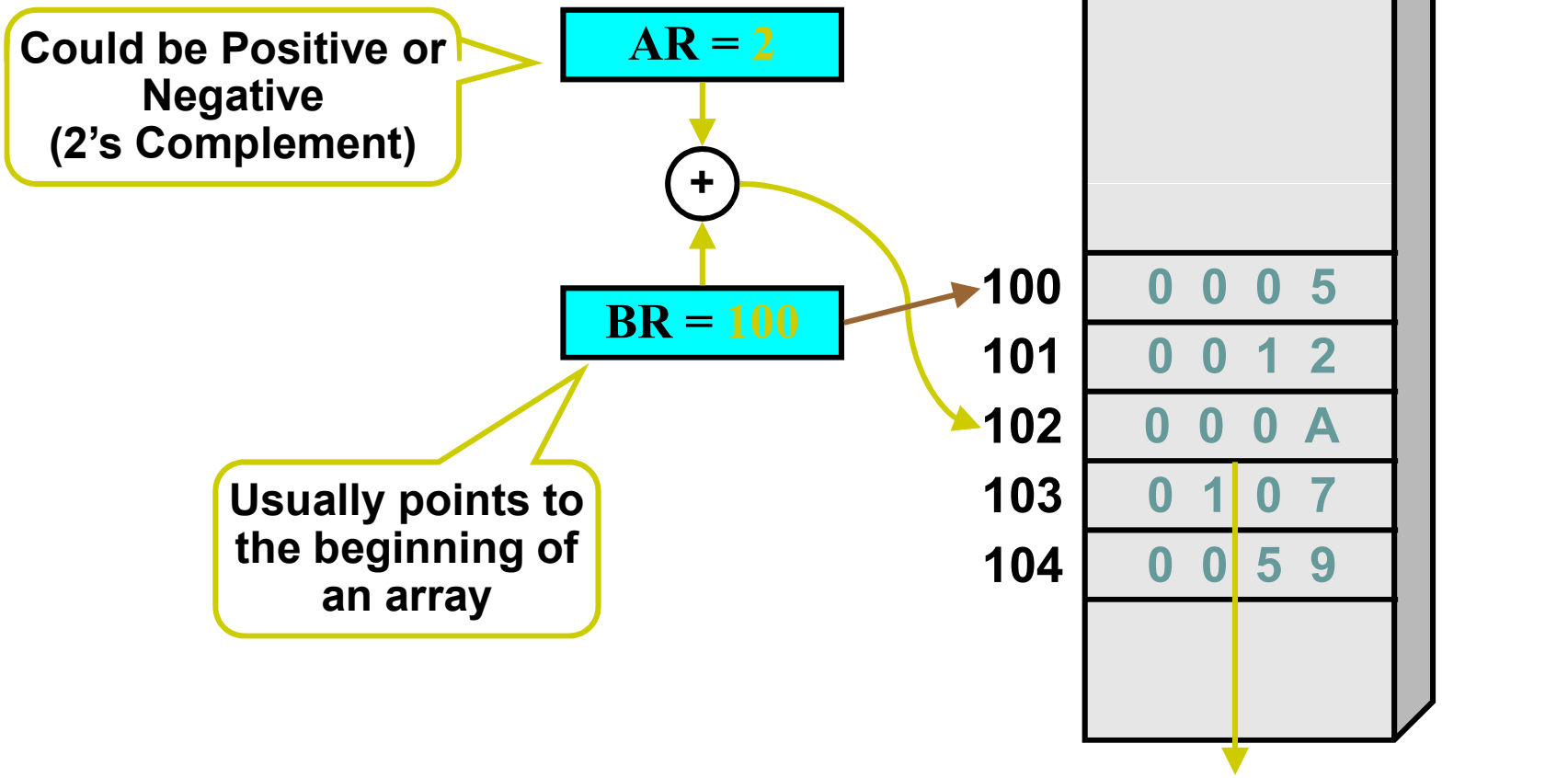
Addressing Modes

- Indexed
  - *EA* = Index Register + Relative Addr



Addressing Modes

- Base Register
  - *EA* = Base Register + Relative Addr

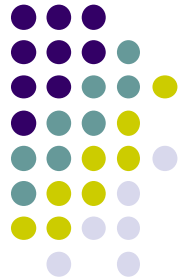


Addressing Modes

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

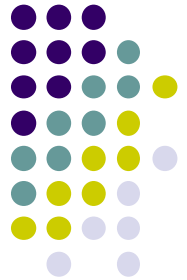
Name	Assembler syntax	Addressingfunction
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Basewith index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Basewith index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Indexing and Arrays



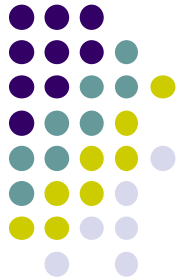
- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register
- $X(R_i)$: $EA = X + [R_i]$
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than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

Indexing and Arrays



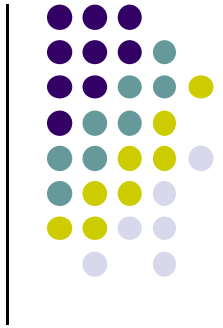
- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
- Several variations:
  (R_i, R_j): EA = $[R_i] + [R_j]$
  $X(R_i, R_j)$: EA = $X + [R_i] + [R_j]$

Relative Addressing



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

Additional Modes

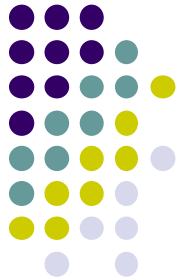


- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- $(R_i)+$. The increment is 1 for byte-sized operands,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

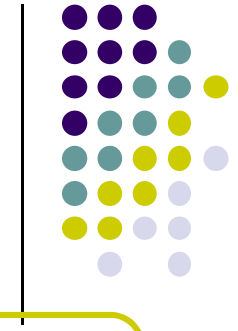
```
                Move        N,R1
                Move        #NUM1,R2      } Initialization
                Clear       R0
LOOP            Add         (R2)+,R0
                Decrement   R1
                Branch>0    LOOP
                Move        R0,SUM
```

Figure 2.16. The Autoincrement addressing mode used in the program of Figure 2.12.

Assembly Language



Types of Instructions

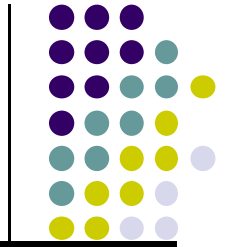


- Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

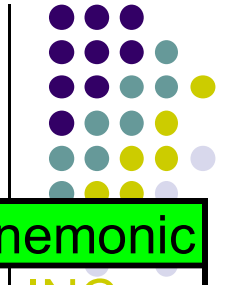
Data value is not modified

Data Transfer Instructions



Mode	Assembly	Register Transfer
Direct address	LD ADR	*AC* ← *M*[*ADR*]
Indirect address	LD @ADR	*AC* ← *M*[*M*[*ADR*]]
Relative address	LD $ADR	*AC* ← *M*[*PC*+*ADR*]
Immediate operand	LD #NBR	*AC* ← NBR
Index addressing	LD ADR(X)	*AC* ← *M*[*ADR*+*XR*]
Register	LD R1	*AC* ← *R1*
Register indirect	LD (R1)	*AC* ← *M*[*R1*]
Autoincrement	LD (R1)+	*AC* ← *M*[*R1*], *R1* ← *R1*+1

Data Manipulation Instructions



- Arithmetic
- Logical & Bit Manipulation
- Shift

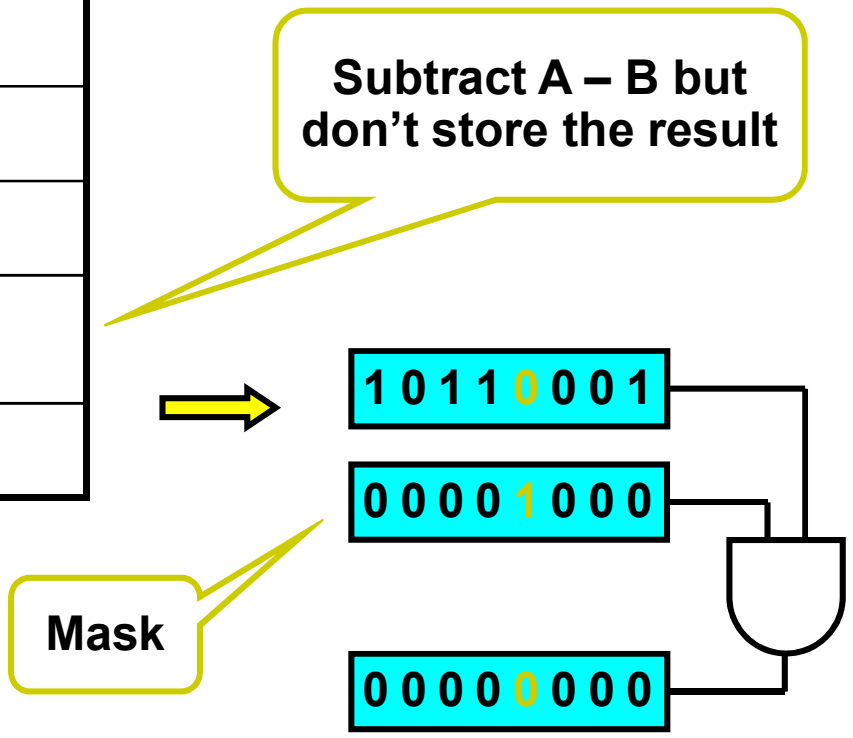
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate	NEG

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

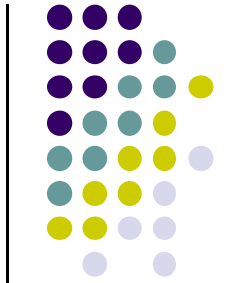
Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (Subtract)	CMP
Test (AND)	TST

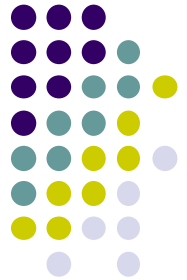


Conditional Branch Instructions

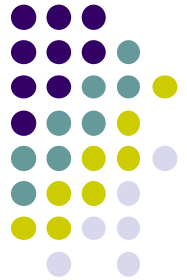


Mnemonic	Branch Condition	Tested Condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0

Basic Input/Output Operations

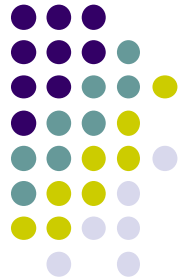


I/O



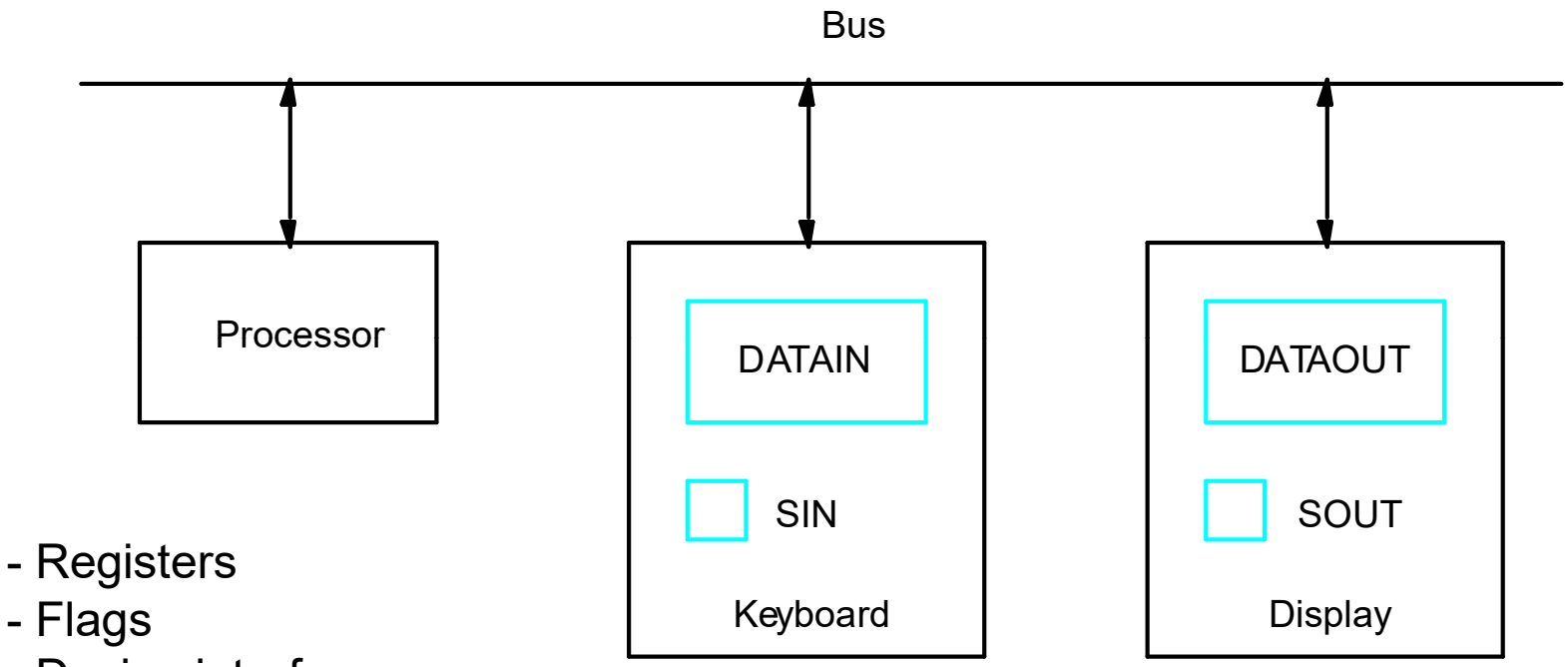
- The data on which the instructions operate are not necessarily already stored in memory.
- Data need to be transferred between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

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.

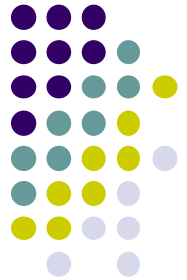
Program-Controlled I/O Example



- Registers
- Flags
- Device interface

Figure 2.19 Bus connection for processor, keyboard, and display

Program-Controlled I/O Example

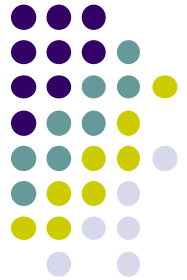


- Machine instructions that can check the state of the status flags and transfer data:

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

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

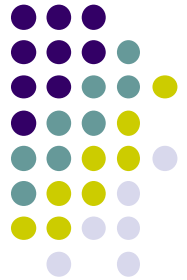
Program-Controlled I/O Example



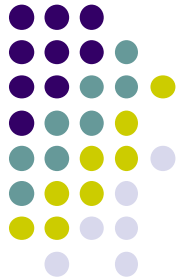
- Memory-Mapped I/O – some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also use device status registers.

```
READWAIT  Testbit   #3, INSTATUS
          Branch=0  READWAIT
          MoveByte  DATAIN, R1
```

Program-Controlled I/O Example

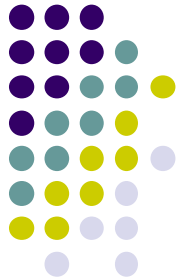


- Assumption – the initial state of SIN is 0 and the initial state of SOUT is 1.
- Any drawback of this mechanism in terms of efficiency?
  - Two wait loops→processor execution time is wasted
- Alternate solution?
  - Interrupt



Stacks

Home Work

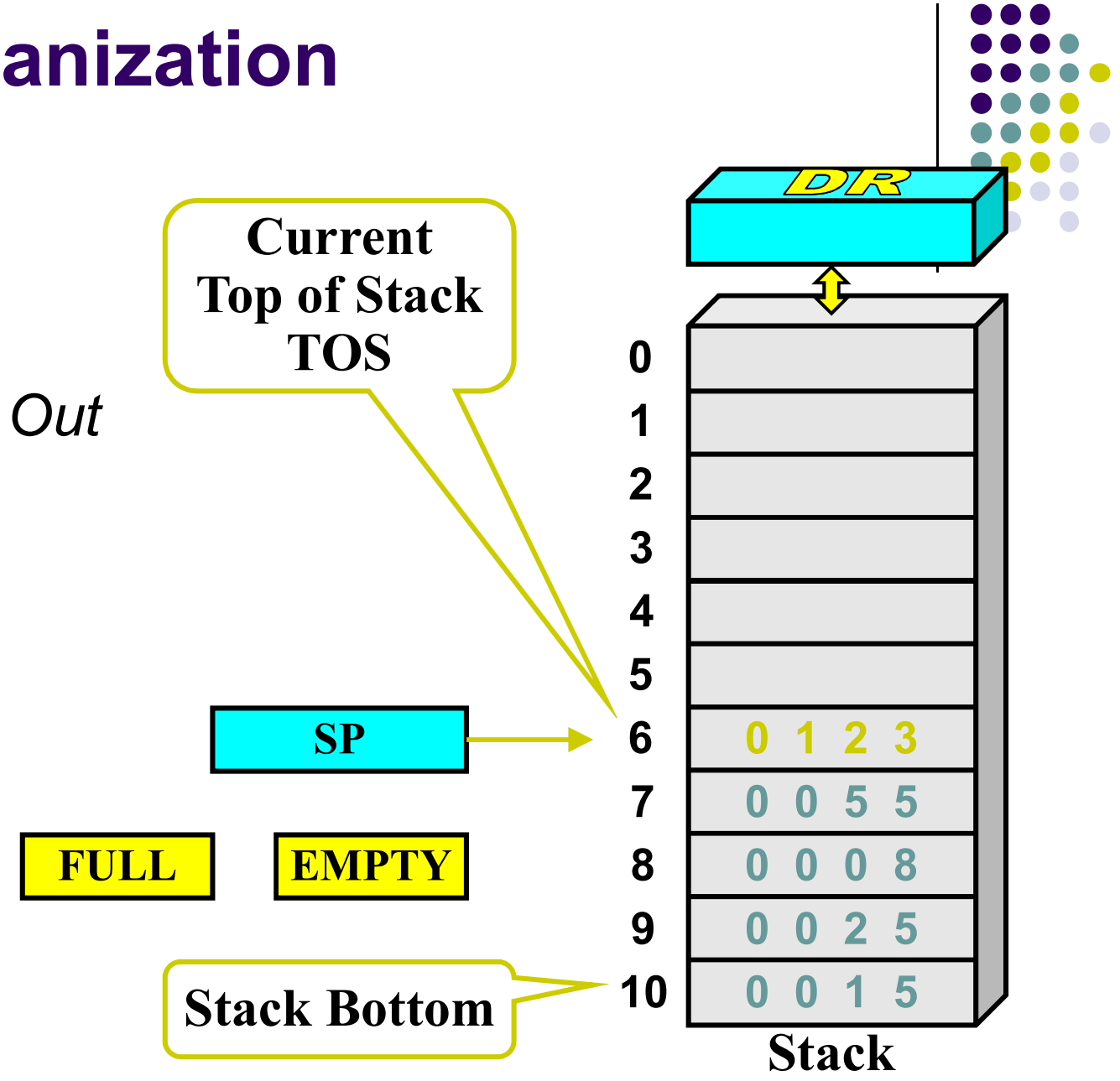


- For each Addressing modes mentioned before, state one example for each addressing mode stating the specific benefit for using such addressing mode for such an application.

Stack Organization

- LIFO

  Last In First Out



Stack Organization

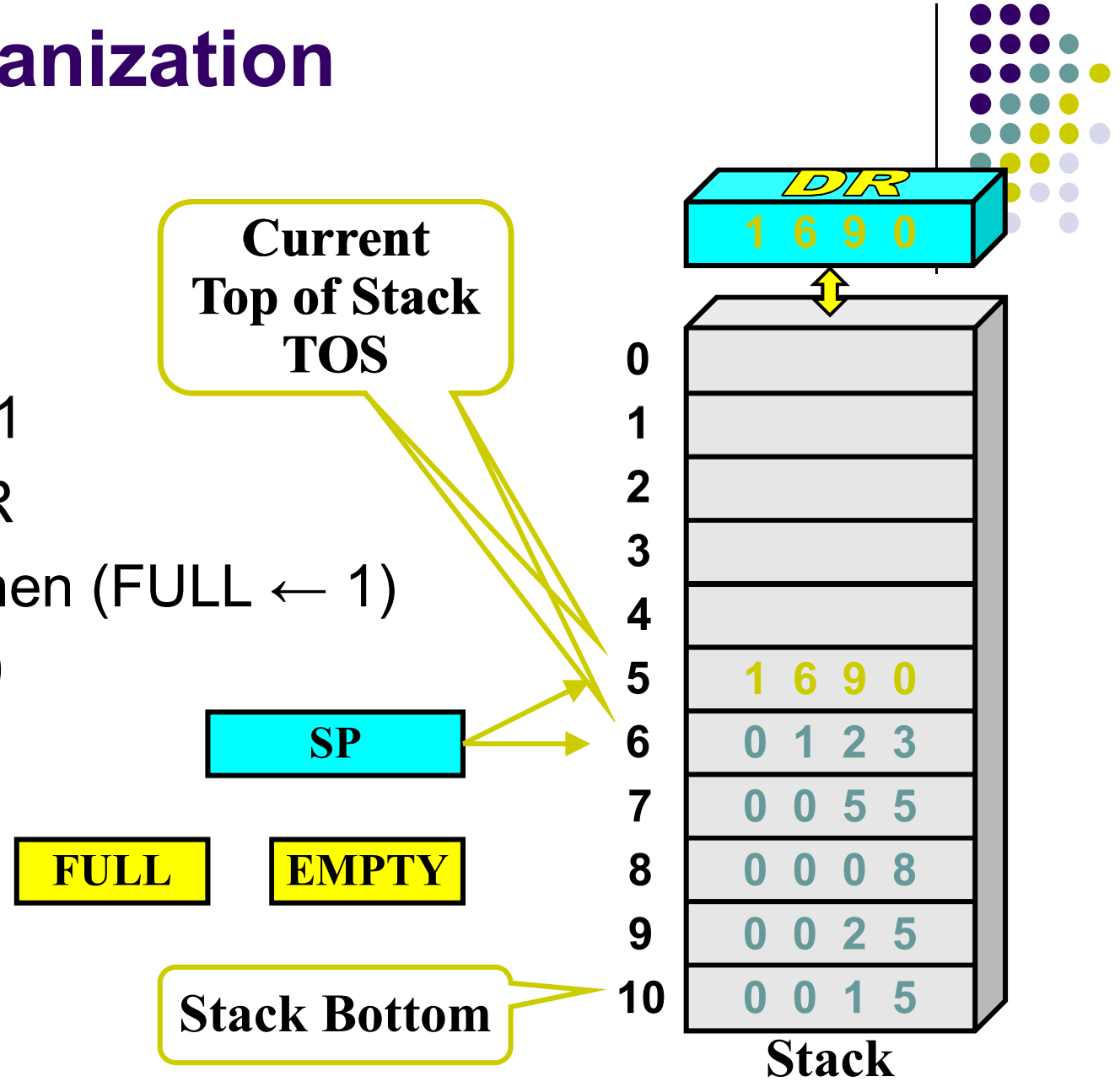
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

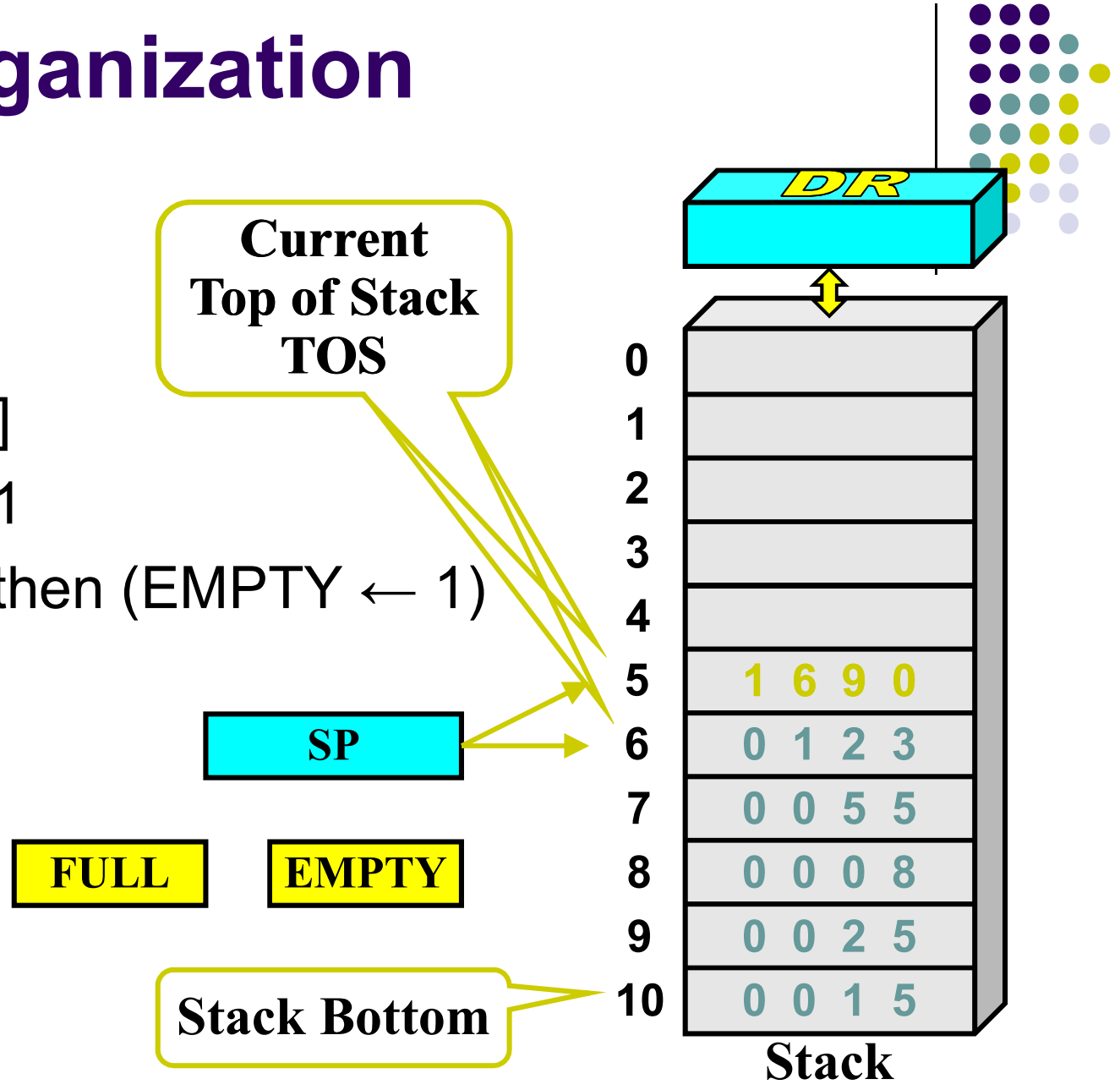
- POP

  DR ← M[SP]

  SP ← SP + 1

  If (SP = 11) then (EMPTY ← 1)

  FULL ← 0



Stack Organization

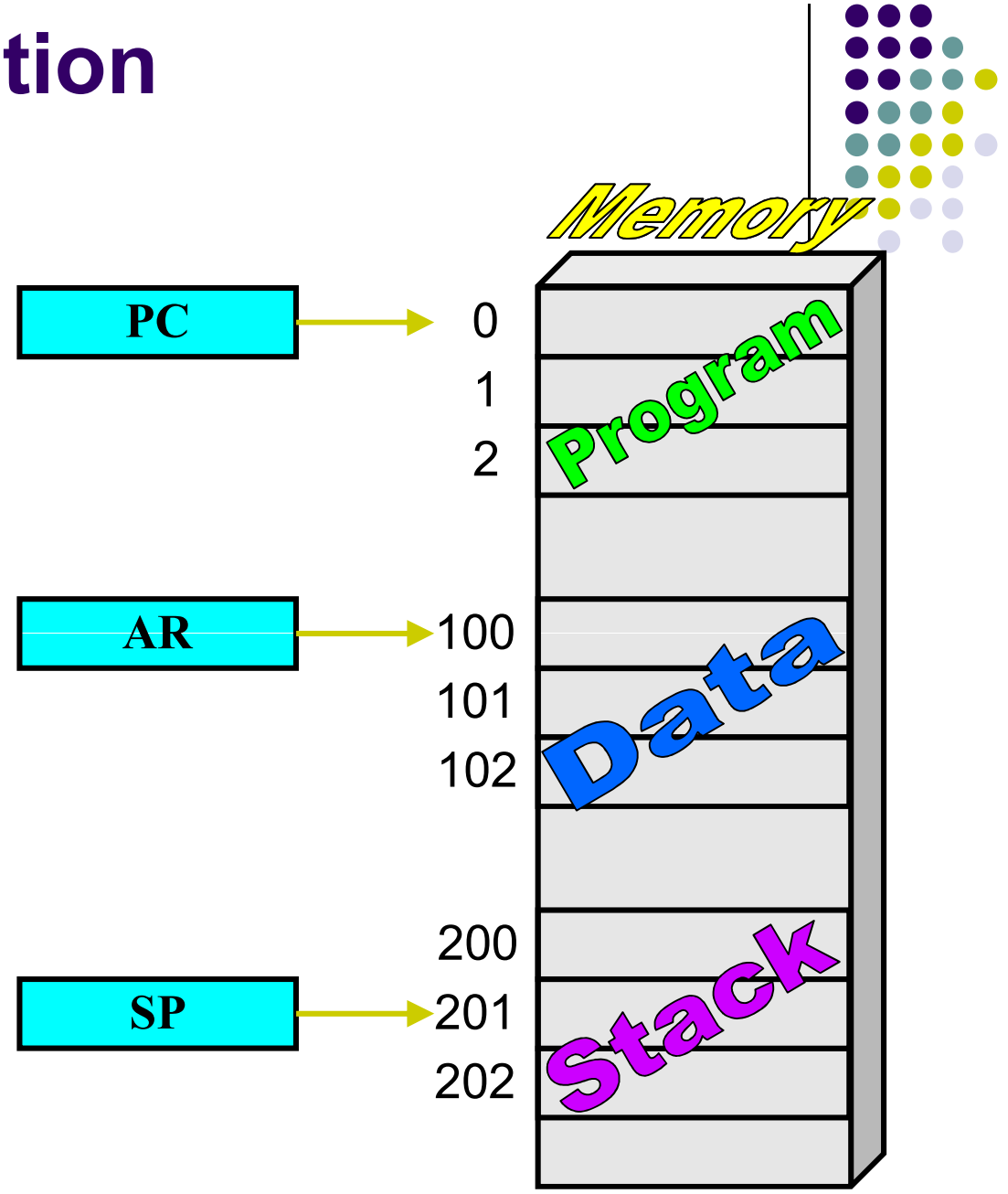
- Memory Stack
  - PUSH

    SP ← SP – 1

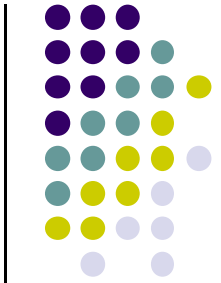
    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1



Reverse Polish Notation



- Infix Notation

  A + *B*

- Prefix or Polish Notation

  + *A B*

- Postfix or Reverse Polish Notation (RPN)

  A B +

$$A * B + C * D \xrightarrow{\text{RPN}} A\ B * C\ D * +$$

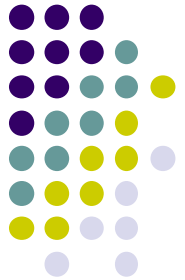
(2) (4) * (3) (3) * +

(8) (3) (3) * +

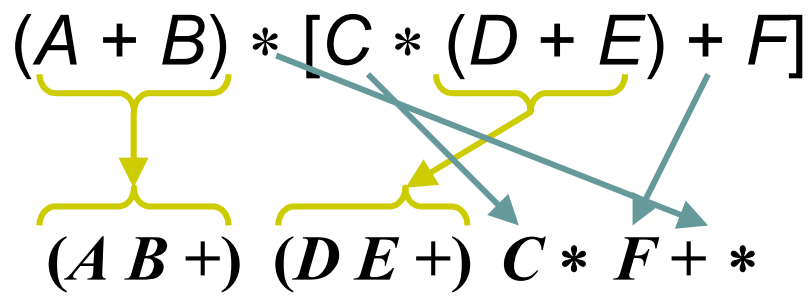
(8) (9) +

17

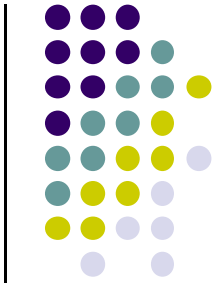
Reverse Polish Notation



- Example



Reverse Polish Notation



- Stack Operation

  (3) (4) ∗ (5) (6) ∗ +

  PUSH 3

  PUSH 4

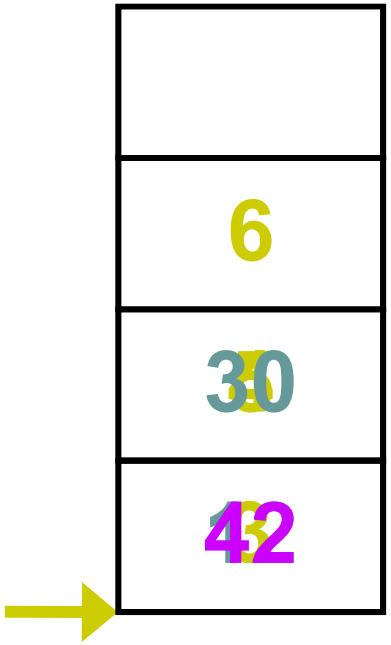
  MULT

  PUSH 5

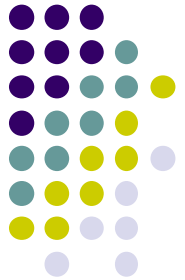
  PUSH 6

  MULT

  ADD

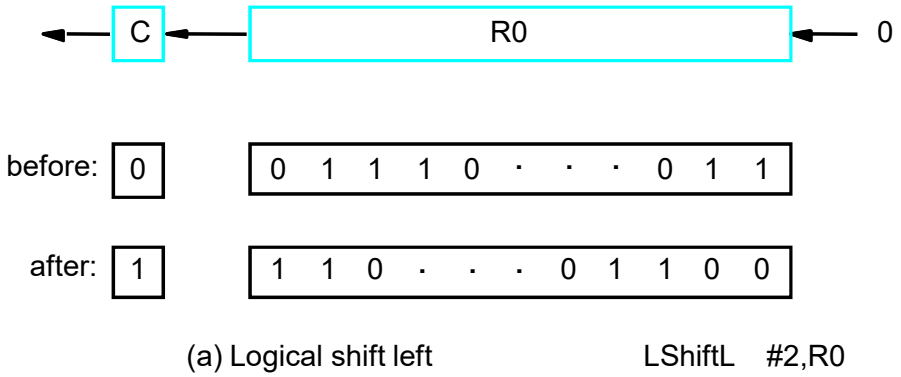


Additional Instructions

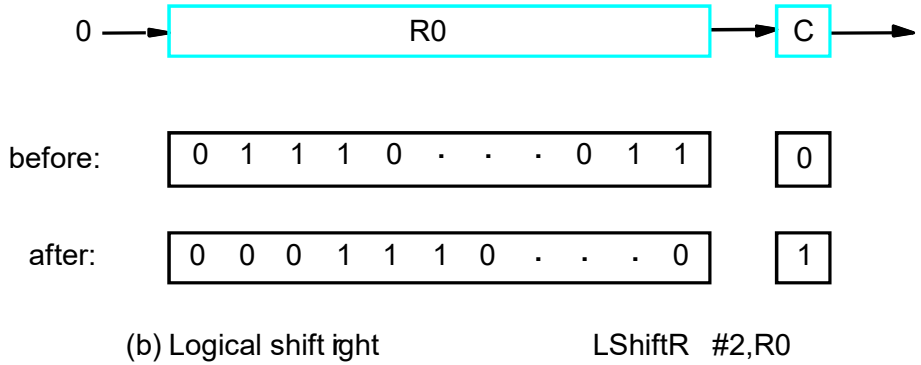


Logical Shifts

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

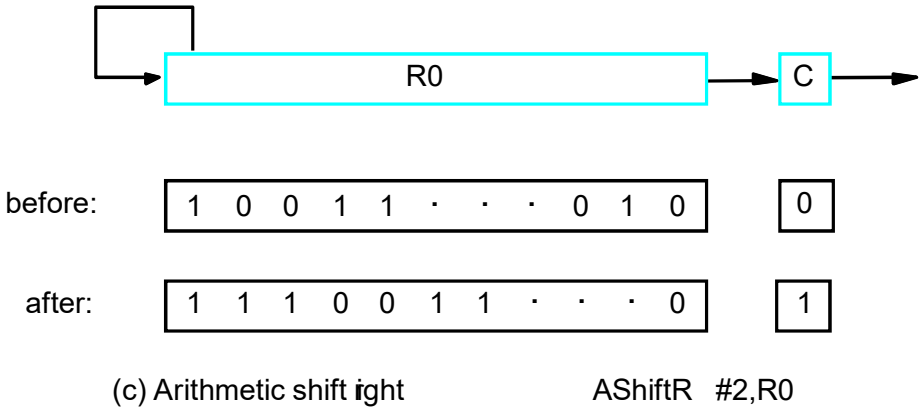


(a) Logical shift left LShiftL #2,R0



(b) Logical shift right LShiftR #2,R0

Arithmetic Shifts



(c) Arithmetic shift right AShiftR #2,R0

Rotate

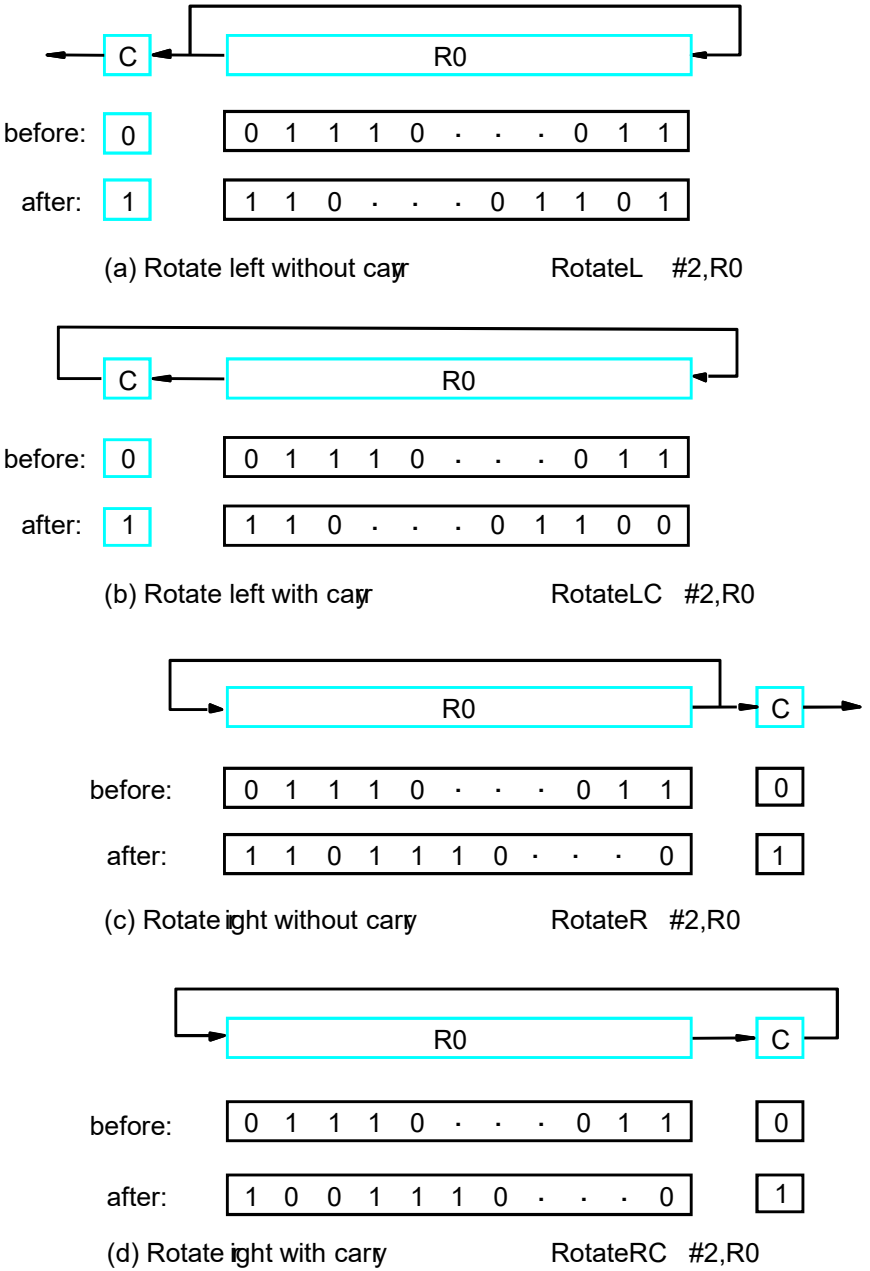
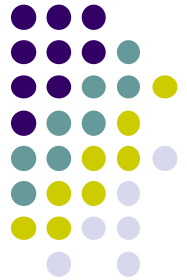


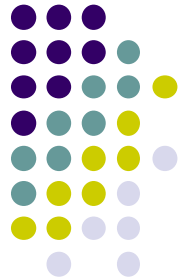
Figure 2.32. Rotate instructions.

Multiplication and Division

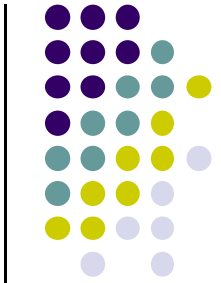


- Not very popular (especially division)
- Multiply R_i, R_j
  $R_j \leftarrow [R_i] \times [R_j]$
- 2n-bit product case: high-order half in R(j+1)
- Divide R_i, R_j
  $R_j \leftarrow [R_i] / [R_j]$
  Quotient is in Rj, remainder may be placed in R(j+1)

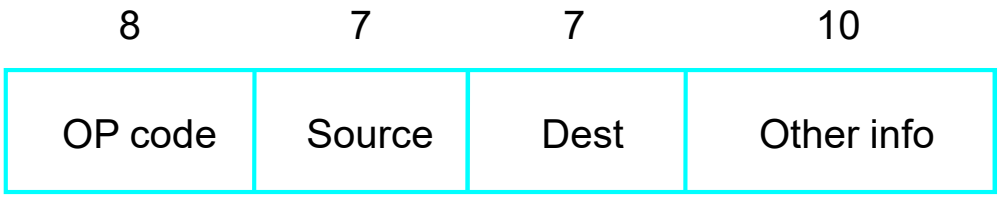
Encoding of Machine Instructions



Encoding of Machine Instructions

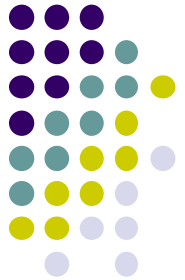


- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)
- In the previous section, an assumption was made that all instructions are one word in length.
- OP code: the type of operation to be performed and the type of operands used may be specified 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
- Branch>0 LOOP

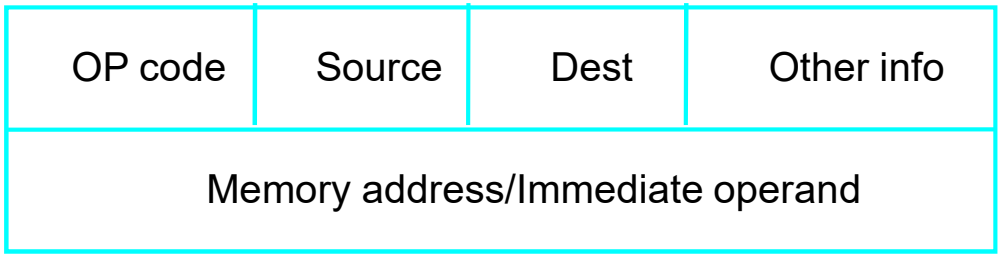


(a) One-word instruction

Encoding of Machine Instructions

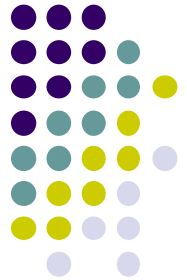


- What happens if we want to specify a memory operand using the Absolute addressing mode?
- Move R2, LOC
- 14-bit for LOC – insufficient
- Solution – use two words



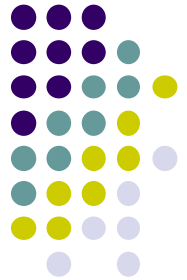
(b) Two-word instruction

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