UNIT 1 - Basic Structure of Computers

Functional Units

Functional Units

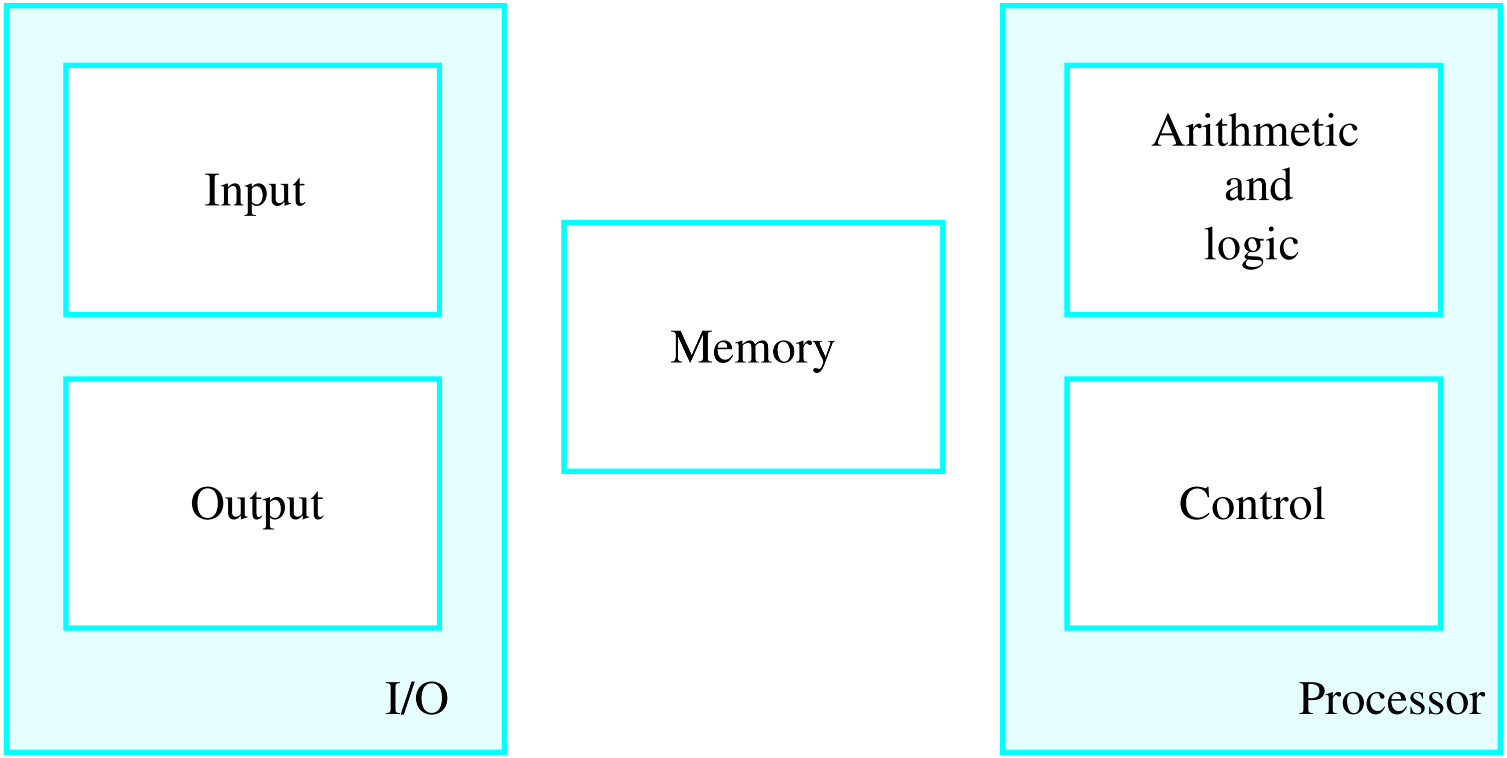


Figure 1.1. Basic functional units of a computer.

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

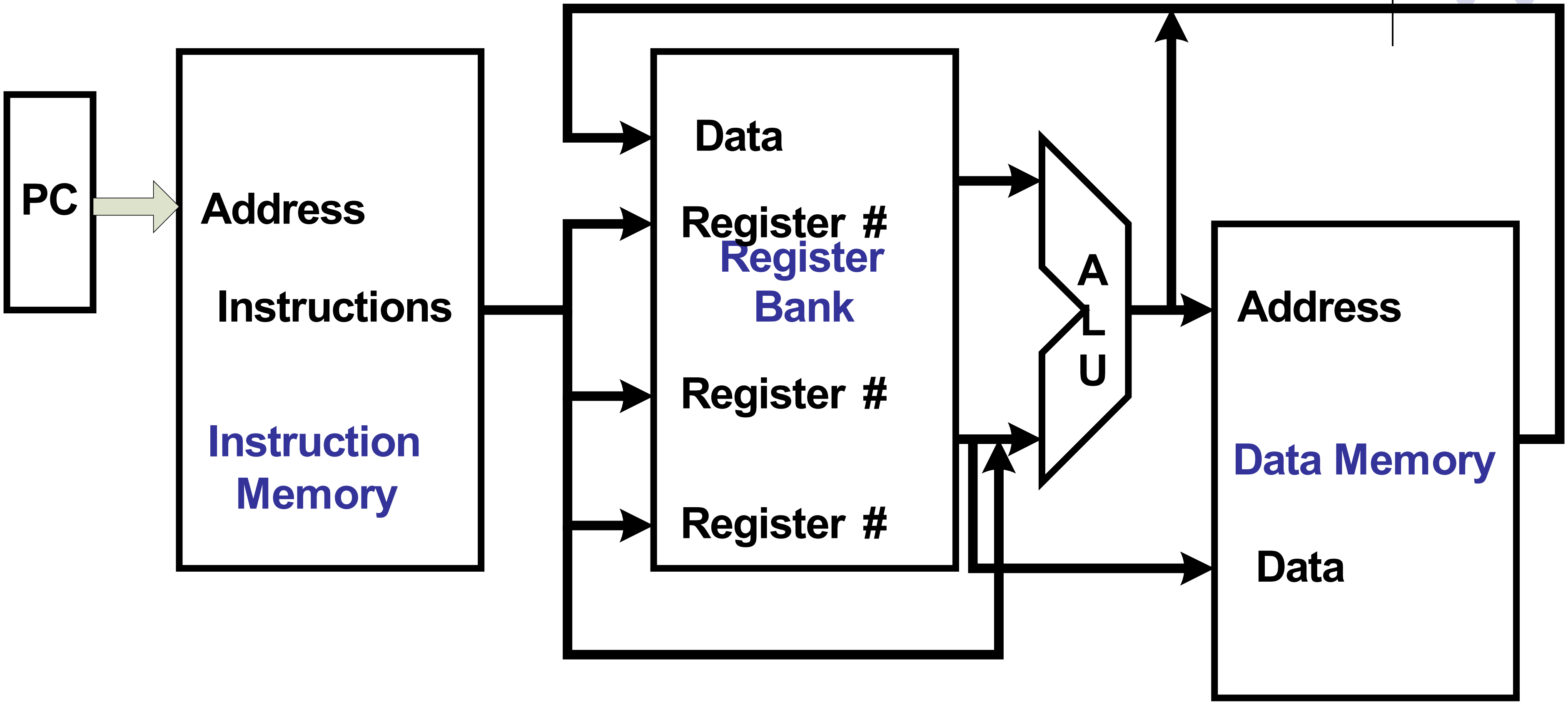
Arithmetic and Logic Unit (ALU)

- 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.
- Operations of a computer:
  - 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 an ALU, where the information is processed
  - Output the processed information through an output unit
  - Control all activities inside the machine through a control unit

The processor : Data Path and Control



- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (state elements)

Five Execution Steps

Step name	Action for R-type instructions	Action for Memory-reference Instructions	Action for branches	Action for jumps

Instruction fetch	IR = MEM[PC]
	PC = PC + 4			

Instruction decode/ register fetch	A = Reg[IR[25-21]]
	B = Reg[IR[20-16]]
	ALUOut = PC + (sign extend (IR[15-0])<<2)			

Execution, address computation, branch/jump completion	ALUOut = A op B	ALUOut = A+sign extend(IR[15-0])	IF(A==B) Then PC=ALUOut	PC=PC[31-28]|| (IR[25-0]<<2)
Memory access or R-type completion	Reg[IR[15-11]] = ALUOut	Load:MDR =Mem[ALUOut] or Store:Mem[ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

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
- Whose contents will be overwritten?

Connection Between the Processor and the Memory

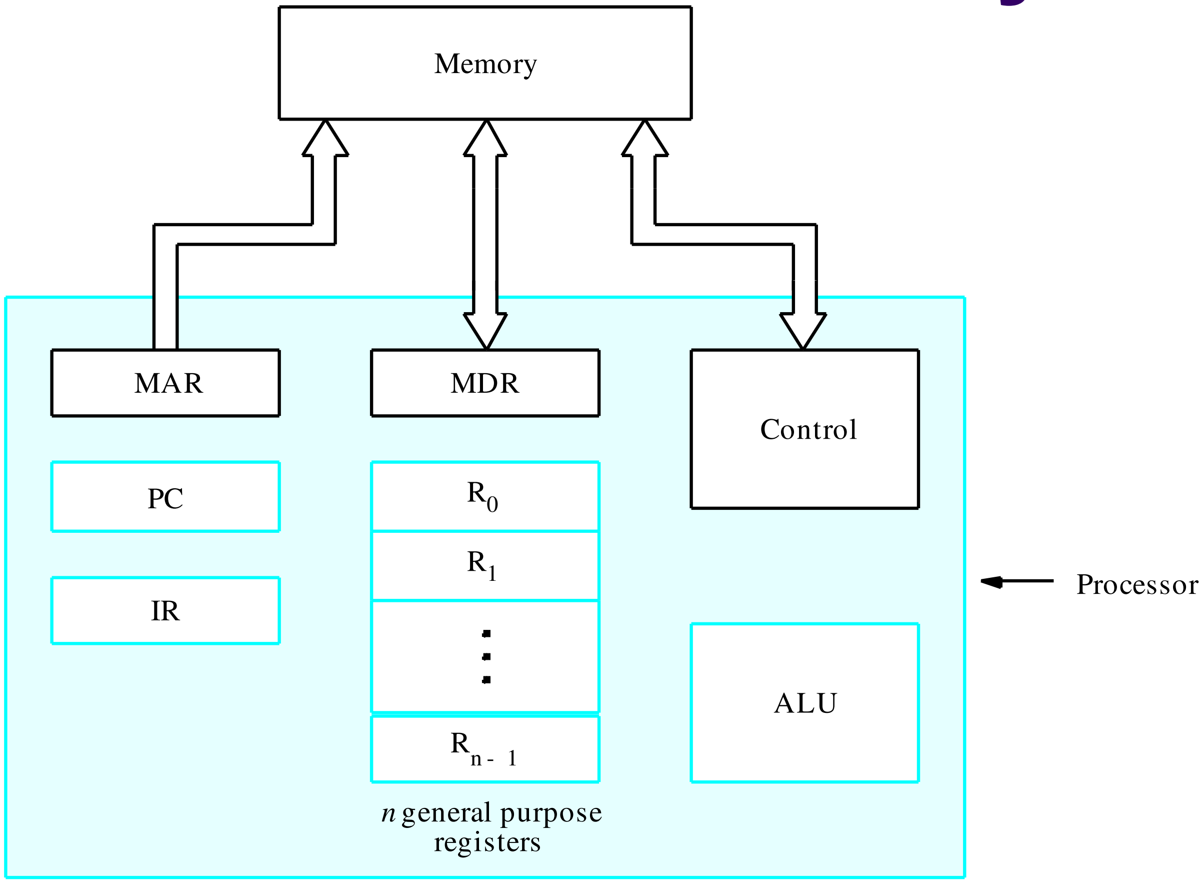


Figure 1.2. Connections between the processor and the memory.

Registers

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

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

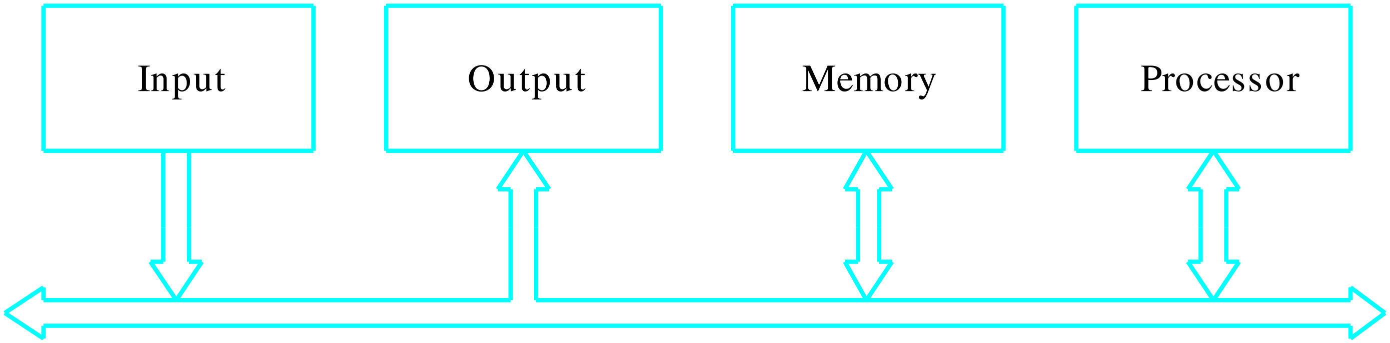


Figure 1.3. Single-bus structure.

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.

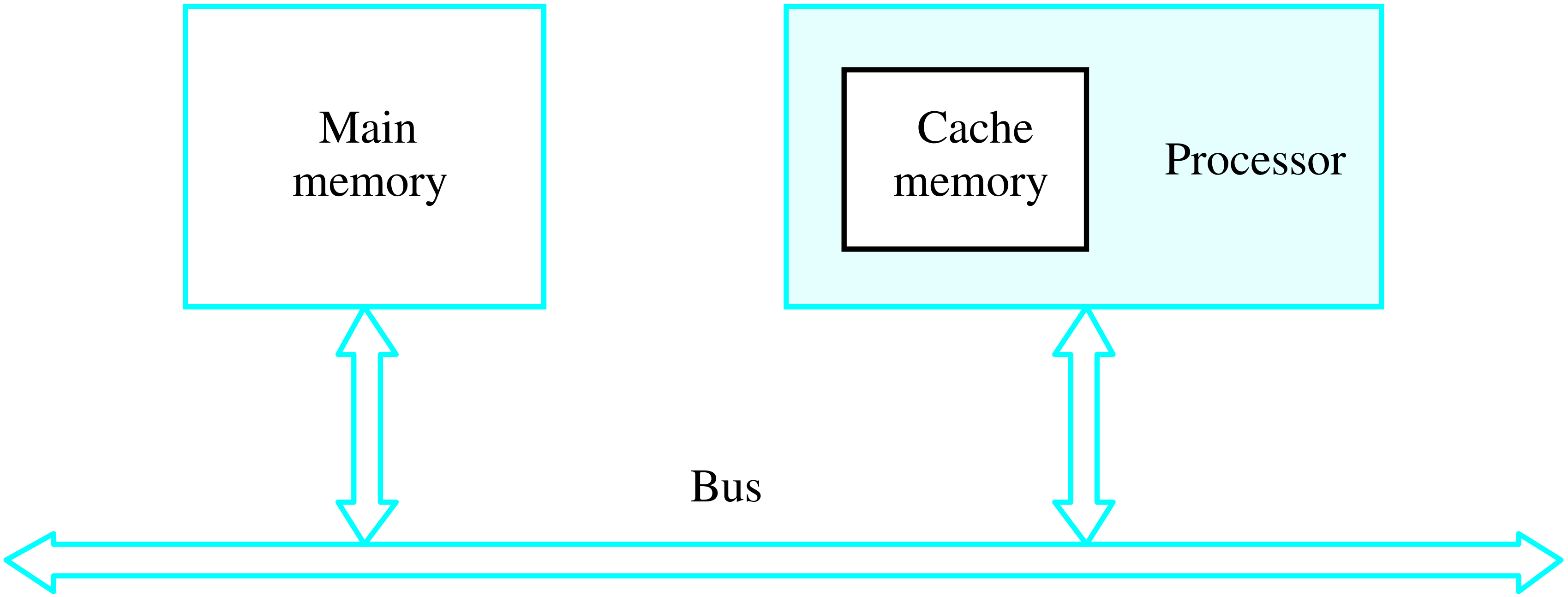


Figure 1.5. The processor cache.

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

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

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

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.

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.

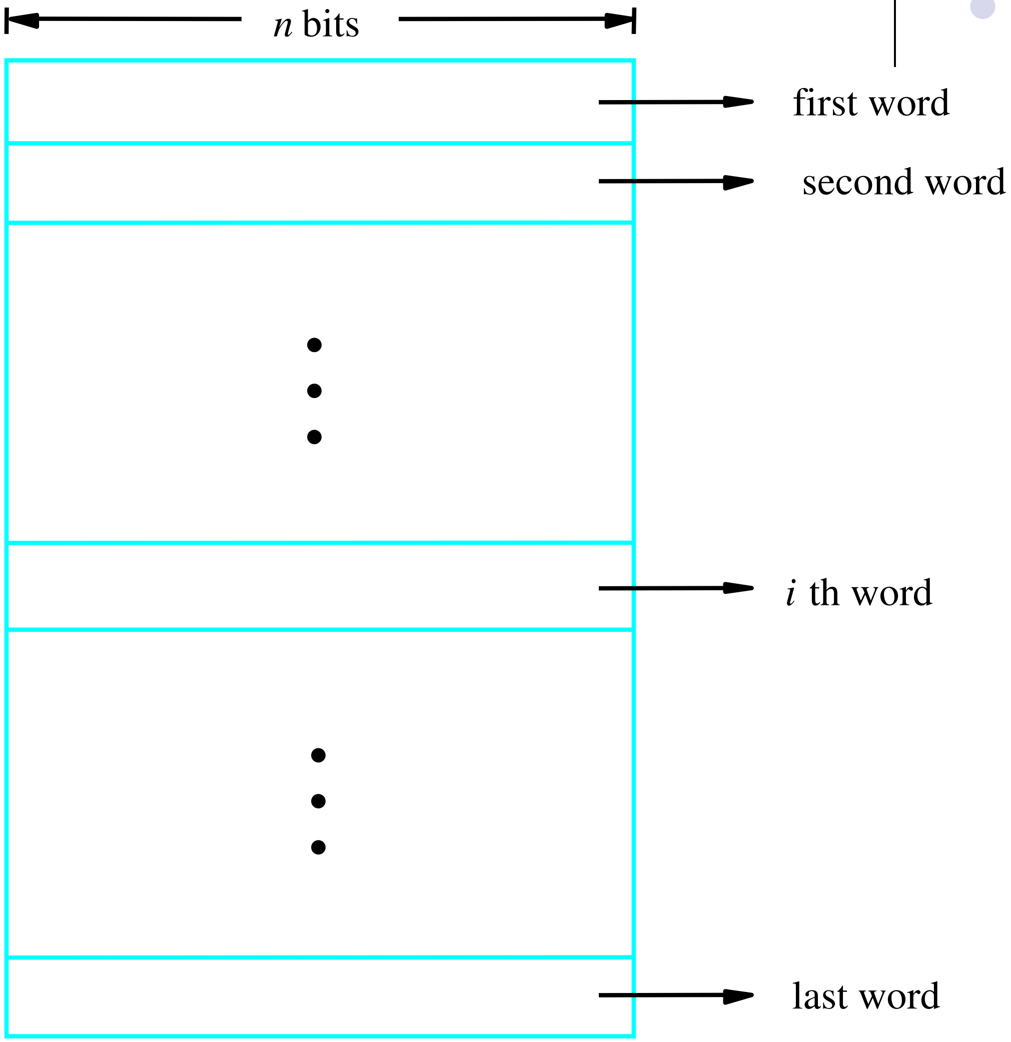
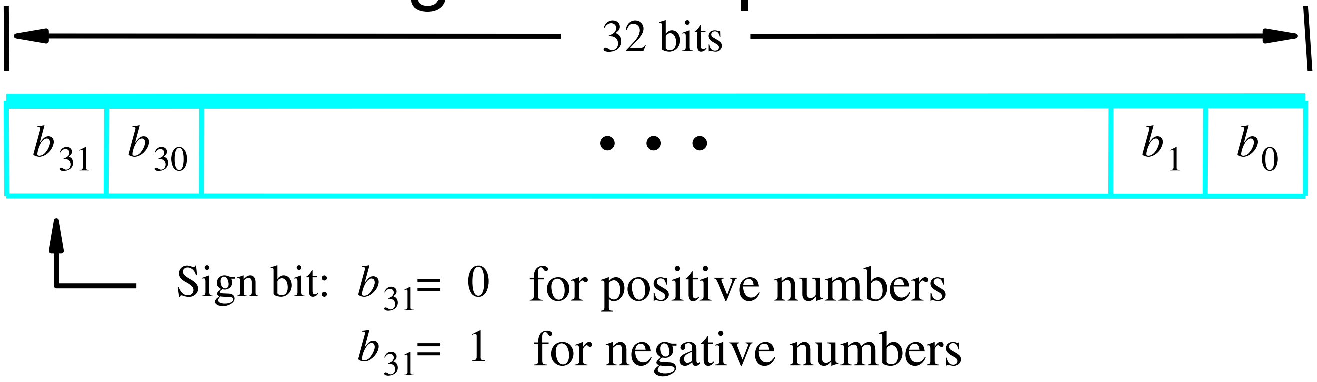


Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example



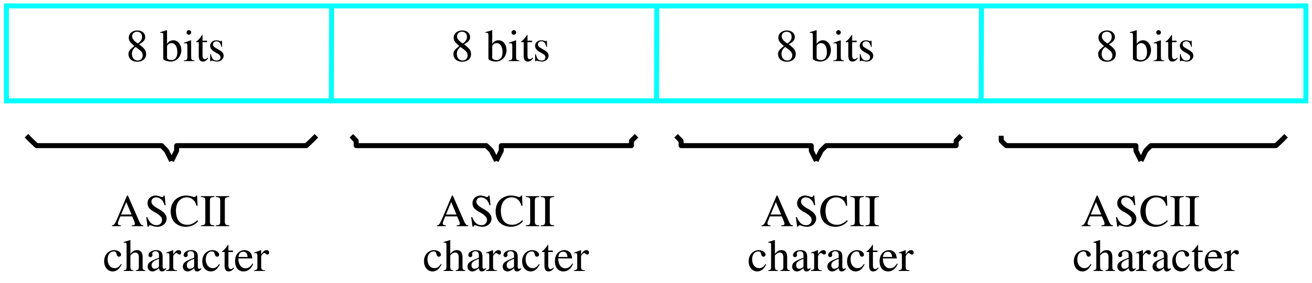
32 bits

b_{31} b_{30} • • • b_1 b_0

Sign bit: $b_{31} = 0$ for positive numbers

$b_{31} = 1$ for negative numbers

(a) A signed integer



8 bits 8 bits 8 bits 8 bits

ASCII character ASCII character ASCII character ASCII character

(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}

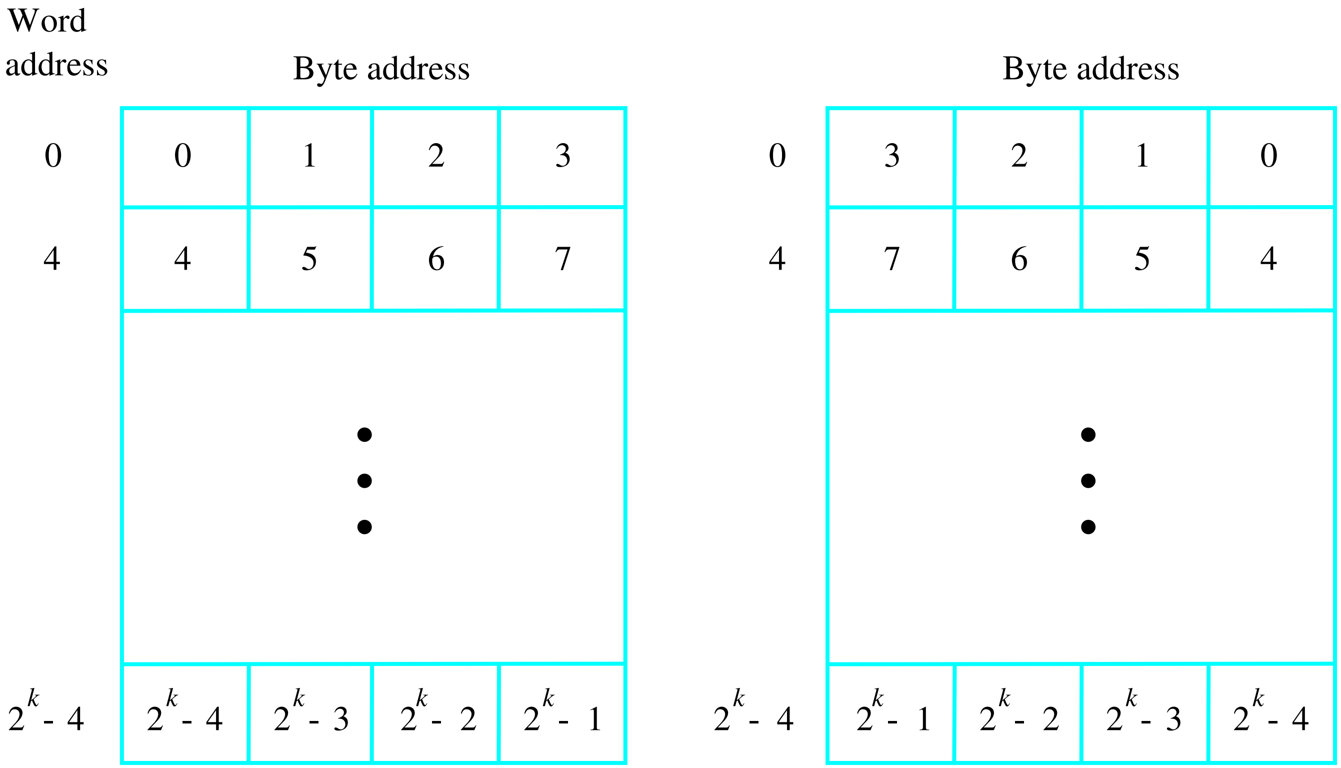
Memory Location, Addresses, and Operation

- 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



Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	⋮			
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

Word address	Byte address			
0	3	2	1	0
4	7	6	5	4
	⋮			
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(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

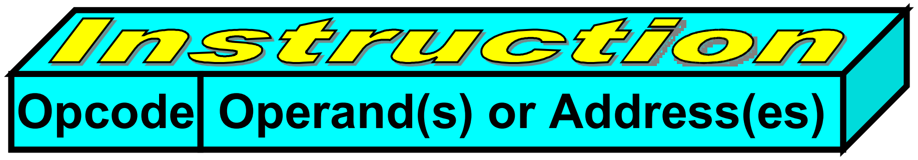
- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

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
  - ADD R1, A, B ; R1 ← M[A] + M[B]
  - ADD R2, C, D ; R2 ← M[C] + M[D]
  - MUL X, R1, R2 ; M[X] ← R1 ∗ R2

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Two-Address
  - MOV R1, A ; R1 ← M[A]
  - ADD R1, B ; R1 ← R1 + M[B]
  - MOV R2, C ; R2 ← M[C]
  - ADD R2, D ; R2 ← R2 + M[D]
  - MUL R1, R2 ; R1 ← R1 ∗ R2
  - MOV X, R1 ; M[X] ← R1

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- One-Address
  - LOAD A ; AC ← M[A]
  - ADD B ; AC ← AC + M[B]
  - STORE T ; M[T] ← AC
  - LOAD C ; AC ← M[C]
  - ADD D ; AC ← AC + M[D]
  - MUL T ; AC ← AC ∗ M[T]
  - STORE X ; M[X] ← AC

Instruction Formats

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

- Zero-Address

```
PUSH   A                ; TOS ← A
PUSH   B                ; TOS ← B
ADD                     ; TOS ← (A + B)
PUSH   C                ; TOS ← C
PUSH   D                ; TOS ← D
ADD                     ; TOS ← (C + D)
MUL                     ; TOS ←
(C+D)*(A+B)
POP    X                ; M[X] ← TOS
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- RISC

```
LOAD   R1, A              ; R1 ← M[A]
LOAD   R2, B              ; R2 ← M[B]
LOAD   R3, C              ; R3 ← M[C]
LOAD   R4, D              ; R4 ← M[D]
ADD    R1, R1, R2         ; R1 ← R1 + R2
ADD    R3, R3, R4         ; R3 ← R3 + R4
MUL    R1, R1, R3         ; R1 ← R1 ∗ R3
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

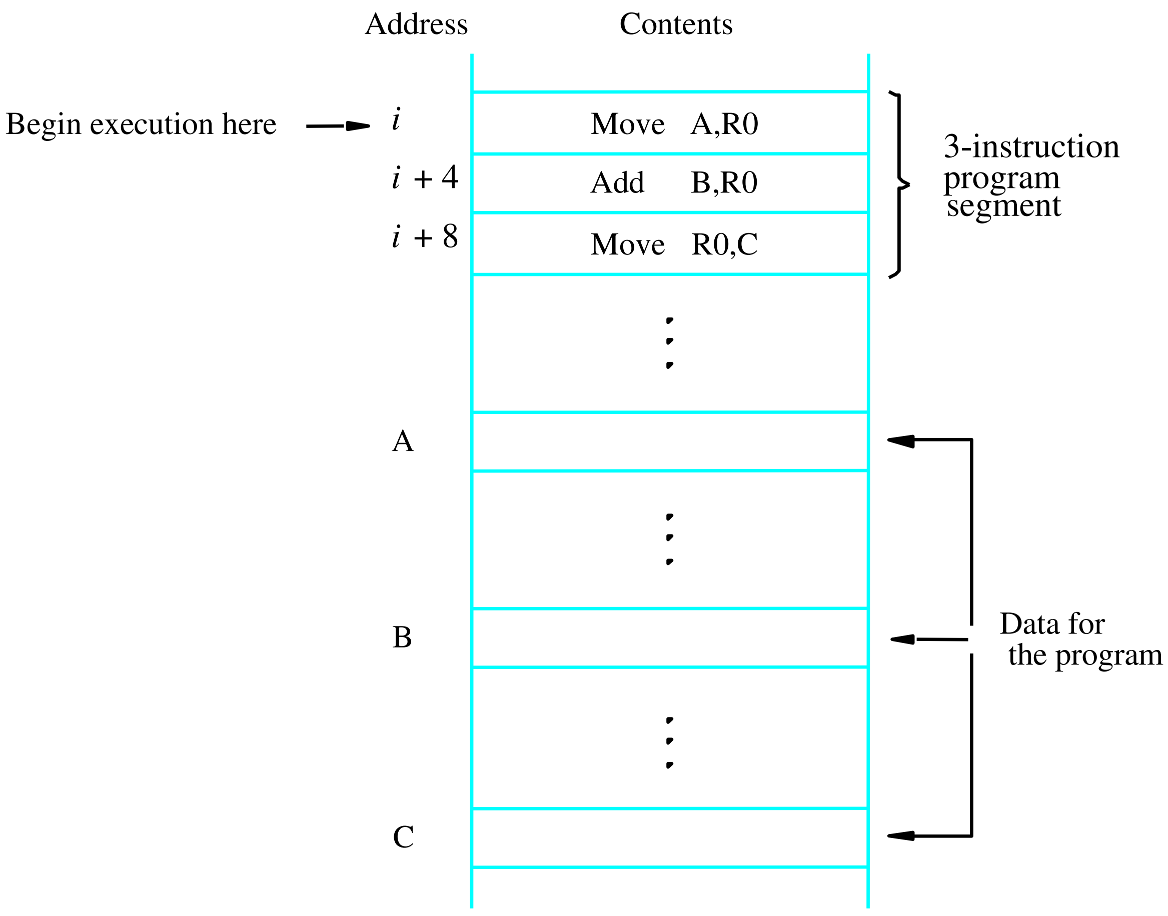


Figure 2.8. A program for C ← [A] + [B].

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

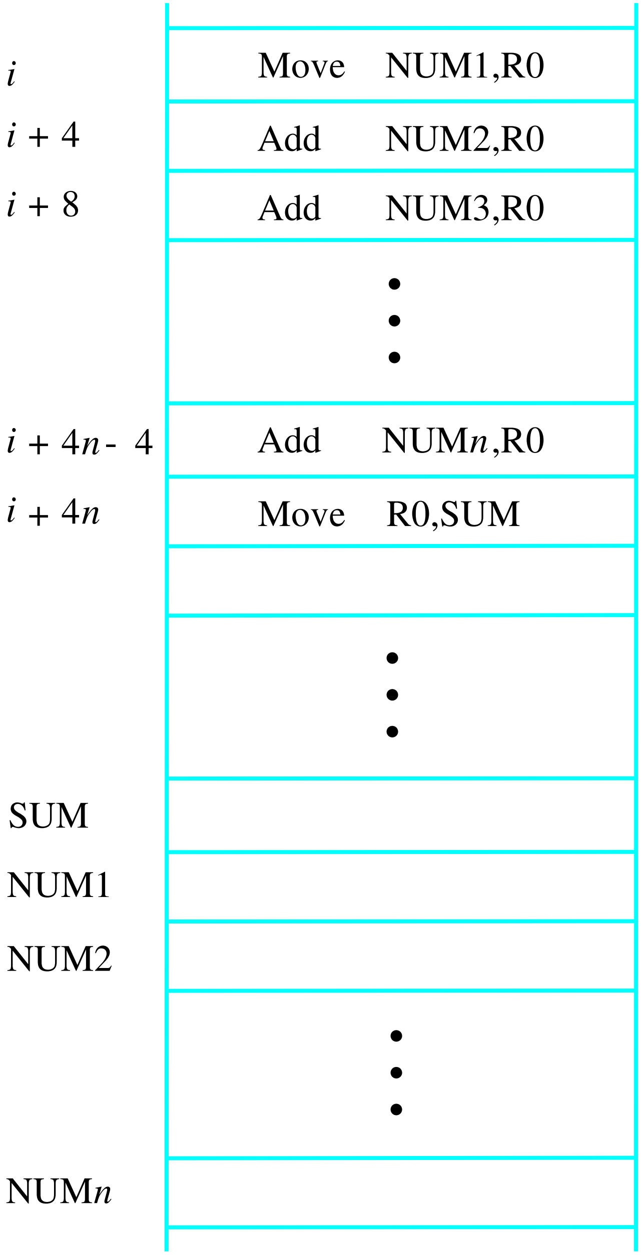
Two-phase procedure

-Instruction fetch

-Instruction execute

Page 43

Branching



i	Move NUM1,R0
$i + 4$	Add NUM2,R0
$i + 8$	Add NUM3,R0
	⋮
$i + 4n - 4$	Add NUMn,R0
$i + 4n$	Move R0,SUM
	⋮
SUM	
NUM1	
NUM2	
	⋮
NUMn	

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

Branching

Branch target

Conditional branch

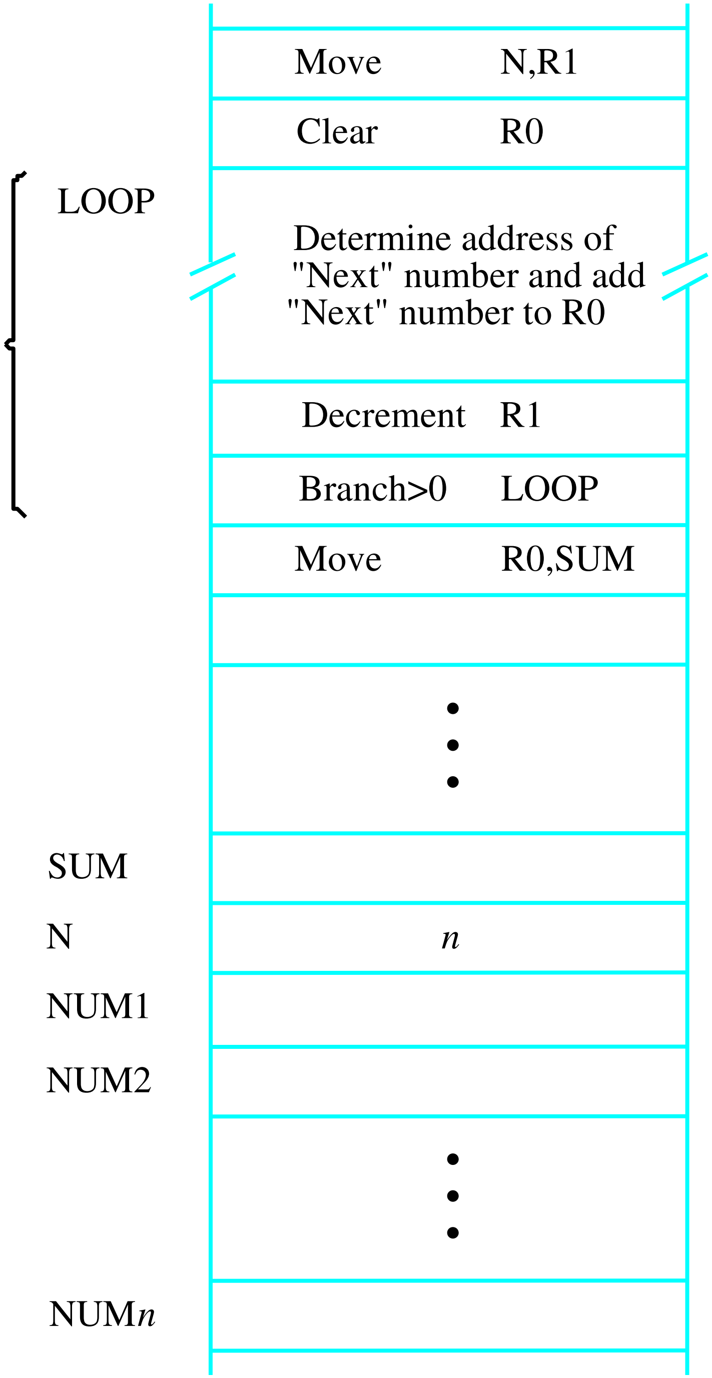


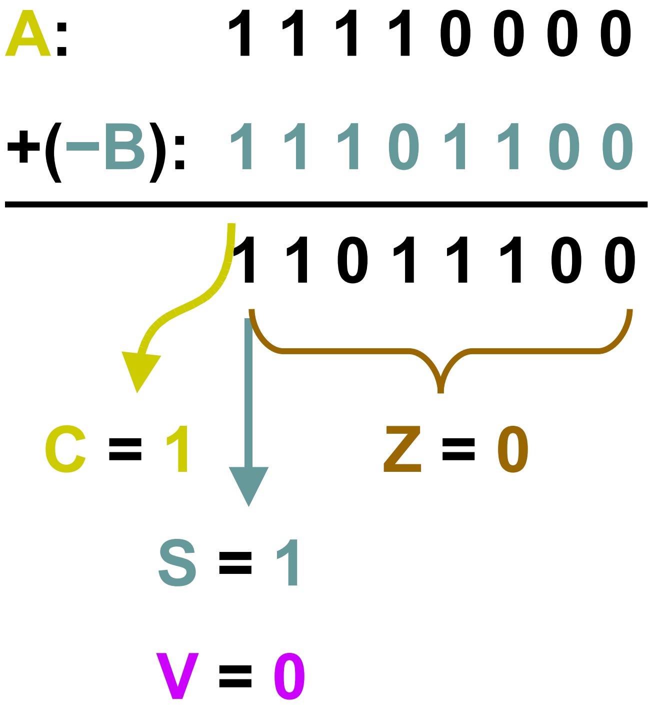
Figure 2.10. Using a loop to add *n* numbers.

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



A: 1 1 1 1 0 0 0 0

+(−B): 1 1 1 0 1 1 0 0

1 1 0 1 1 1 0 0

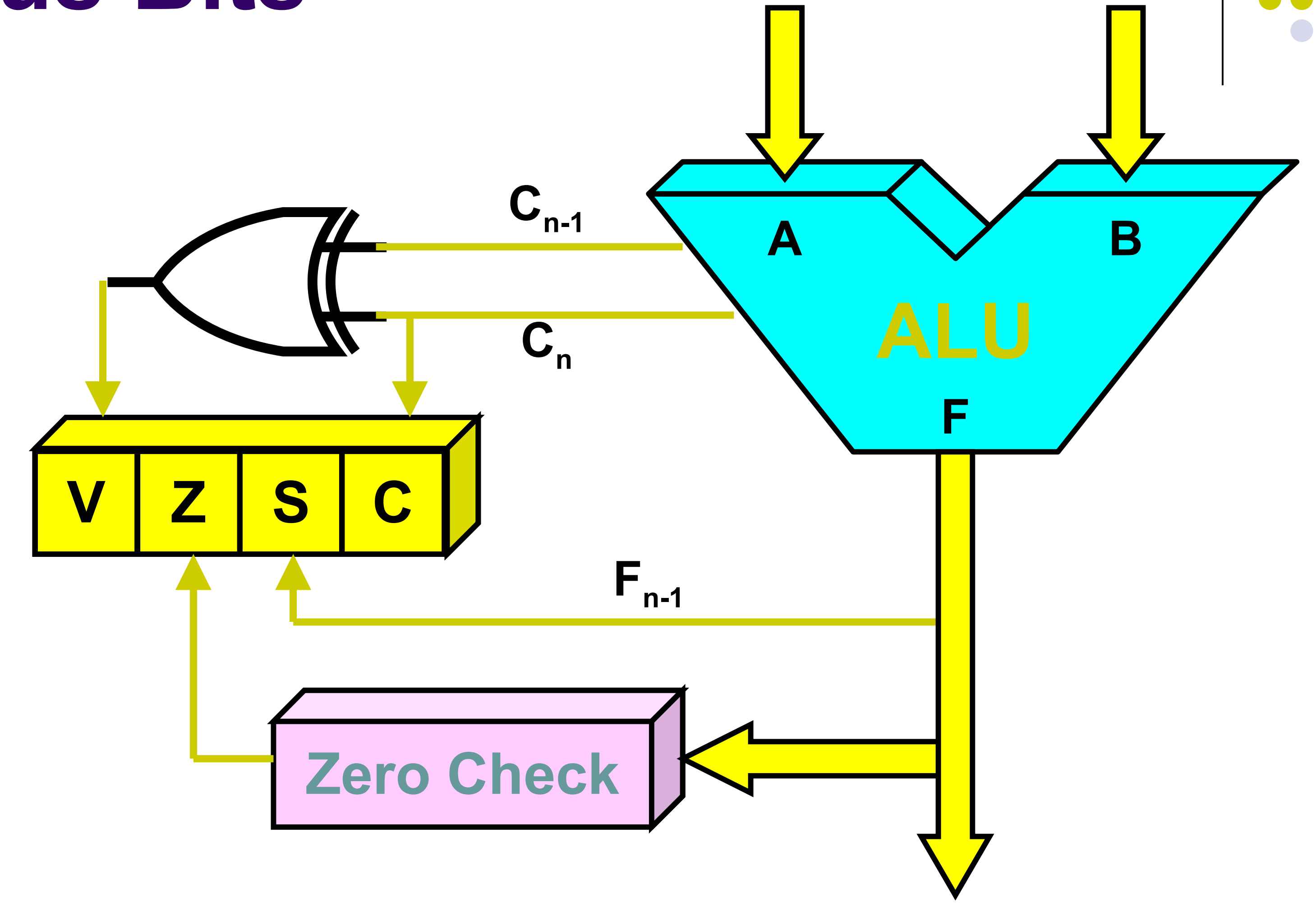
C = 1

Z = 0

S = 1

V = 0

Status Bits

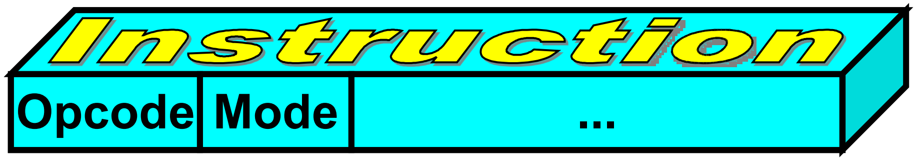


Addressing Modes

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


Instruction

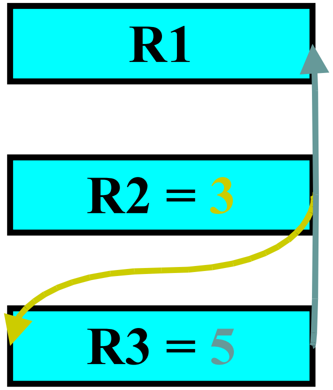
Opcode | Mode | ...

- 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

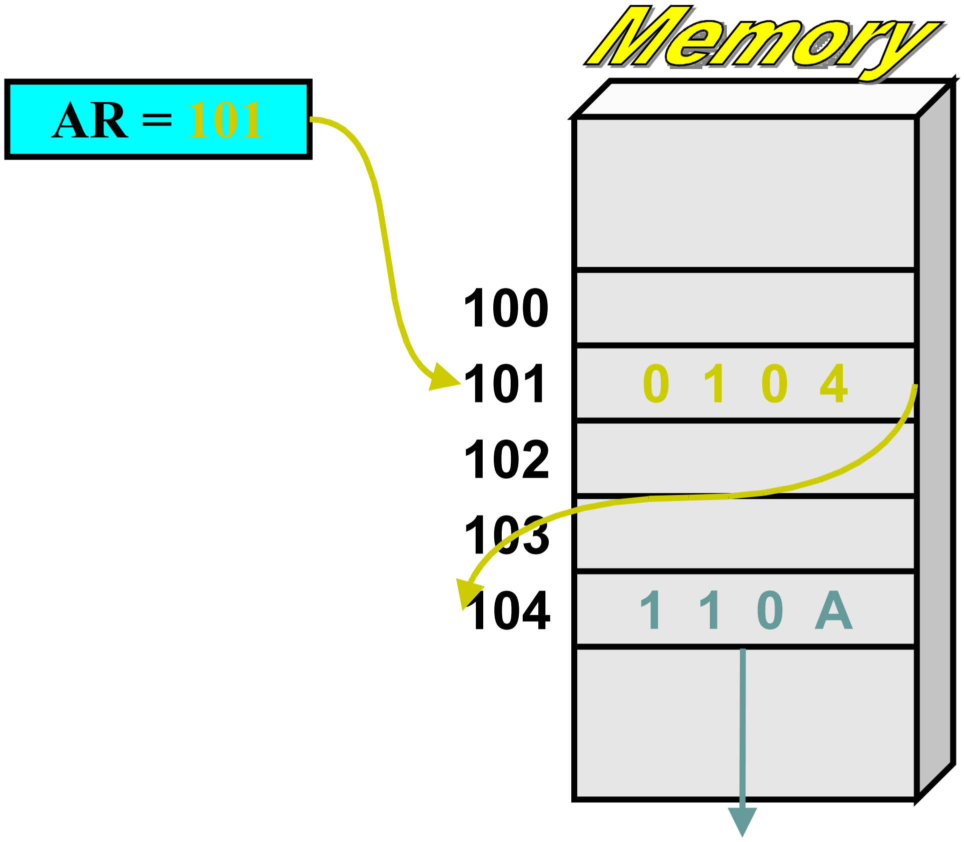
    MOV R1, (R2)



- Autoincrement / Autodecrement
  - Access & update in 1 instr.
- Direct Address
  - Use the given address to access a memory location

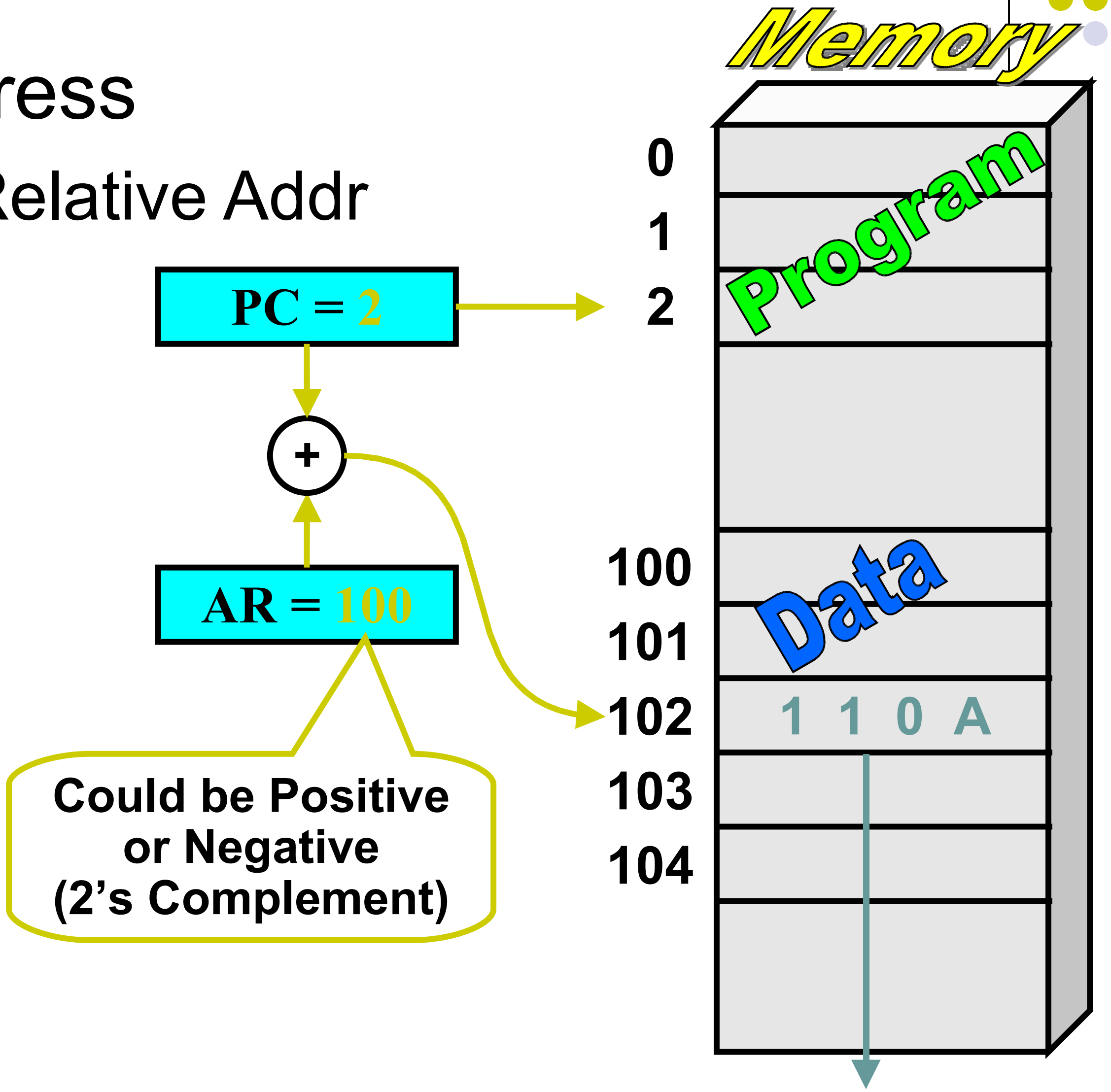
Addressing Modes

- 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



PC = 2

+

AR = 100

Could be Positive or Negative (2's Complement)

Memory

Program

Data

0

1

2

100

101

102

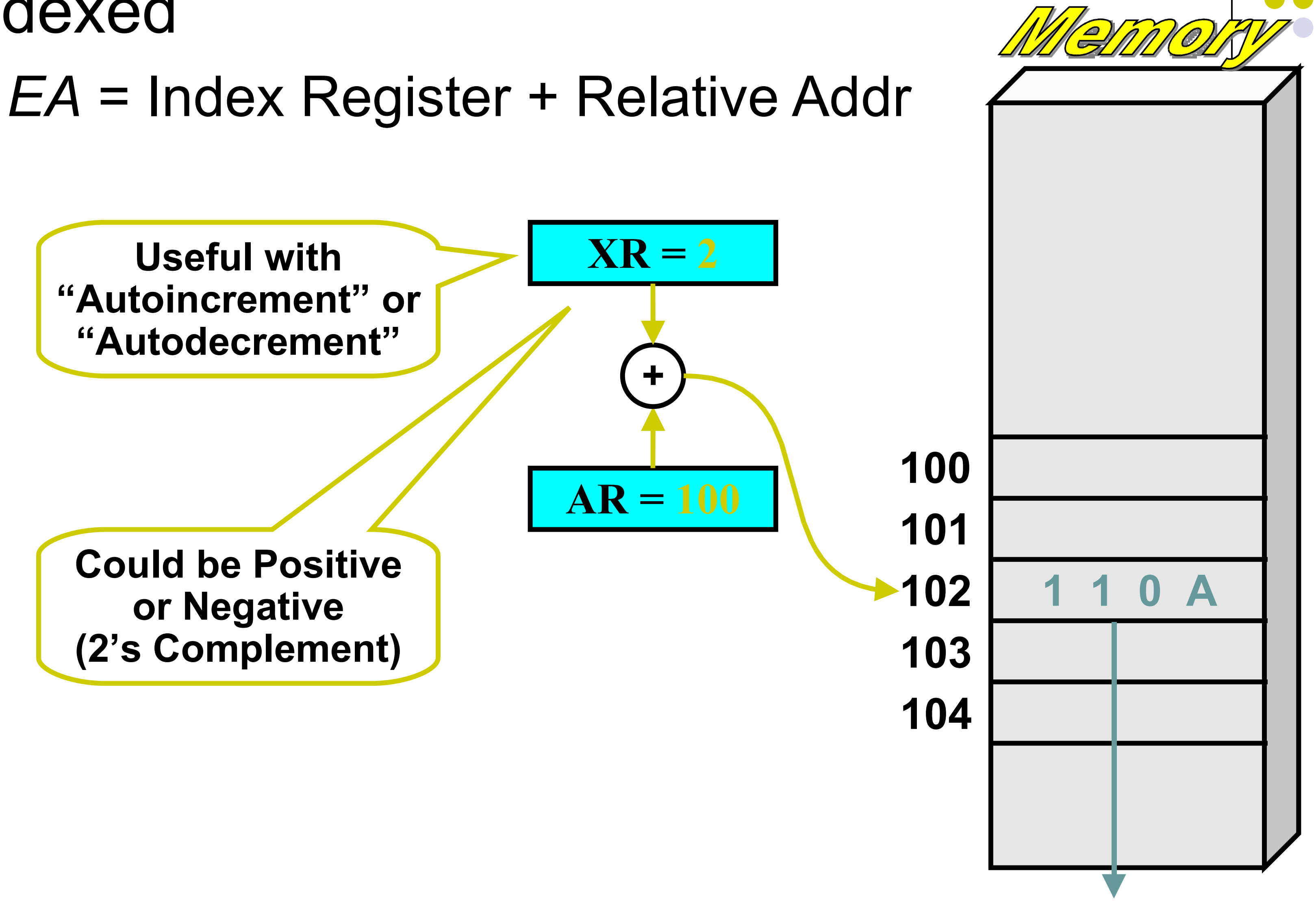
103

104

1 1 0 A

Addressing Modes

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

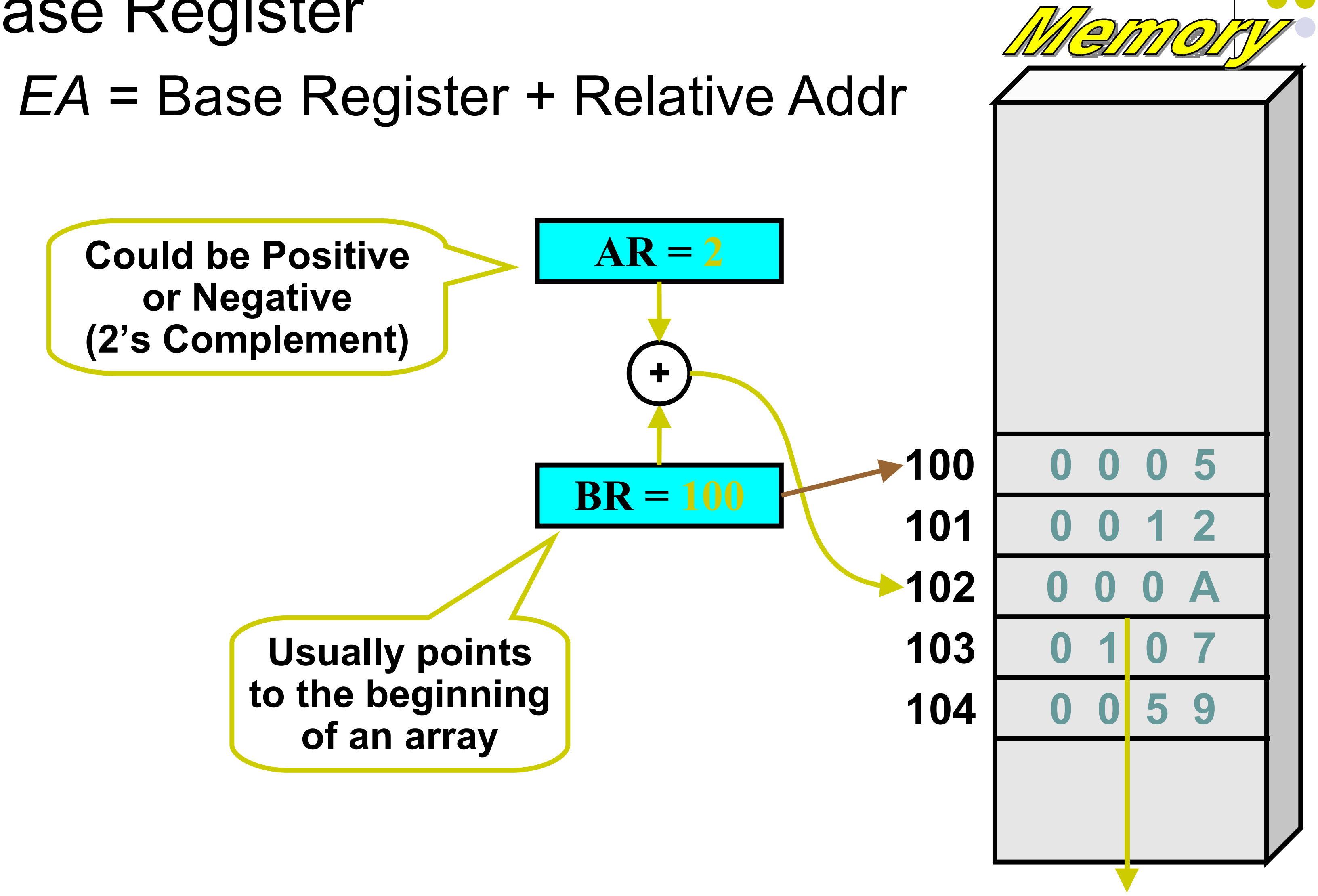


Useful with “Autoincrement” or “Autodecrement”

Could be Positive or Negative (2’s Complement)

Addressing Modes

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


Memory
Could be Positive or Negative (2's Complement)
AR = 2
+
BR = 100
Usually points to the beginning of an array
100
0 0 0 5
101
0 0 1 2
102
0 0 0 A
103
0 1 0 7
104
0 0 5 9

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	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
Base with index	(R*I*,R*J*)	EA = [R*I*] + [R*J*]
Base with 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

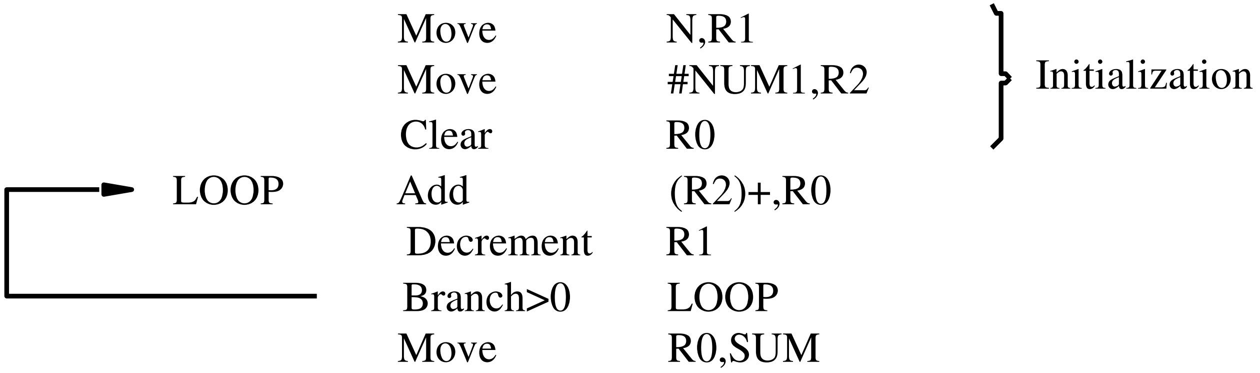
- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
- Several variations:
  (R_i, R_j): EA = $[R_i] + [R_j]$
  $X(R_i, R_j)$: EA = $X + [R_i] + [R_j]$

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	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC+ADR]$
Immediate operand	LD #NBR	$AC \leftarrow \text{NBR}$
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR+XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1]$, $R1 \leftarrow R1+1$

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

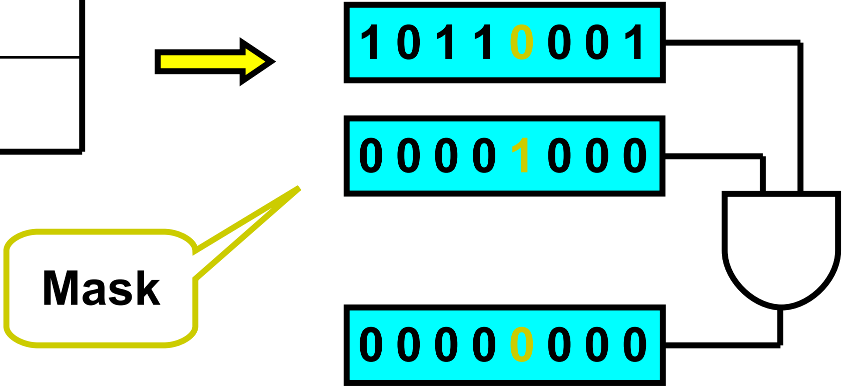
Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

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

Subtract A – B but don't store the result



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

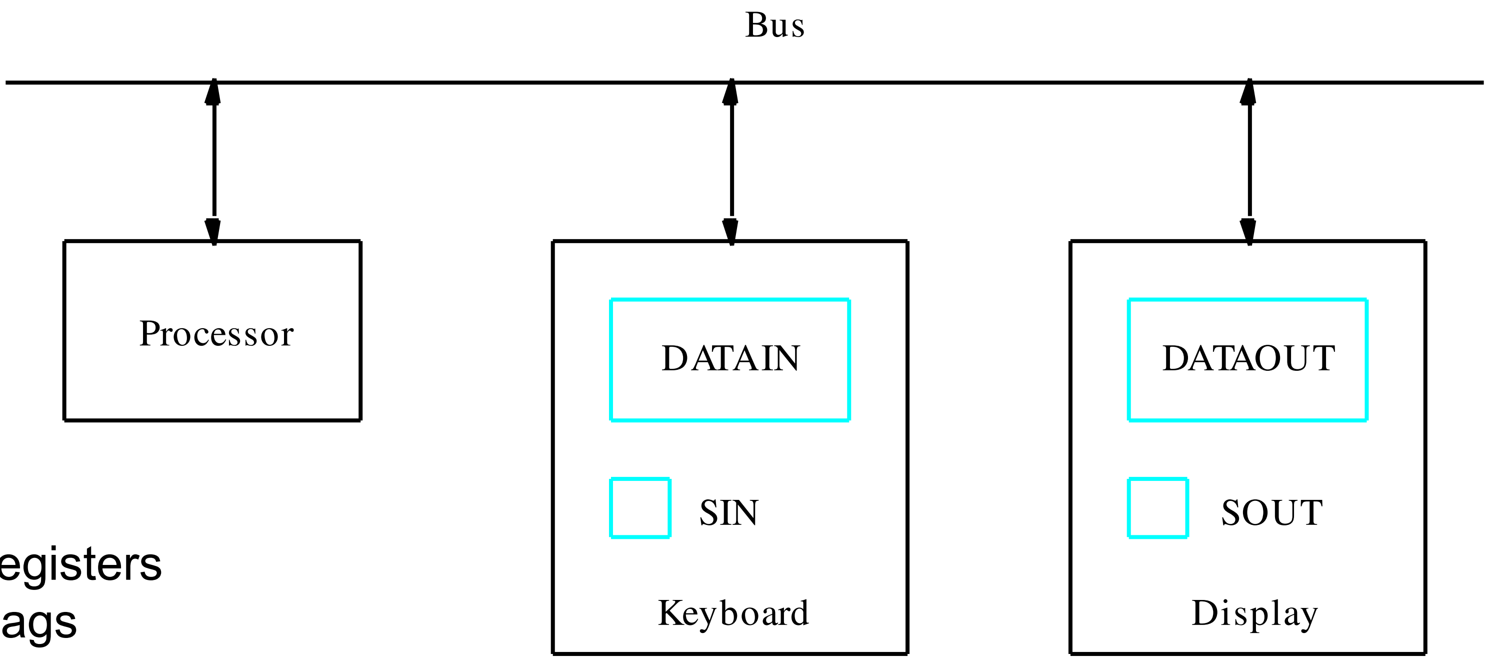
I/O

- 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

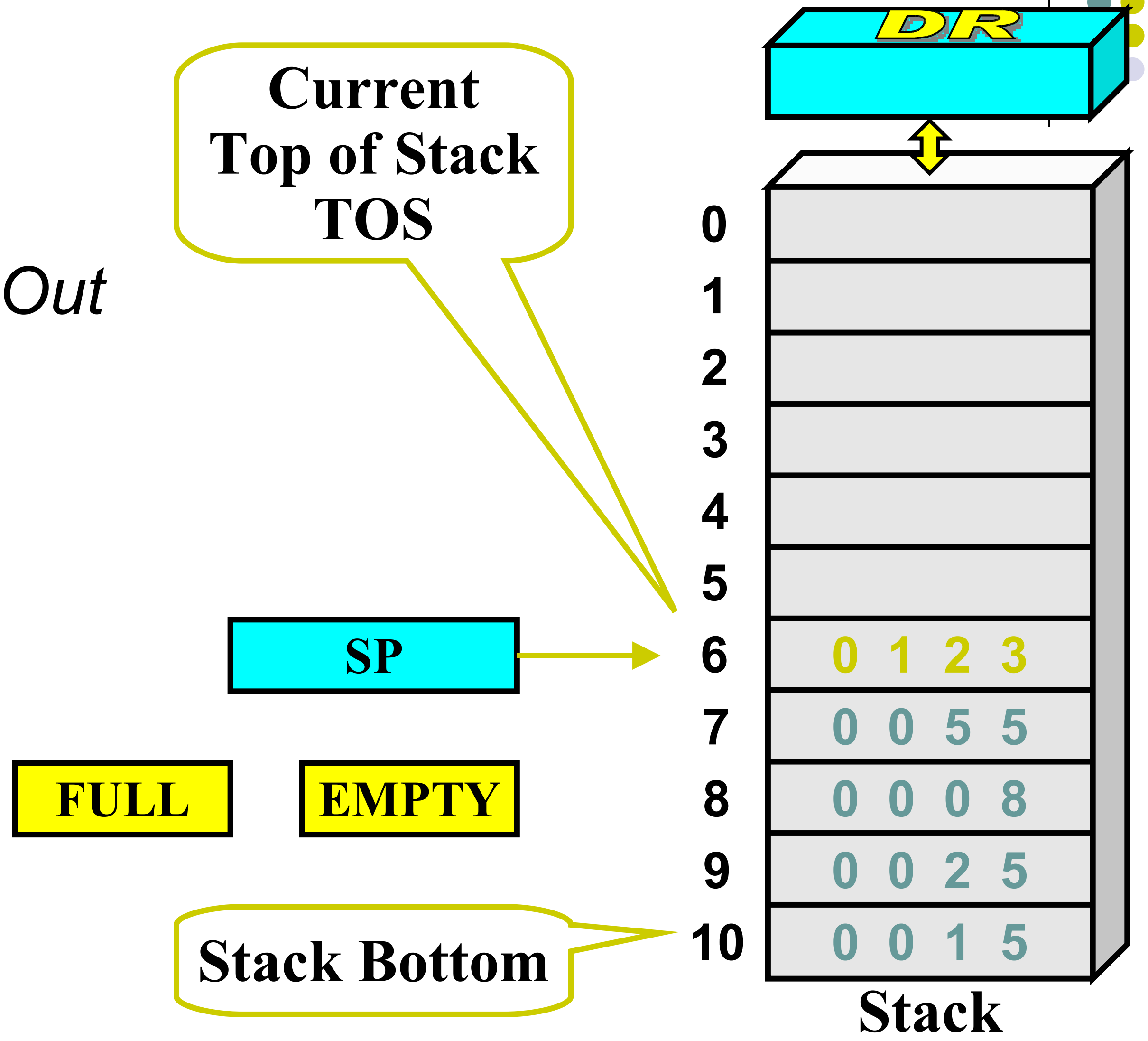
- Assumption – the initial state of SIN is 0 and the initial state of SOUT is 1.
- Any drawback of this mechanism in terms of efficiency?
  - Two wait loops→processor execution time is wasted
- Alternate solution?
  - Interrupt

Stacks

Stack Organization

- LIFO

  Last In First Out



Stack Organization

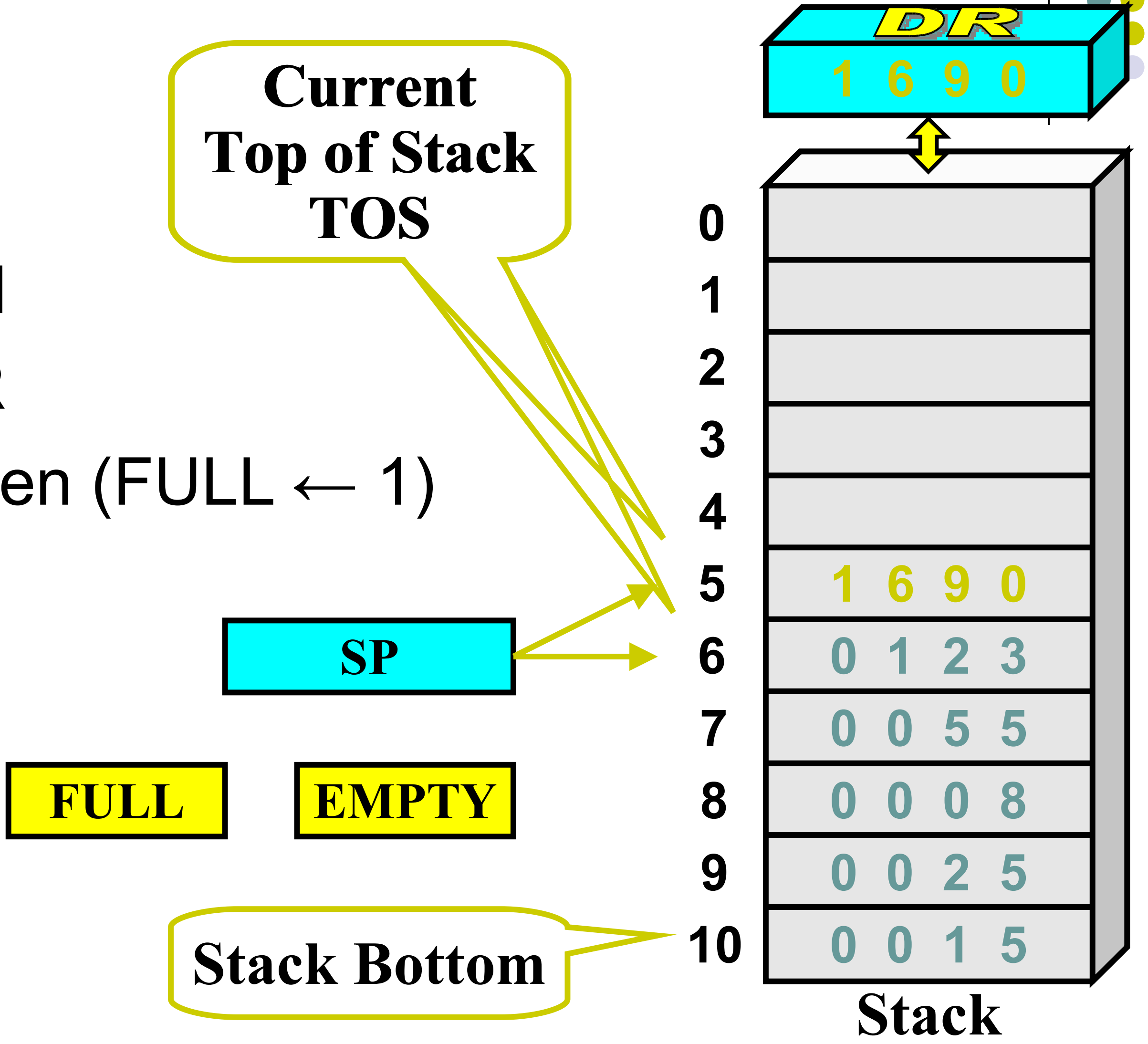
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

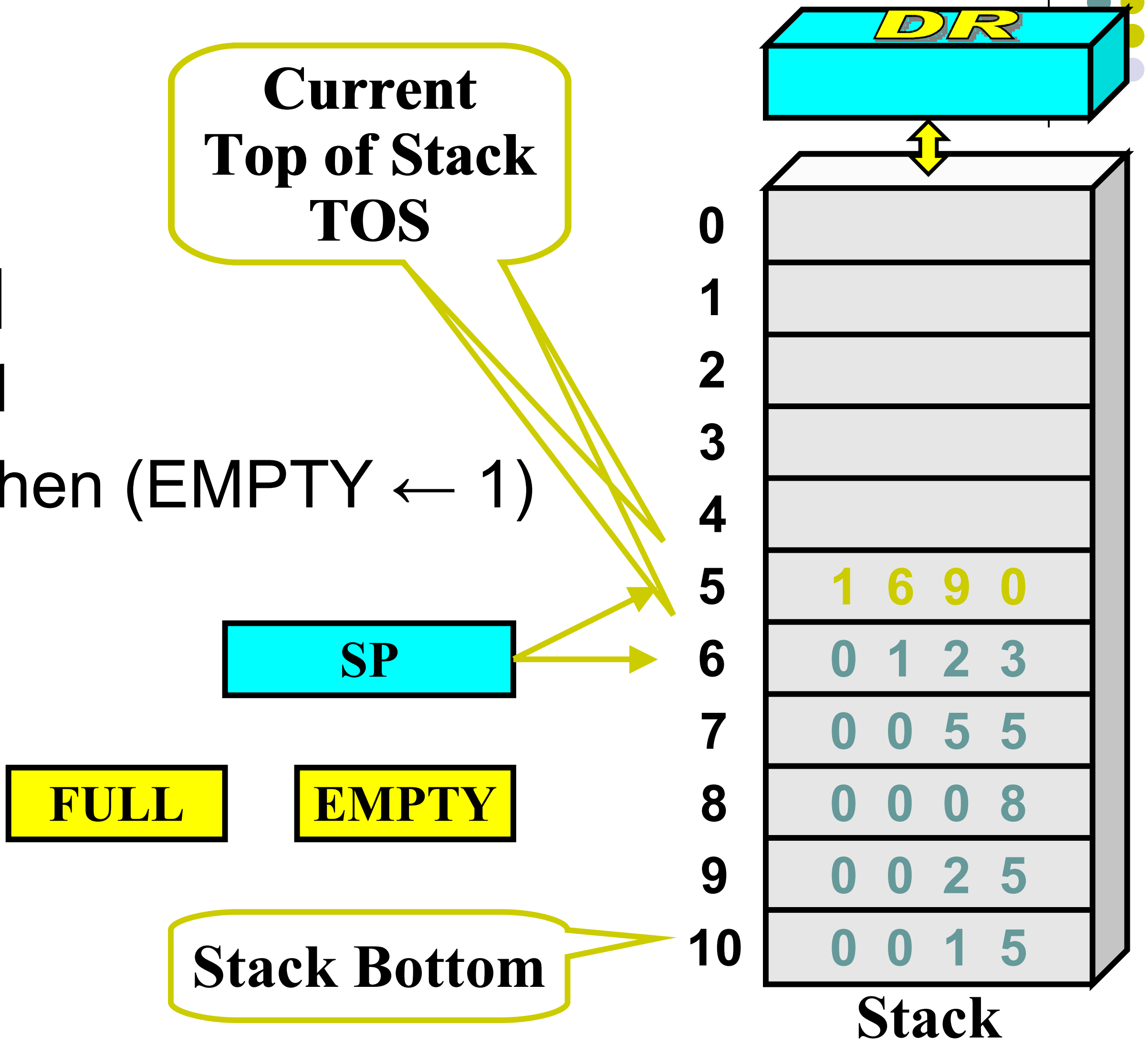
- POP

  DR ← M[SP]

  SP ← SP + 1

  If (SP = 11) then (EMPTY ← 1)

  FULL ← 0



Stack Organization

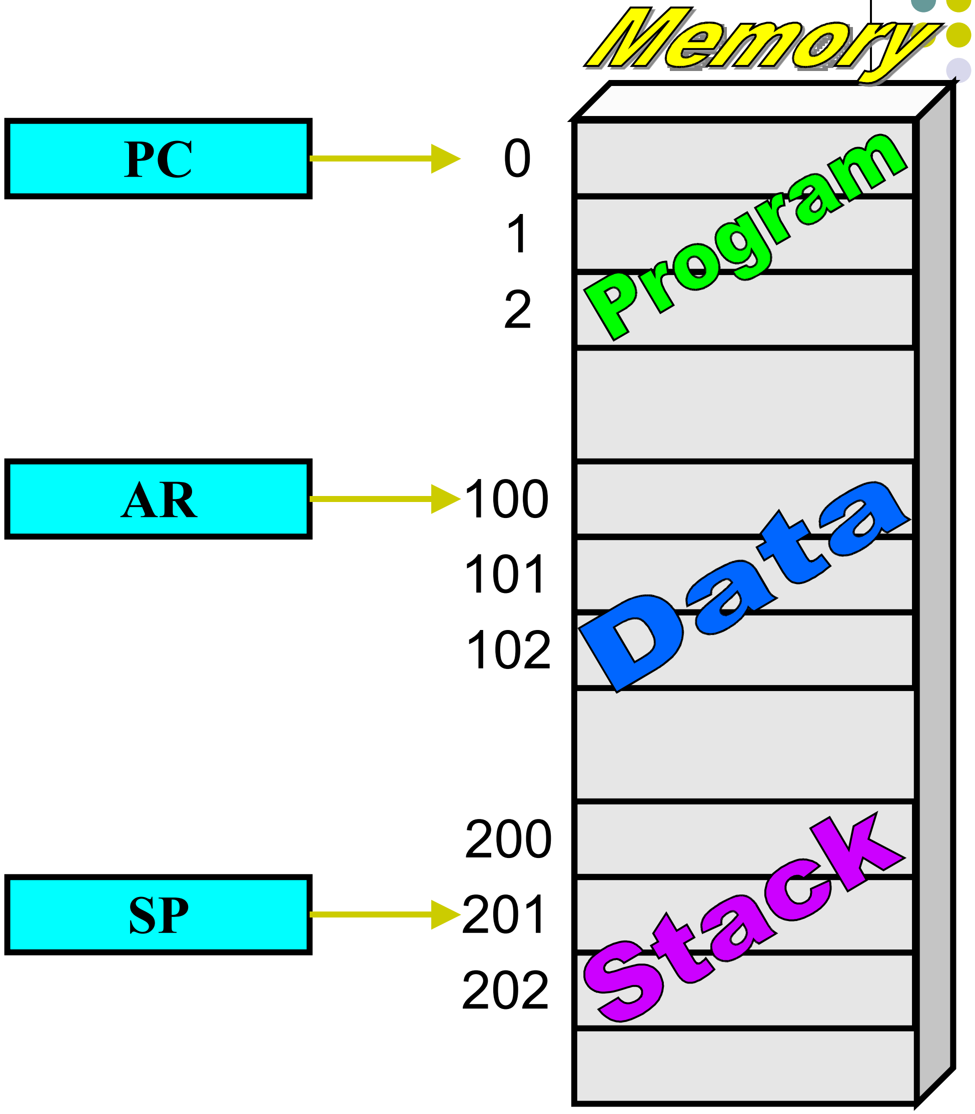
- Memory Stack
  - PUSH

    SP ← SP – 1

    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1


Memory
PC
AR
SP
0
1
2
100
101
102
200
201
202
Program
Data
Stack