



Module 3

Basic Structure of Computers

OUTLINE

- **Basic Structure of Computers:**
- Basic Operational Concepts, Bus Structures, Performance – Processor Clock, Basic Performance Equation, Clock Rate, Performance Measurement
- **Machine Instructions and Programs:** Memory Location and Addresses, Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language, Basic Input and Output Operations, Stacks and Queues, Subroutines, Additional Instructions, Encoding of Machine Instructions



INTRODUCTION

Functional Units

Functional Units

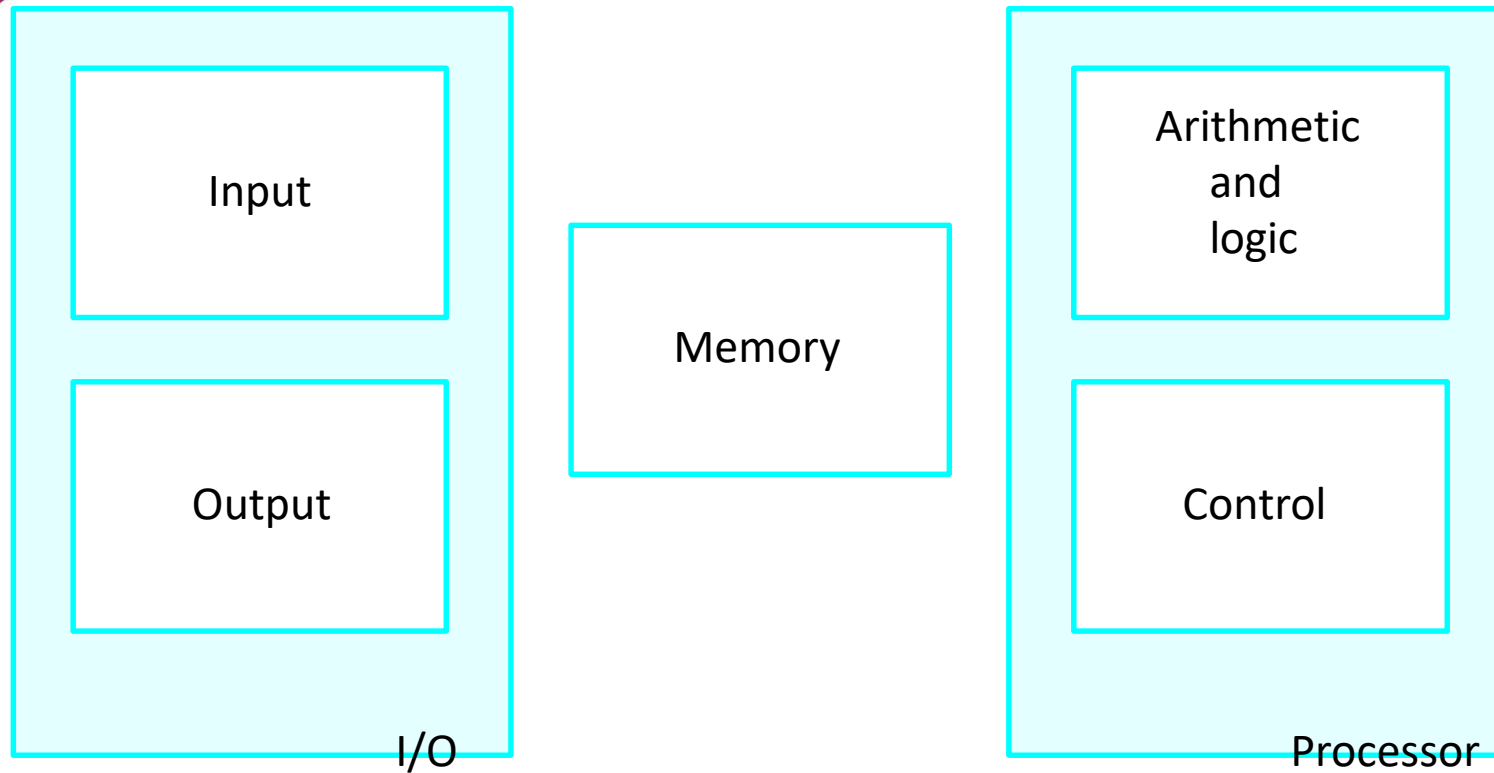


Figure 1.1. Basic functional units of a computer.



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

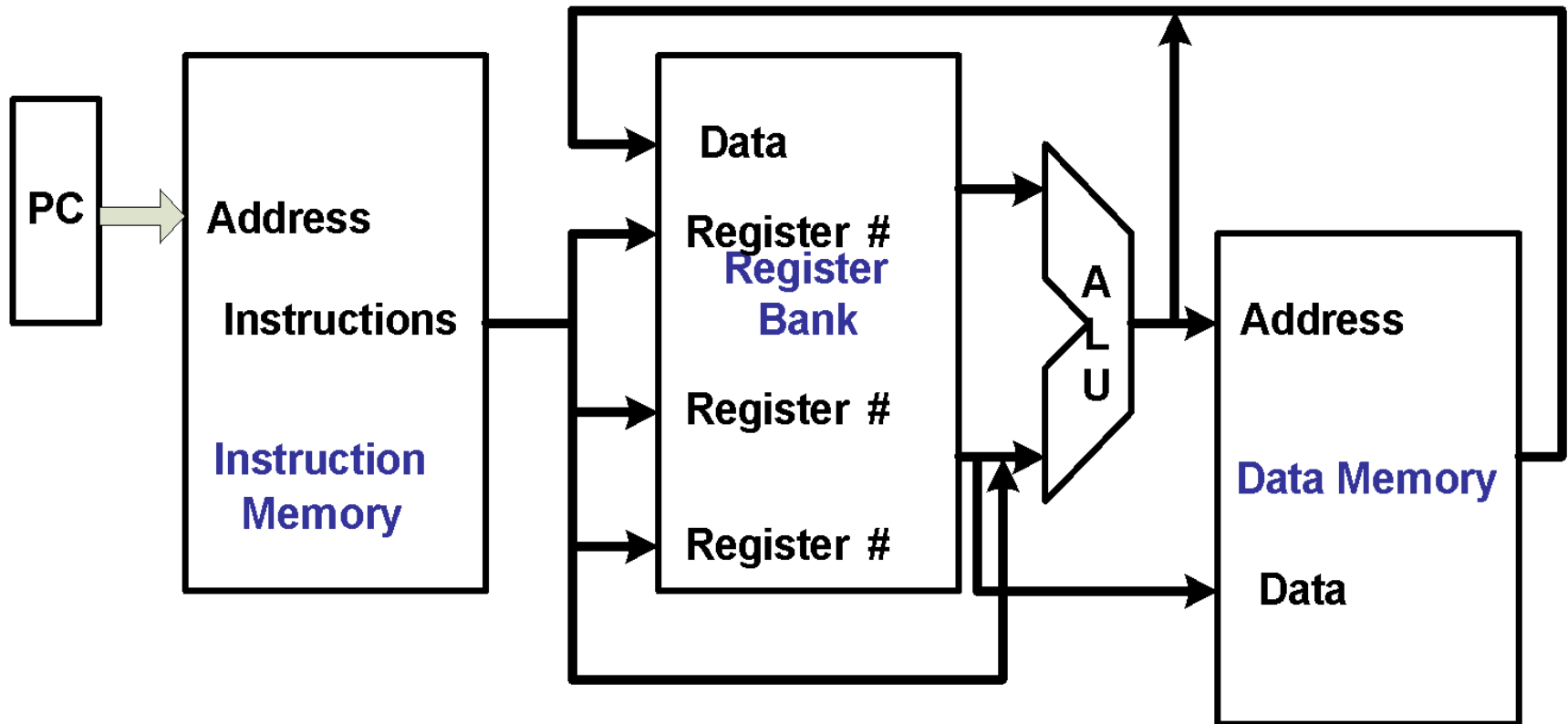
Arithmetic and Logic Unit (ALU)

- 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





Five Execution Steps

Step name	Action for R-type instructions	Action for Memory	Action for	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

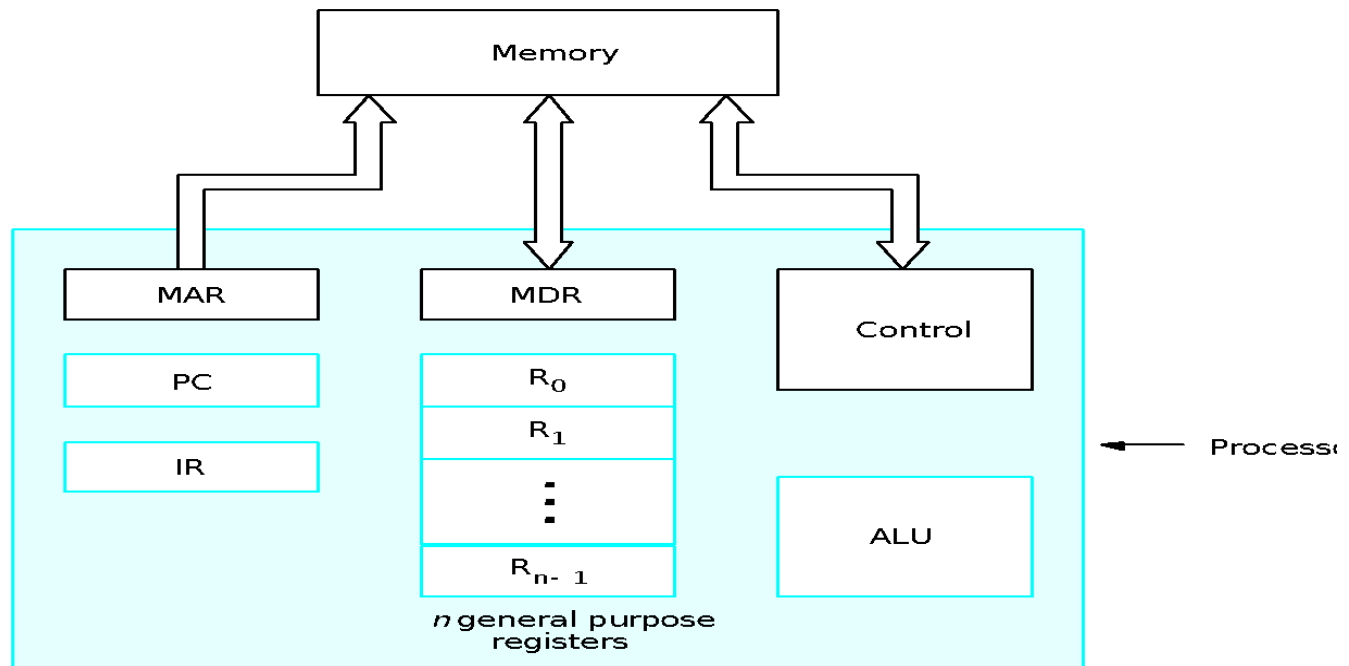


Figure 1.2. Connections between the processor and the memory.

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

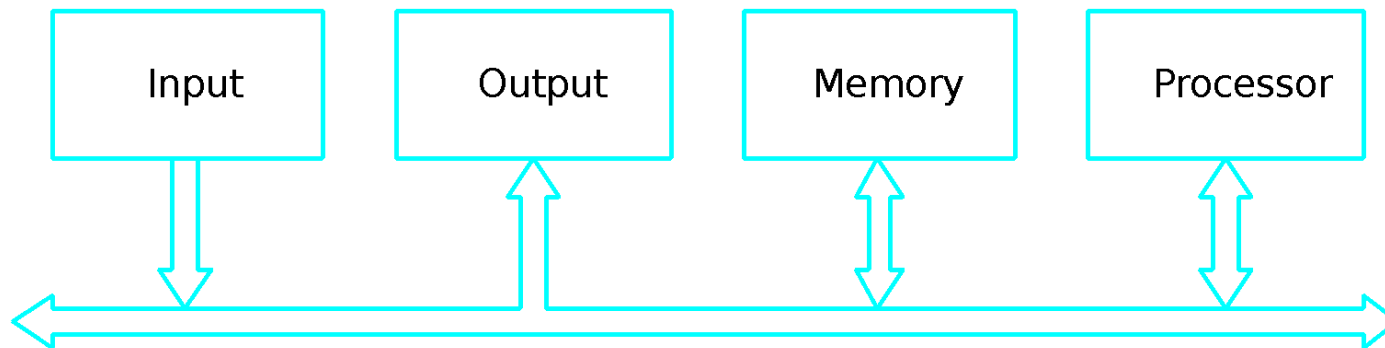


Figure 1.3. Single-bus structure.

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.



RV Institute of
Technology and
Management®

Go, change the world

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
- Elapsed time- The total time required to execute the program(measure of the performance of the entire computer system)
- Processor Time- The time during which the processor is active

Performance

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

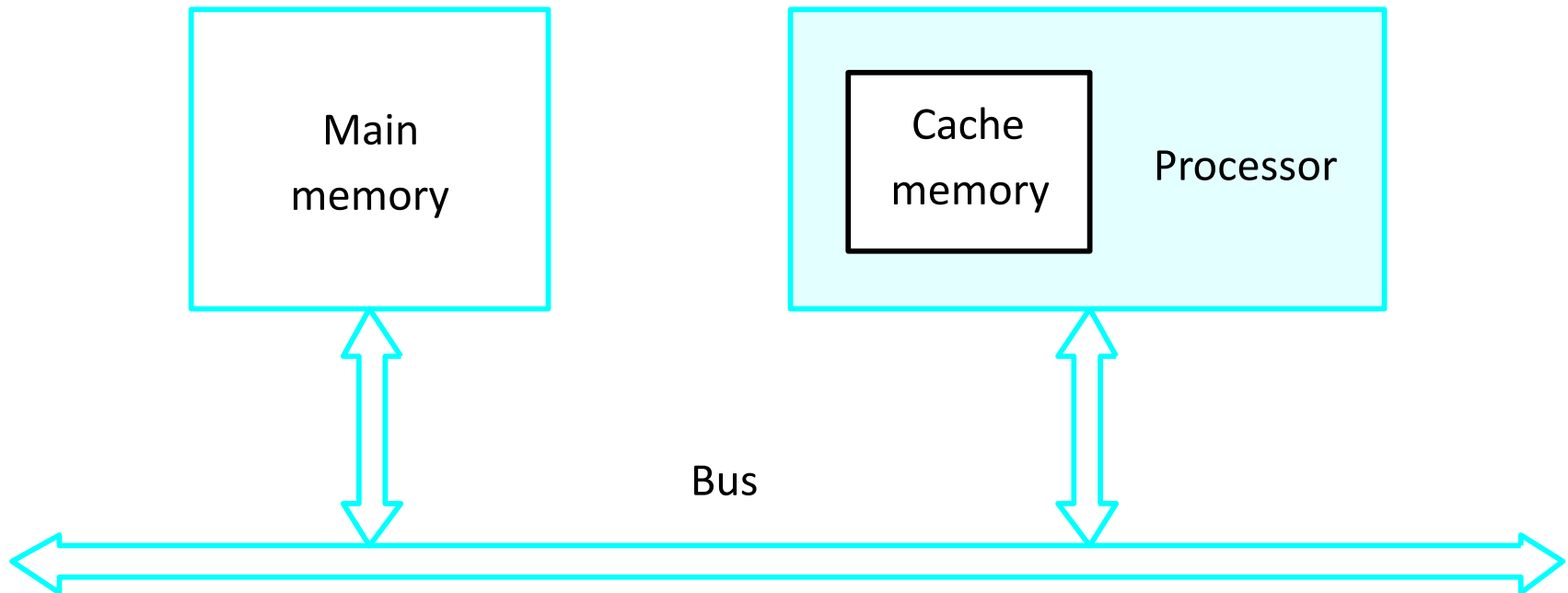


Figure 1.5. The processor cache.



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
- Processor circuits are controlled by a timing signal called **Clock**.
- It defines the regular time intervals called **Clock cycle**.
- The execution of each instruction is divided into several steps, each of which completes in one clock cycle.
- The length P of one clock cycle is an important parameter that affects the processor performance.
- Inverse of is **Clock rate**. $R = 1/P$
- Hertz – cycles per second
- Eg. 500 million cycles per second and 1250 milion per second (How much time is needed to execute one basic step)

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.

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

Memory Location, Addresses, and Operation

Go, change the world

- 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.

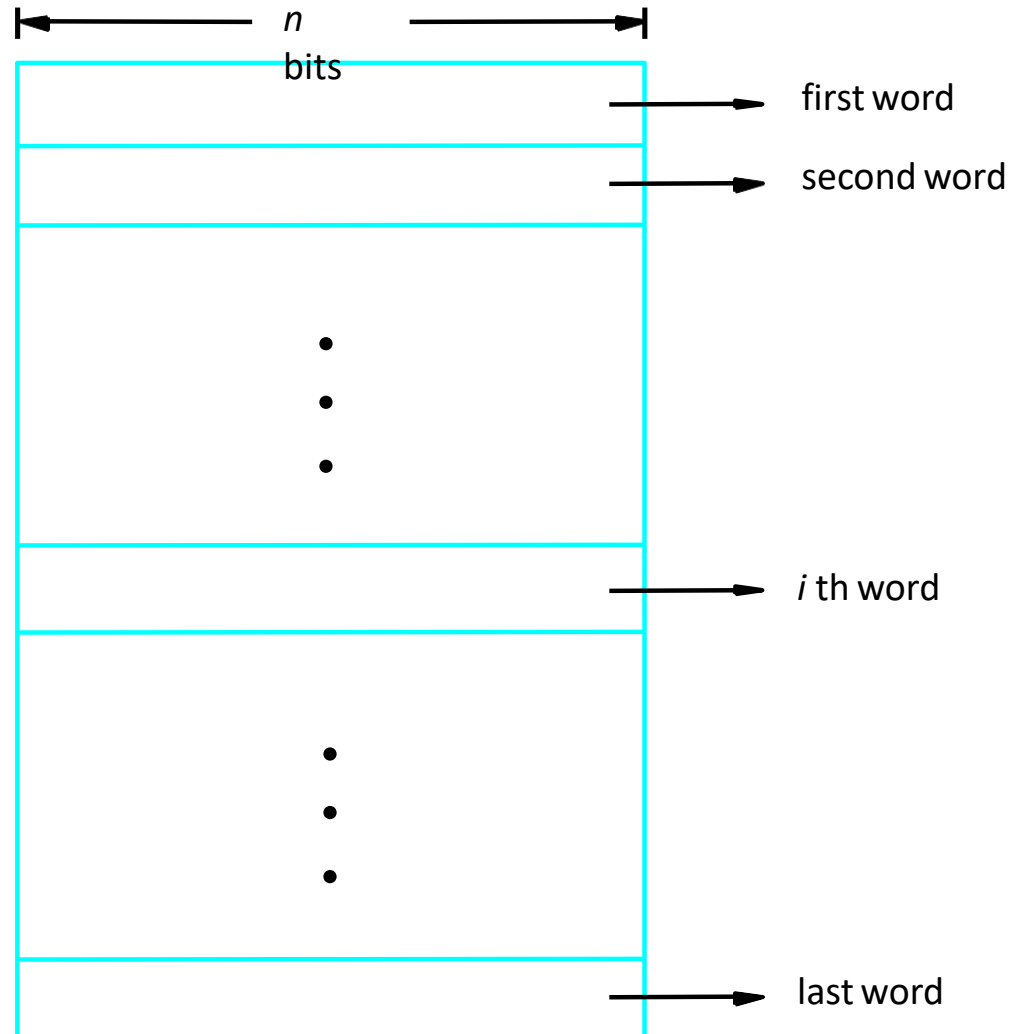
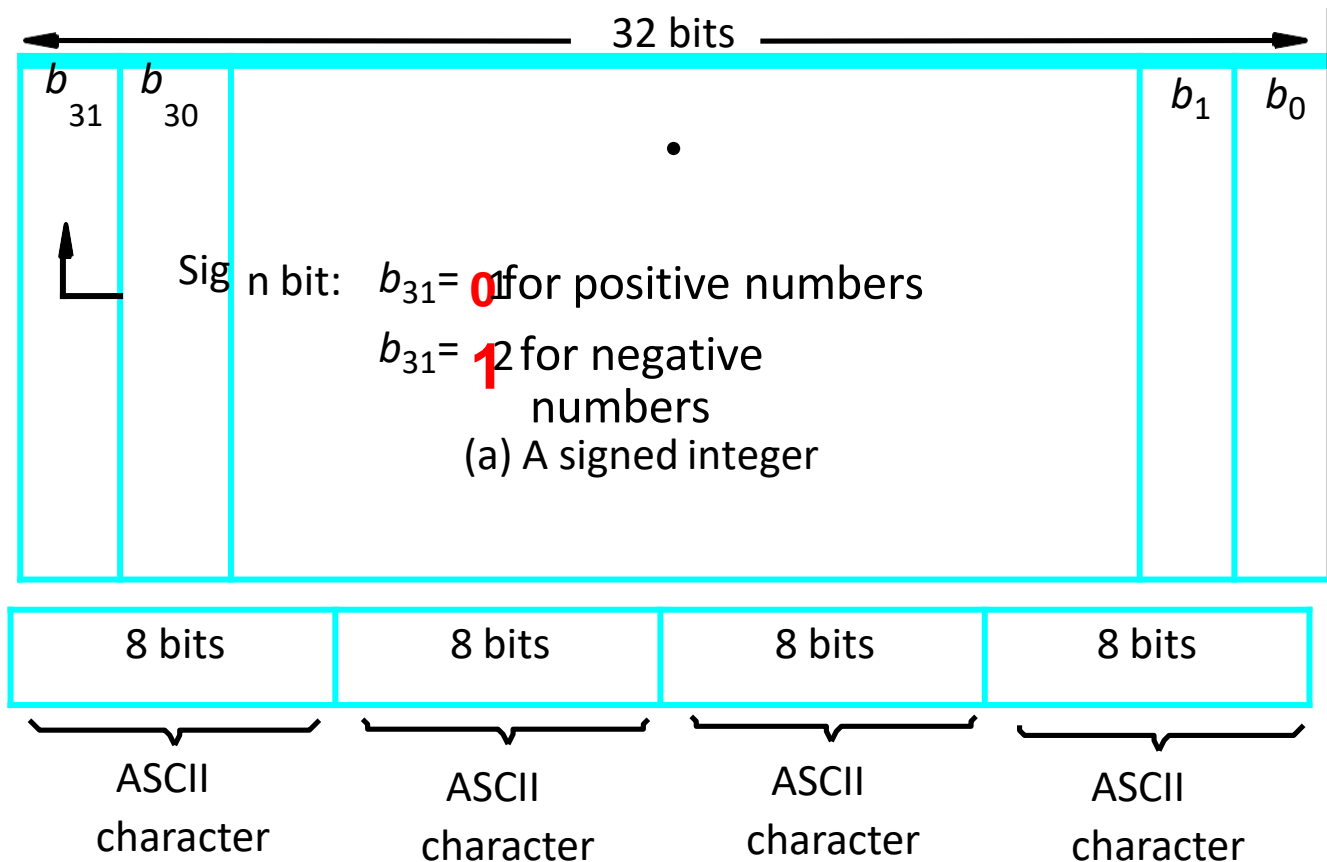


Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example



(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: $2^{24} = 16,777,216 = 16\text{M}$ ($1\text{M} = 2^{20}$)
- 32-bit memory: $2^{32} = 4\text{G}$ ($1\text{G} = 2^{30}$)
- $1\text{K(kilo)} = 2^{10}$
- $1\text{T(tera)} = 2^{40}$

Memory Location, Addresses, and Operation

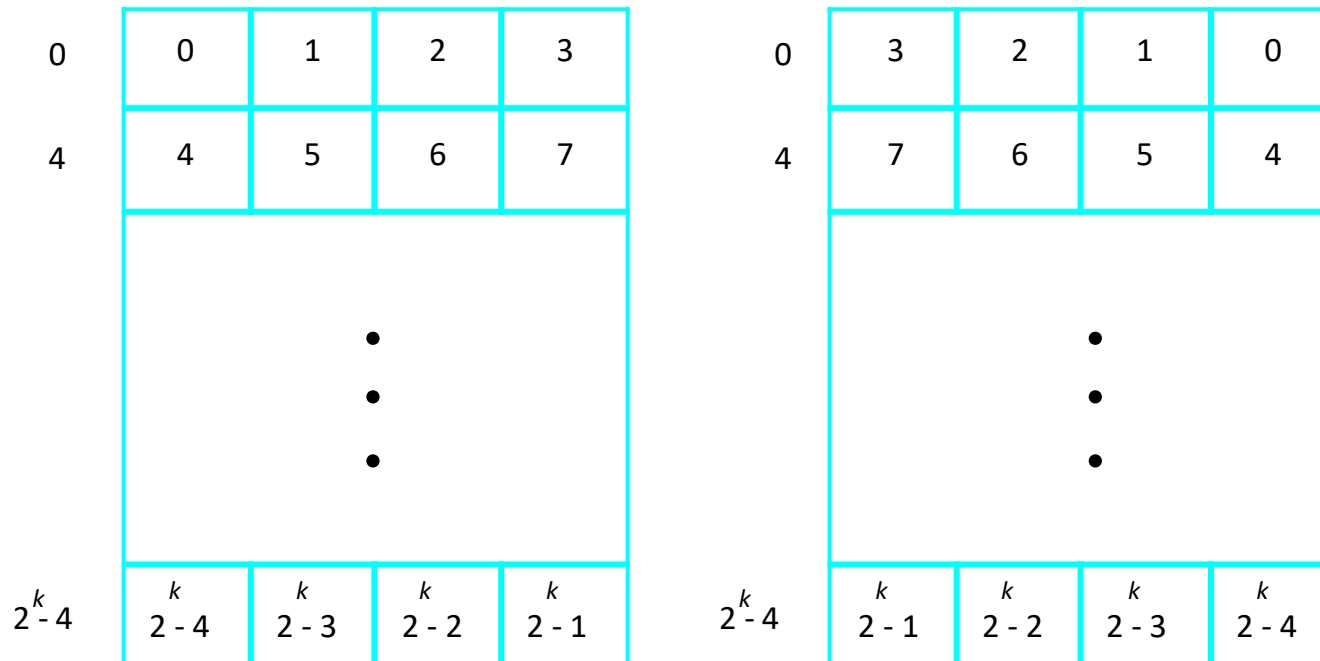
- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, ... If word length is 32 bits,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

Big-Endian and Little-Endian Assignments

Go, change the world

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: wordaddresses: 0, 2, 4,....
 - 32-bit word: wordaddresses: 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 ($R1 \leftarrow [LOC]$, $R3 \leftarrow [R1] + [R2]$)
- Register Transfer Notation (RTN)

Assembly Language Notation

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

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

$R3 \leftarrow R1 + R2$

- Two-Address Instructions

- ADD R1, R2

$R2 \leftarrow R1 + R2$

- One-Address Instructions

ADD M

$AC \leftarrow AC + M[AR]$

Zero-Address Instructions

- ADD

$TOS \leftarrow TOS + (TOS - 1)$

- RISC Instructions

- Lots of registers. Memory is restricted to Load & Store



Instruction Formats

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

- Three-Address

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

Instruction Formats

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

- Two-Address

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

Instruction Formats

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

- One-Address

- | | |
|------------|-----------------------------|
| 1. LOAD A | ; $AC \leftarrow M[A]$ |
| 2. ADD B | ; $AC \leftarrow AC + M[B]$ |
| 3. STORE T | ; $M[T] \leftarrow AC$ |
| 4. LOAD C | ; $AC \leftarrow M[C]$ |
| 5. ADD D | ; $AC \leftarrow AC + M[D]$ |
| 6. MULT T | ; $AC \leftarrow AC * M[T]$ |
| 7. STORE X | ; $M[X] \leftarrow AC$ |

Instruction Formats

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

- Zero-Address

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

Instruction Formats

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

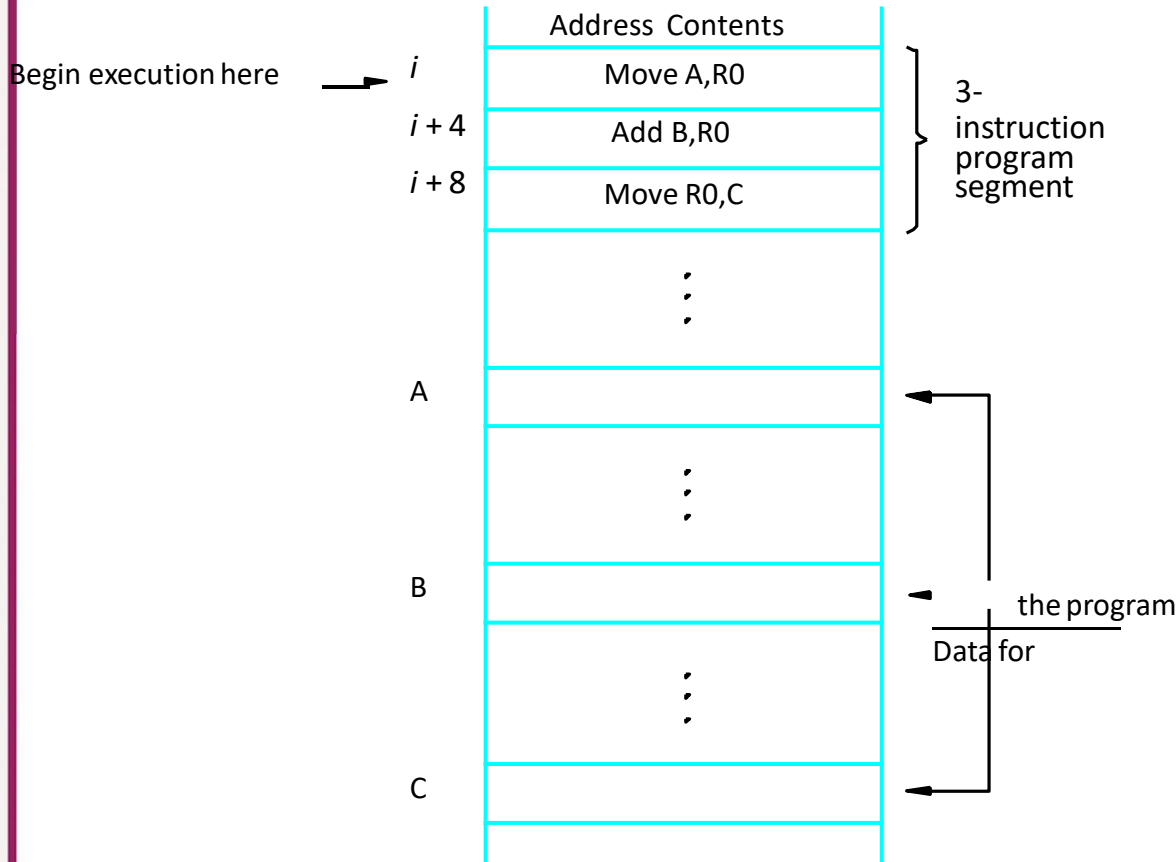
- RISC

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

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

Page 43

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



Branching

Go, change the world

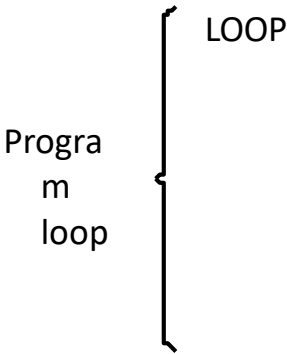
i	Move	NUM1,R0
$i + 4$	Add	NUM2,R0
$i + 8$	Add	NUM3,R0
	•	
	•	
	•	
$i + 4n - 4$	Add	NUM n ,R0
$i + 4n$	Move	R0,SUM
	•	
	•	
	•	
SUM		
NUM1		
NUM2		
	•	
	•	
NUM n	•	

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



Branching

Branch target
Conditional branch



Move	N,R1
Clear	R0
Determine address of "Next" number and add "Next" number to R0	
Decrement	R1
Branch>0	LOOP
Move	R0,SUM
	•
	•
	•
	<i>n</i>
	•
	•
	•

Go, change the world

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

Condition Codes

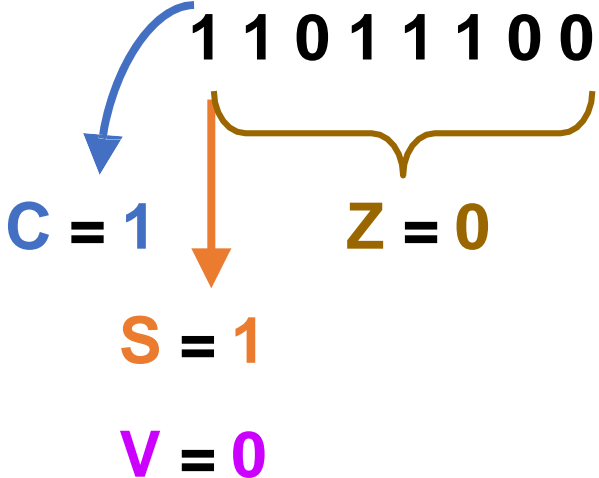
- Condition code flags
- Condition code register / status register
- N (negative)
- Z (zero)
- V (overflow)
- C (carry)
- Different instructions affect different flags

Conditional Branch Instructions

- Example:

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

$$\begin{array}{r}
 \text{A:} \quad 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\
 +(-\text{B}): 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0 \\
 \hline
 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0
 \end{array}$$



C = 1
S = 1
V = 0



Addressing Modes

Generating Memory Addresses

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

Addressing Modes

Implementation of variables and constants

1. Register Addressing Mode : the operand is the contents of a processor register.

Add R0, R1

2. Absolute or Direct Addressing Mode : the operand is in a memory location; the address of this location is given explicitly in the instruction.

Move LOC, R2

3. Immediate Addressing Mode : the operand is given explicitly in the instruction.

Move 200_{immediate}, R0

Move #200, R0

Addressing Modes

Pointers

Indirect Mode

1. **Register Indirect Addressing Mode** : one of the register in the instruction holds the address of the operand

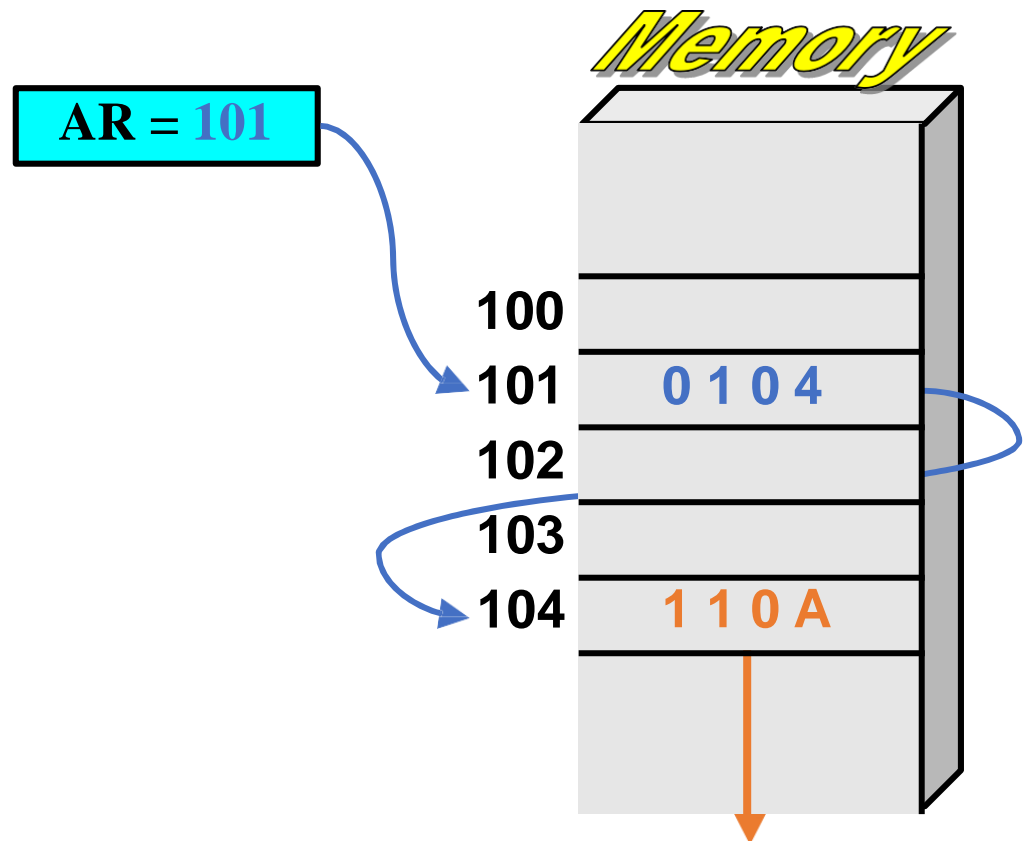
$$EA=[Ri]$$

2. **Indirect Addressing Mode** : the effective address of the operand is the memory location whose address appears in the instruction

$$EA=[LOC]$$

Addressing Modes

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



Indexing and Arrays

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

Addressing Modes

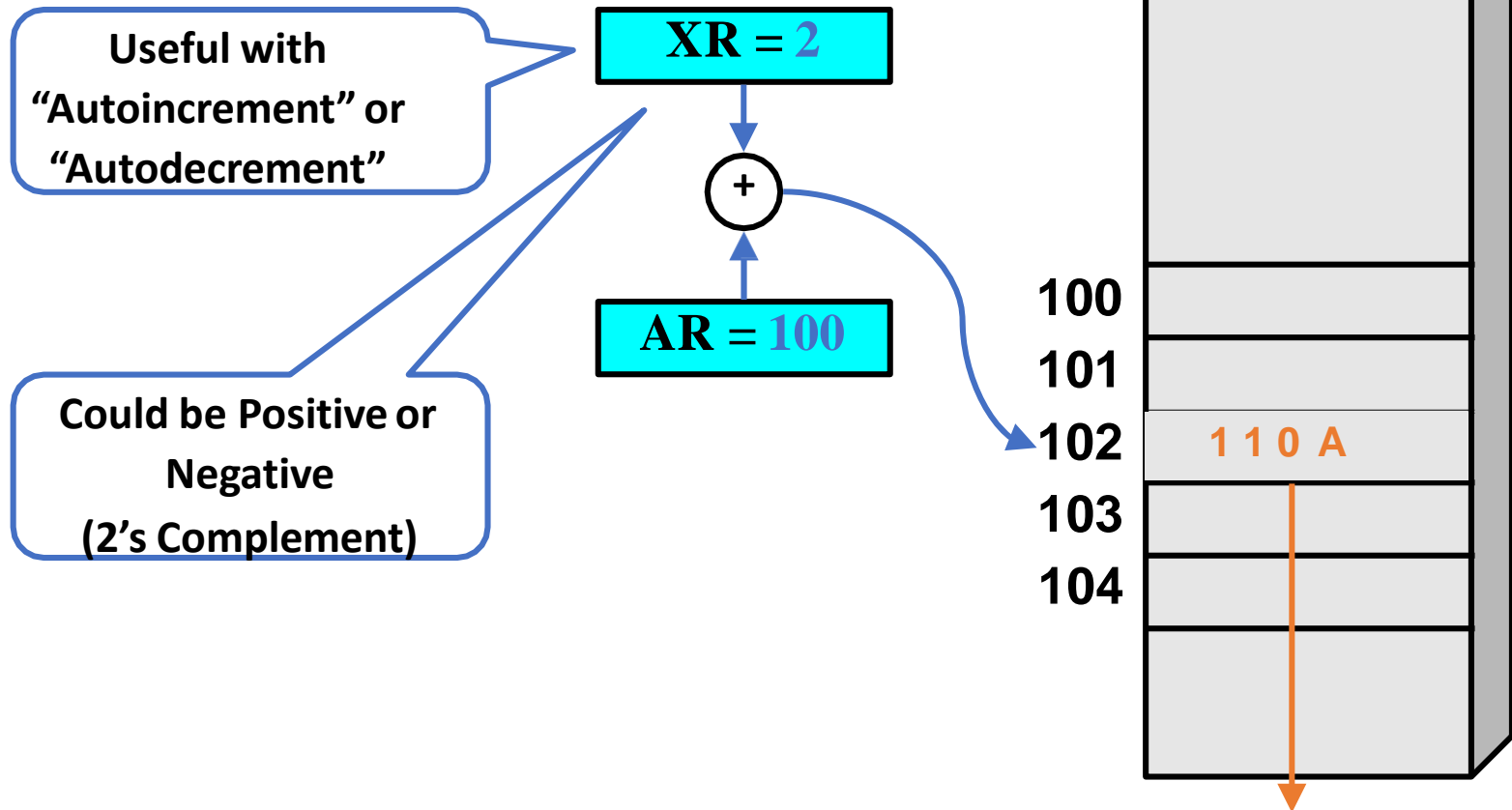
Arrays

1. **Indexed Addressing Mode** : the EA of the operand is generated by adding a constant value to the contents of a register
$$X(R_i) \Rightarrow EA = [R_i] + X$$
2. **Base with Index** : $(R_i, R_j) \Rightarrow EA = [R_i] + [R_j]$
3. **Base with index and Offset** : $X(R_i, R_j) \Rightarrow EA = [R_i] + [R_j] + X$

Addressing Modes

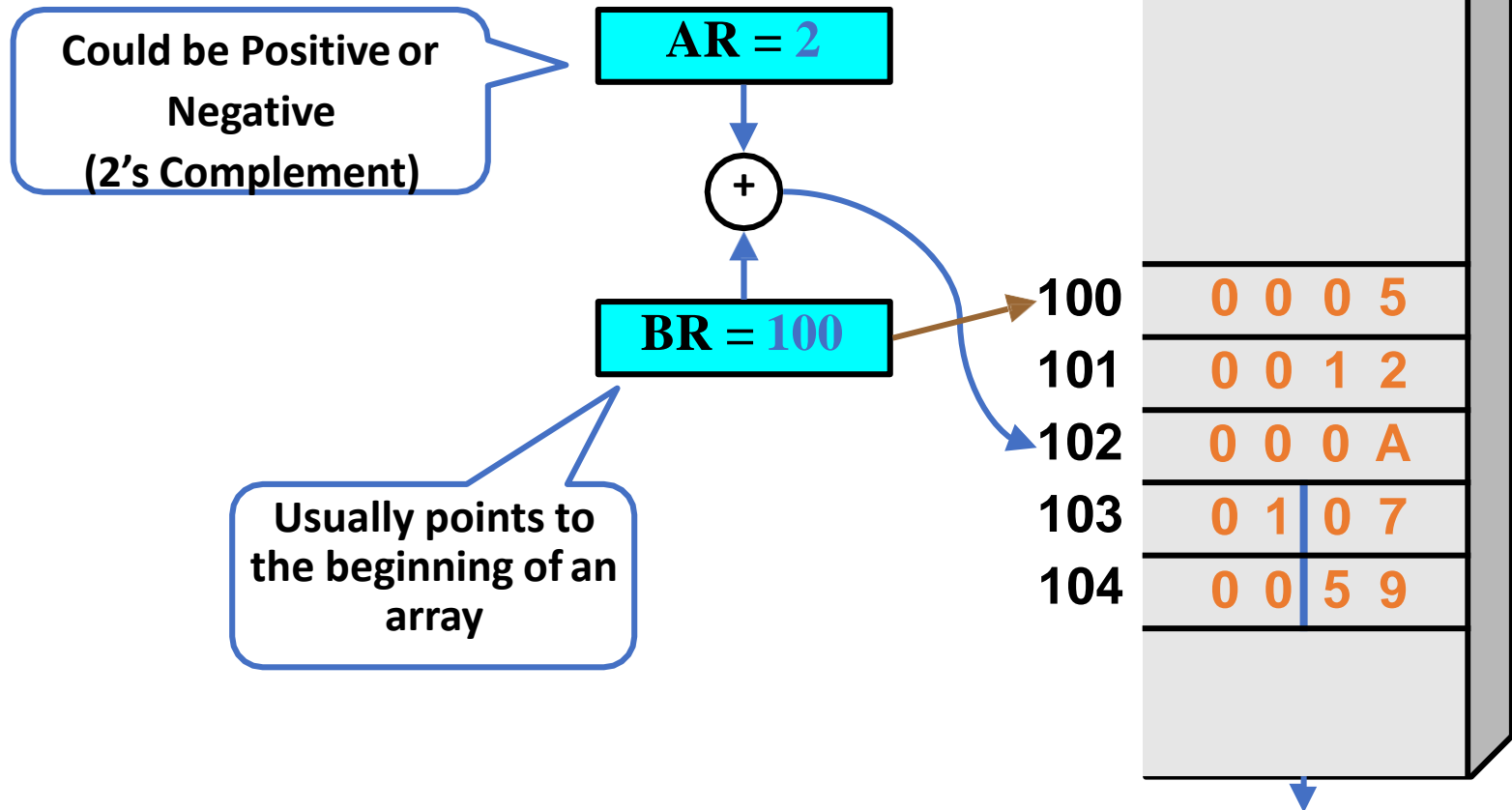
Go, change the world

- Indexed
- $EA = \text{Index Register} + \text{Relative Addr}$



Addressing Modes

- Base Register
- $EA = \text{Base Register} + \text{Relative Addr}$



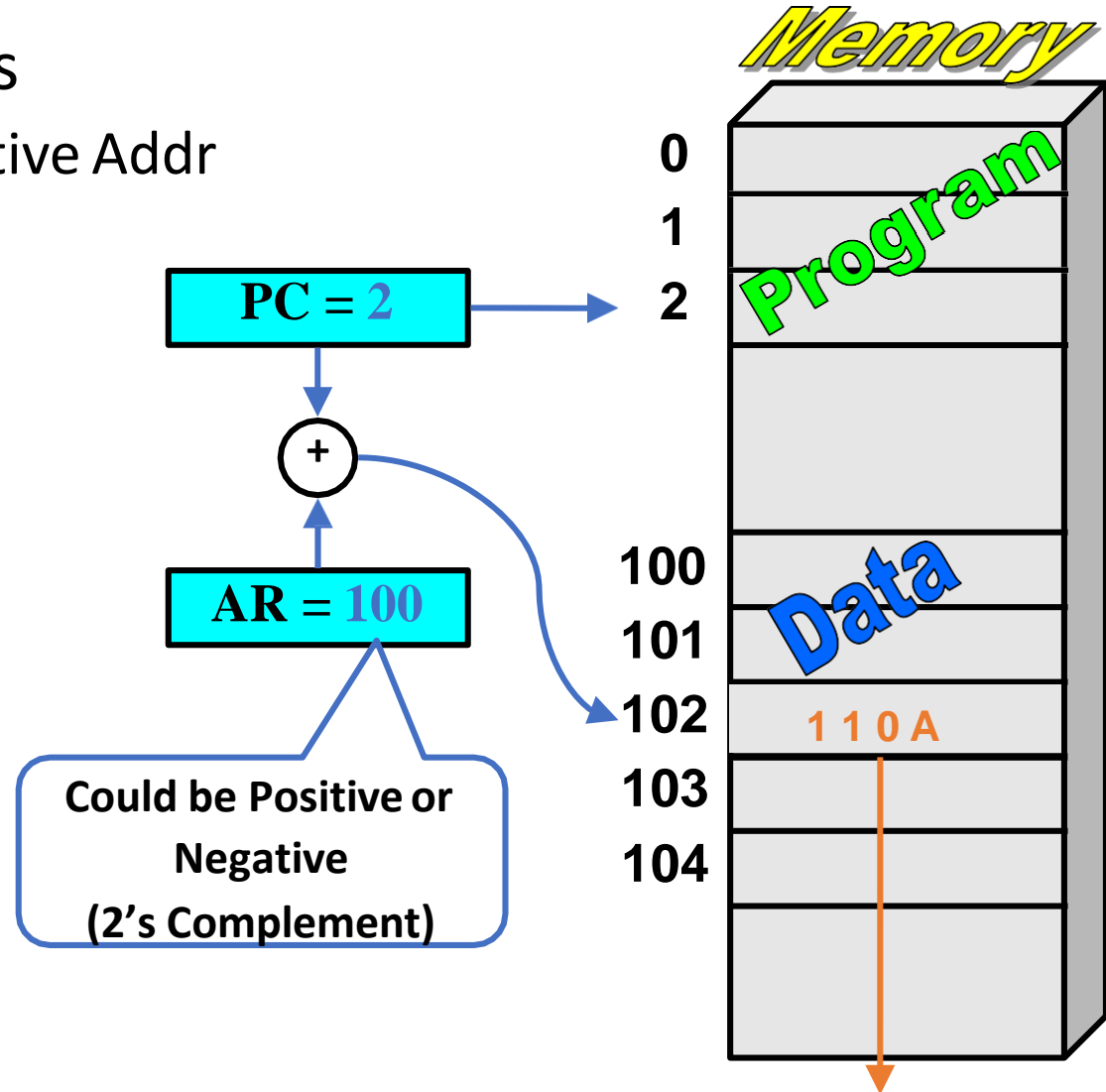
Relative Addressing

- 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 signed num.

Addressing Modes

Go, change the world

- Relative Address
 - $EA = PC + \text{Relative Addr}$



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

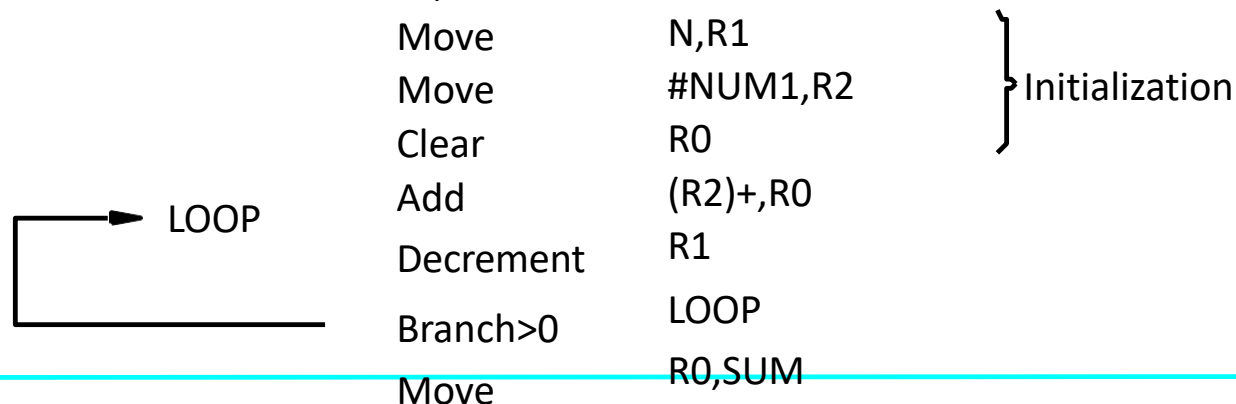


Figure 2.16. The Autoincrement addressing mode used in the program of Figure 2.12.

Addressing Modes

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

Name	Assem bler	syn tax	Addressing function
Immediate #V alue		Op erand = 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 $R\ i;$ $EA = [R\ i]$



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.



Assembly Language

Assembly Language

- Human understandable notation for machine level language
- Mnemonics – symbolic names
- Set of rules – syntax

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 Manipulation Instructions *Go, change the world*

- Arithmetic
- Logical & Bit Manipulation
- Shift

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

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

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	$V = 0$

Assembler Directives

- It allows the programmer to specify other information needed to translate the source program
- `SUM EQU 200`
- It will not appear in the object code
- It simply informs assembler that the Name SUM should be replaced by the value 200

Assembler Directives

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

Figure 2.18 Assembly language representation for the program in Figure 2.17.



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.

I/O

★ Three methods

1. Program controlled IO
2. Interrupt IO
3. Direct Memory Access DMA

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

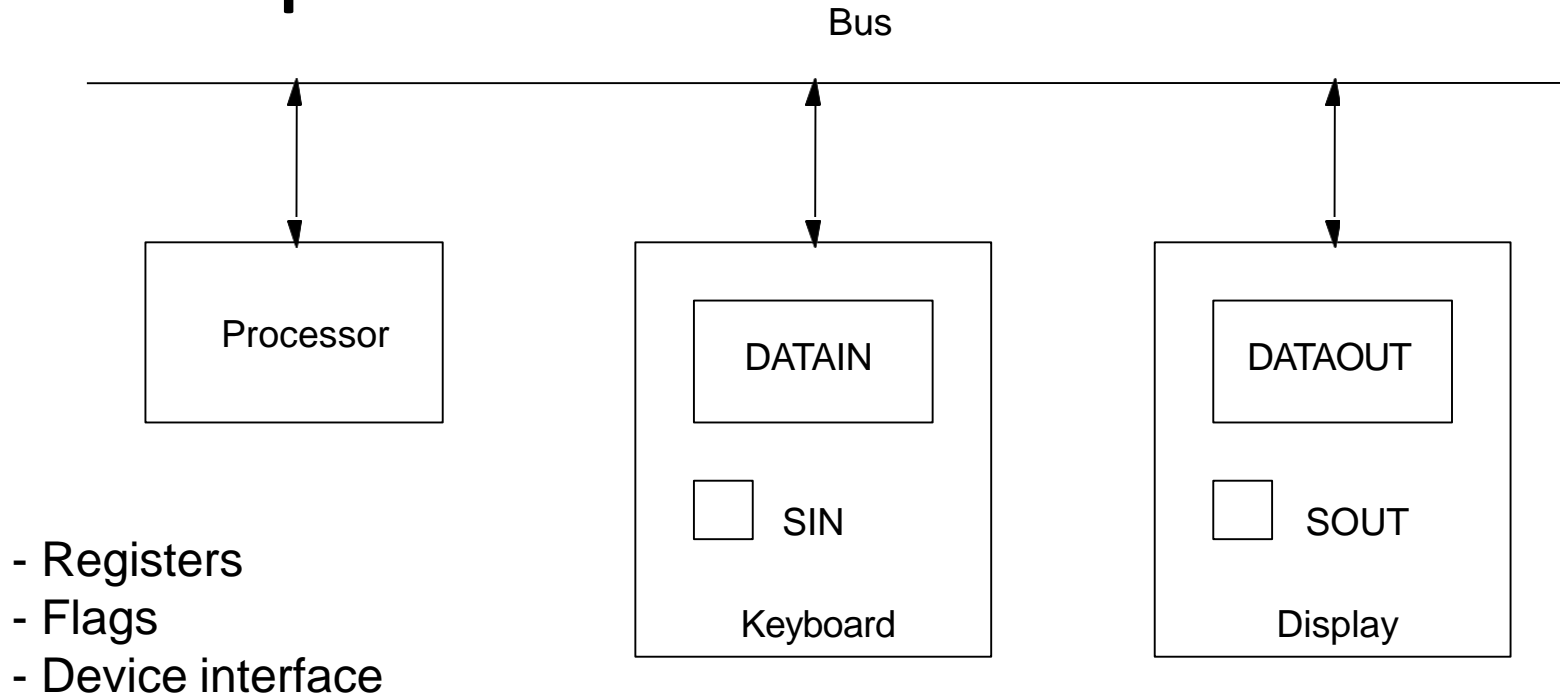


Figure 2.19 Bus connection for proces,sor keyboard, and d.isplay

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

Testbit -> This instruction tests the state of one bit in the destination location, where the bit position to be tested is indicated by the first operand.

Program-Controlled I/O Example

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



Stacks



Stacks

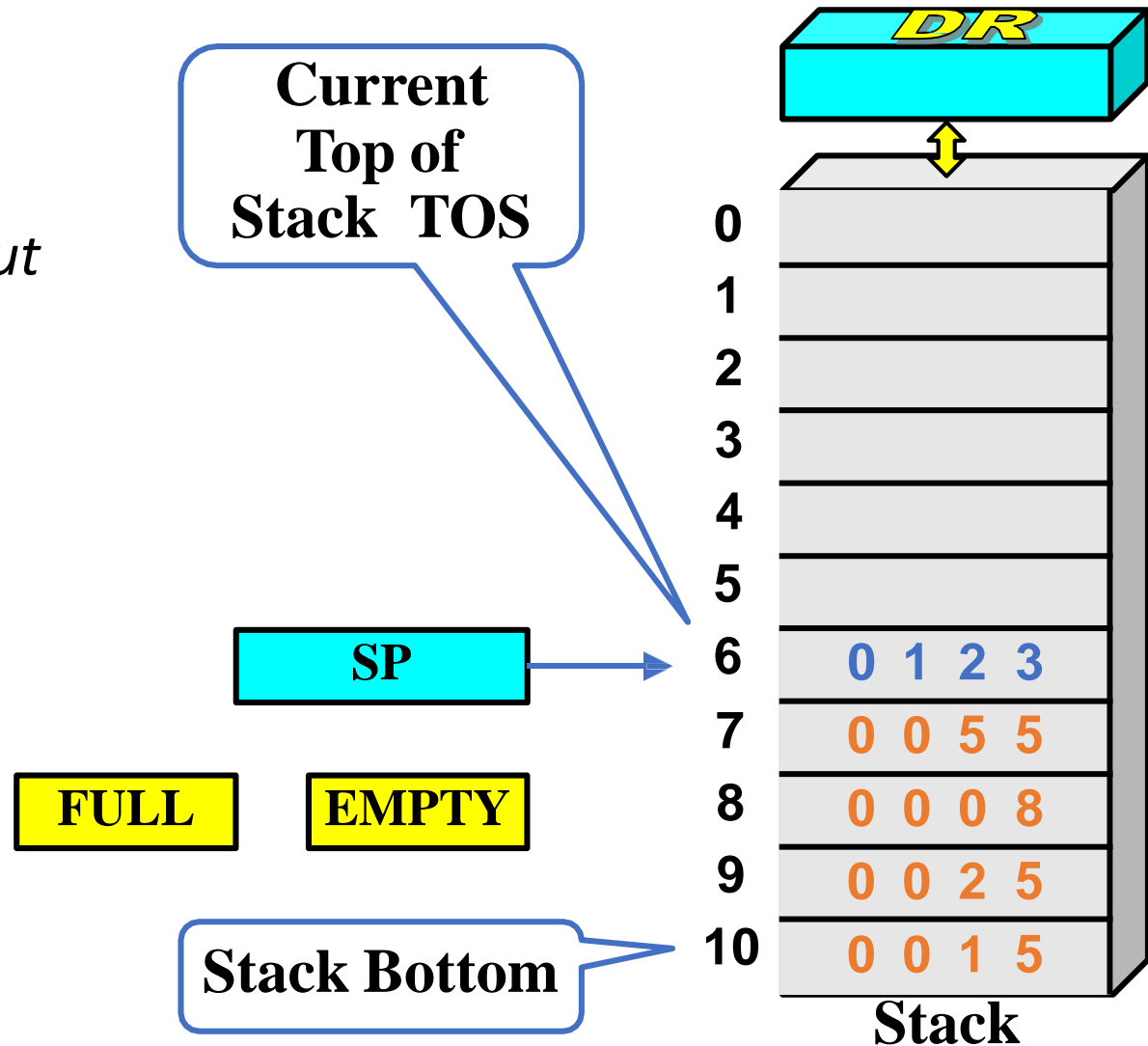
- ★ Data Structure – list of data elements
- ★ Access Restriction - Elements can be added or removed at one end of the list only
- ★ LIFO Stack

Stack Organization

Go, change the world

- LIFO

Last In First Out



Stack Organization

Go, change the world

- PUSH**

$SP \leftarrow SP - 1$

$M[SP] \leftarrow DR$

If $(SP = 0)$ then $(FULL \leftarrow 1)$

$EMPTY \leftarrow 0$

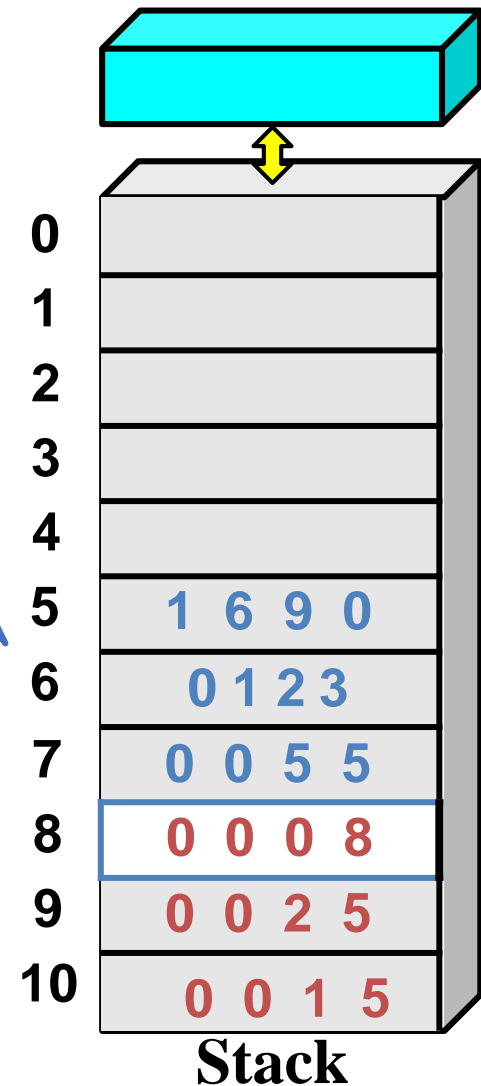
**Current
Top of
Stack TOS**

SP

FULL

EMPTY

Stack Bottom



Stack Organization

Go, change the world

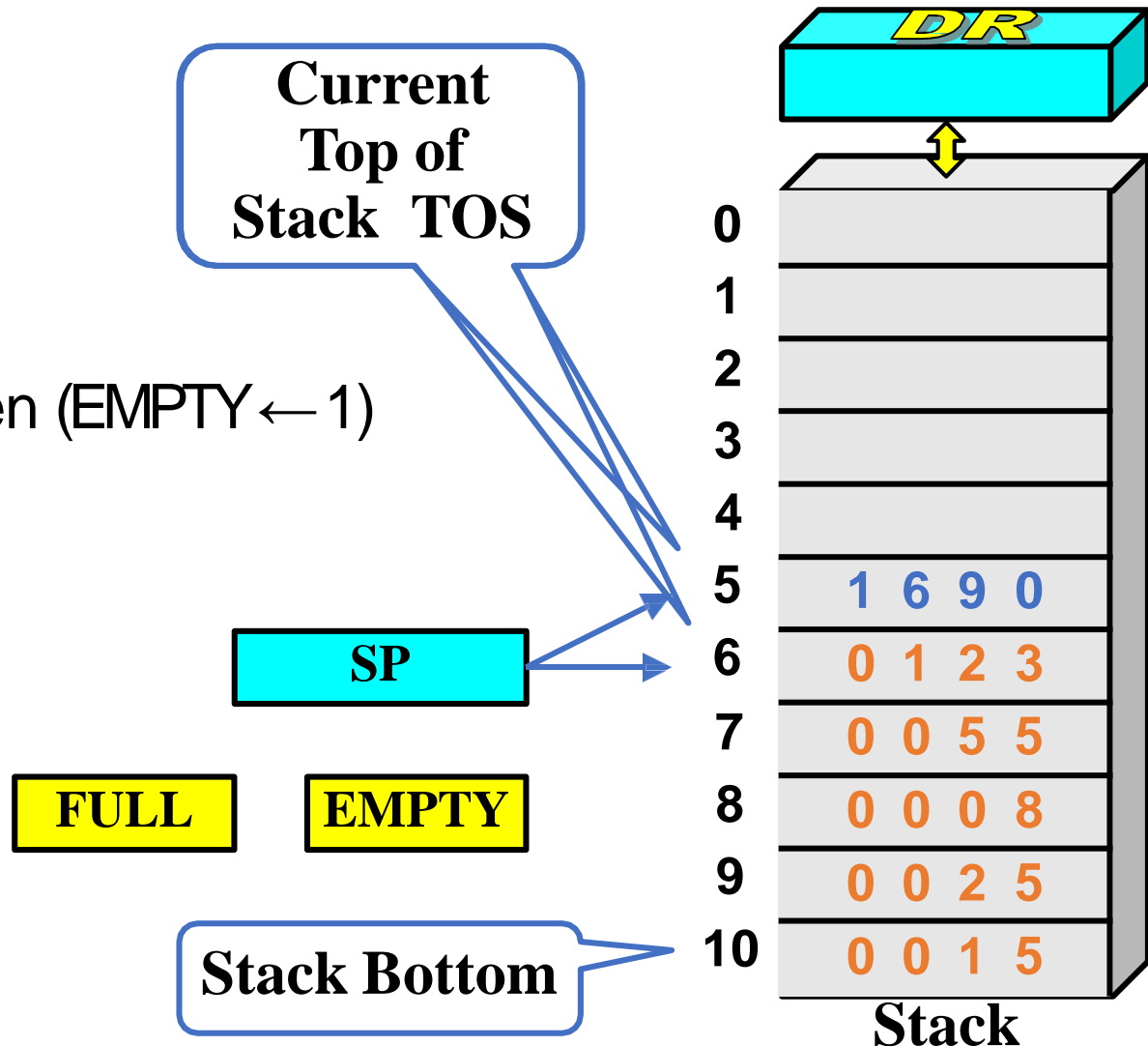
- POP

$DR \leftarrow M[SP]$

$SP \leftarrow SP + 1$

If $(SP = 11)$ then $(EMPTY \leftarrow 1)$

$FULL \leftarrow 0$



Stack Organization

Go, change the world

- Memory Stack

- PUSH**

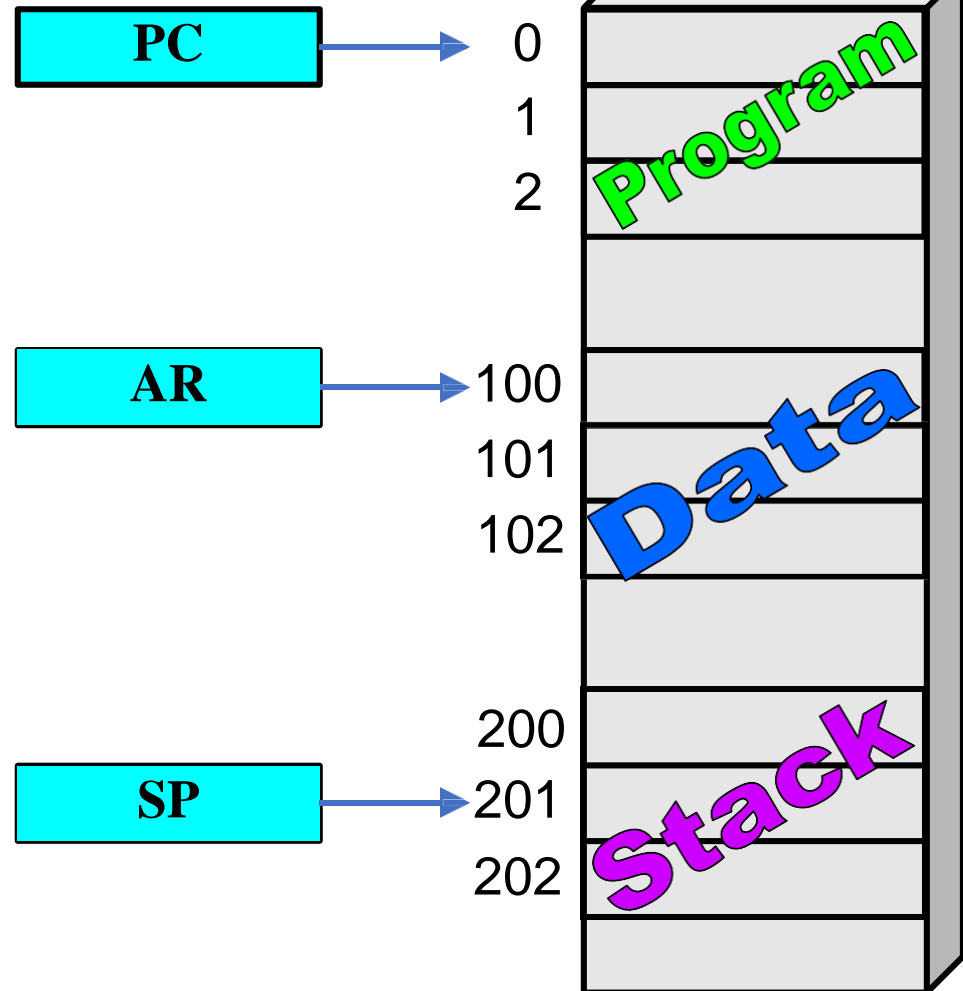
$SP \leftarrow SP - 1$

$M[SP] \leftarrow DR$

- POP**

$DR \leftarrow M[SP]$

$SP \leftarrow SP + 1$





RV Institute of
Technology and
Management®

Go, change the world

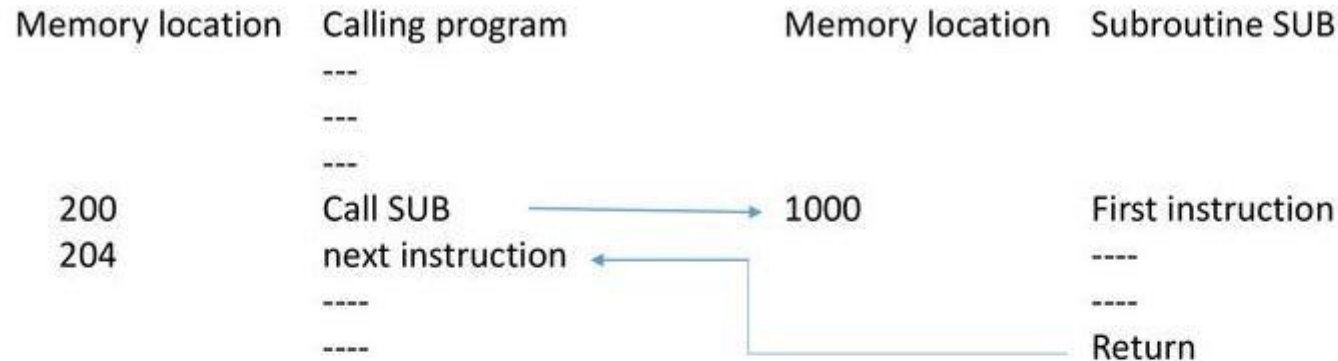
Sub Routines

Subroutines

- It is a subtask
- **CALL instruction**
 - ❑ Store the contents of the PC in the link register
 - ❑ Branch to the target address specified by the instruction
- **RETURN instruction**
 - ❑ Branch to the address contained in the link register
- The way in which a computer makes it possible to call and return from subroutine – **subroutine linkage**

Subroutine

Go, change the world



Here, address of next instruction must be saved by the Call instruction to enable returning to Calling program

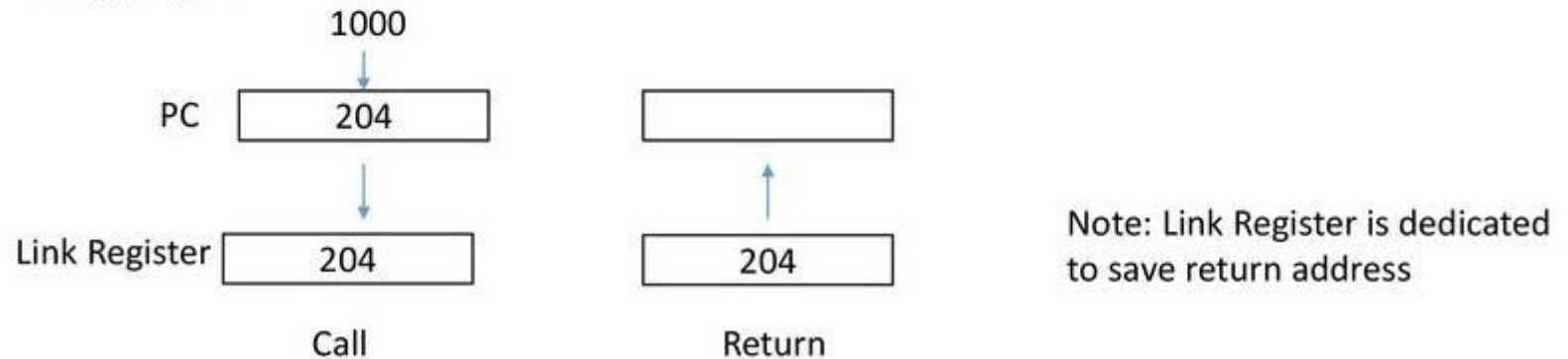


Figure 2.24 Subroutine linkage using a link register

Parameter Passing

Calling program

Move	N,R1	R1 serves as a counter.
Move	#NUM1,R2	R2 points to the list.
Call	LISTADD	Call subroutine.
Move	R0,SUM	Save result.
:		
:		

Subroutine

LISTADD	Clear	R0	Initialize sum to 0.
LOOP	Add	(R2)+,R0	Add entry from list.
	Decrement	R1	
	Branch>0	LOOP	
	Return		Return to calling program.

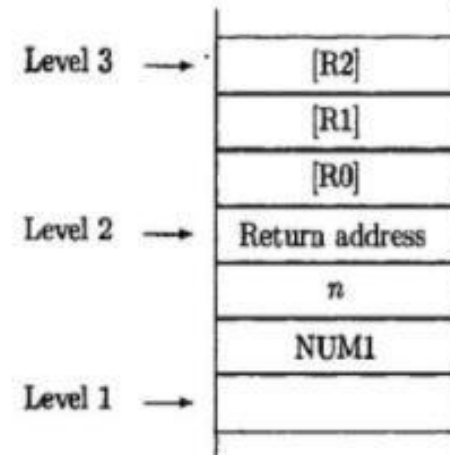
Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

Program of subroutine Parameters passed on the stack

Go, change the world

Assume top of stack is at level 1 below.

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

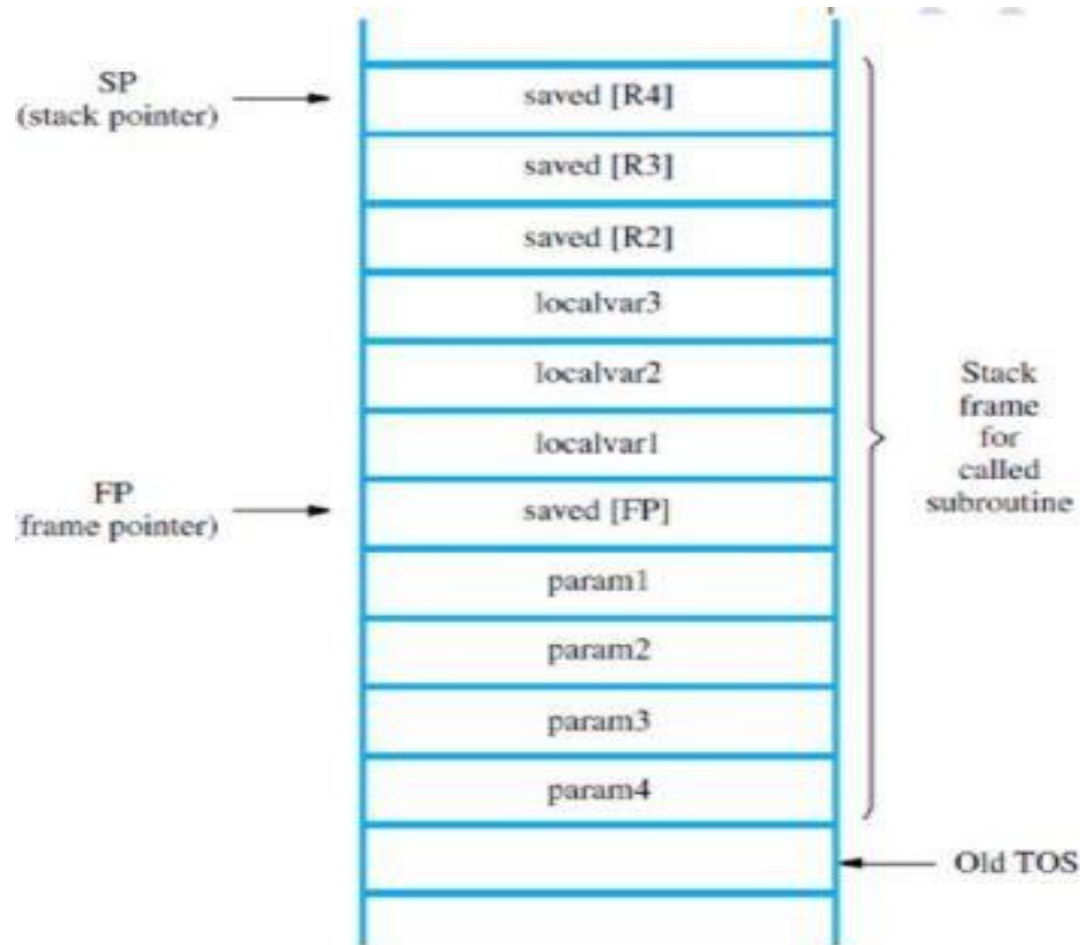


Stack Frame

- Location constitute a private work space for the subroutine
- Created at the time the subroutine is entered and freed up when the subroutine returns
- Frame Pointer – to access the local variables of subroutine

The StackFrame

Go, change the world



Stack Frame for Nested Subroutine

Main program

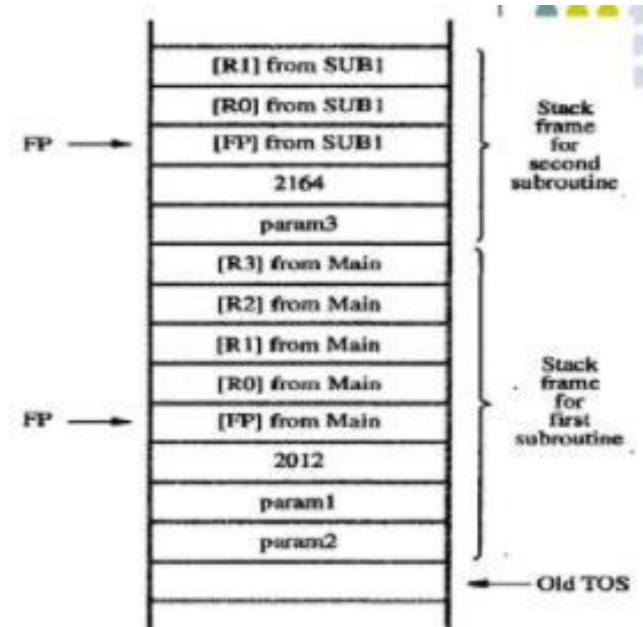
```

:
2000 Move    PARAM2, -(SP)  Place parameters on stack.
2004 Move    PARAM1, -(SP)
2008 Call    SUB1
2012 Move    (SP), RESULT   Store result.
2016 Add     #8, SP         Restore stack level.
2020 next instruction
  
```

First subroutine

```

2100 SUB1 Move    FP, -(SP)   Save frame pointer register.
2104      Move    SP, FP      Load the frame pointer.
2108      MoveMultiple R0-R3, -(SP) Save registers.
2112      Move    8(FP), R0    Get first parameter.
      Move    12(FP), R1      Get second parameter.
:
      Move    PARAM3, -(SP)   Place a parameter on stack.
2160 Call    SUB2
2164      Move    (SP)+, R2    Pop SUB2 result into R2.
:
      Move    R3, 8(FP)       Place answer on stack.
      MoveMultiple (SP)+, R0-R3 Restore registers.
      Move    (SP)+, FP       Restore frame pointer register.
      Return
  
```



Second subroutine

```

3000 SUB2 Move    FP, -(SP)   Save frame pointer register.
      Move    SP, FP      Load the frame pointer.
      MoveMultiple R0-R1, -(SP) Save registers R0 and R1.
      Move    8(FP), R0    Get the parameter.
:
      Move    R1, 8(FP)     Place SUB2 result on stack.
      MoveMultiple (SP)+, R0-R1 Restore registers R0 and R1.
      Move    (SP)+, FP     Restore frame pointer register.
      Return
  
```



Additional Instructions

Logical Shifts

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



before: 0 0 1 1 1 0 · 0 1 1

after: 1 1 1 0 ... 0 1 1 0 0

(a) Logical shift left · LShiftL #2, R0



before: 0 1 1 1 0 · 0 1 1 0

after: 0 0 0 1 1 1 0 · 0 1

(b) Logical shift right · LShiftR #2, R0

Arithmetic Shifts



Rotate

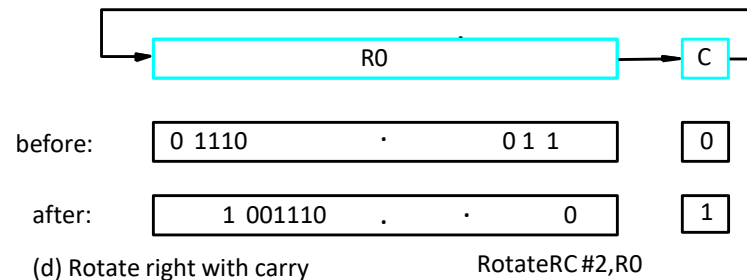
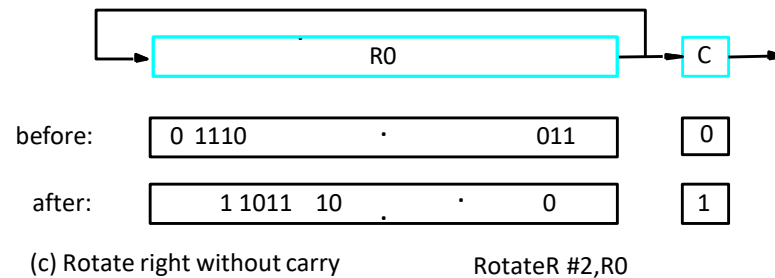
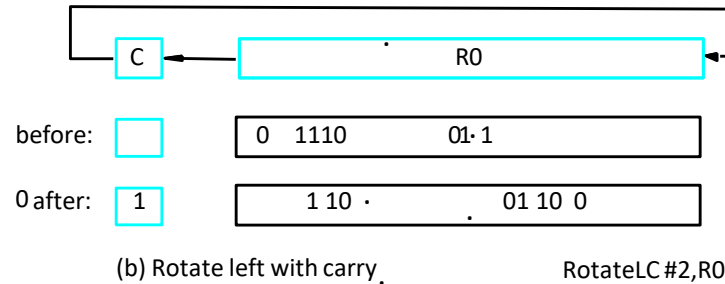
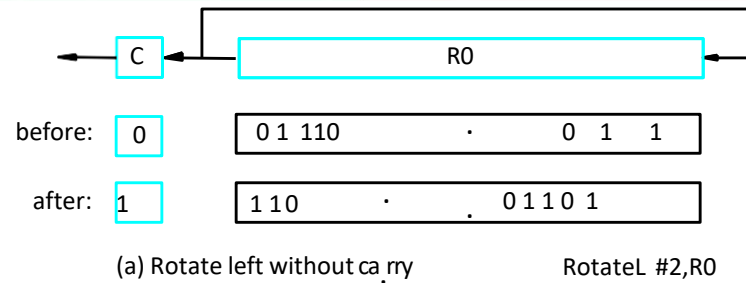


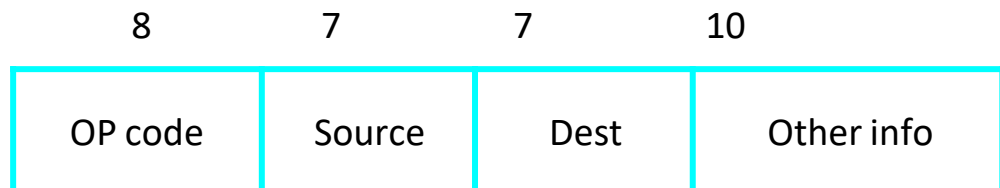
Figure 2.32. Rotate instructions.



Encoding of Machine Instructions

Encoding of Machine Instructions

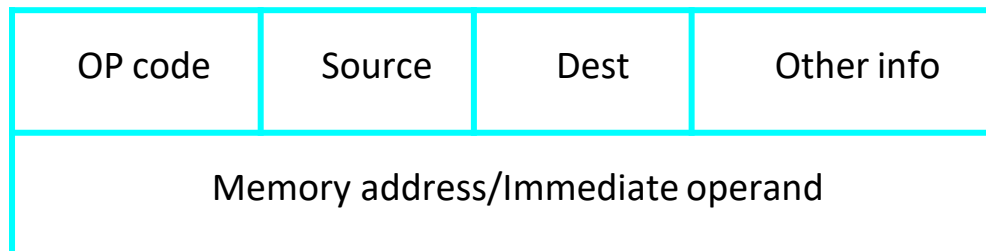
- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)
- In the previous section, an assumption was made that all instructions are one word in length.
- OP code: the type of operation to be performed and the type of operands used may be specified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