Chapter 2. Machine Instructions and Programs

Objectives

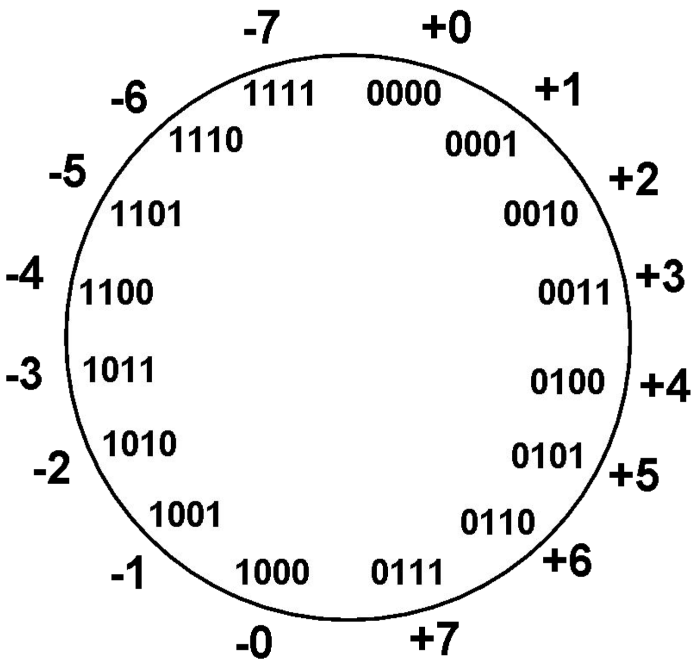
- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2's-complement system.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions,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

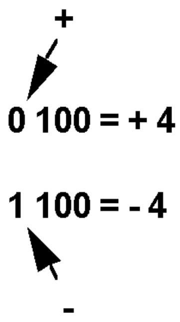
Number, Arithmetic Operations, and Characters

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

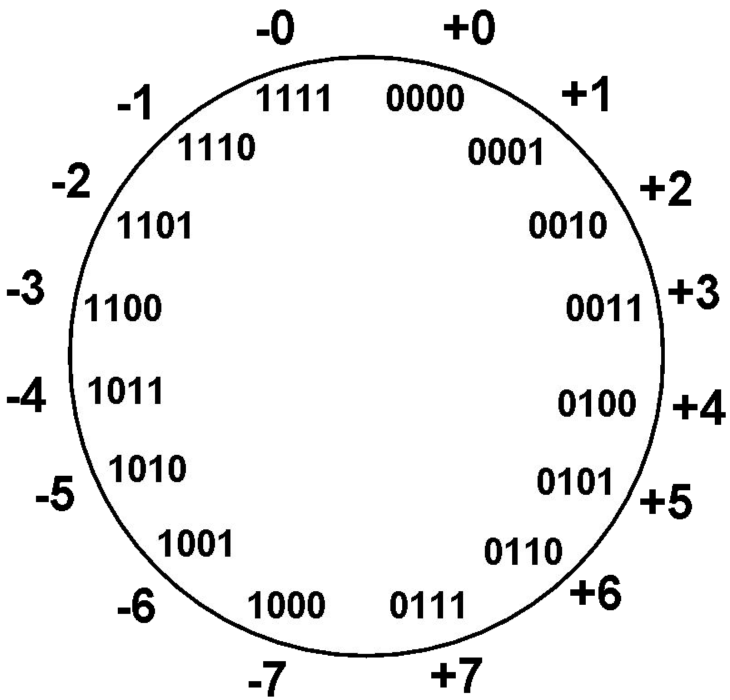
Sign and Magnitude Representation

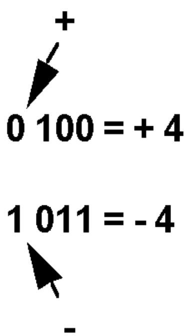




High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Number range for n bits = +/-2^{n-1} -1
Two representations for 0

One's Complement Representation

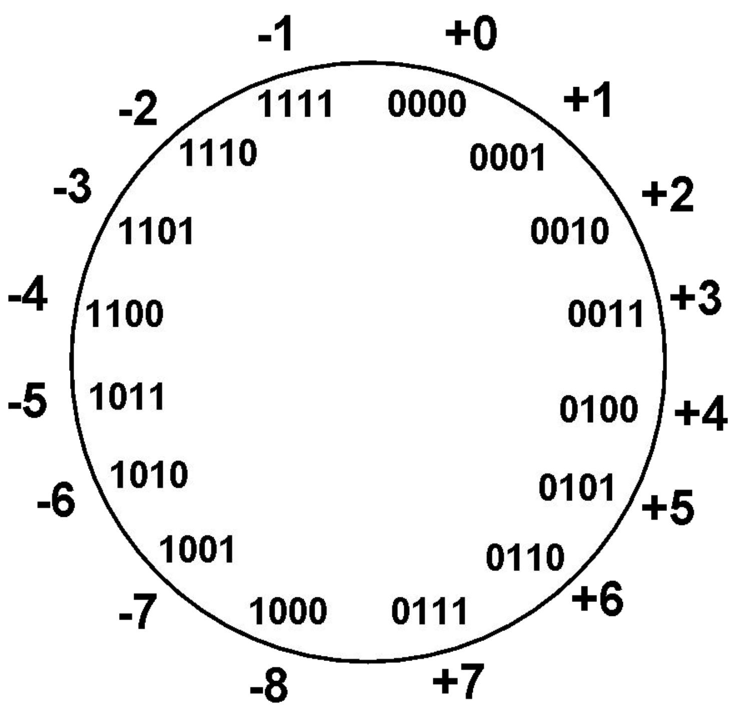


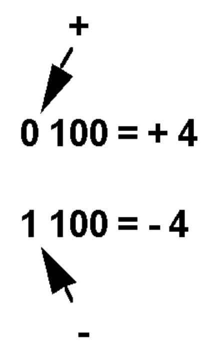


- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

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

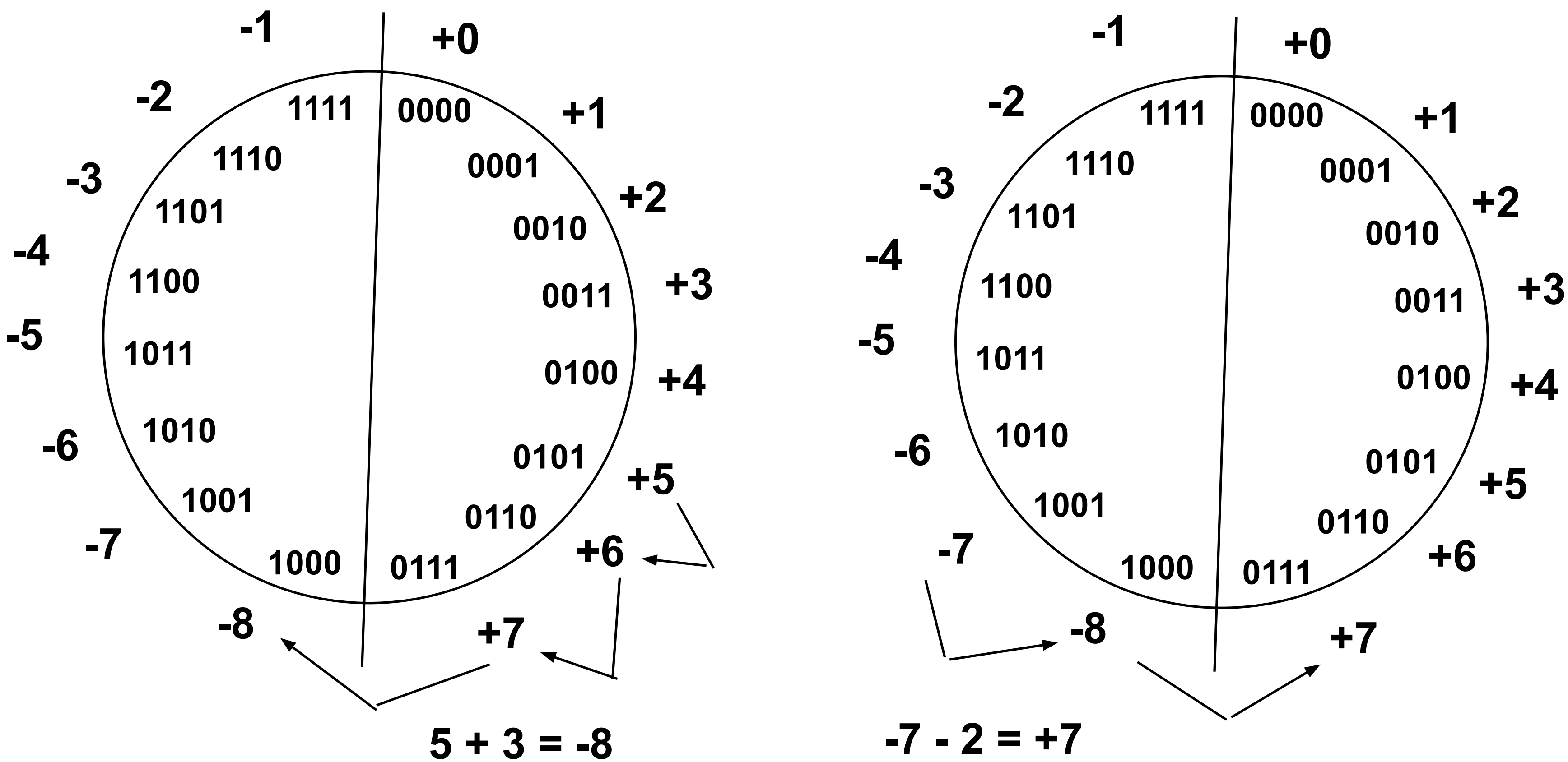
2’s-Complement Add and Subtract Operations

Page 31

(a)	0010	(+2)	(b)	0100	(+4)
	+ 0011	(+3)		+ 1010	(- 6)
	0101	(+5)		1110	(- 2)
(c)	1011	(- 5)	(d)	0111	(+7)
	+ 1110	(- 2)		+ 1101	(- 3)
	1001	(- 7)		0100	(+4)
(e)	1101	(- 3)	⇒	1101	
	- 1001	(- 7)		+ 0111	
				0100	(+4)
(f)	0010	(+2)	⇒	0010	
	- 0100	(+ 4)		+ 1100	
				1110	(- 2)
(g)	0110	(+6)	⇒	0110	
	- 0011	(+3)		+ 1101	
				0011	(+3)
(h)	1001	(- 7)	⇒	1001	
	- 1011	(- 5)		+ 0101	
				1110	(- 2)
(i)	1001	(- 7)	⇒	1001	
	- 0001	(+1)		+ 1111	
				1000	(- 8)
(j)	0010	(+2)	⇒	0010	
	- 1101	(- 3)		+ 0011	
				0101	(+ 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

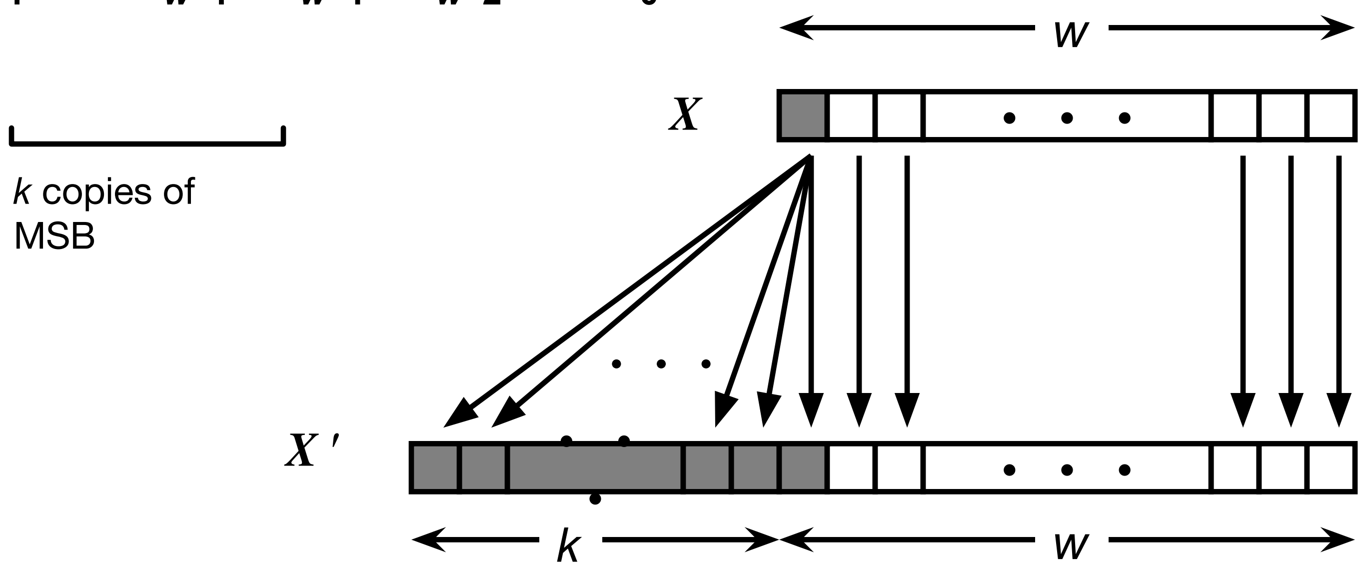
	1 1 1 1
-3	1 1 0 1
-5	1 0 1 1
-8	1 1 0 0 0

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:
  - $\boldsymbol{X}' = \boldsymbol{x}_{w-1}, \ldots, \boldsymbol{x}_{w-1}, \boldsymbol{x}_{w-1}, \boldsymbol{x}_{w-2}, \ldots, \boldsymbol{x}_0$



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.

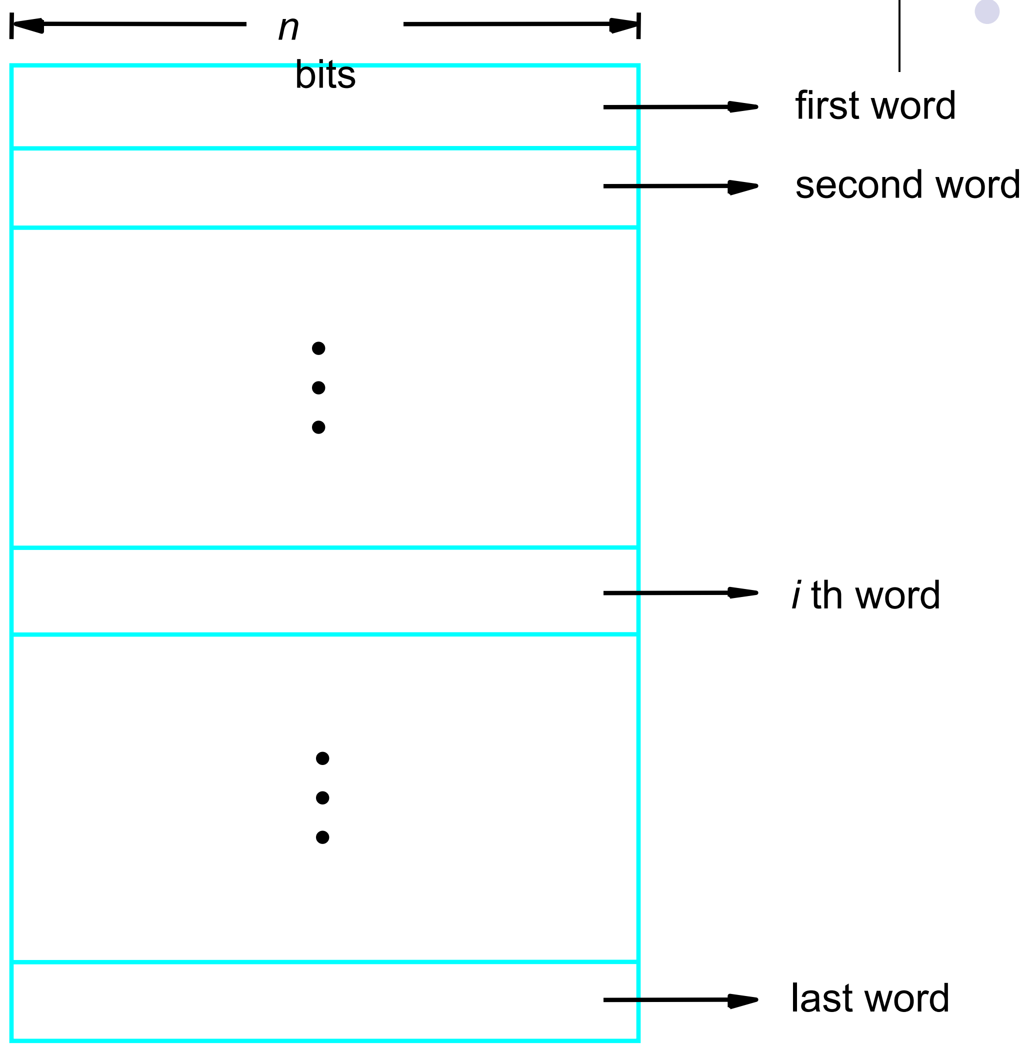
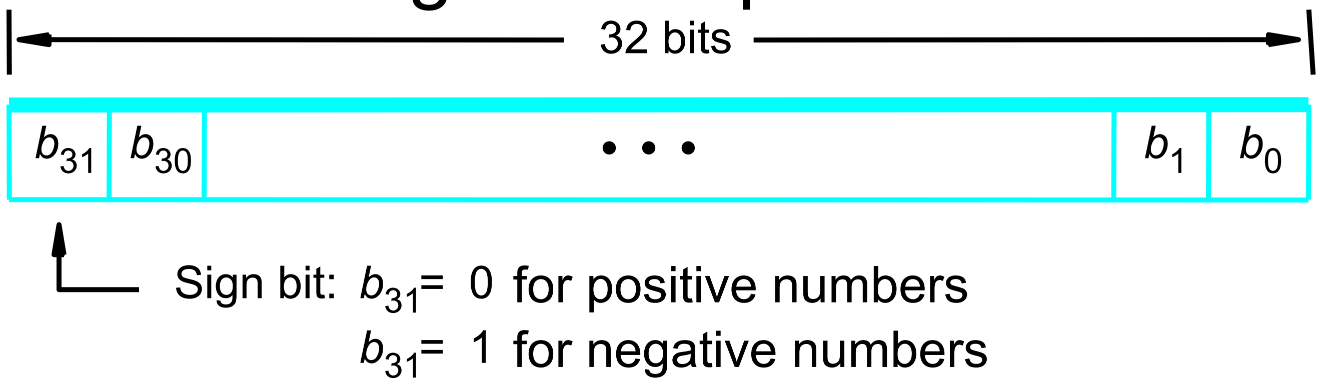


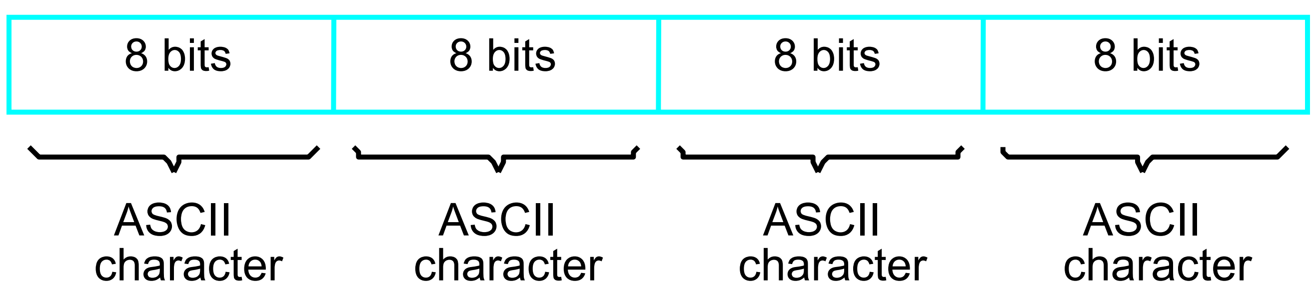
Figure 2.5. Memory words.

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

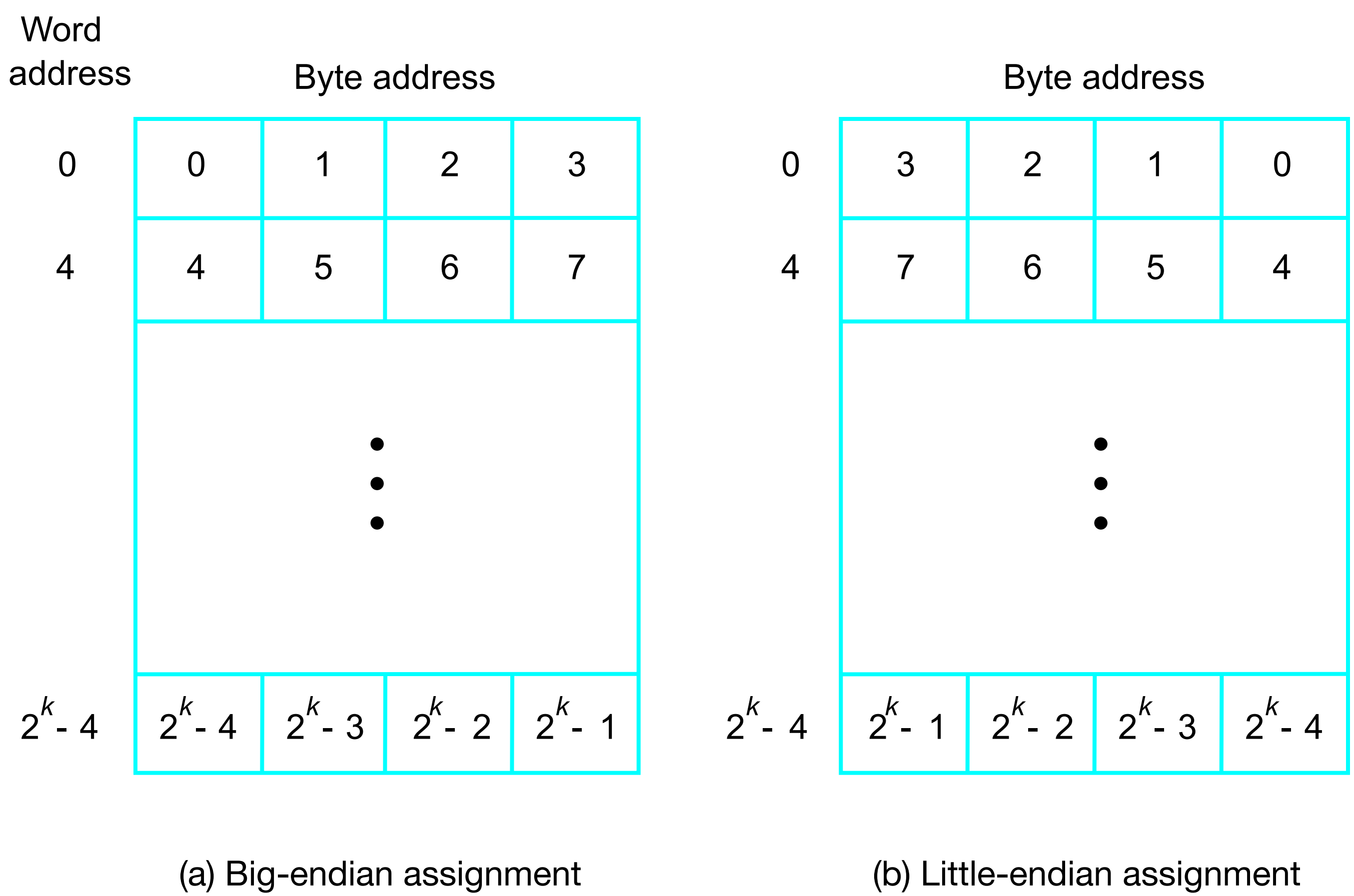


Figure 2.7. Byte and word addressing.

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

Instruction Formats

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

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

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 $\leftarrow$ 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 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. STORE X, 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

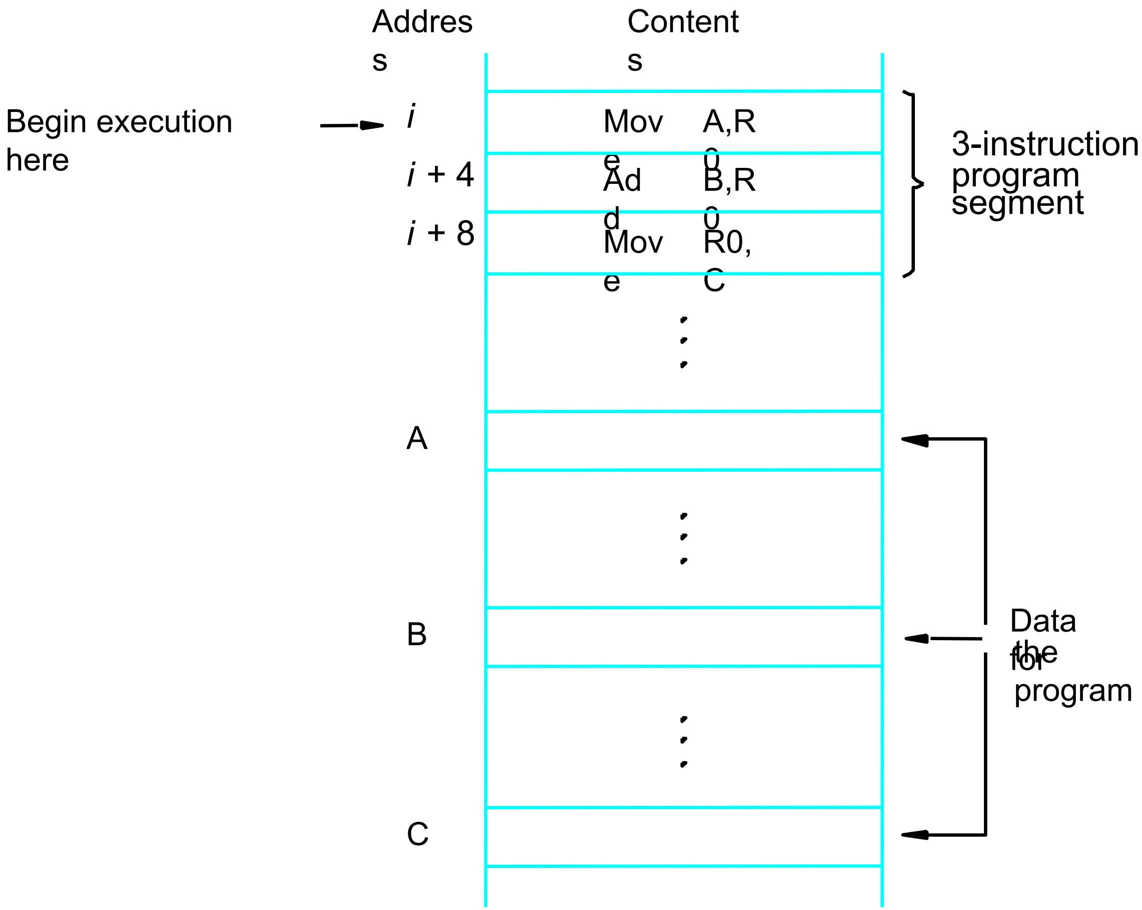


Figure 2.8. A program for C ← [A] + [B].

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

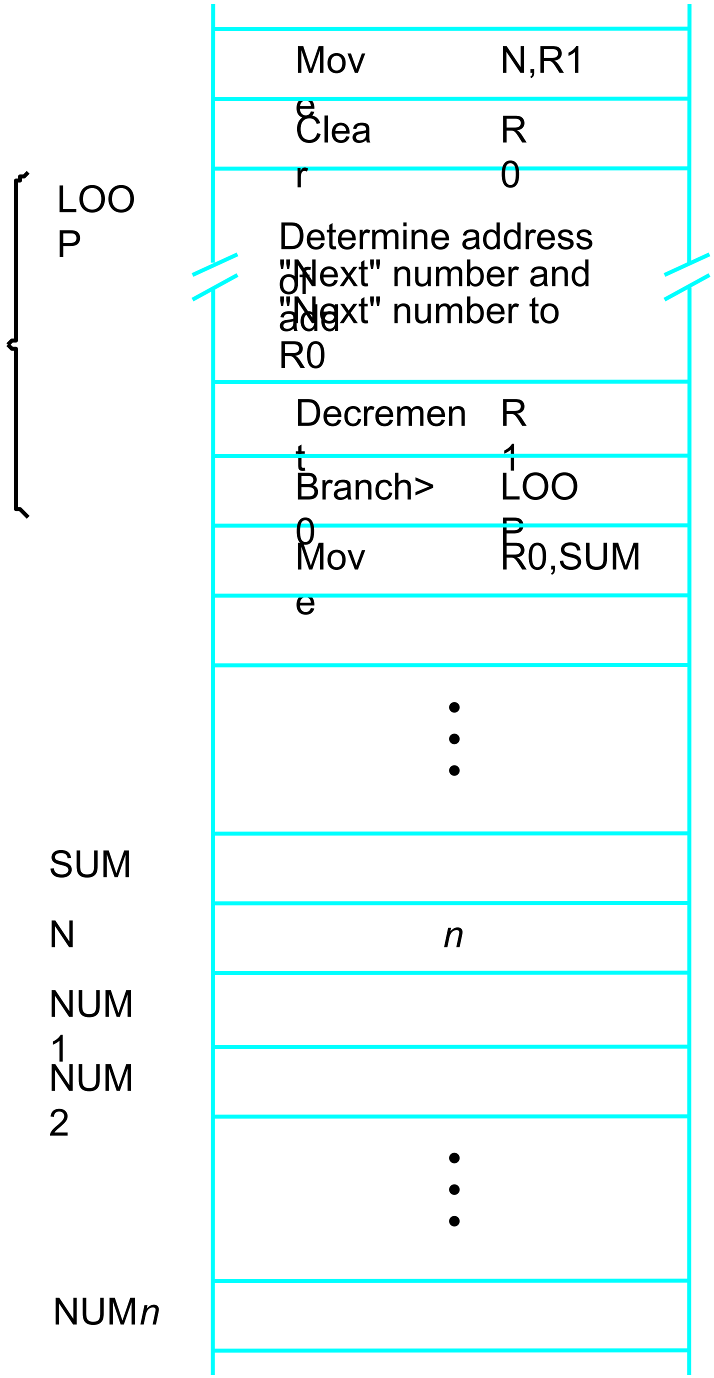


Figure 2.10. Using a loop to add *n* numbers.

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

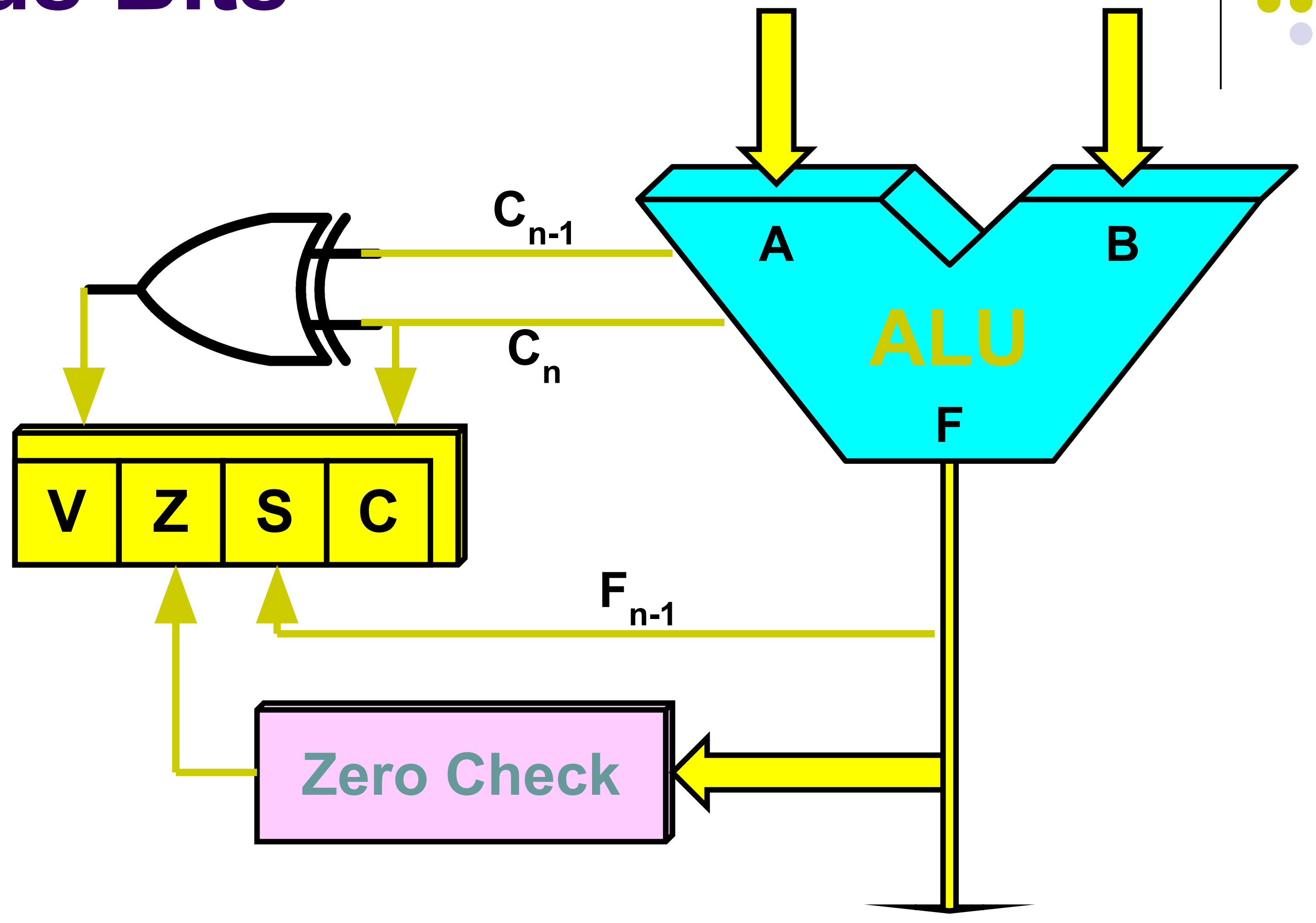
C = 1

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.

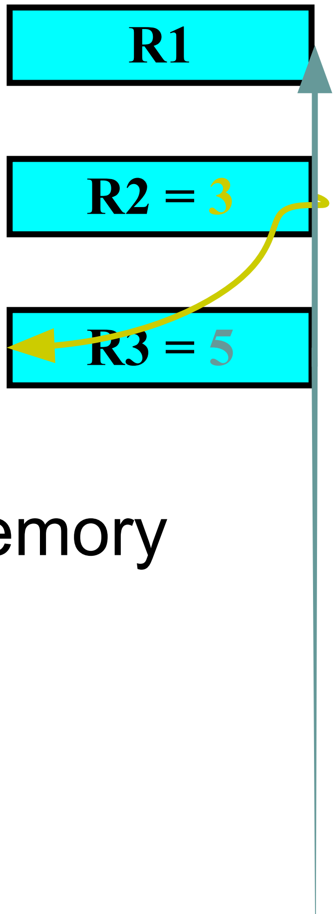
Addressing Modes

- 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

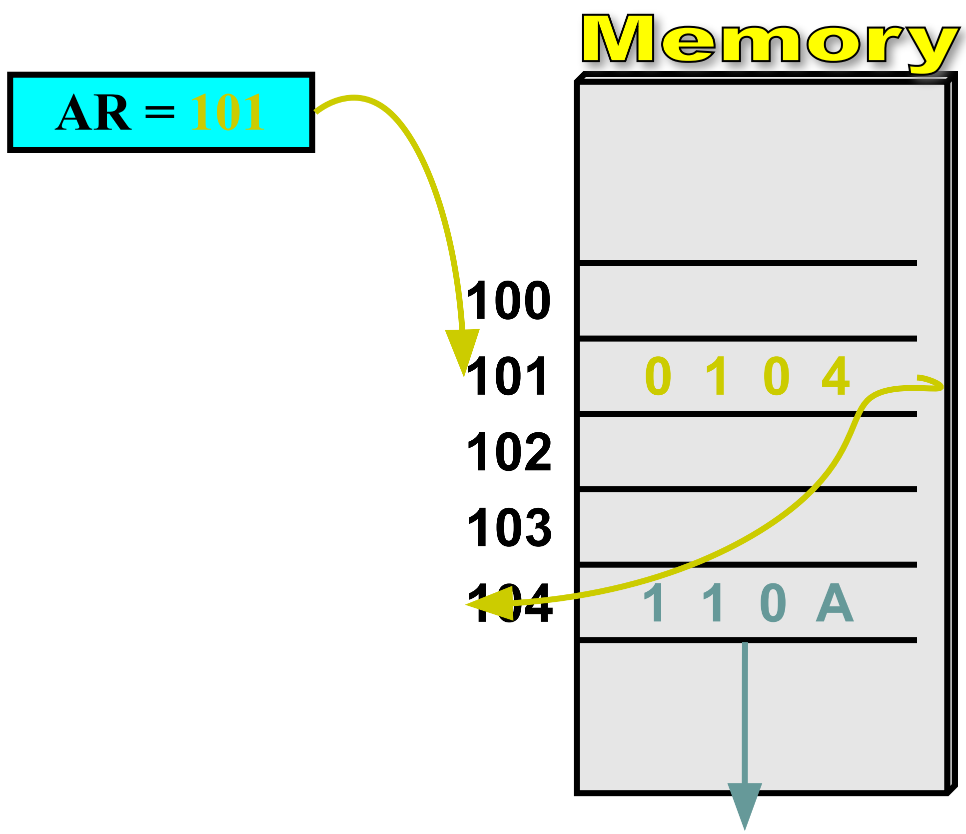
    MOV R1, (R2)



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

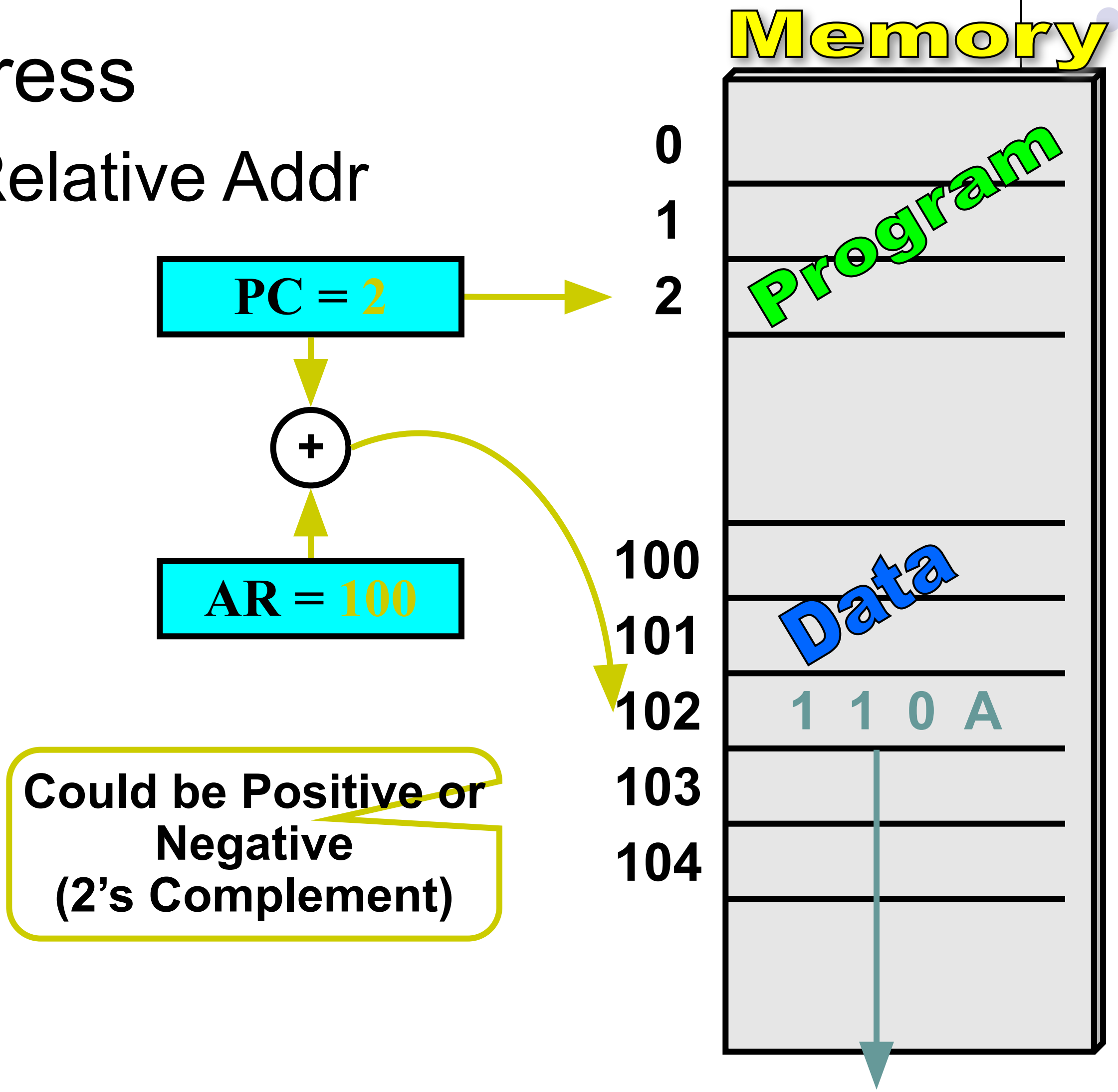
Addressing Modes

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



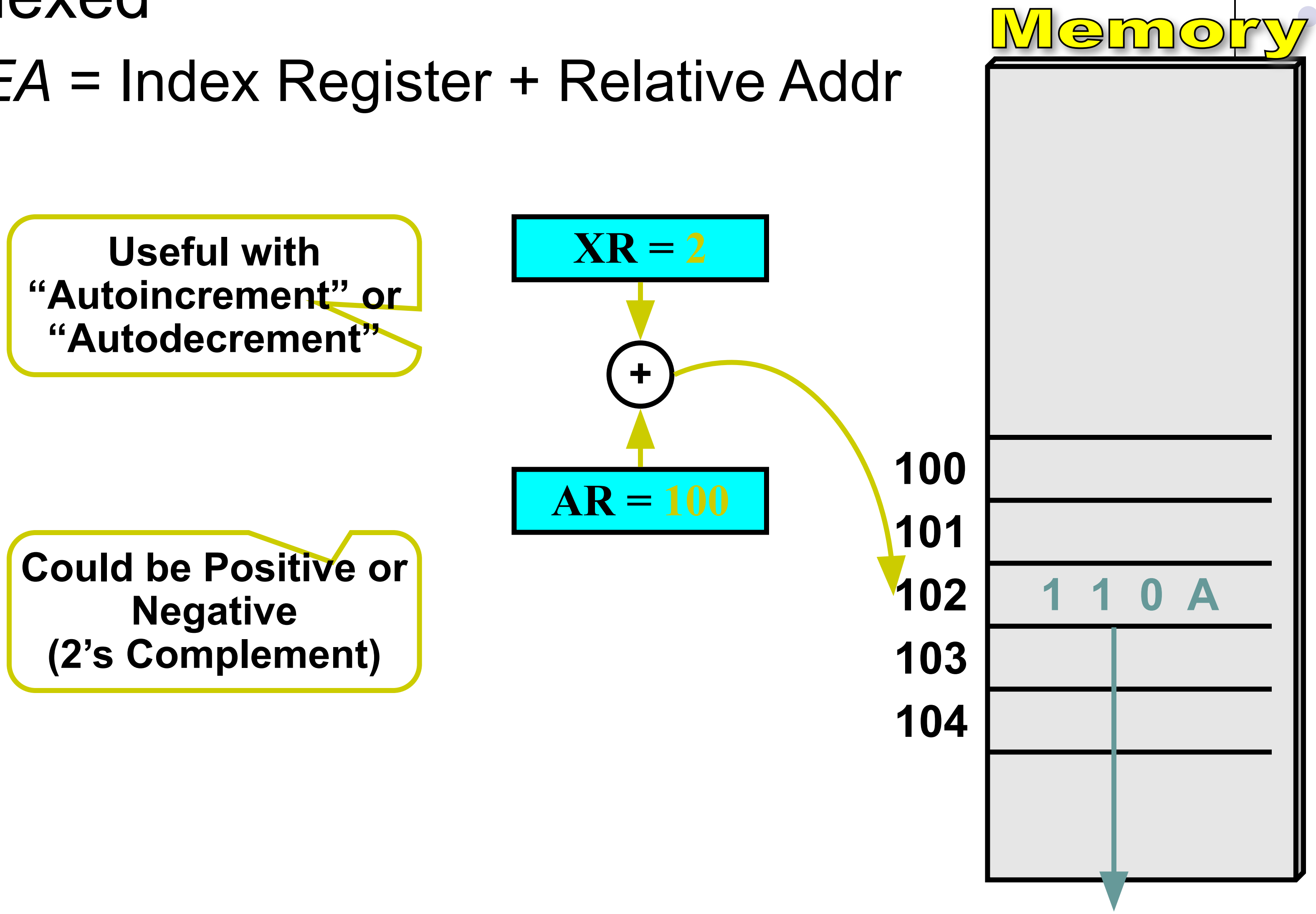
Addressing Modes

- Relative Address
  - EA = PC + Relative Addr



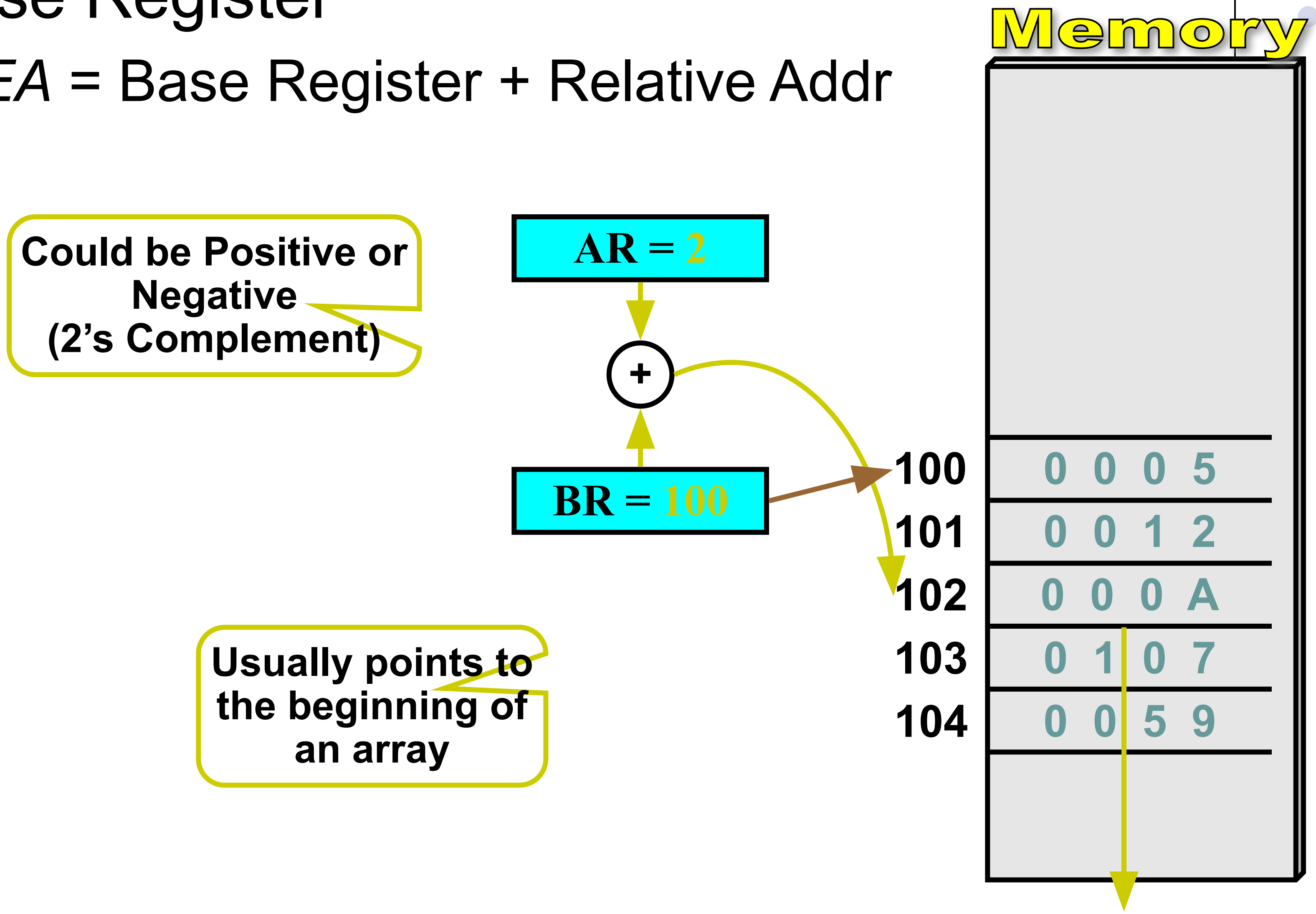
Addressing Modes

- 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	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

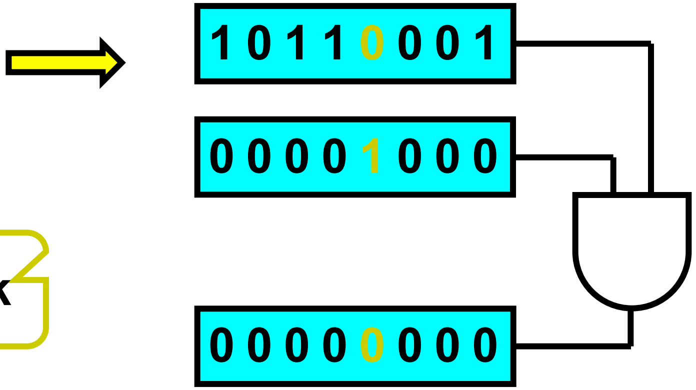
Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

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

Mask



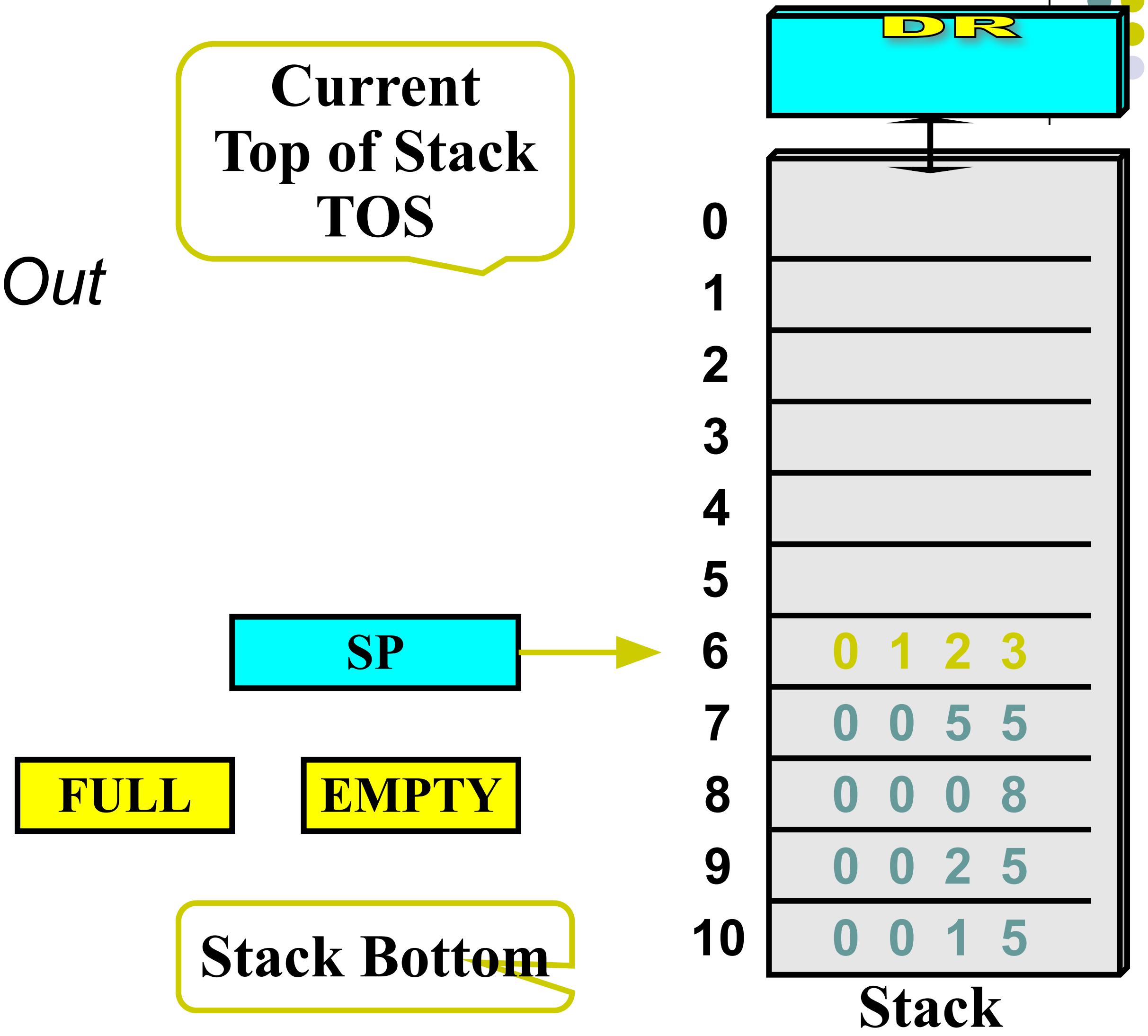
Conditional Branch Instructions

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

Stacks

Stack Organization

- LIFO
  Last In First Out



Stack Organization

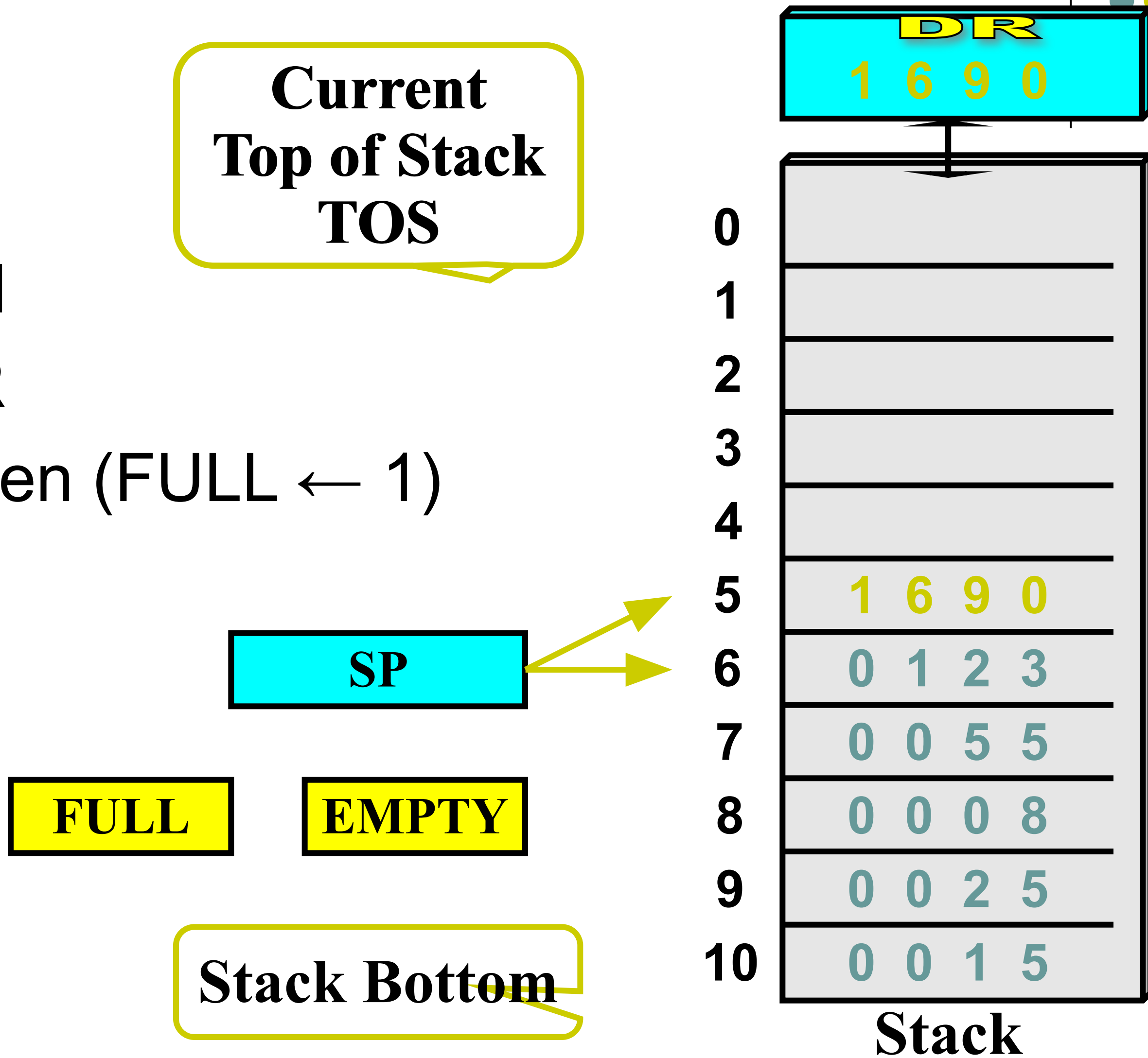
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

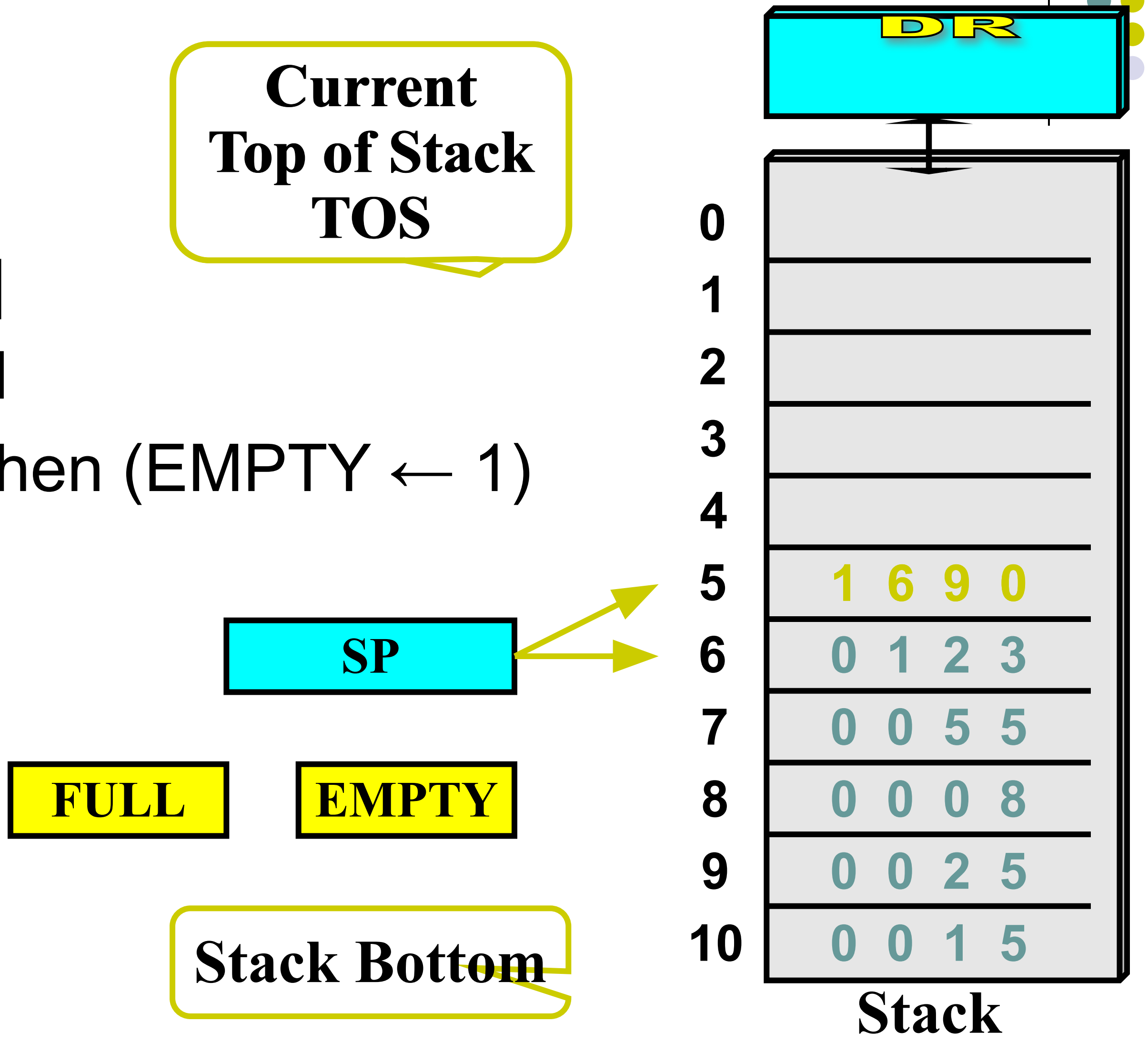
- POP

  DR ← M[SP]

  SP ← SP + 1

  If (SP = 11) then (EMPTY ← 1)

  FULL ← 0



Stack Organization

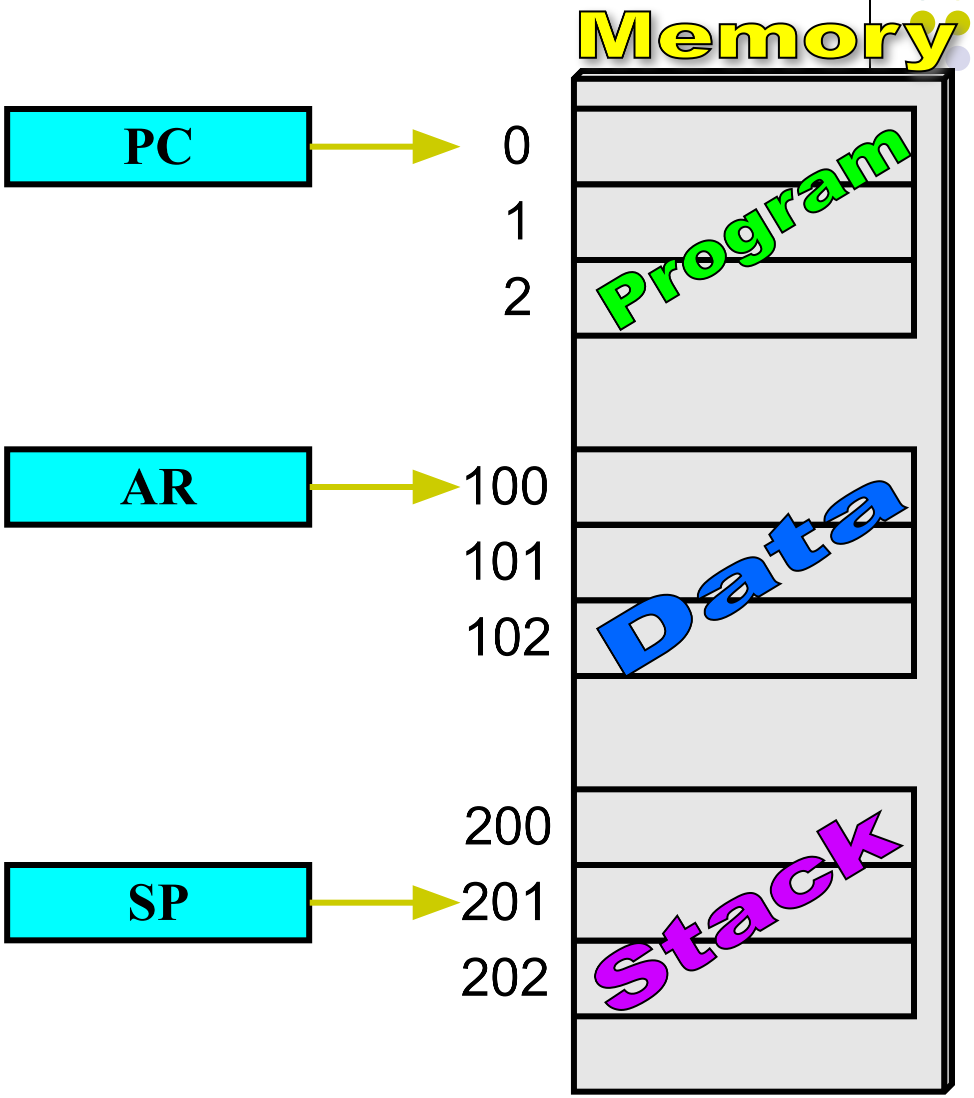
- Memory Stack
  - PUSH

    $SP \leftarrow SP - 1$

    $M[SP] \leftarrow DR$
  - POP

    $DR \leftarrow M[SP]$

    $SP \leftarrow SP + 1$



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

  $(A + B) * [C * (D + E) + F]$

  $(A\ B\ +)\ (D\ E\ +)\ C\ *\ F\ +\ *$

Reverse Polish Notation

- Stack Operation

  (3) (4) * (5) (6) * +

PUSH 3

PUSH 4

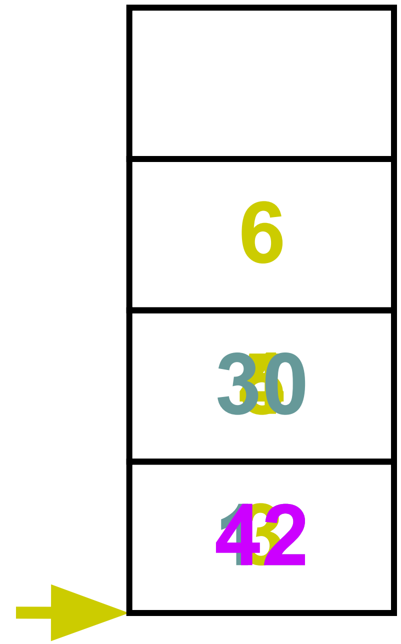
MULT

PUSH 5

PUSH 6

MULT

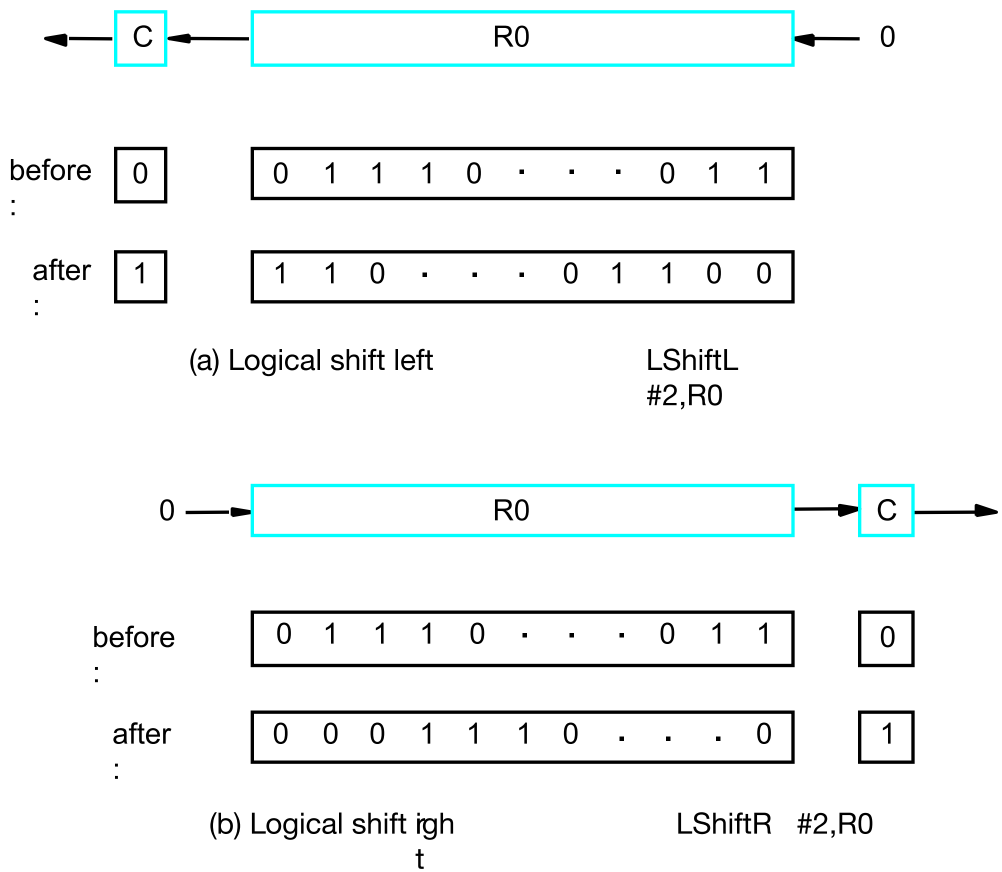
ADD



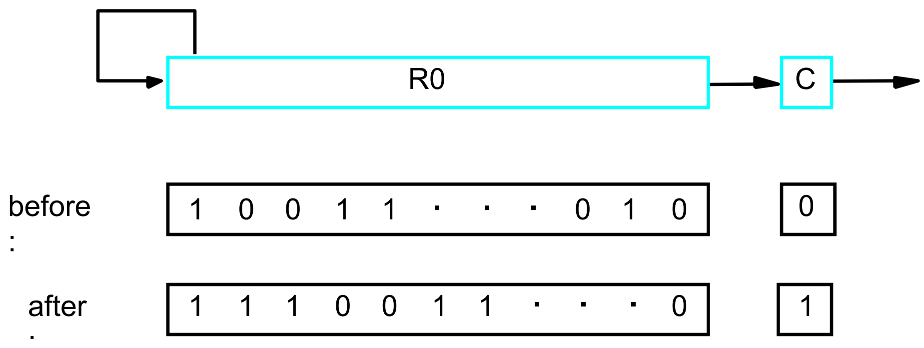
Additional Instructions

Logical Shifts

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



Arithmetic Shifts



(c) Ar ithmetic shift igh
r t

AShiftR #2,R0

Rotate

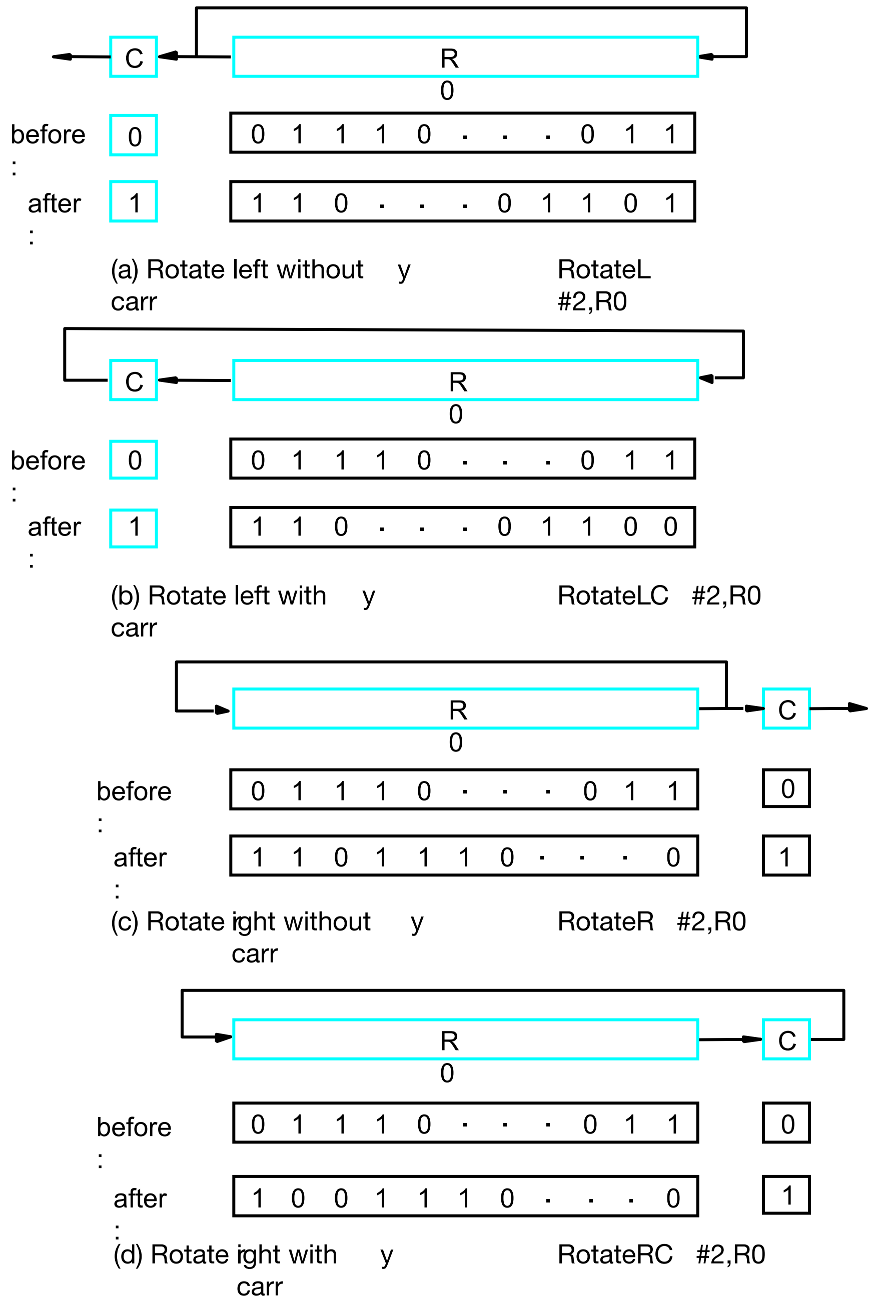


Figure 2.32. Rotate instructions.

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

8	7	7	10
OP code	Source	Dest	Other info

(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

OP code	Source	Dest	Other info
Memory address/Immediate operand			

(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