Chapter 2. Machine Instructions and Programs

Objectives

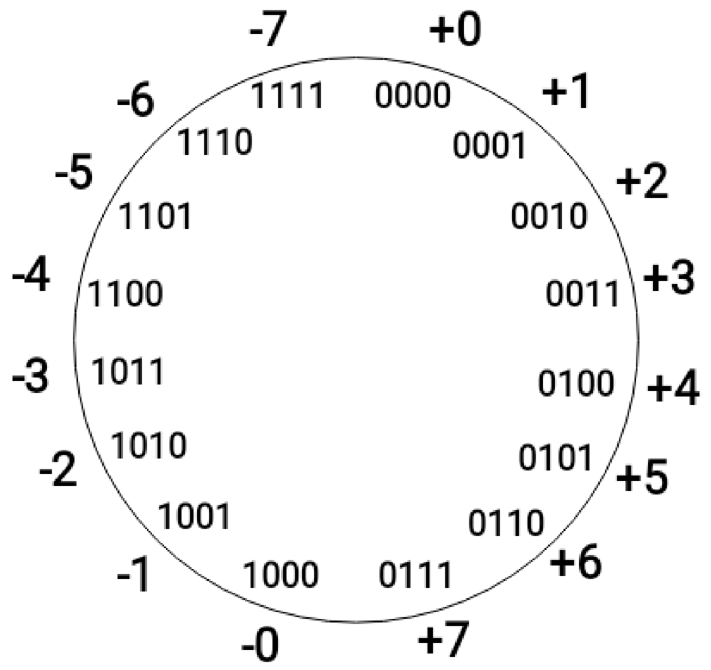
- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2’s-complement system.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions,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

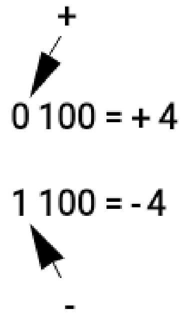
Number, Arithmetic Operations, and Characters

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

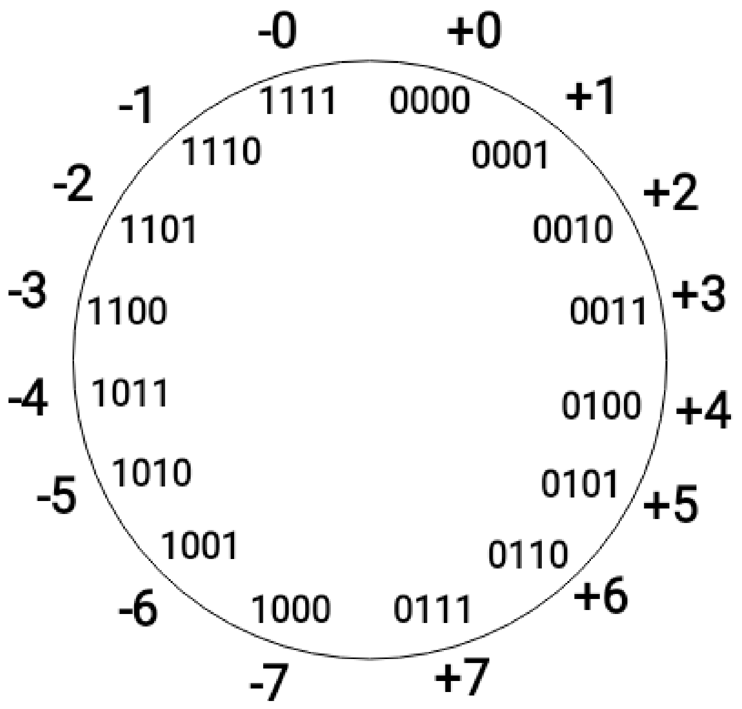
Sign and Magnitude Representation

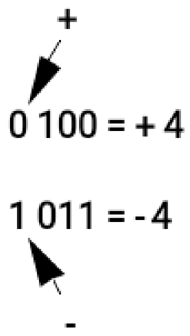




High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Number range for n bits = $+/-2^{n-1}$ -1
Two representations for 0

One's Complement Representation

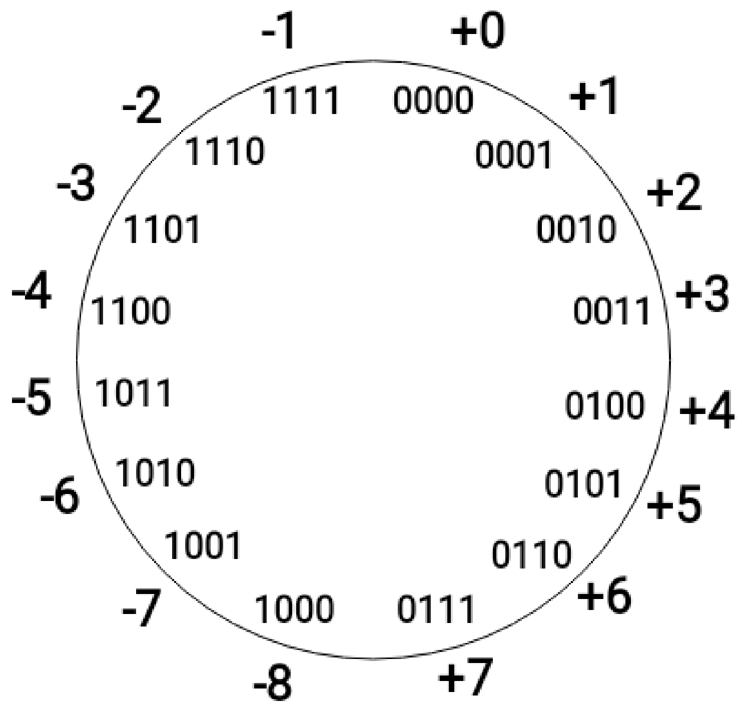


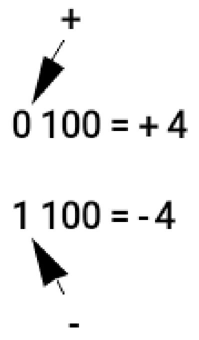


- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems

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      1000           -1      1111
            1
```

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

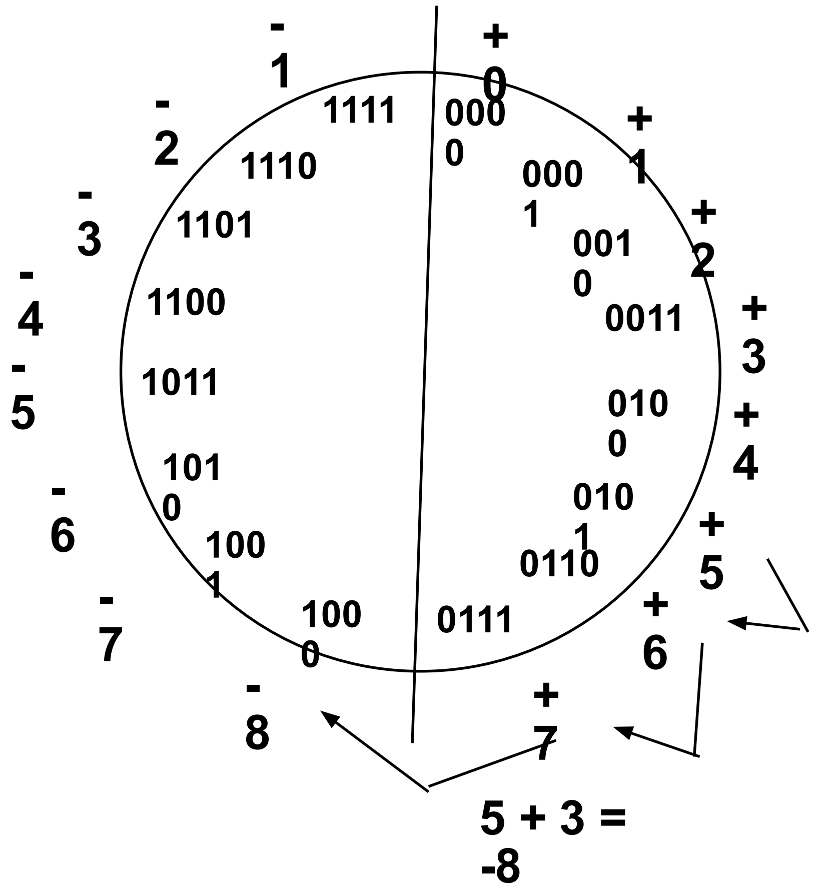
2’s-Complement Add and Subtract Operations

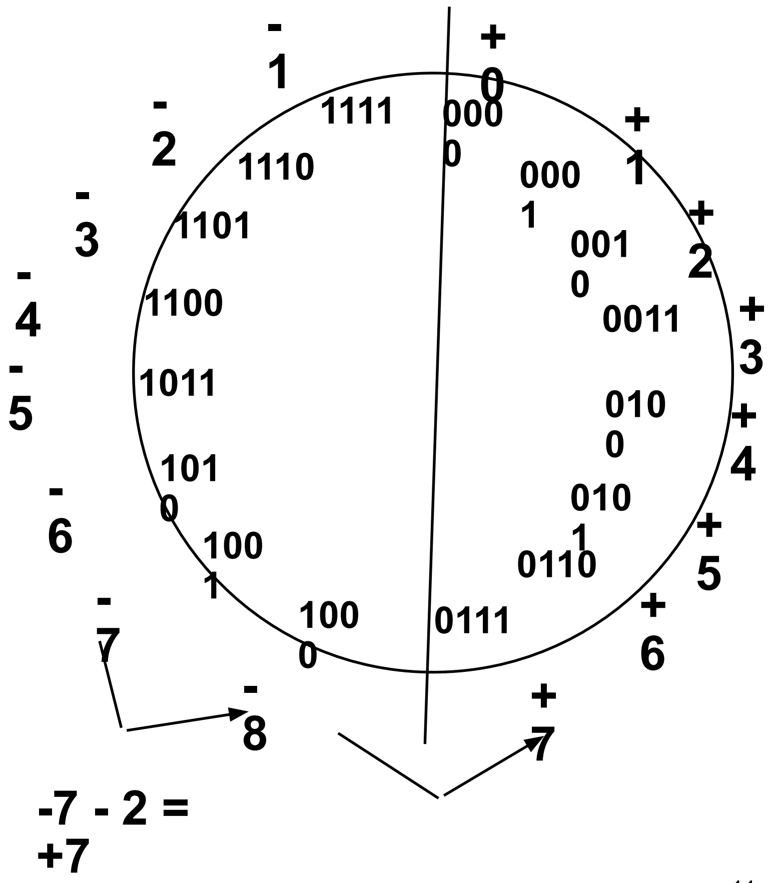
Page 31

```
(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

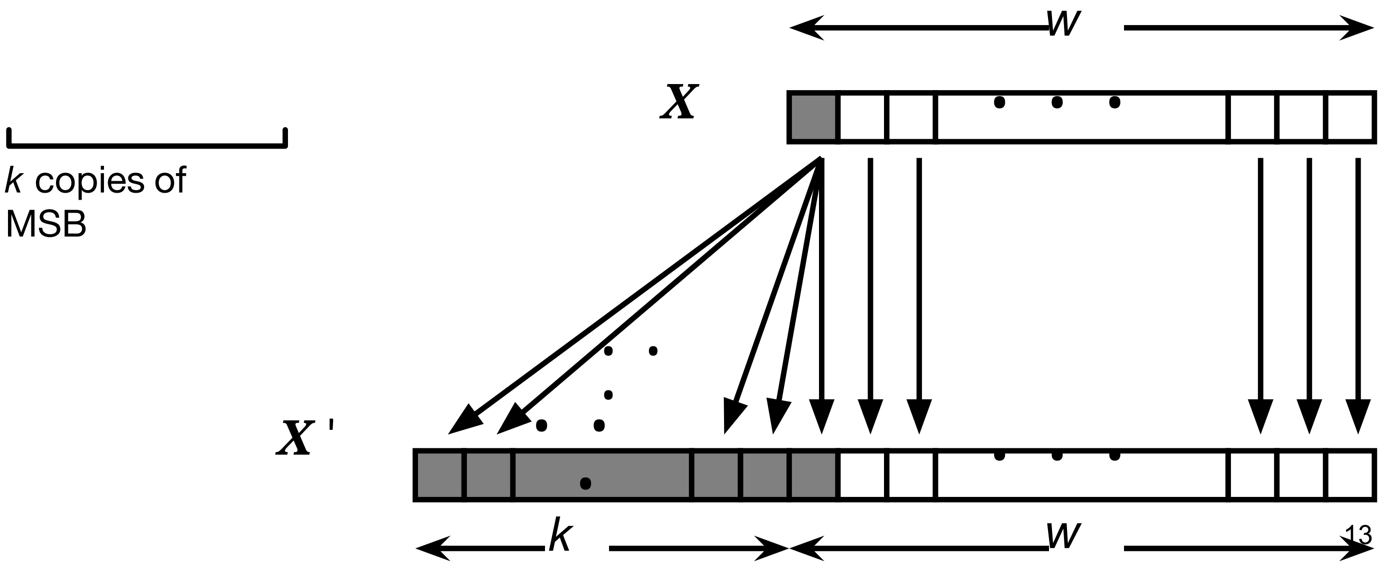
```
                0 1 1 1
  5               0 1 0
                1
  3
 ____           ________
                  0 0 1
  -             1
  8
Overflo           1 0 0
w               0
                0 0 0 0
  5               0 1 0
                1
  2
 ____           ________
                  0 0 1
  7             0

No                0 1 1
overflow        1
```

```
                1 0 0 0
  -               1 0 0
  7             1
 ____           ________
  -               1 1 0
  2             0
                 |________
Overflo         1 0 1 1
w  7            1
                1 1 1 1
  -               1 1 0 1
  3
                  1 0 1 1
 ____           ________
  -
  5              1 1 0 0
                0  |________
No -
8
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.

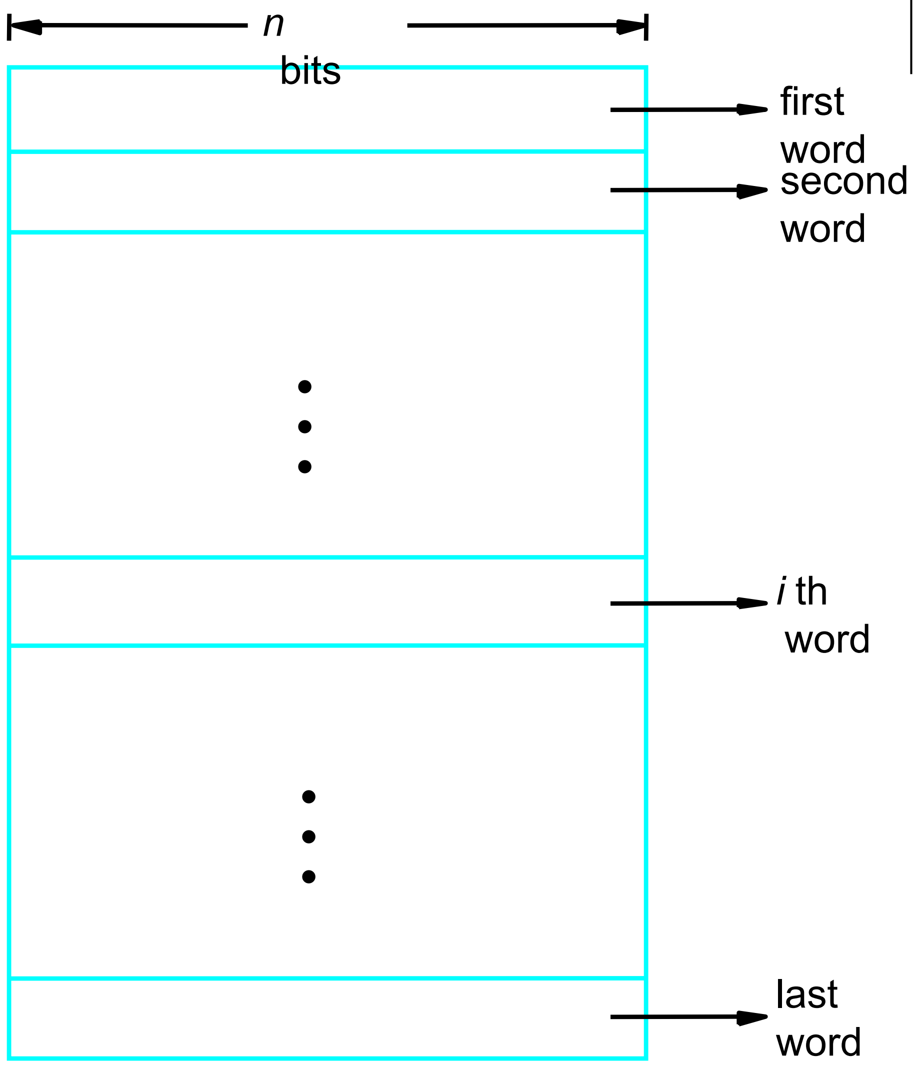
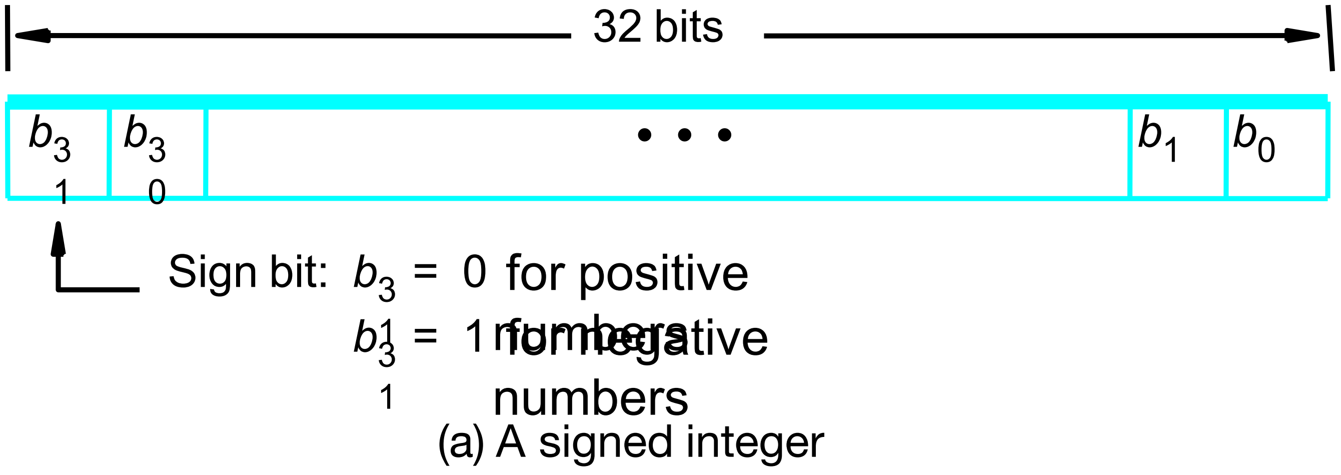


Figure 2.5. Memory words.

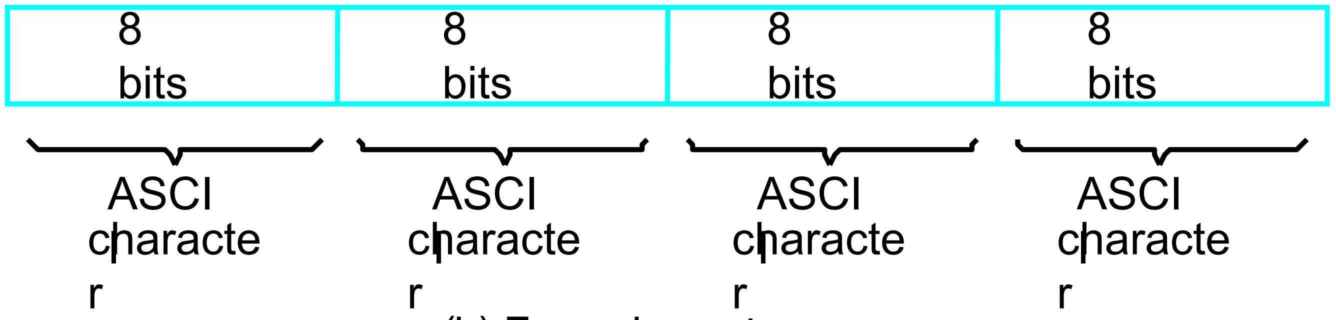
Memory Location, Addresses, and Operation

- 32-bit word length example



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



(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}

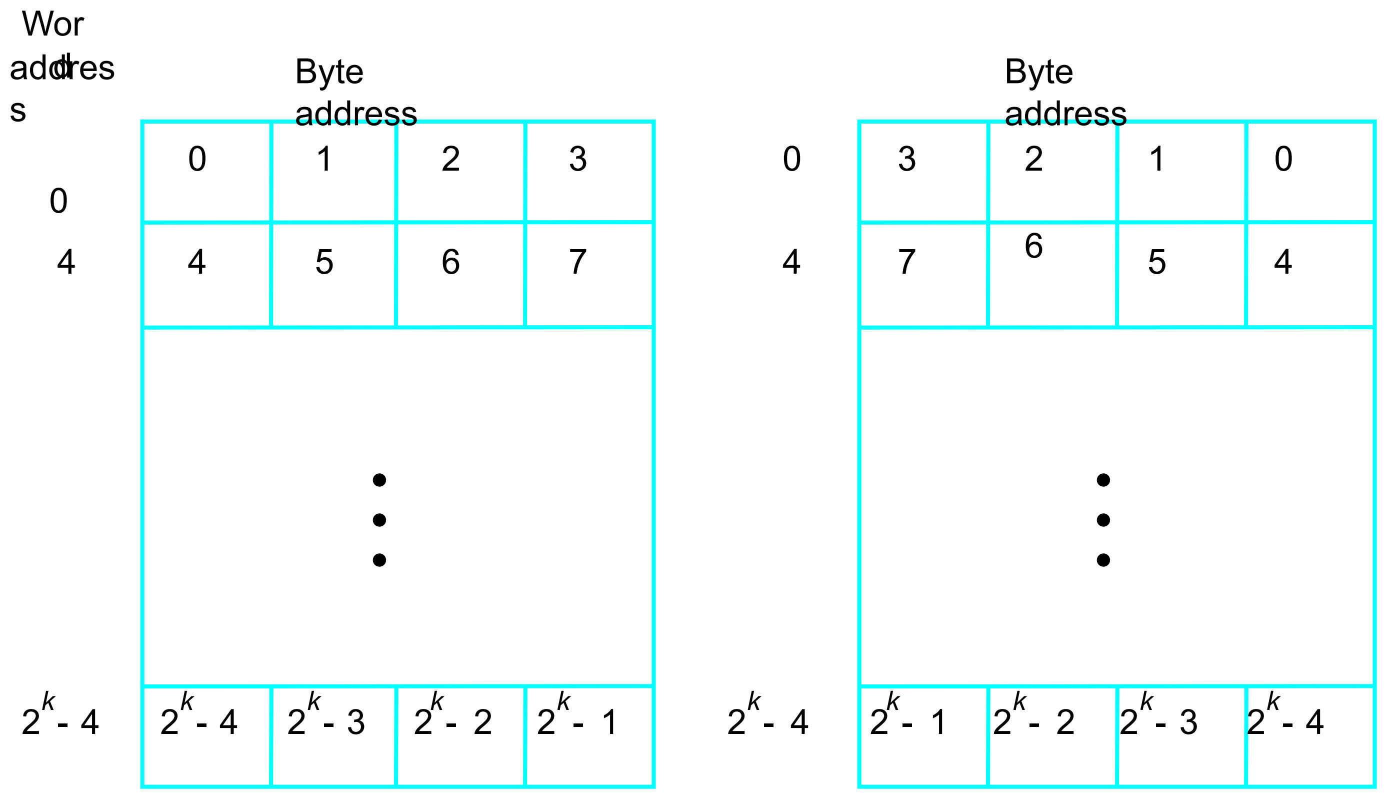
Memory Location, Addresses, and Operation

- 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 ($R1 \leftarrow [LOC]$, $R3 \leftarrow [R1]+[R2]$)
- Register Transfer Notation (RTN)

Assembly Language Notation

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

Basic Instruction Types

- Consider statement

  C = A + B

How in high level language?

the action C ←[A]+[B] takes place in the computer

Three address instruction Add A,B,C

Operation source1,source2,Destination.

This is too large to fit in one word length.

- An alternative

Add A,B,C can be written as

```
Add A,B
Move B, C
```

```
OR    Move B,C
      Add A,C
```

But even two address instruction normally will not fit into usual word length.

A processor register ,called accumulator may be used for this purpose.

```
Load A
Add B
Store C
```

- Early computers were designed around single accumulator structure.
- Modern computers have a number of general purpose registers (8 to 32).
- Access to data in these registers is much faster than memory , as being inside the processor and very few bits(5 bits for 32) are required to specify which register.

Load A, Ri

Store Ri, A

Add A, Ri

Are generalization of the Load, Store and Add instructions for the single accumulator cases, in which Ri performs the function of accumulator.

In modern computer Add Ri, Rj

or

Add Ri,Rj,Rk

Such instructions will normally will fit into one word.

Data transfer between different registers

- Move Source,destination

Move A,Ri   is same as   Load A,Ri

Move Ri,A   is same as   Store Ri,A

DO   C=A+B

operations are allowed only on processor register

operations are allowed one in memory but other in register

CPU Organization

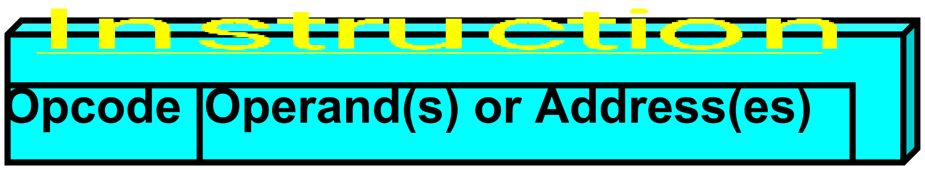
- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats

- Three-Address Instructions
  - ADD R2, R3, R1 $R1 \leftarrow R2 + R3$

  Or ADD R1, R2, R3
- Two-Address Instructions
  - ADD R1, R2 $R1 \leftarrow R1 + R2$

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

Instruction Formats

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

- One-Address
  1. LOADA ; AC $\leftarrow$ M[A]
  2. ADD B ; AC $\leftarrow$ AC + M[B]
  3. STORE T ; M[T] $\leftarrow$ AC
  4. LOADC ; AC $\leftarrow$ M[C]
  5. ADD D ; AC $\leftarrow$ AC + M[D]
  6. MUL 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. LOADB, R2 ; R2 $\leftarrow$ M[B]
  3. LOADC, 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

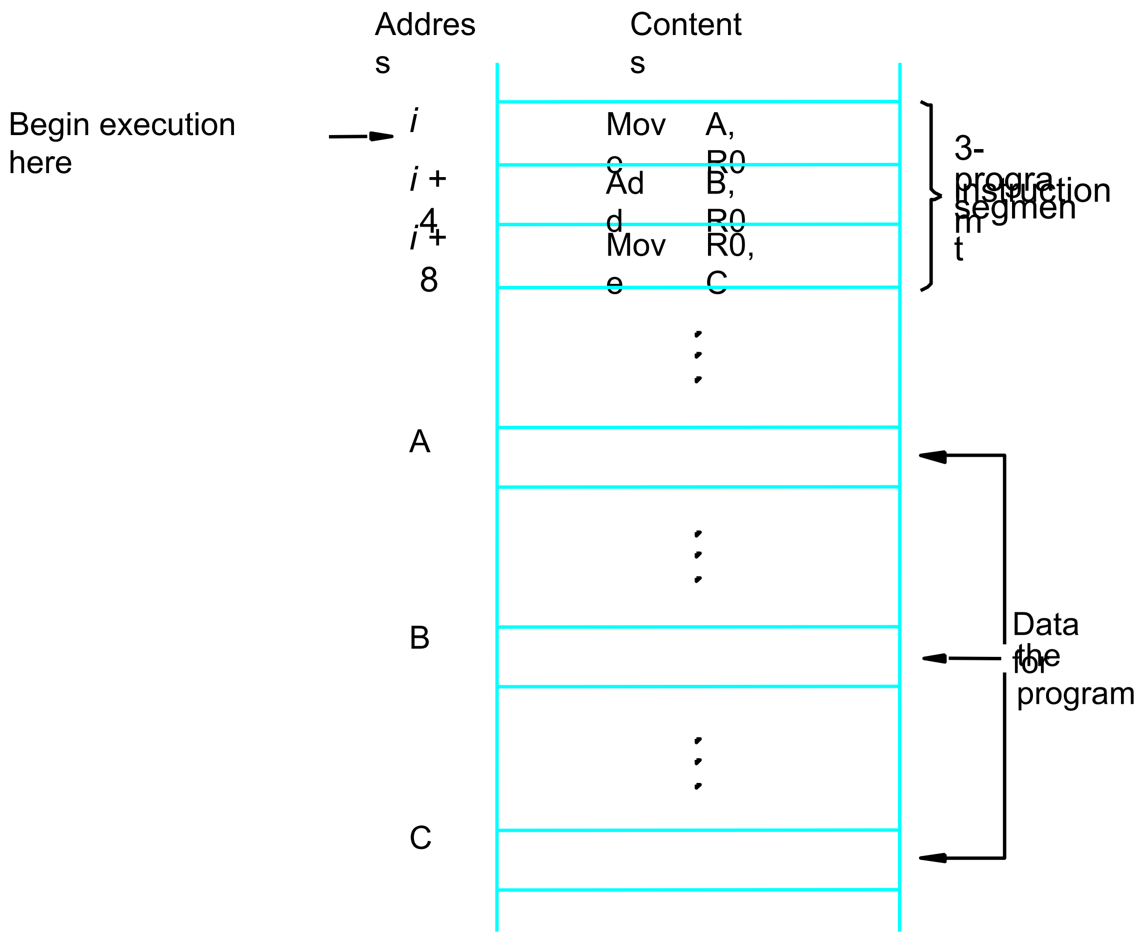


Figure 2.8. A program for C ← [A] + [B].

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

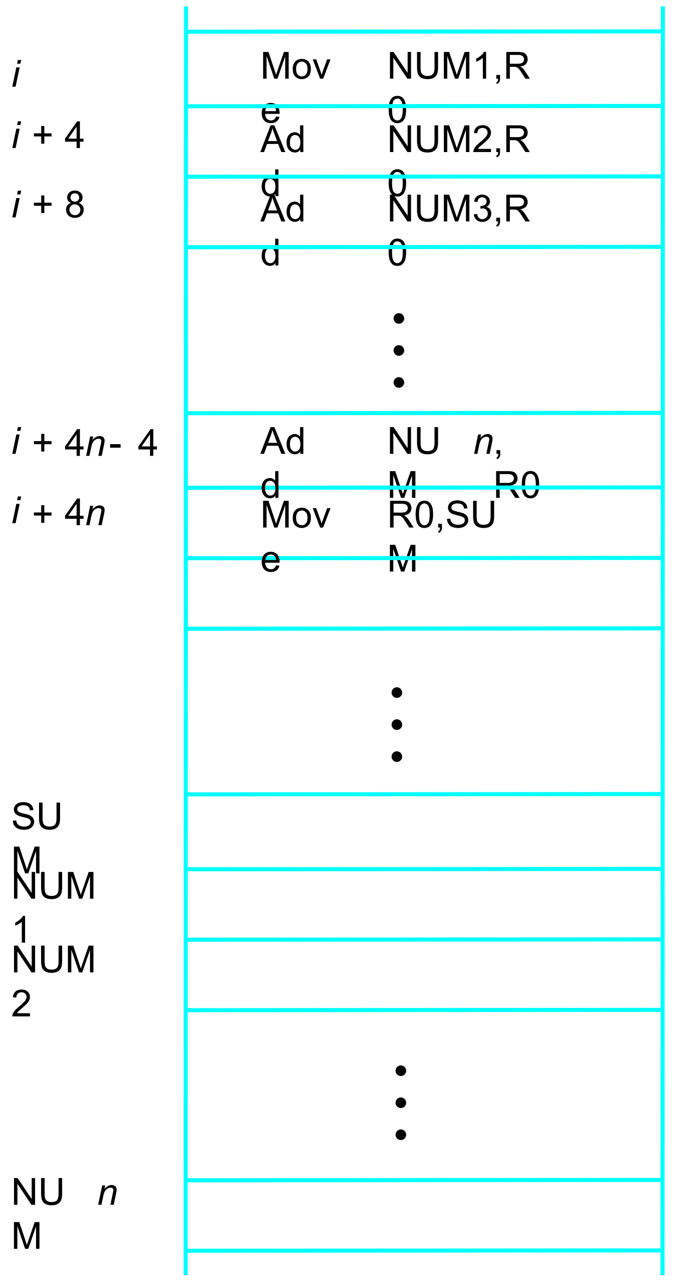


Figure 2.9. A straight-line program for adding *n* numbers

Branching

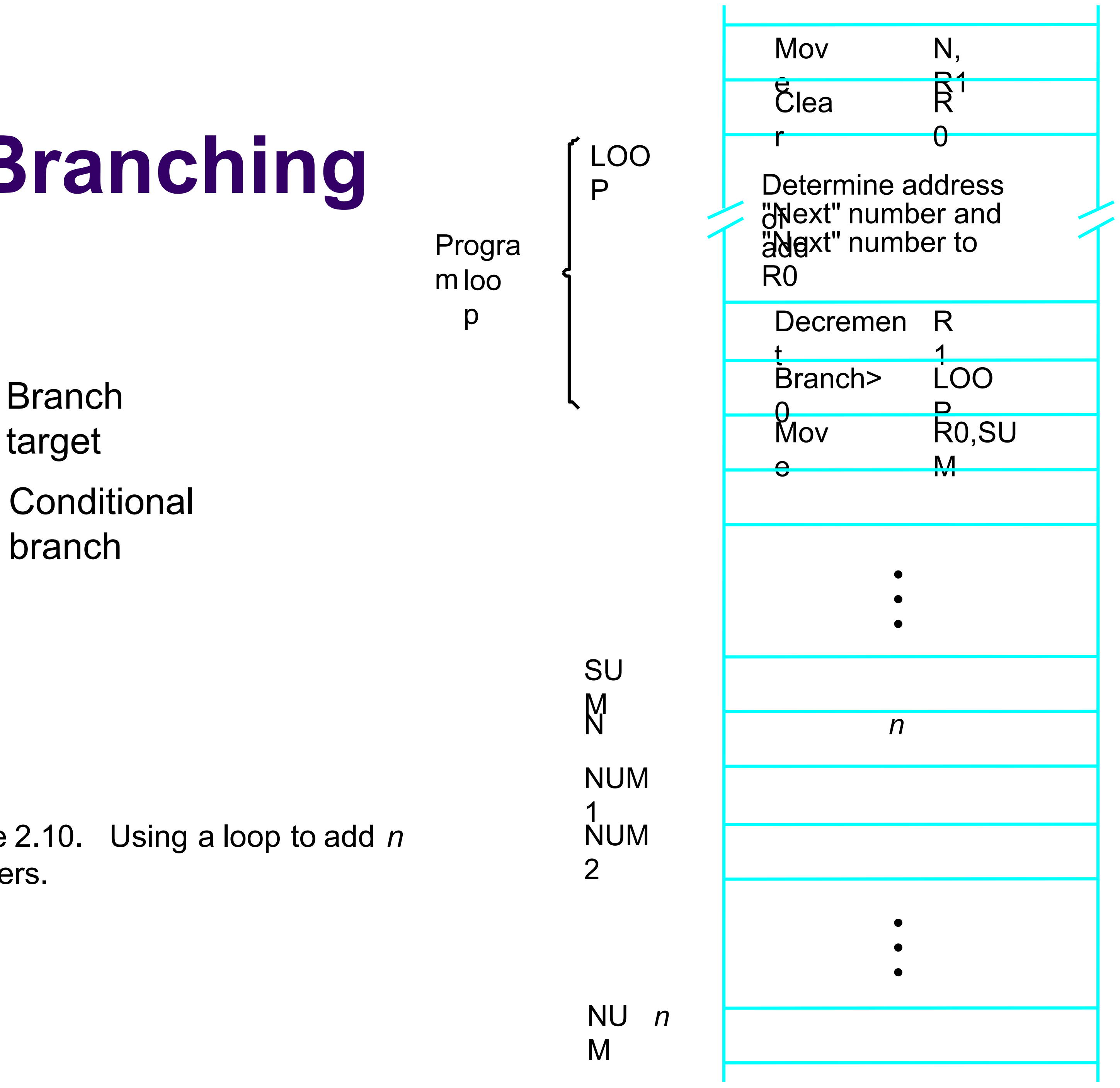


Figure 2.10. Using a loop to add *n* numbers.

Condition Codes

- Condition code flags
- Condition code register / status register
- N (negative)
- Z (zero)
- V (overflow)
- C (carry)
- Different instructions affect different flag
- Branch > 0 condition to be tested based on logical expression.

Conditional Branch Instructions

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

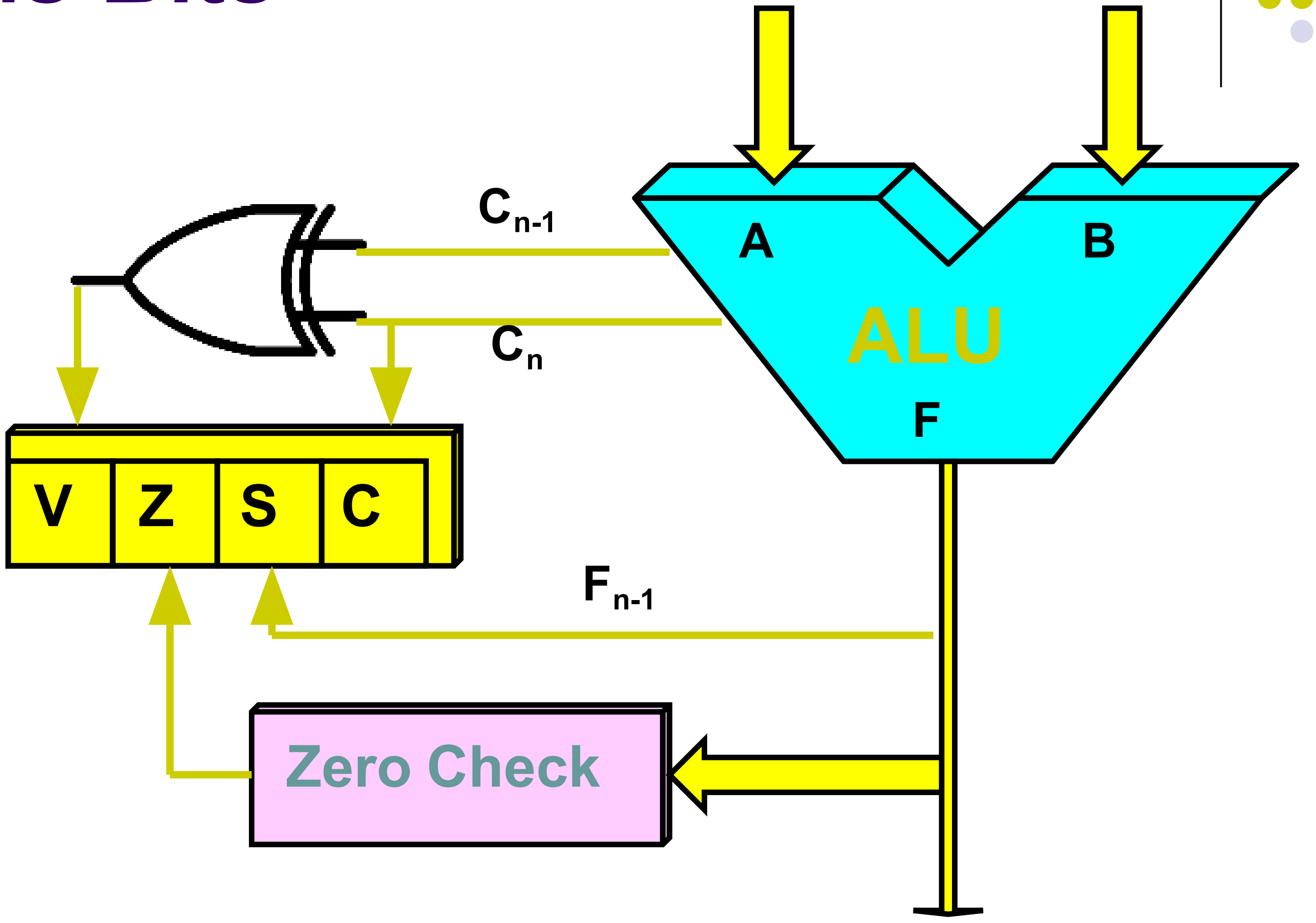
$$\begin{array}{lr} \text{A:} & 1\,1\,1\,1\,0\,0\,0\,0 \\ +(-\text{B}): & 1\,1\,1\,0\,1\,1\,0\,0 \\ \hline & 1\,\underbrace{1\,0\,1\,1\,1\,0\,0}_{\text{Z} = 0} \end{array}$$

C = 1

S = 1

V = 0

Status Bits

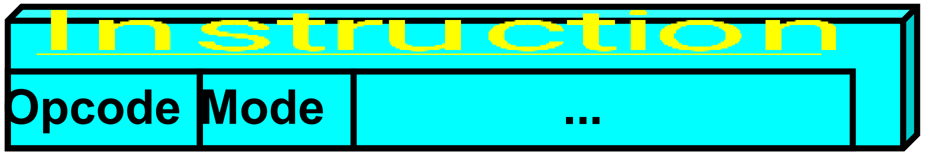


Addressing Modes

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

- Programmers use organizations called data structures to represent data (lists, queues, linkedlists,arrays, …….)
- When translating high level Language to assembly language ,compiler must be able to implement these constructs using the facilities provided in the instruction set.
- Different ways in which the location of an operand is specified in an instruction referred to as add. mode

Addressing Modes

Implementation of Variables and Constants

Register Mode: Operand is content of register

Absolute Mode: Operand is content of Memory location. Also called as **Direct A Mode**

Move LOC,R2 uses these two modes.

Absolute mode can represent global variables in a program.

Immediate Mode: Move $200_{\text{immediate}}$,R0

Move #200,R0

Addressing Modes

High level language statement

A=B+6

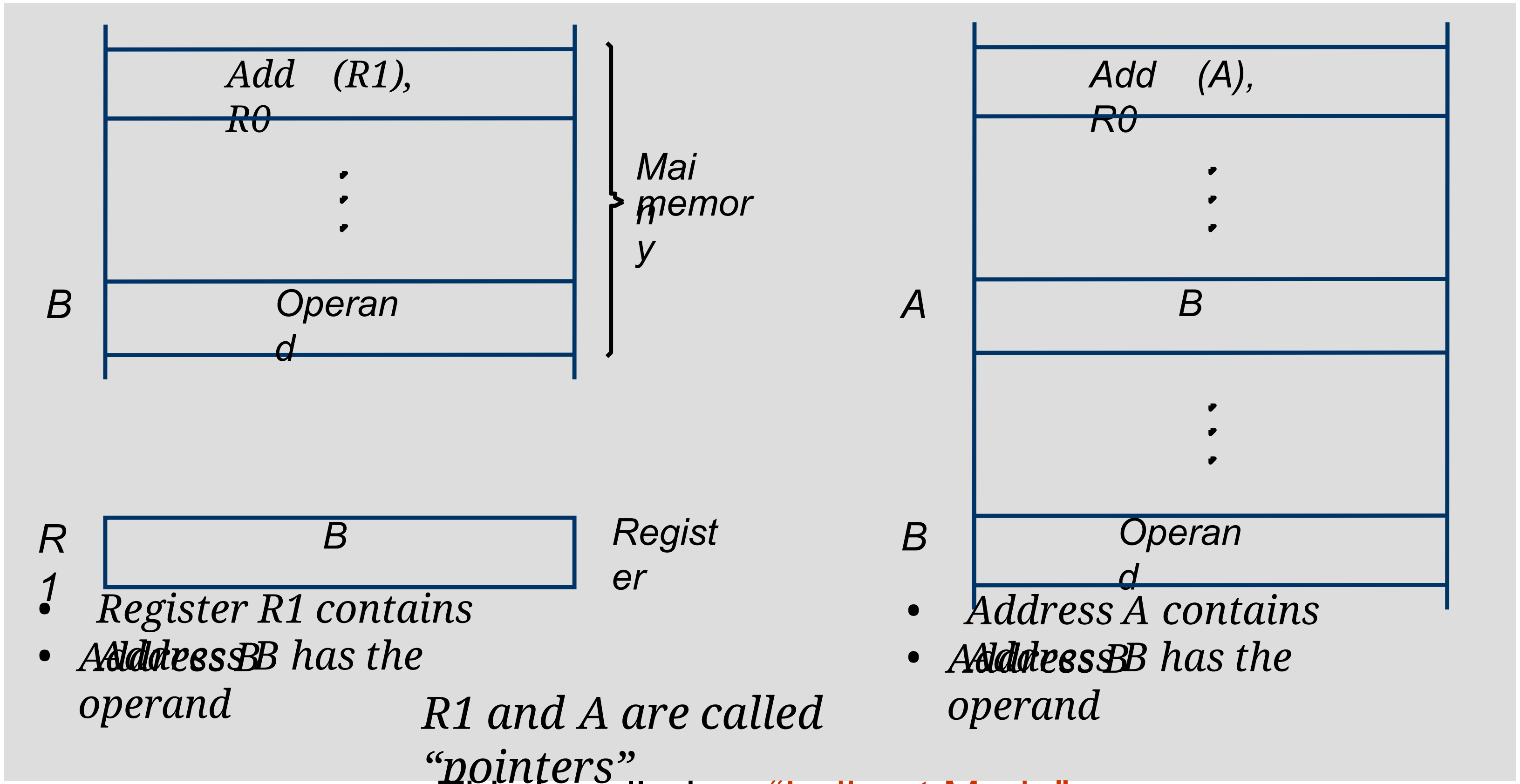
```
Move B,R1
Add   #6,R1
Move R1,A
```

Indirection And pointers

- Register, Absolute and Immediate modes contained either the address of the operand or the operand itself.
- Some instructions provide information from which the memory address of the operand can be determined
  - That is, they provide the "Effective Address" of the operand.
  - They do not provide the operand or the address of the operand explicitly.
- Different ways in which "Effective Address" of the operand can be generated.

Addressing modes (contd..)

Effective Address of the operand is the contents of a register or a memory location whose address appears in the instruction.



- Register R1 contains Address B
- Address B has the operand

- Address A contains Address B
- Address B has the operand

R1 and A are called "pointers"

This is called as "Indirect Mode"

Addressing modes (contd..)

The register or memory location that contains the address of an operand is called a Pointer.

Analogy of Hunt : By changing the note , location of the treasure can be changed , but instruction for hunt remains the same.

By changing the content of R1 or location A , add instruction fetches different operands to add to register R0.

Addressing modes (contd..)

Use of indirect addressing in the program

```
        Move            N,R1
        Move            #Num1,R2
        Clear           R0
Loop:   Add             (R2),R0
        Add             #4,R2
        Decrement       R1
        Branch>0        Loop
        Move            R0,SUM
```

Addressing modes (contd..)

Consider C-Language statement

```
   A  =  *B;
```

B is pointer variable, This statement may be compiled into

```
Move   B,R1
Move   (R1),A
```

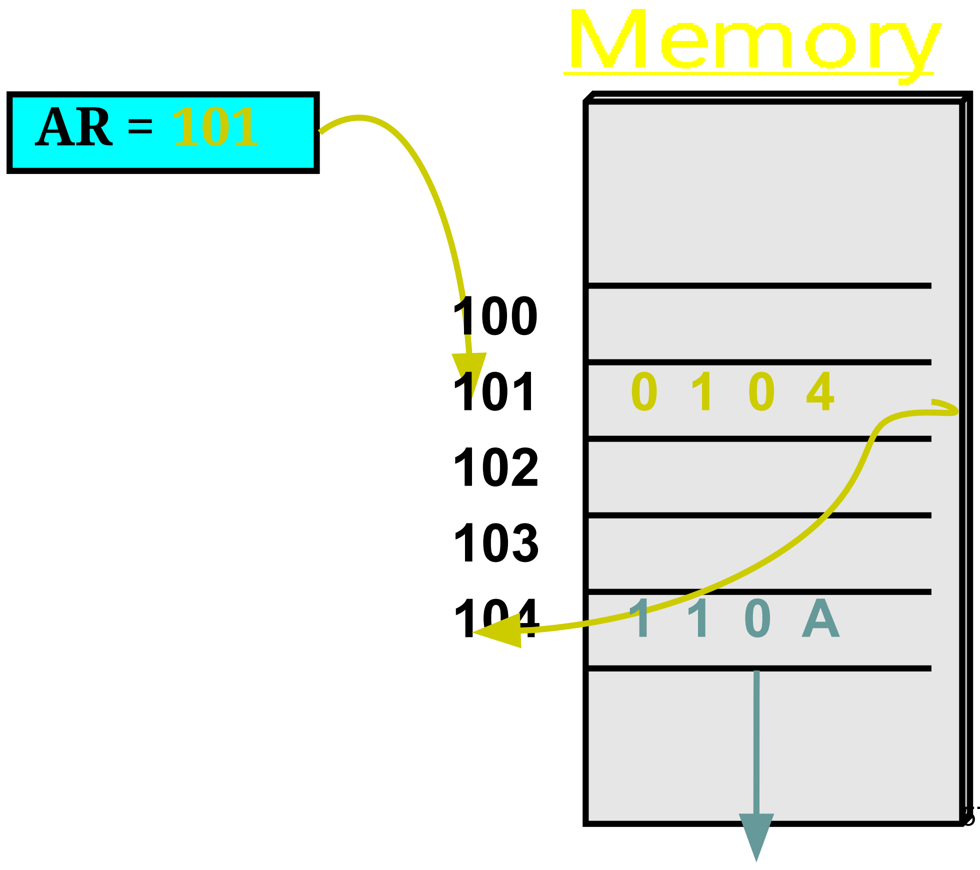
Using Indirect addressing through memory,

```
Move   (B),A
```

Proven to be limitation and hence seldom found modern computers.

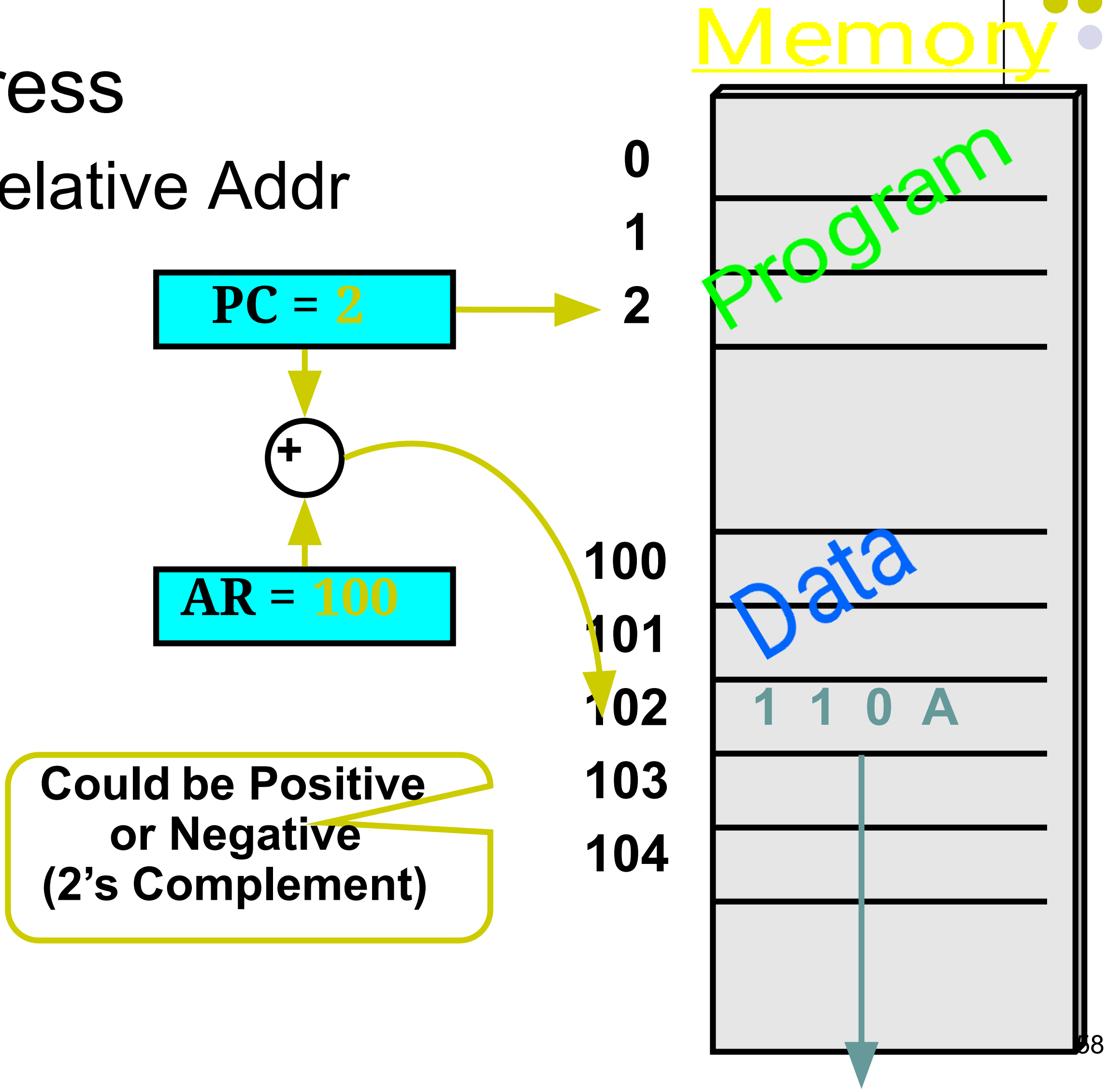
Addressing Modes

- 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

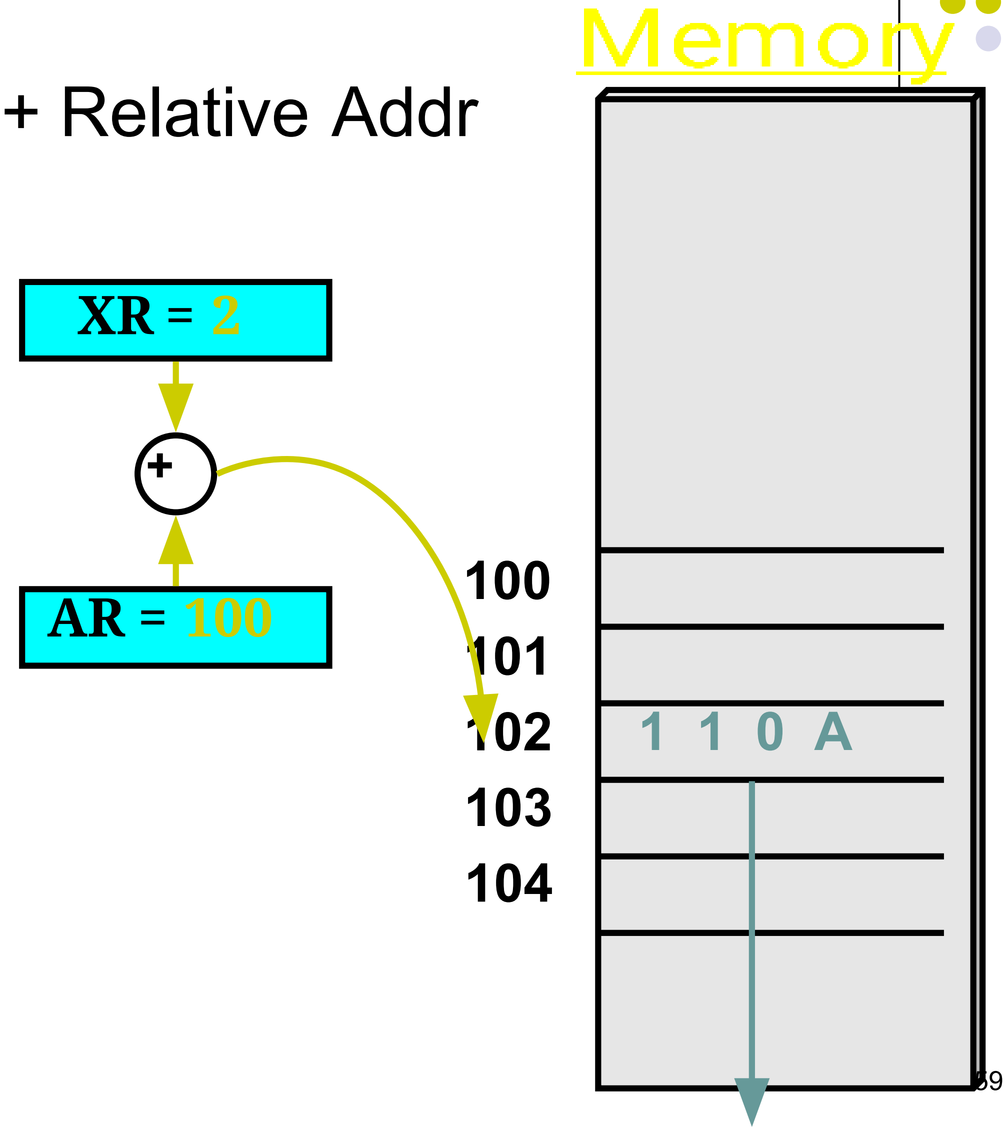


Addressing Modes

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

Useful with "Autoincrement" or "Autodecrement"

Could be Positive or Negative (2's Complement)



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*)	EA = [R*i*]
	(LOC)	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

Assembly Language

- Machine instructions are represented by patterns of 0s and 1s. Awkward to prepare prg
- Use symbolic names such as Move, Add, Increment ,Branch etc.(Such words are normally called mnemonics).
- similarly use R3 to refer to register 3 and LOC to refer memory location.
- Complete set of such symbolic names and rules for their use constitute programming language called assembly language

Assembly Language

- Program written in assembly language can automatically translated into a sequence of machine instructions by a program called assembler.
- Assembler like any other program is stored in as a sequence of m/c instruction in comp. Memory
- When assembler prg is executed ,it reads the user program, analyze it and then generates desired machine language program.
- Later content of 0s and 1s will be executed

Assembly Language

- User Prg in alphanumeric text format is called **source program** and assembled machine language program is called an **object Prg**.
- Assembly language may or may not be case sensitive,
- MOVE R0,SUM
- MOVE opcode will be converted in binary opcode that the computer understands. one blank space source, destination.
- Assembly language must indicate which mode is being used.

Assembly Language

- ADD #5,R3
- ADDI 5,R3
- MOVE #5,(R2)

  OR

  MOVEI 5,(R2)

Assembly language

- Recall that information is stored in a computer in a binary form, in a patterns of 0s and 1s.
- Such patterns are awkward when preparing programs.
- Symbolic names are used to represent patterns.
  - So far we have used normal words such as *Move, Add, Branch,* to represent corresponding binary patterns.
- When we write programs for a specific computer, the normal words need to be replaced by acronyms called mnemonics.
  - E.g., *MOV, ADD, INC*
- A complete set of symbolic names and rules for their use constitute a programming language, referred to as the assembly language.

Assembly language (contd..)

- Programs written in assembly language need to be translated into a form understandable by the computer, namely, binary, or machine language form.
- Translation from assembly language to machine language is performed by an <u>assembler</u>.
  - Original program in assembly language is called source program.
  - Assembled machine language program is called object program.
- Each mnemonic represents the binary pattern, or <u>*OP code*</u> for the operation performed by the instruction.
- Assembly language must also have a way to indicate the addressing mode being used for operand addresses.
- Sometimes the addressing mode is indicated in the OP code mnemonic.
  - E.g., *ADDI* may be a mnemonic to indicate an addition operation with an immediate operand.

Assembly language (contd..)

- Assembly language allows programmer to specify other information necessary to translate the source program into an object program.
  - How to assign numerical values to the names.
  - Where to place instructions in the memory.
  - Where to place data operands in the memory.
- The statements which provide additional information to the assembler to translate source program into an object program are called <u>assembler directives or commands.</u>

Assembly language (contd..)

	100	Move	N, R1
	104	Move	#NUM1,R2
	108	Clear	R0
LOOP	112	Add	(R2), R0
	116	Add	#4,R2
	120	Decrement	R1
	124	Branch>0	LOOP
	128	Move	R0,SUM
	132		
		⋮	
SUM	200		
N	204	100	
NUM1	208		
NUM2	212		
		⋮	
NUM100	604		

- *What is the numeric value assigned to SUM?*
- *What is the address of the data NUM1 through NUM100?*
- *What is the address of the memory location represented by the label LOOP?*
- *How to place a data value into a memory location?*

Assembly language (contd..)

	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
		BGT	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

EQU:
- Value of SUM is 200.

ORIGIN:
- Place the datablock at 204.

DATAWORD:
- Place the value 100 at 204
- Assign it label N.
- N EQU 100

RESERVE:
- Memory block of 400 words is to be reserved for data.
- Associate NUM1 with address 208

ORIGIN:
- Instructions of the object program to be loaded in memory starting at 100.

RETURN:
- Terminate program execution.

Assembly language (contd..)

- Assembly language instructions have a generic form:

  Label Operation Operand(s) Comment

- Four fields are separated by a delimiter, typically one or more blank characters.
- Label is optionally associated with a memory address:
  - May indicate the address of an instruction to be executed.
  - May indicate the address of a data item.
- How does the assembler determine the values that represent names?
  - Value of a name may be specified by EQU directive.
    - SUM EQU 100
  - A name may be defined in the Label field of another instruction, value represented by the name is determined by the location of that instruction in the object program.
    - E.g., BGTZ LOOP, the value of LOOP is the address of the instruction ADD (R2) R0

Assembly language (contd..)

- Assembler scans through the source program, keeps track of all the names and corresponding numerical values in a "symbol table".
    - When a name appears second time, it is replaced with its value from the table.
- What if a name appears before it is given a value, for example, branch to an address that hasn't been seen yet (forward branch)?
    - Assembler can scan through the source code twice.
    - First pass to build the symbol table.
    - Second pass to substitute names with numerical values.
    - Two pass assembler.

Assembly directive

- The assembler stores the object program on a magnetic disk. The object program must be loaded into the memory of the computer before it is executed. For this to happen, another utility program called a *loader must already be in the memory.*
- When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can detect and report syntax errors. To help the user find other programming errors, the system software usually includes a *debugger program.*

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

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 reads a line of character and display

- Move #LOC,R0 ; Initialize pointer register R0 to point to the address of the first location in memory where the characters are to be stored.
- READ TestBit #3,INSTATUS ; Wait for a character to be entered
- Branch=0 READ ;in the keyboard buffer DATAIN.
- MoveByte DATAIN,(R0) ;Transfer the character from DATAIN into the memory (this clears ;SINto 0).
- ECHO TestBit #3,OUTSTATUS ; Wait for the display to become ready.
- Branch=0 ECHO
- MoveByte (R0),DATAOUT ; Move the character just read to the display buffer register (this ;clears SOUT to 0).
- Compare #CR,(R0)+ ; Check if the character just read is CR (carriage return). If it is ;not CR, then
- Branch=0 READ ; branch back and read another character.

  ; Also, increment the pointer to store the next character.

Figure 2.20

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