Lecture 2

Instruction Sequencing

Decimal and Binary

- Decimal $8547 = 8*10^3 + 5*10^2 + 4*10^1 + 7*10^0$

- Common Form $D = d_{n-1}d_{n-2} \ldots\ldots d_1 d_0$

  $V(D) = d_{n-1}*10^{n-1} + d_{n-2}*10^{n-2} \ldots d_0*10^0$

- Binary $B = b_{n-1}b_{n-2} \ldots\ldots b_1 b_0$

  $V(B) = b_{n-1}*2^{n-1} + b_{n-2}*2^{n-2} \ldots b_0*2^0$

Based Convert

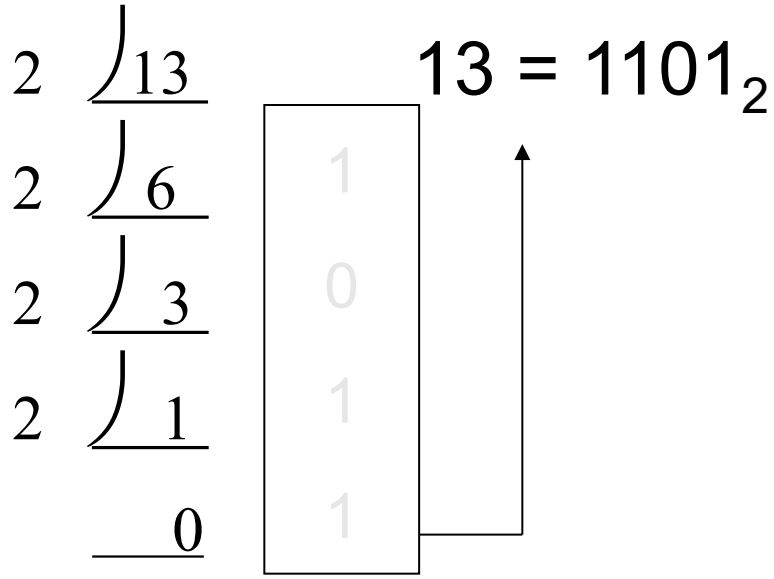
- Binary to Decimal

$(1101)_2 = 1*2^3 + 1*2^2 + 0*2^2 + 1*2^0$

$= 8 + 4 + 1$

$= 13$

- Decimal to Binary



Octal

- Digits range from 0 to 7
- A single octal digit can be represented using 3 bits.
- A binary number can be converted to an octal number by taking a group of 3 bits, starting from the LSB (least significant bit)

- EX. $(101011010111)_2$ = $\underline{101}\ \underline{011}\ \underline{010}\ \underline{111}$
  = $(5327)_8$

Hexadecimal

ใช้ระบุ address หรือ data ของ instruction ทั้งหมด

- Digits range from 0-9, A-F
- A single hexadecimal digit can be represented using 4 bits

- Ex. $(AF25)_{16}$ = $(\underline{1010}\ \underline{1111}\ \underline{0010}\ \underline{0101})_2$

เป็นที่นิยมสำหรับงานระดับวงจรคอมพิวเตอร์

- Octal and Hexadecimal numbers are used as shorthand notations for binary numbers.

Number Representations

- Sign-and-magnitude
  - Similar to decimal numbering format.
  - An extra bit is used as a sign.
  - MSB = 0 : number is positive and MSB = 1 : number is negative
  - Ex. 0101 = +5 , 1101 = -5
- 1’s-complement
  - The negative number can be obtained by complementing each bit of the number including the sign bit (MSB)
  - Ex. 0101 = +5 , -5 = 1010
- 2’s-complement
  - The negative number can be obtained by adding 1 to a 1’s complement representation.
  - Ex. 0101 = +5 , -5 = 1011

Number Representations

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

Addition of Sign-and-magnitude

- Although the simplest representation, adding two numbers of opposite signs can be complicated. We need to find a smaller number to be subtracted from the larger one
- The extra logic circuit will be needed to compare numbers.
- This representation is not used in computer systems nowadays

Addition of 1's-complement

- In some cases, a correction is needed, which means an extra addition will have to be performed.
- The correction is done by adding the carry bit to the result.

- Ex.

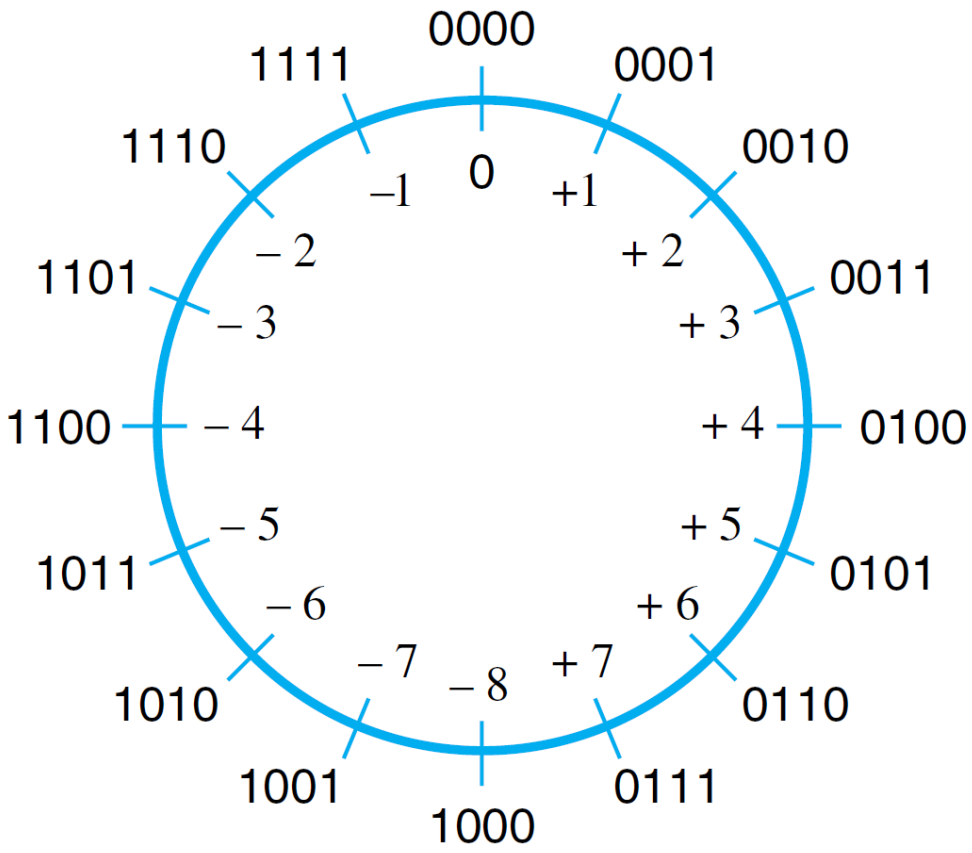
5_+	0101_+
-2	1101
3	10010_+
	→ 1
	0011

Mod 16 System for 2's-complement Numbers

6 +
2
−8

11
0110 +
0010
1000

↓ −8 2's complement



Addition of 2's-complement

- Addition of 2's complement is simple. The carry out can be ignore.
- The addition can be done in the same way regardless of the sign.

- Ex.

5_+	0101_+
-2	1110
3	10011

ignore

Arithmetic Overflow

ผลที่ได้ อยู่นอกขอบเขตของเลขที่นำเสนอได้
เช่น 6+2 = 8 overflow
มักเกิดใน sign เดียวกัน

- Arithmetic Overflow occurs when addition or subtraction produce a result that doesn't fit within the significant bit used to represent the number.
- Usually happens when 2 numbers have the same sign.
- A circuit can be added to detect an overflow

  Overflow $= C_{n-1} + C_n$

  (C is a carry occurs from bit n)

Arithmetic Overflow Example

- Ex.

	$c_4c_3c_2c_1$
$+7_+$	0111_+
+2	0010 $C_3 = 1$
+9	1001 $C_4 = 0$

$C_3 + C_4 = 1$ ⟶ overflow occurs

Arithmetic Overflow Example

- Ex.

	$c_4c_3c_2c_1$
-7_+	1001_+
$+2$	0010 $C_3 = 0$
-5	1011 $C_4 = 0$

$C_3 + C_4 = 0$ ⟶ no overflow

Memory Locations and Addresses

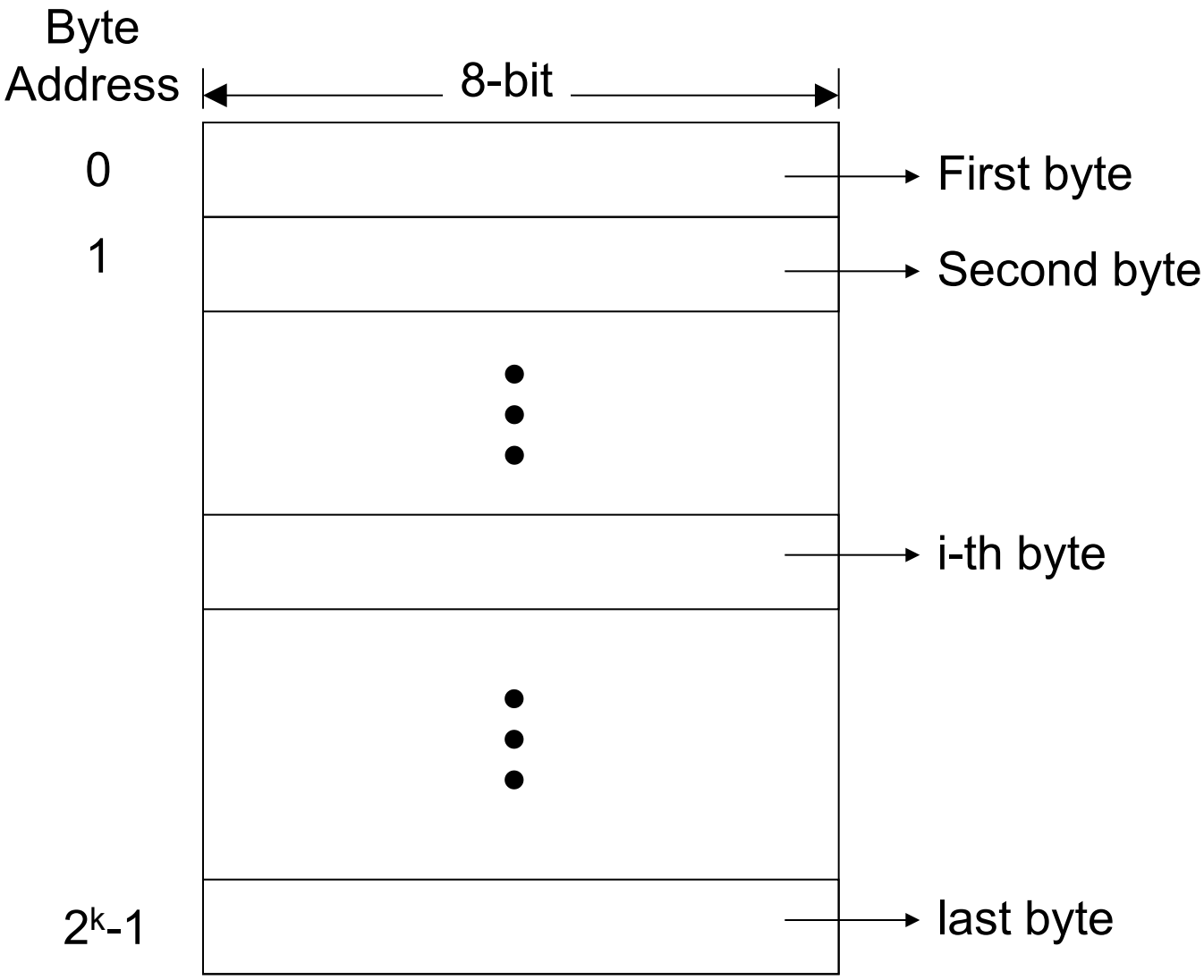
- Memory consists of a set of storage cells, where each cell stores 1 bit of information. A group of bits form a word, which is a unit use when retrieving the stored information
- Accessing a word need a distinct address. Memory address is typically a number from 0 to 2^k-1 , for some value k. 2^k is the size of the address space.
- Ex. 32-bit address creates 2^{32} (4 gig) address space

Memory Locations

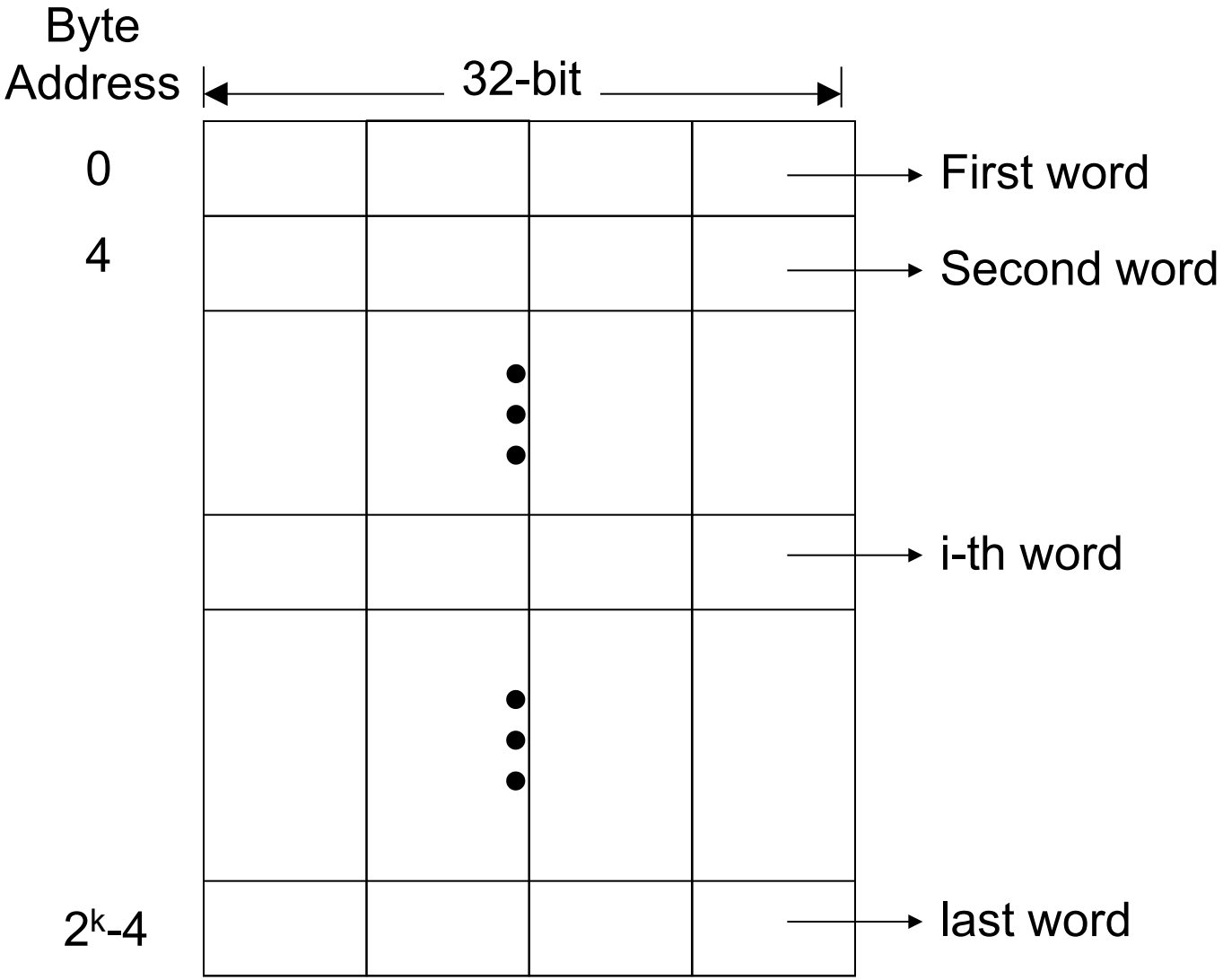
- 1 byte = 8 bits
- 1 word = 16 to 64 bits, normal is 32 bits
- Most computers have a byte–addressable memory. All bytes in one word usually are stored consecutively in the address space.
- If a word has 4 bytes, successive word locations are 0,4,8,12,……

32 bits

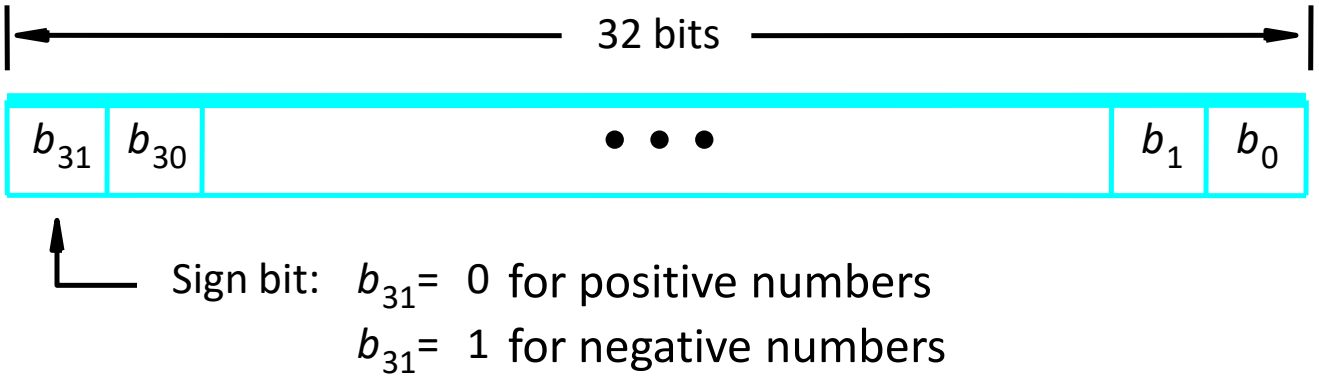
Byte Address



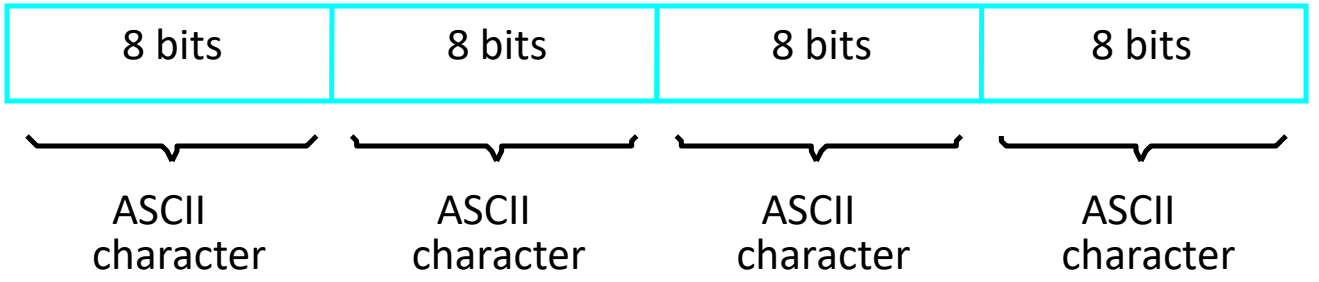
Word Address



Encoded Information

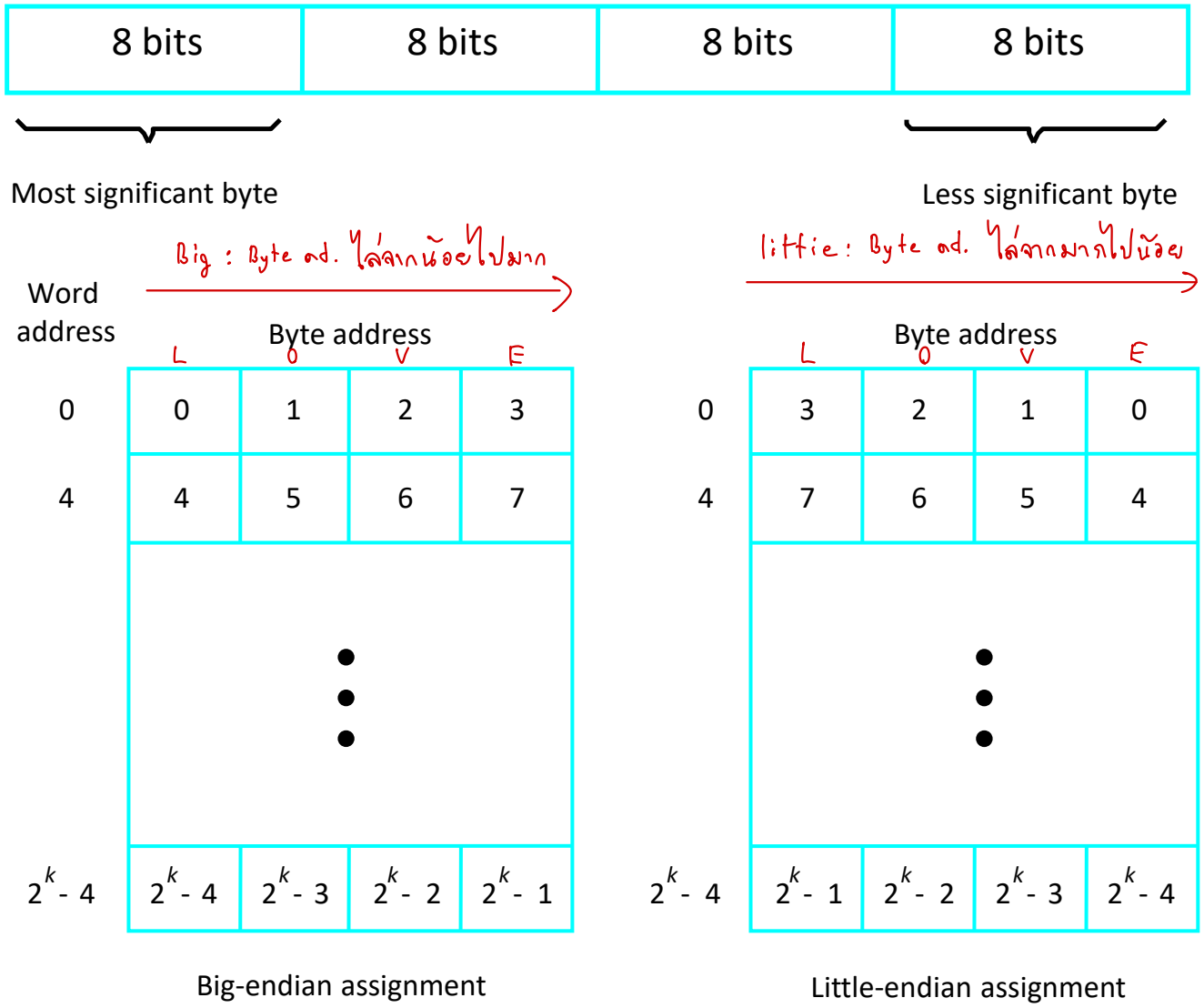


(a) A signed integer



(b) Four characters

Big-endian and Little-endian



Big-endian assignment

Little-endian assignment

Instruction Sequencing

Instructions (1)

- A task carried out by a computer consists of a sequence of small steps.
- These small steps are called instructions.
- Ex
  - Adding 2 numbers
  - Testing condition (if)
  - Reading/writing a word
- So .. computer task is a *Instructions Sequencing*

Instructions (2)

- There are four types of instructions, which are
  - Data transfers (between the memory and the processor)
  - Arithmetic and logic operations
  - Program control operations
  - I/O transfers
- Notations for machine instructions
  - Register Transfer Notation (RTN)
  - Assembly Notation

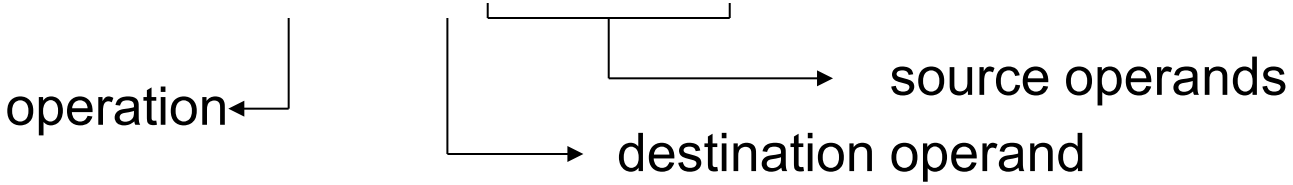
Register Transfer Notation (RTN)

หน่วยความจำที่อยู่ใน CPU

- Notations often used for data transfer
  - Processor's registers : R0, R1,…., Rn
  - Memory locations : VAR, LOCA, A
  - I/O : DATAIN, DATAOUT
- The content in each location are denoted using a bracket ([R0])
  - R1 ⟵ [LOC] : put the content of the memory location, LOC, into register R1.
  - R3 ⟵ [R1] + [R2] : add the content of registers R1 and R2, and places the result into register R3.

Assembly Notation

- Notations in assembly language format can be shown below
  - Load R1, LOC Ξ R1 ⟵ [LOC]
  - Add R3, R1, R2 Ξ R3 ⟵ [R1] + [R2]



- Some processors use *mnemonics* instead Eng.
  - Ex. LD for Load from mem or ST for Store to mem

Instruction Operands (1)

- ADD A, B, C (3 Operands)
  - A 3-address instruction possibly be too large to fit in a word of a reasonable size. We may need an instruction format that allows multiple words to be used for a single instruction, which make things more complex.
- ADD A, C (2 Operands)
  - A 2-address instruction can sometime fix the problem.

Instruction Operands (2)

- For some machines, 2-address instructions are still too large to fit in one word. We can use a 1-address instruction, assuming that the second operand is stored in a fix location (processor register call "accumulator"). ACC → เป็น Buffer ในการจัดเก็บ register ในขั้นตอนของ ADD
- Ex ADD A Ξ add the content of A to the accumulator and save the result in the accumulator

  LOAD A Ξ copy the content of A to the accumulator

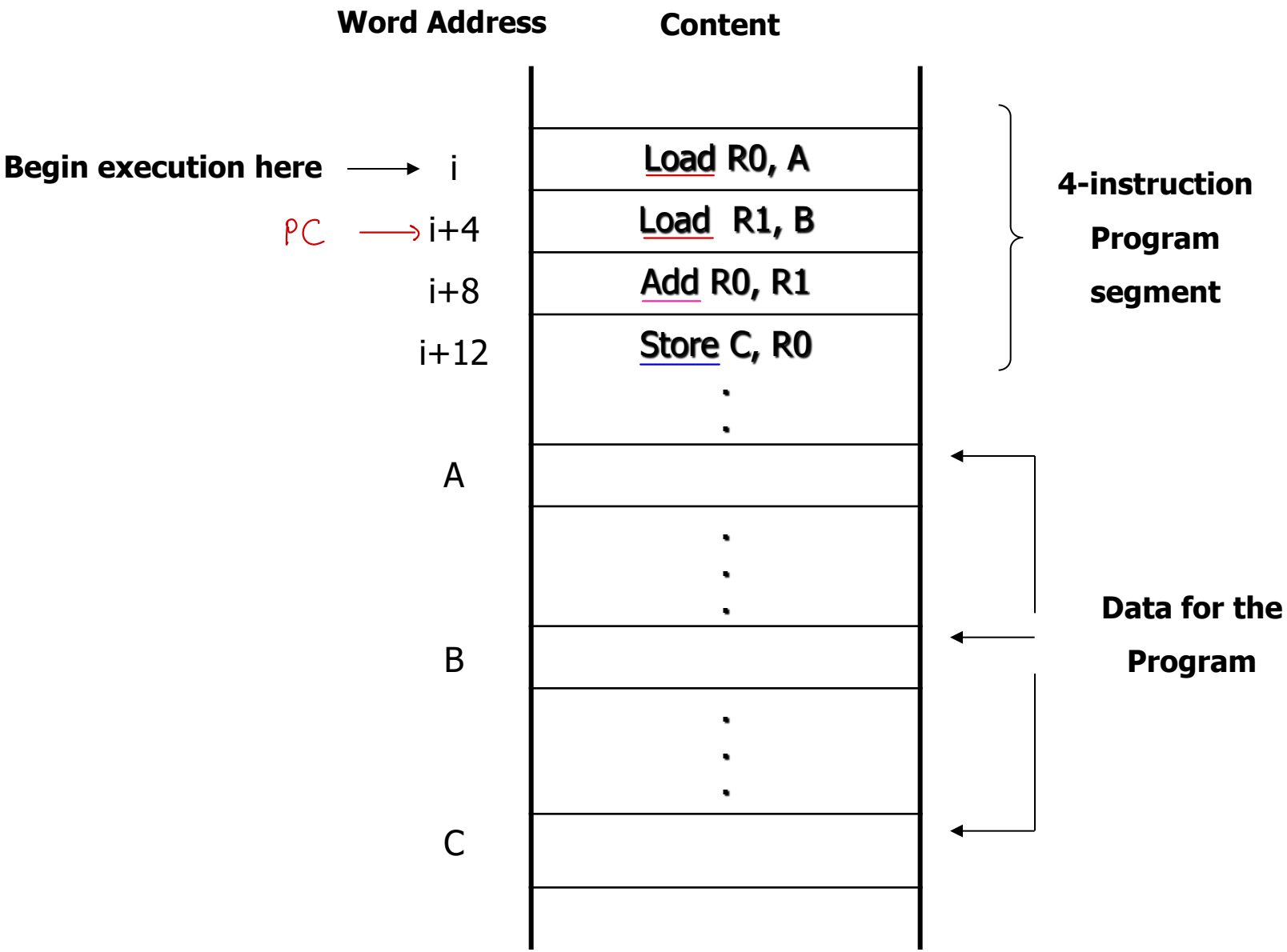
  STORE A Ξ copy the content of the accumulator to A

RISC and CISC Instruction Sets

- All instructions in the set need to fit in one word to reduce the complexity and the number of different types of instructions, called Reduced Instruction Set Computer (RISC) 1 instruction จะมีความยาวสม่ำเสมอ แบบ 1 word
  ก็จะ= 1 word ตลอด Instruction

- More complex instructions may span more than one word of memory per an instruction for specifically complicated task, called Complex Instruction Set Computer (CISC) 1 instruction จะมีความยาวไม่ สม่ำเสมอ

Instruction execution: C = [A] + [B]



Instructions Execution (1)

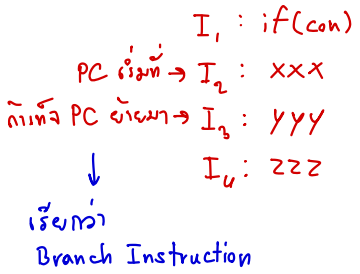
- Each instruction has two phases of execution, which are fetch and execution.
- Instruction is placed in the instruction register (IR), which is decoded to determine what operation is performed.
- In the mean time of operation PC is pointing to the next instruction

Instructions Execution (2)

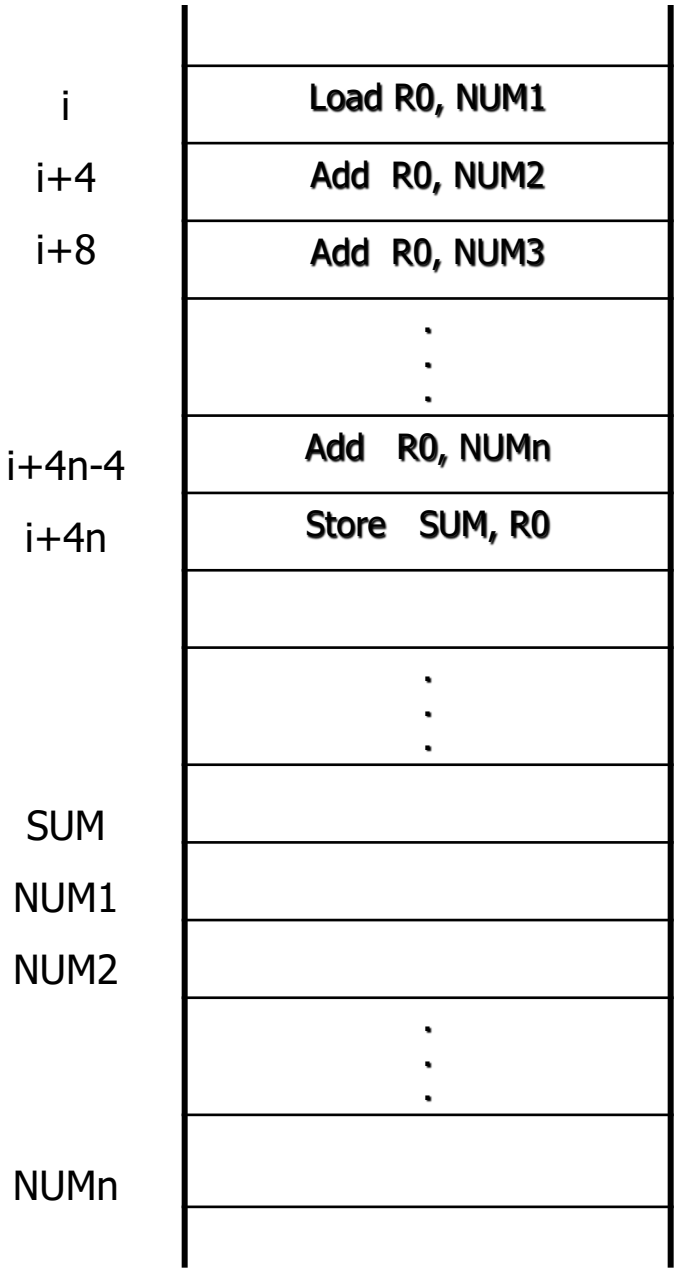
- A PC or a program counter is a register on a processor that holds the address of the next instruction to be executed.
- The address of the first instruction (i) is placed in the PC, the processor fetches and executes the instruction, and the PC is incremented by 4 (assume byte addressable).
- After an instruction stored at i+12 is executed, the PC will contain i+16, which should hold the first instruction of the next program.

Branching Concept

- The following slides show programs that add *n* number together.
  - First program: No branch instruction is used
  - Second program: Uses a loop structure (via branch instruction) to optimize the program

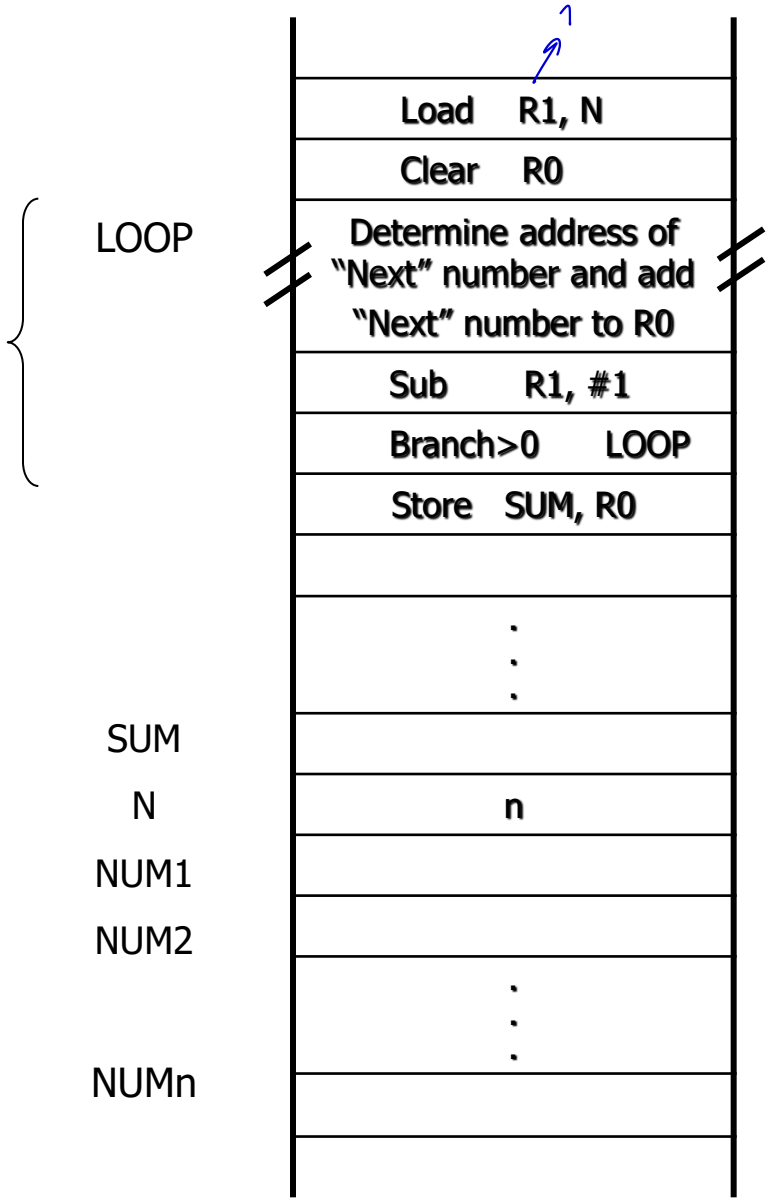


No Branching



Branching

Program loop



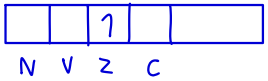
Conditional Branching

- The branch instruction loads the new target address to the PC, instead of the next instruction in the sequence.
- Conditional branch, only causes a branch when the condition is met.
  - The instruction “Branch > 0 LOOP” causes a program to branch to the location, *LOOP* , when the result of the preceding instruction is > 0.

Condition Codes with Flags

- Condition codes are used to determine the condition of the instruction so that a branch can be determined. These codes are grouped together and stored in status register.
  - N (negative), 1 if negative
  - V (overflow), 1 if arithmetic overflow
  - Z (zero), 1 if result is 0
  - C (carry), 1 if have carry-out

Flag Register



Addressing Modes (1)

เกี่ยวกับ operand ของ instruction

- There are different ways in which the location of an operand is specified in an instruction.
- Address modes in modern processors are listed in the above table.
- Most variables and constants use two addressing modes, which are register and absolute modes.
- The absolute mode is also used for global variables.

Addressing Modes (2)

Name	Assembly syntax	Addressing function
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
Index	X(Ri)	EA = [Ri] + X
Base with index	(Ri,Rj)	EA = [Ri] + [Rj]
Base with index and offset	X(Ri,Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri;
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri;]

EA = effective address

Value = a signed number

Immediate Addressing Mode

- Immediate mode : Move R0, #200
  - Moves the value, 200, into the register R0
- A = B+6 can be translated to
- Load R1, B

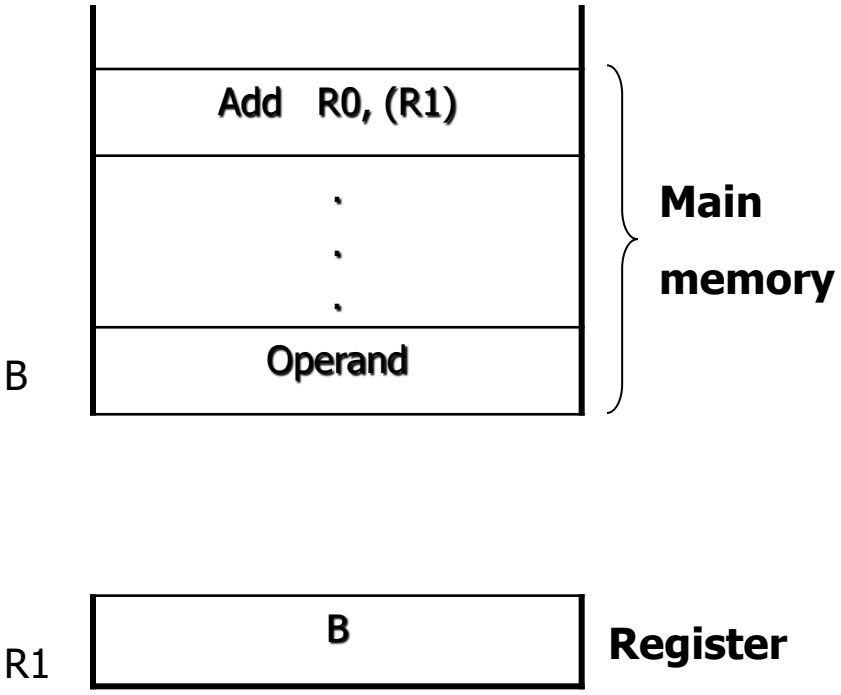
  Add R1, #6

  Store A, R1

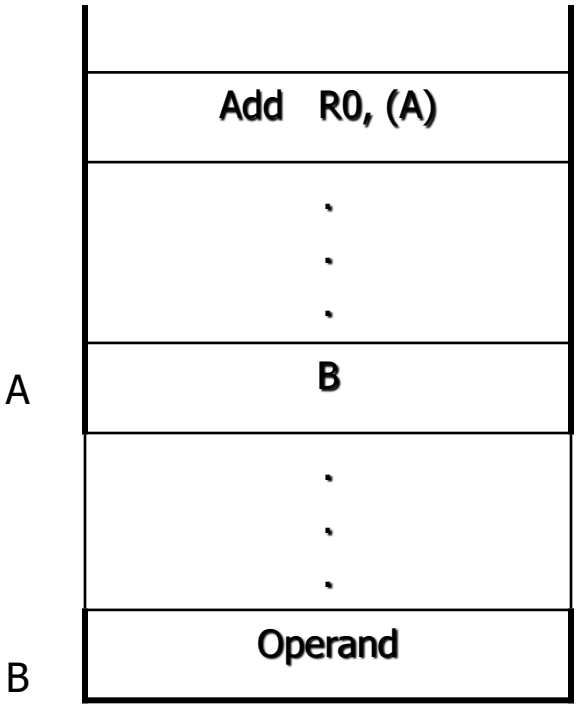
Indirect Addressing Mode (1)

- In the Indirect mode, the effective address is the content of a register or a memory location.
- Indirect mode can be used to implement the concept of pointers in a high-level language.

Indirect Addressing Mode (2)



(a) Through a general-purpose register



(b) Through a memory location

Indirect Addressing Mode (3)

Add n numbers together (Use R2 as a pointer)

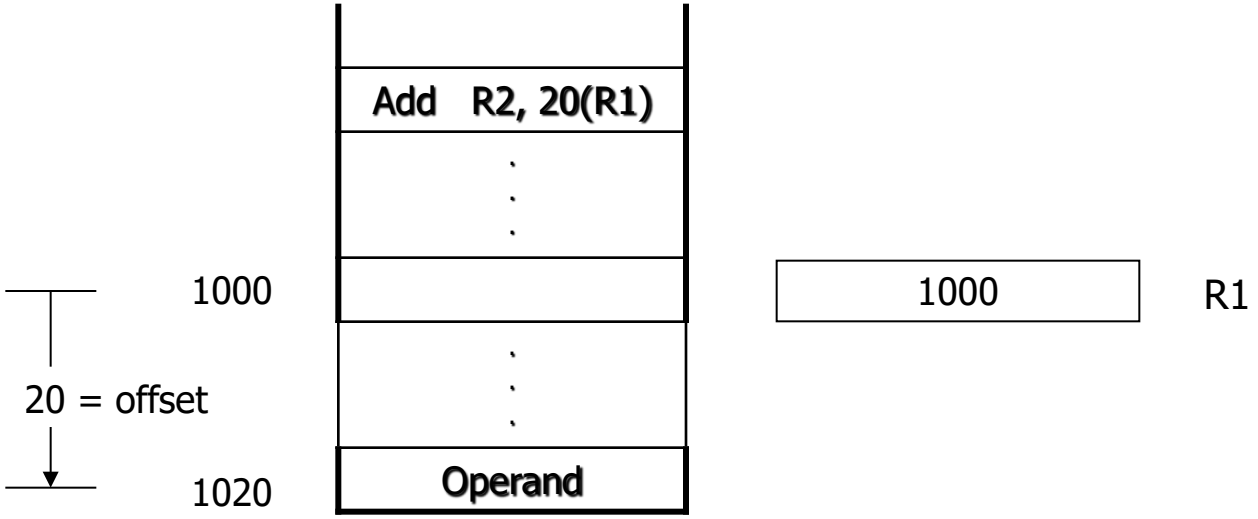
Address	Content		
	Load	R1, N	Initialization
	Load	R2, #NUM1	
	Clear	R0	
LOOP	Add	R0, (R2)	
	Add	R2, #4	
	Decrement	R1	
	Branch>0	LOOP	
	Store	SUM, R0	

Index Addressing Mode (1)

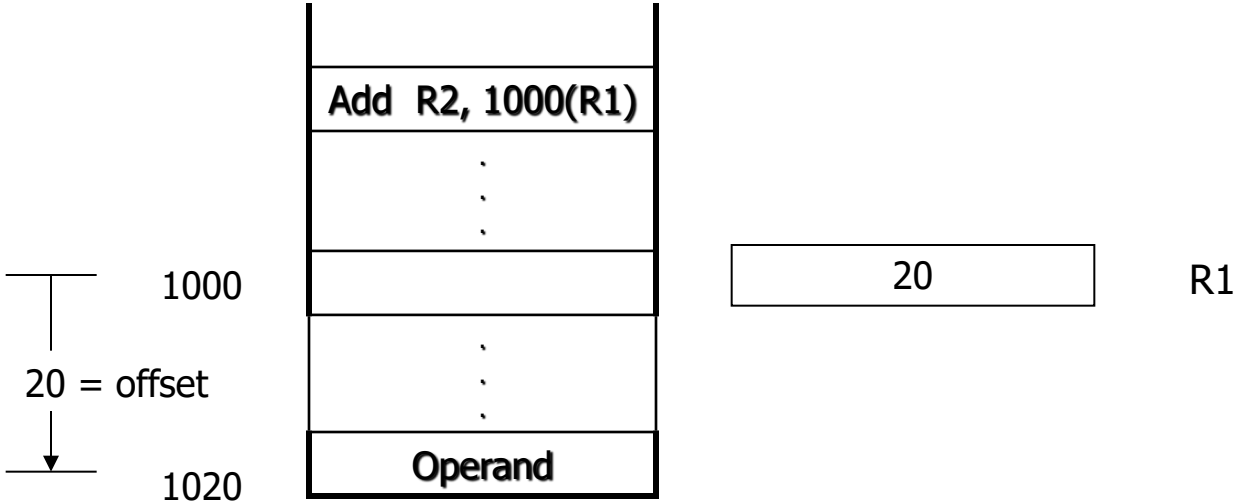
- In the index mode, the effective address is generated by adding a constant value to the content of the register.

$$X\,(Ri) \longrightarrow ea = [Ri] + X$$

Index Addressing Mode (2)



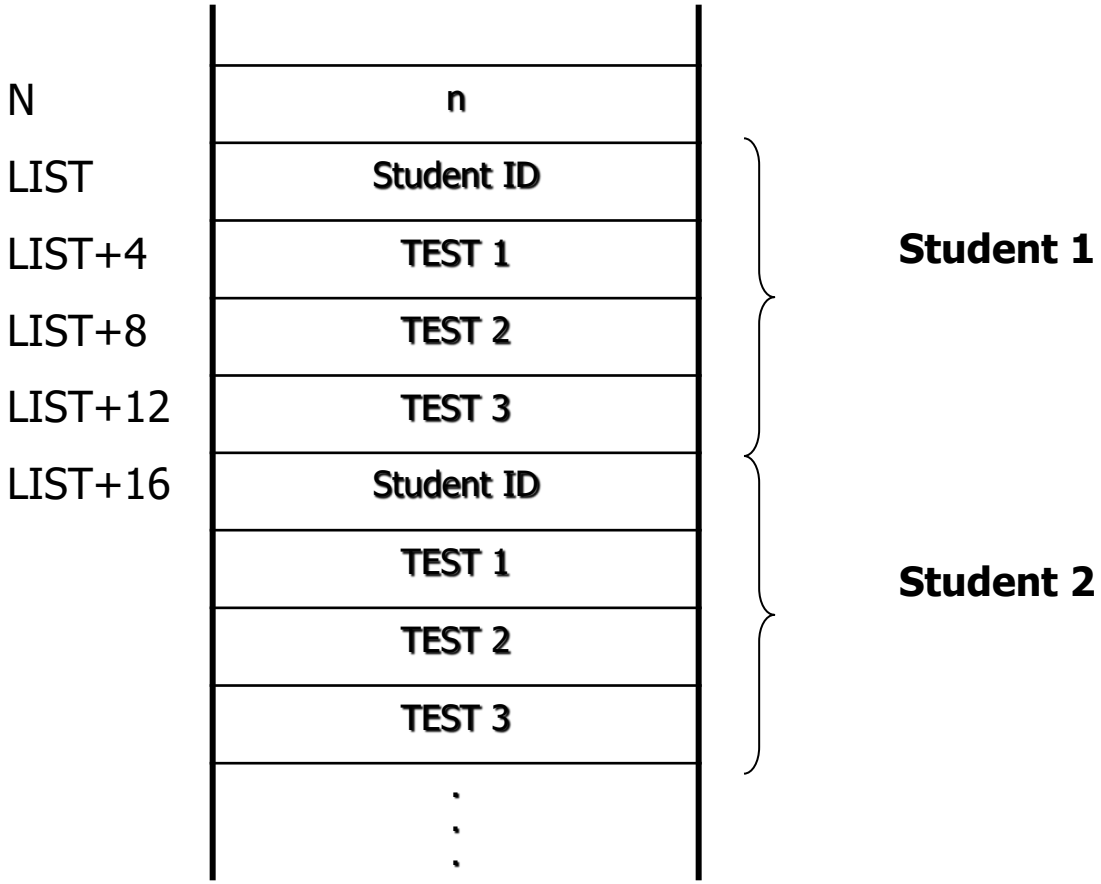
(a) Offset is given as a constant



(b) Offset is in the index register

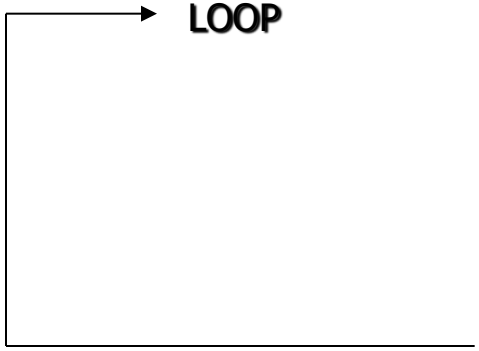
Index Addressing Mode (3)

A program uses a 2 dimensional array of 4xn. There are n students in the class and each student takes 3 exams. The program calculate the accumulative score for each test.



Index Addressing Mode (4)

Index addressing mode is used in the program. R0 only changes in the last ‘add’ instruction (move to the next student).



	Load	R0, #LIST
	Clear	R1
	Clear	R2
	Clear	R3
	Load	R4, N
LOOP	Add	R1, 4(R0)
	Add	R2, 8(R0)
	Add	R3, 12(R0)
	Add	R0. #16
	Decrement	R4
	Branch>0	LOOP
	Store	SUM1, R1
	Store	SUM2, R2
	Store	SUM3, R3

Relative Addressing Mode

- Relative mode: is similar to the index mode, except that a PC is used instead of a general purpose register. This mode can be used to specify target address in branch instructions

Autoincrement/autodecrement Modes

- Autoincrement (Ri)+ and Autodecrement -(Ri)

- These two modes combine 2 instructions, which are Load and Increment/Decrement.