III Sem(2021-22)
Computer Architecture and Organization
(CCT 201/CDT202)

Mr. P.M.Sonsare
Assistant Professor
Computer Science and Engineering
RCOEM, Nagpur

Course Objectives:

- To impart to students the basic structure of Computers and different data representation techniques.
- To familiarize students with designing of memory hierarchy and control unit.
- To make students aware of I/O organization.

Course Contents

Unit 1. **Basic Structure Of Computers**

Unit 2. **Basic Processing Unit**

Unit 3. **Data Representation**

Unit 4. **Memory System Design**

Unit 5. **Memory Organization**

Unit 6. **Input / Output Organization**

Grading Policy

- Teacher Assessment (30 marks)
- Internal Tests(30 Marks)
- End Semester Exam (40 marks)

Books:

- V. C. Hamacher, Z. G. Vranesic and S. G. Zaky; Computer Organisation; 5th edition; Tata McGraw Hill, 2002.
- W. Stallings; Computer Organization & Architecture; PHI publication; 2001.
- J. P. Hayes; Computer Architecture & Organization; 3rd edition; McGraw-Hill; 1998.

Course Outcomes:

On completion of the course, student will be able to–

CO1: Understand the basic components of a computer, including CPU, memories, and input/output, and their organization.

CO2: Understand the cost performance tradeoff in designing memory hierarchy and instruction sets.

CO3: Understand the execution of complete instruction and design of control unit.

CO4: Perform mathematical operations on arithmetic and floating point numbers.

UNIT1:Basic Structure Of Computers

- Objectives

- Functional units of computer.
- Instructions set architecture of a CPU Instruction sequencing
- Addressing modes
- Instruction set classification
- Subroutine & parameter passing
- Expanding opcode

Functional Units

Functional Units

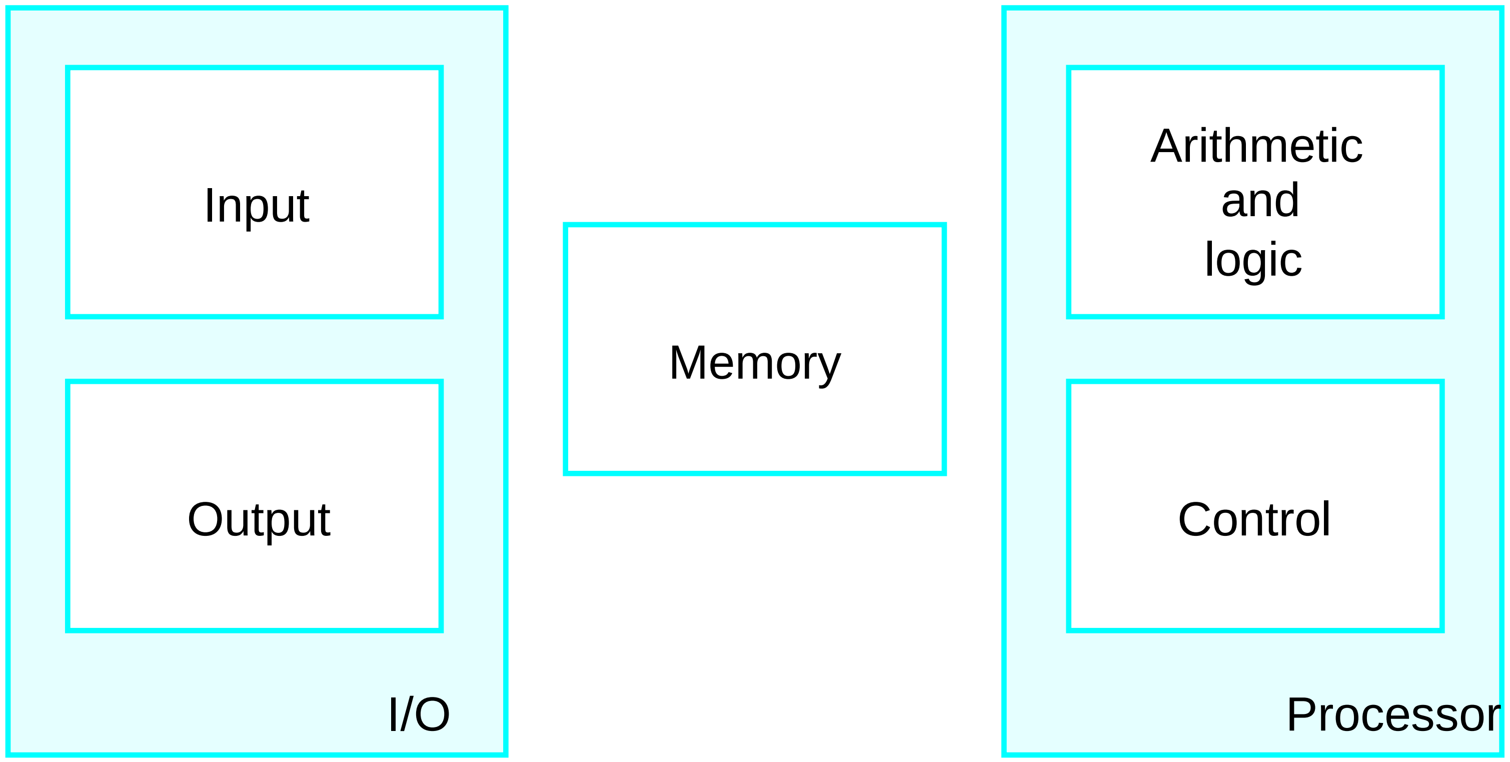


Figure 1.1. Basic functional units of a computer.

Information Handled by a Computer

- Instructions/machine instructions
  - Govern the transfer of information within a computer as well as between the computer and its I/O devices
  - Specify the arithmetic and logic operations to be performed
  - Program
- Data
  - Used as operands by the instructions
  - Source program
- Encoded in binary code – 0 and 1

Memory Unit

- Store programs and data
- Two classes of storage
  - Primary storage
    - Fast
    - Programs must be stored in memory while they are being executed
    - Large number of semiconductor storage cells
    - Processed in words
    - Address
    - RAM and memory access time
    - Memory hierarchy – cache, main memory
  - Secondary storage – larger and cheaper

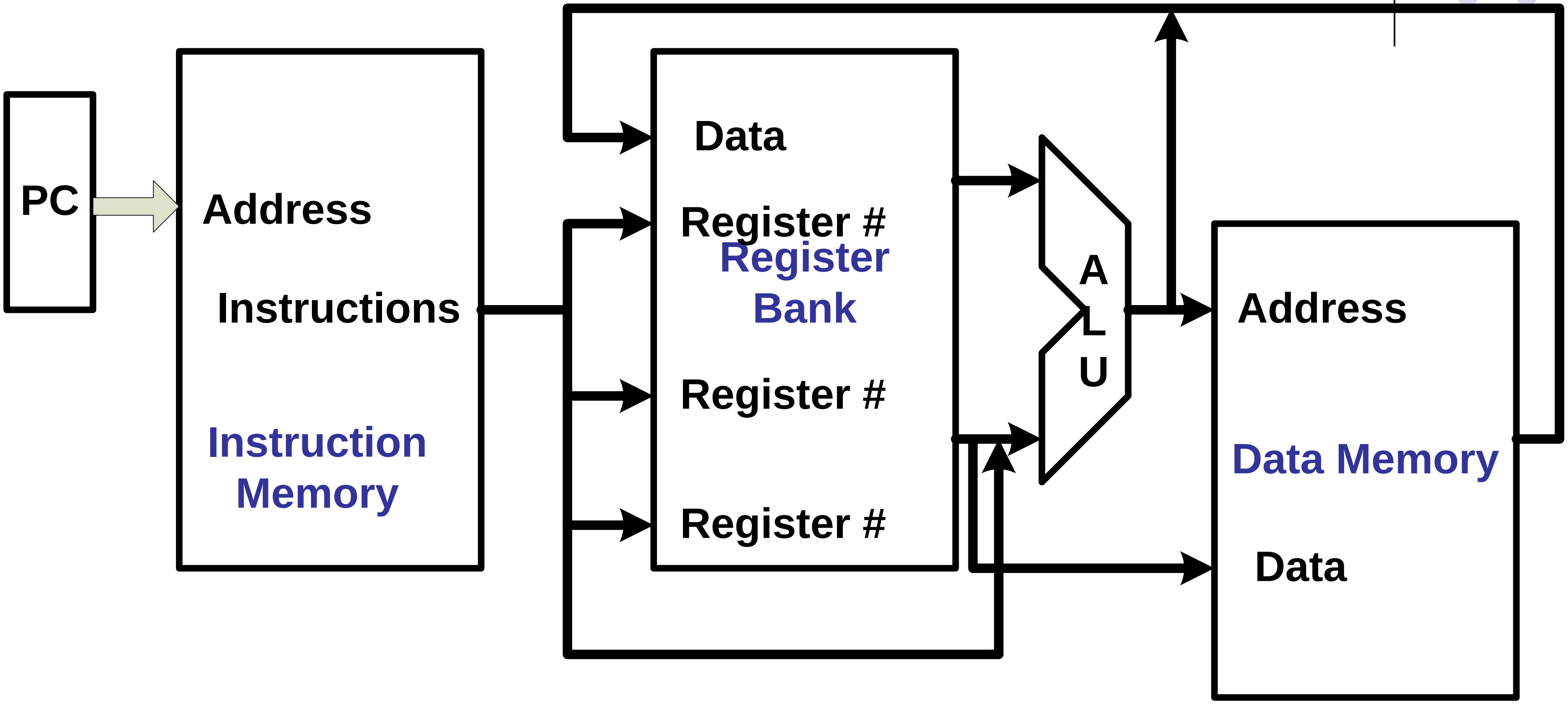
Arithmetic and Logic Unit (ALU)

- Most computer operations are executed in ALU of the processor.
- Load the operands into memory – bring them to the processor – perform operation in ALU – store the result back to memory or retain in the processor.
- Registers
- Fast control of ALU

Control Unit

- All computer operations are controlled by the control unit.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Control unit is usually distributed throughout the machine instead of standing alone.
- Operations of a computer:
  - Accept information in the form of programs and data through an input unit and store it in the memory
  - Fetch the information stored in the memory, under program control, into an ALU, where the information is processed
  - Output the processed information through an output unit
  - Control all activities inside the machine through a control unit

The processor : Data Path and Control



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

Five Execution Steps

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

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

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

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

Basic Operational Concepts

Review

- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

A Typical Instruction

- Add LOCA, R0
- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.

Separate Memory Access and ALU Operation

- Load LOCA, R1
- Add R1, R0
- Whose contents will be overwritten?

Connection Between the Processor and the Memory

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

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

Speed Issue

- Different devices have different transfer/operate speed.
- If the speed of bus is bounded by the slowest device connected to it, the efficiency will be very low.
- How to solve this?
- A common approach – use buffers.

Performance

Performance

- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
  - Hardware design
  - Instruction set
  - Compiler

Performance

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

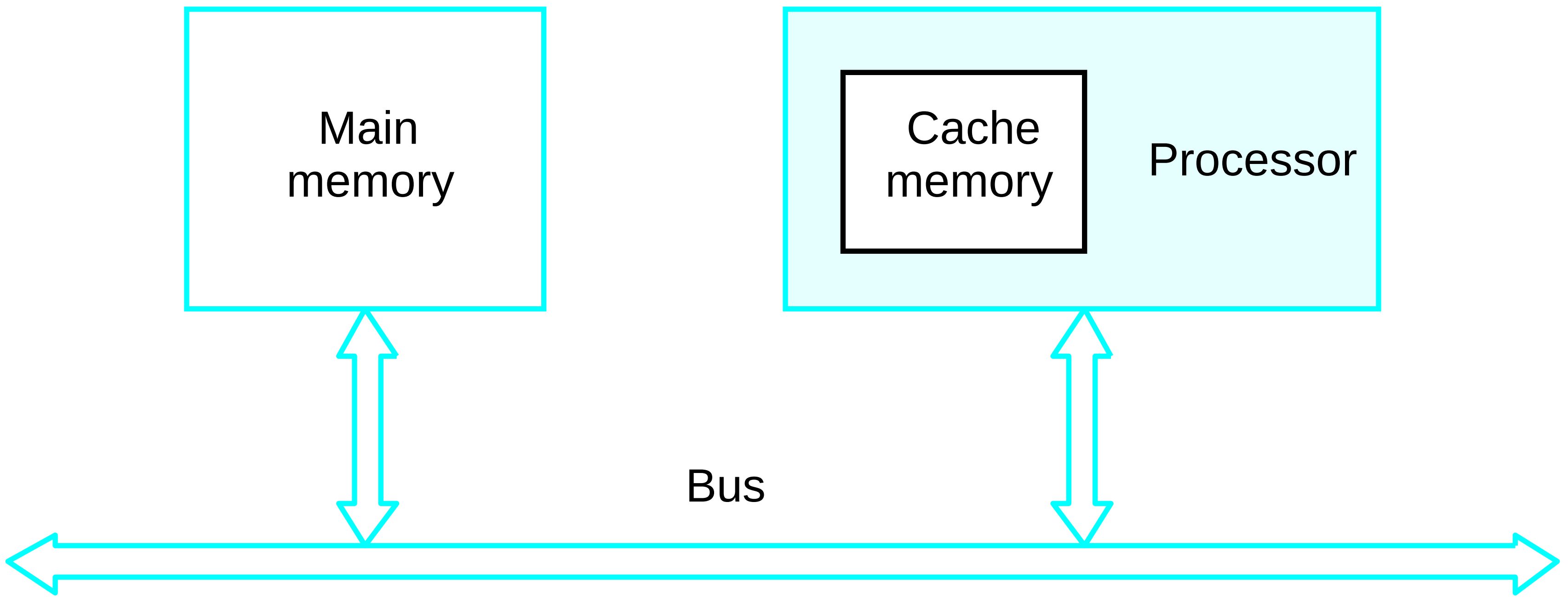


Figure 1.5. The processor cache.

Performance

- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip.
- Speed
- Cost
- Memory management

Processor Clock

- Clock, clock cycle, and clock rate
- The execution of each instruction is divided into several steps, each of which completes in one clock cycle.
- Hertz – cycles per second

Basic Performance Equation

- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- Note: these are not independent to each other

$$T = \frac{N \times S}{R}$$

How to improve T?

CISC and RISC

- Tradeoff between N and S
- A key consideration is the use of pipelining
  - S is close to 1 even though the number of basic steps per instruction may be considerably larger
  - It is much easier to implement efficient pipelining in processor with simple instruction sets
- Reduced Instruction Set Computers (RISC)
- Complex Instruction Set Computers (CISC)

- **Reduced Instruction Set Architecture (RISC)** The main idea behind is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, store command will store the data.
- **Complex Instruction Set Architecture (CISC)** The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it’s complex.

- Both approaches try to increase the CPU performance
- **RISC:** Reduce the cycles per instruction at the cost of the number of instructions per program.
- **CISC:** The CISC approach attempts to minimize the number of instructions per program but at the cost of increase in number of cycles per instruction.

- Earlier when programming was done using assembly language, a need was felt to make instruction do more task because programming in assembly was tedious and error-prone due to which CISC architecture evolved but with the uprise of high-level language dependency on assembly reduced RISC architecture prevailed.

Characteristic of RISC

- Simpler instruction, hence simple instruction decoding.
- Instruction comes undersize of one word.
- Instruction takes a single clock cycle to get executed.
- More number of general-purpose registers.
- Simple Addressing Modes.
- Less Data types.
- Pipeline can be achieved.

Characteristic of CISC

- Complex instruction, hence complex instruction decoding.
- Instructions are larger than one-word size.
- Instruction may take more than a single clock cycle to get executed.
- Less number of general-purpose registers as operation get performed in memory itself.
- Complex Addressing Modes.
- More Data types.

Example:

- Suppose we have to add two 8-bit number:
- **CISC approach:** There will be a single command or instruction for this like ADD which will perform the task.
- **RISC approach:** Here programmer will write the first load command to load data in registers then it will use a suitable operator and then it will store the result in the desired location.
- So, add operation is divided into parts i.e. load, operate, store due to which RISC programs are longer and require more memory to get stored but require fewer transistors due to less complex command.

Difference

RISC	CISC
Focus on software	Focus on hardware
Uses only Hardwired control unit	Uses both hardwired and micro programmed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction execute in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word	Instructions are larger than the size of one word

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.

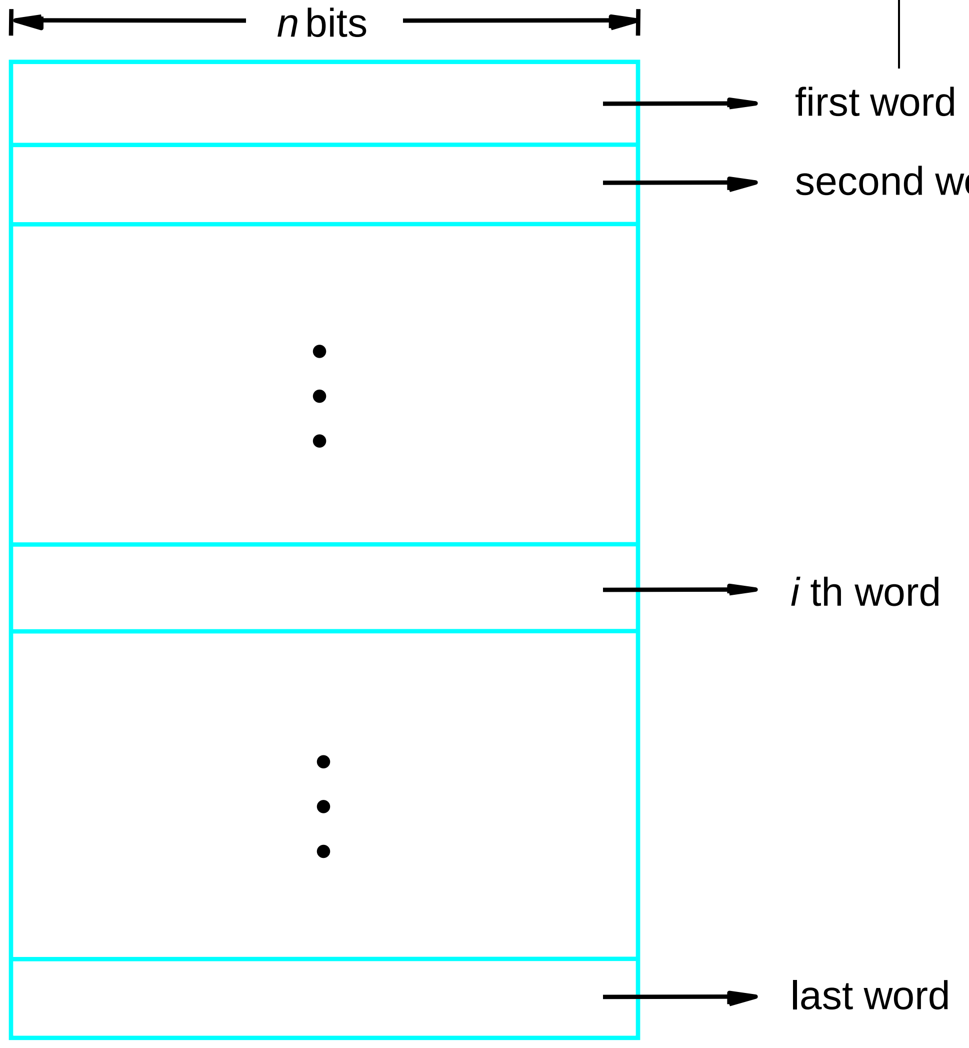
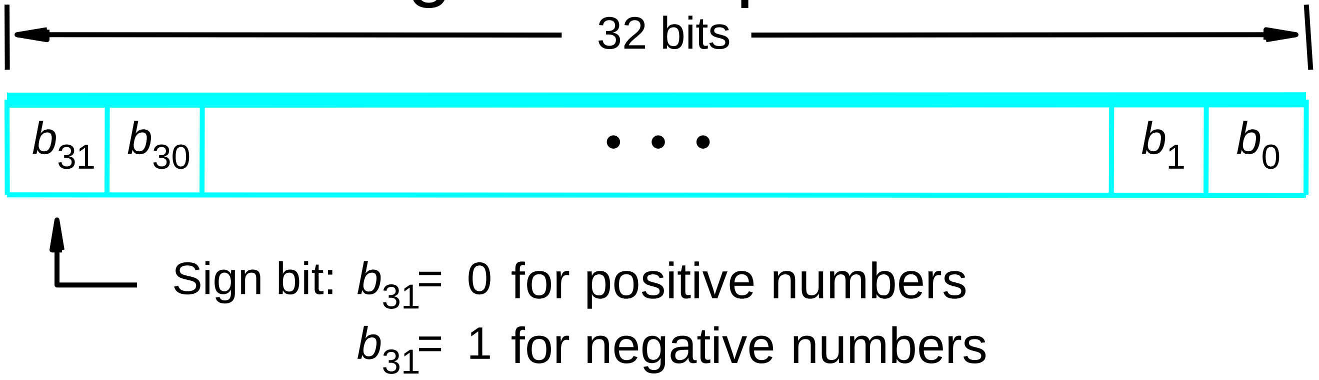


Figure 2.5. Memory words.

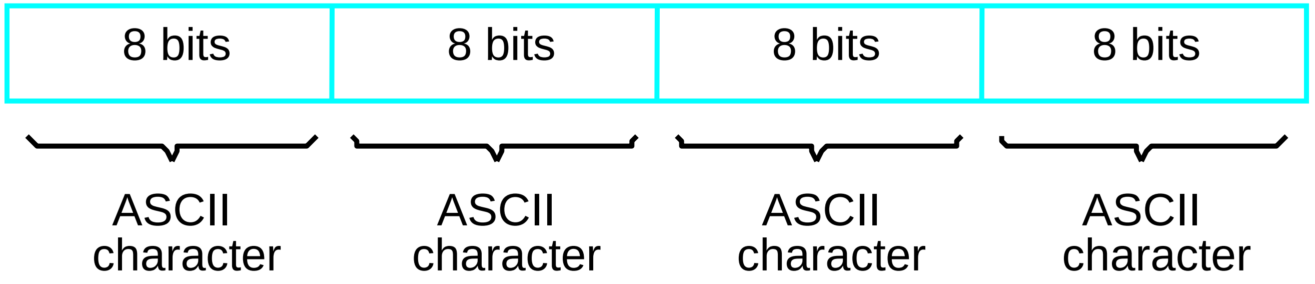
Memory Location, Addresses, and Operation

- 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

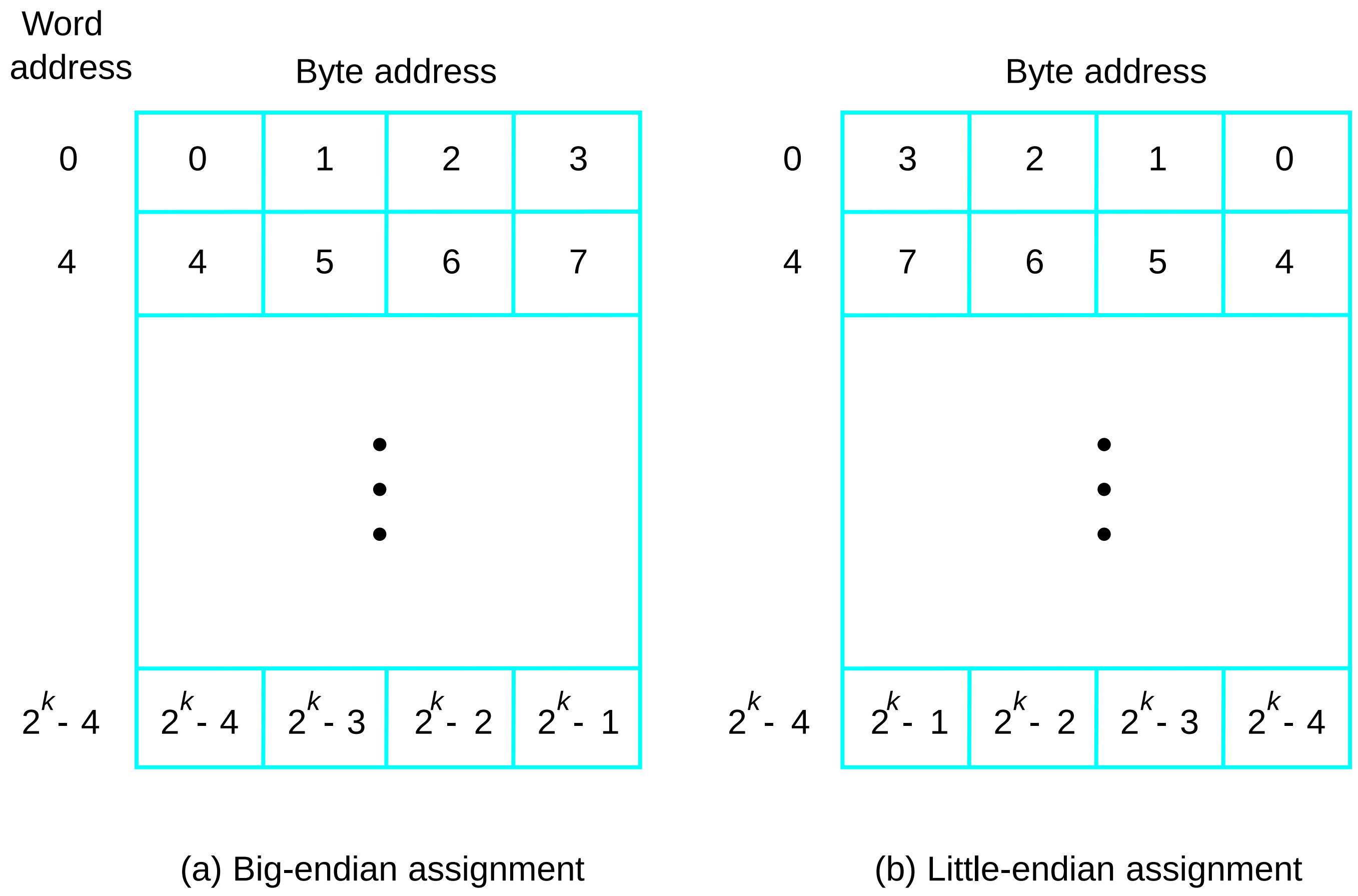


Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….
- Access numbers, characters, and character strings

Memory Operation

- Load (or Read or Fetch)
  - Copy the content. The memory content doesn’t change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - Registers can be used

Instruction and Instruction Sequencing

“Must-Perform” Operations

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location (R1←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation

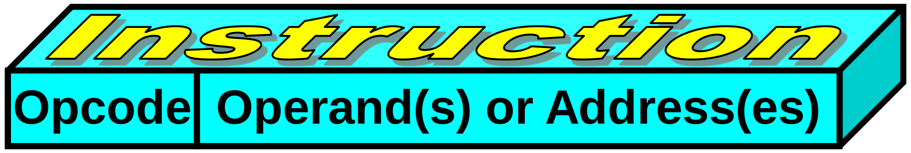
- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

CPU Organization

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

Instruction Formats

- Three-Address Instructions
  - ADD R1, R2, R3 — $R3 \leftarrow R1 + R2$
- 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. 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   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. STOREX, R1              ; M[X] ← R1
```

Using Registers

- Registers are faster
- Shorter instructions
  - The number of registers is smaller (e.g. 32 registers need 5 bits)
- Potential speedup
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing

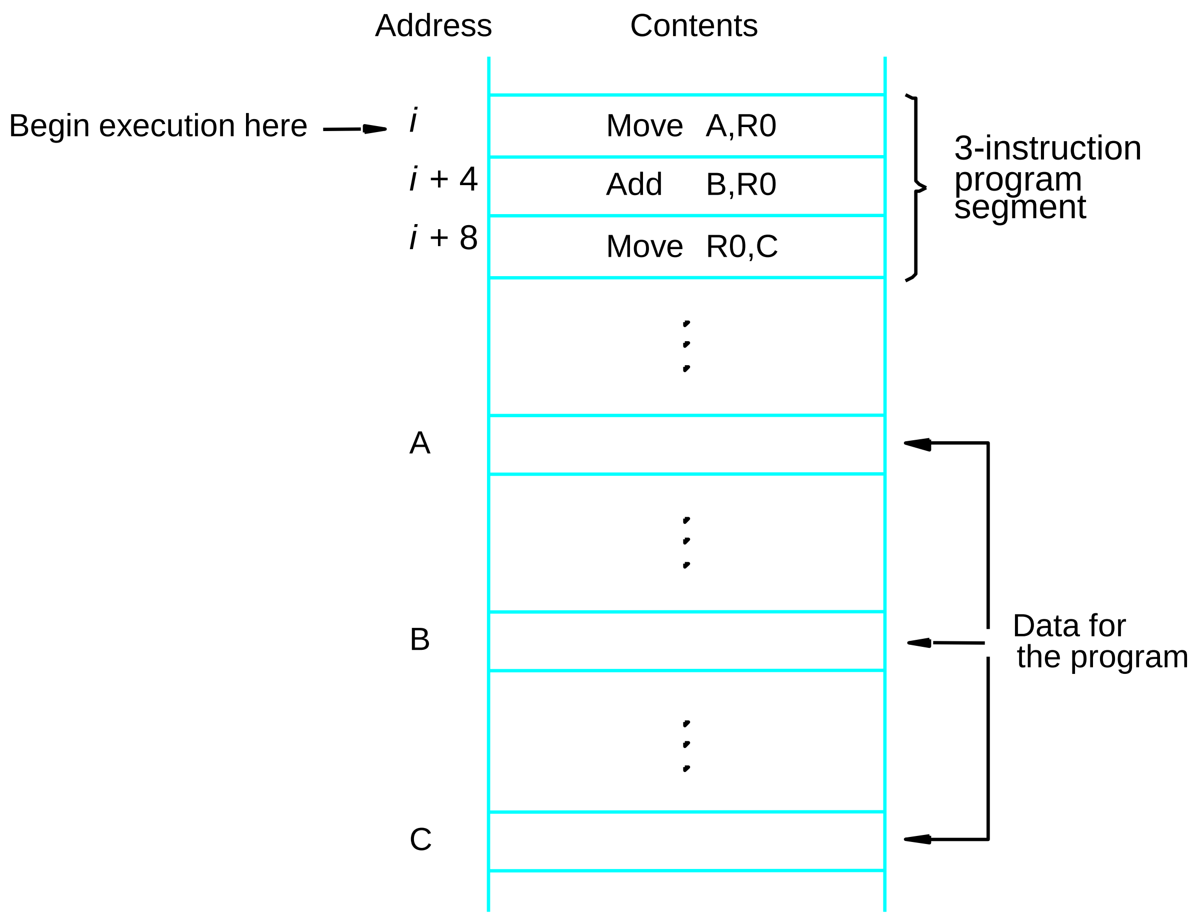


Figure 2.8. A program for C ← [A] + [B].

Assumptions:
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure
- Instruction fetch
- Instruction execute

Branching

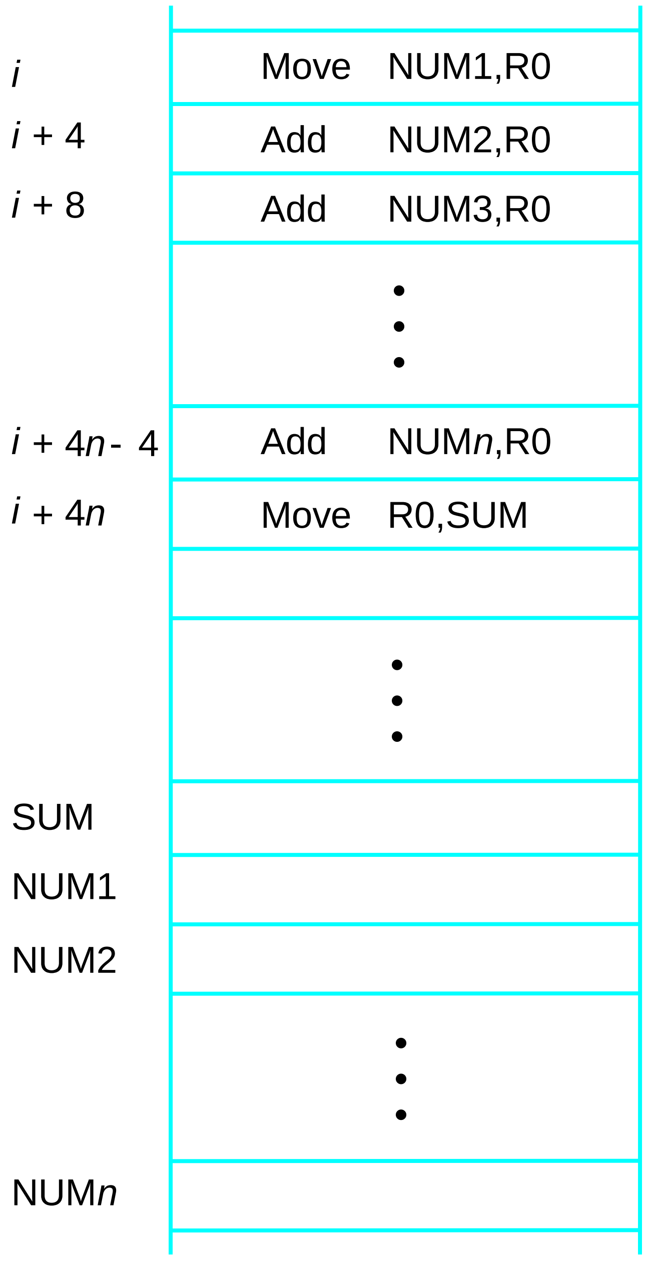


Figure 2.9. A straight-line program for adding *n* numbers.

Branching

Branch target

Conditional branch

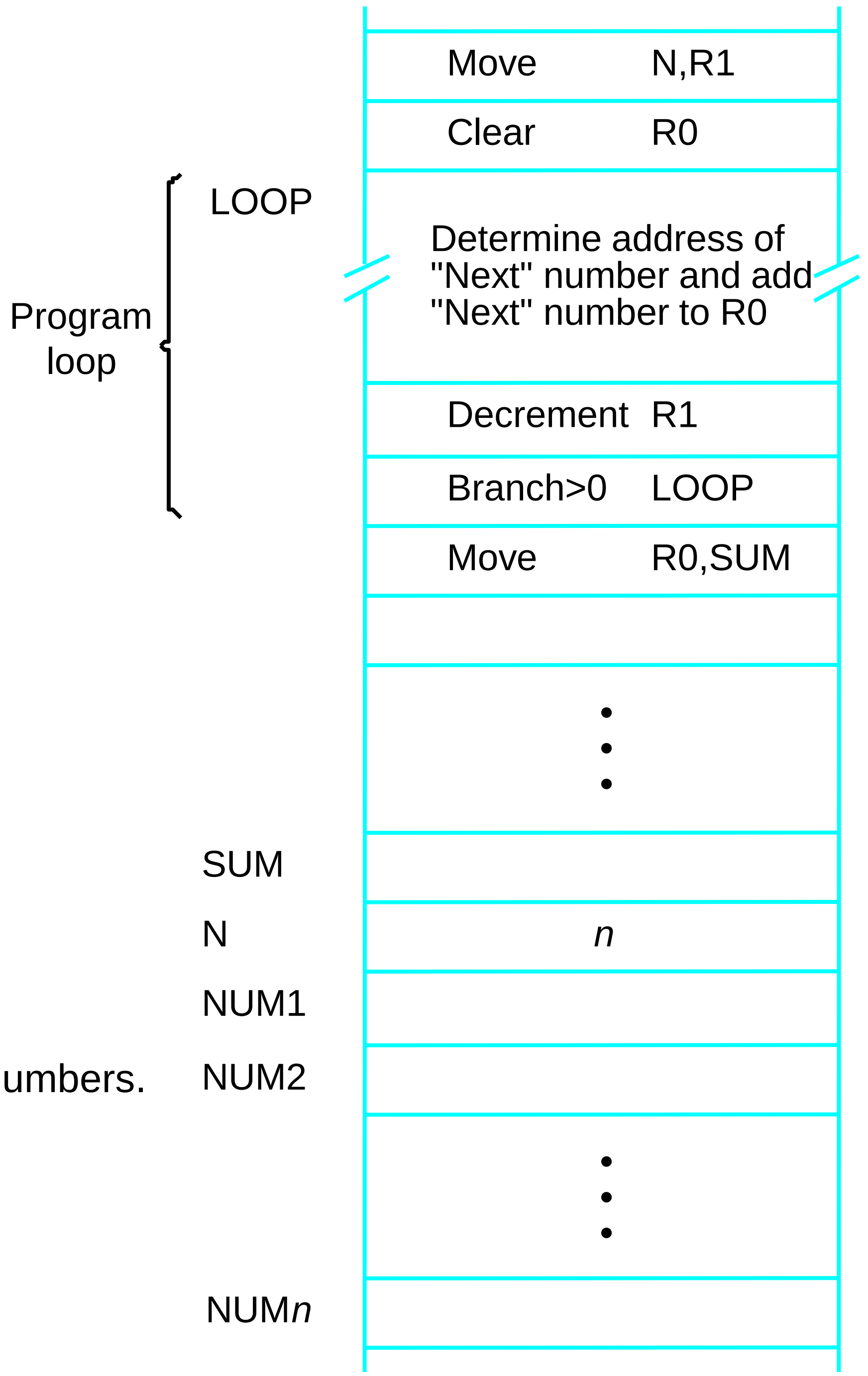


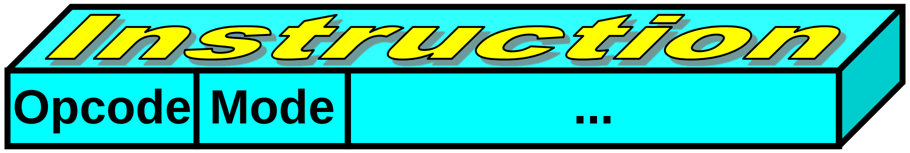
Figure 2.10. Using a loop to add *n* numbers.

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

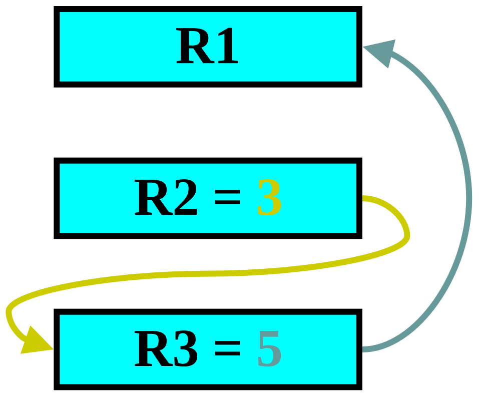


- Implied
  - AC is implied in “ADD   M[AR]” in “One-Address” instr.
  - TOS is implied in “ADD” in “Zero-Address” instr.
- Immediate
  - The use of a constant in “MOV #5,R1”, i.e. R1 ← 5
- Register
  - Indicate which register holds the operand

Addressing Modes

- Register Indirect
  - Indicate the register that holds the number of the register that holds the operand

    MOV (R2), R1



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

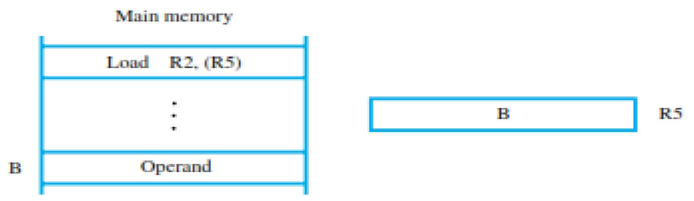
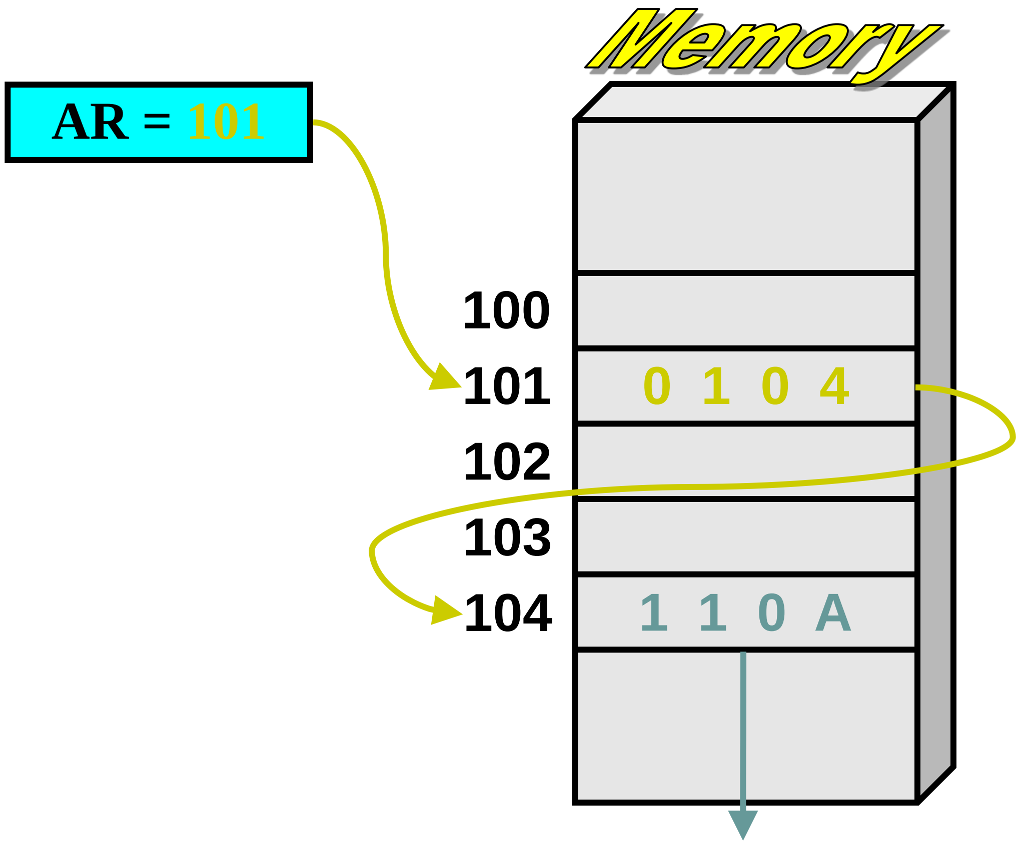
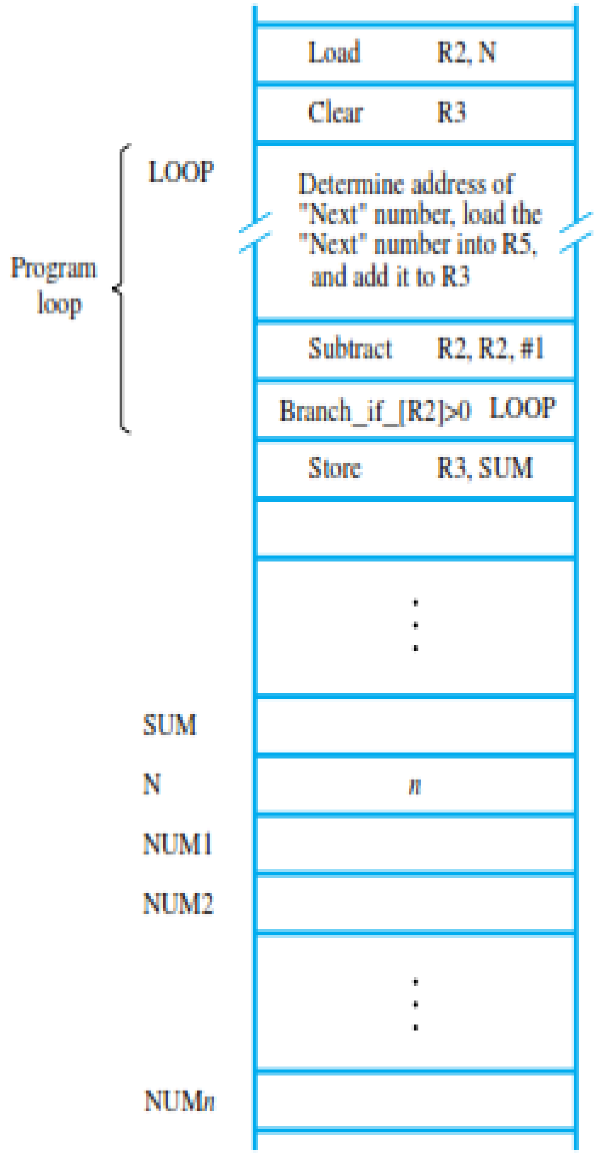


Figure 2.7 Register indirect addressing.

Addressing Modes

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





```
       Load              R2, N        Load the size of the list.
       Clear             R3           Initialize sum to 0.
       Move              R4, #NUM1    Get address of the first number.
LOOP:  Load              R5, (R4)     Get the next number.
       Add               R3, R3, R5   Add this number to sum.
       Add               R4, R4, #4   Increment the pointer to the list.
       Subtract          R2, R2, #1   Decrement the counter.
       Branch_if_[R2]>0  LOOP         Branch back if not finished.
       Store             R3, SUM      Store the final sum.
```

Indexing and Arrays

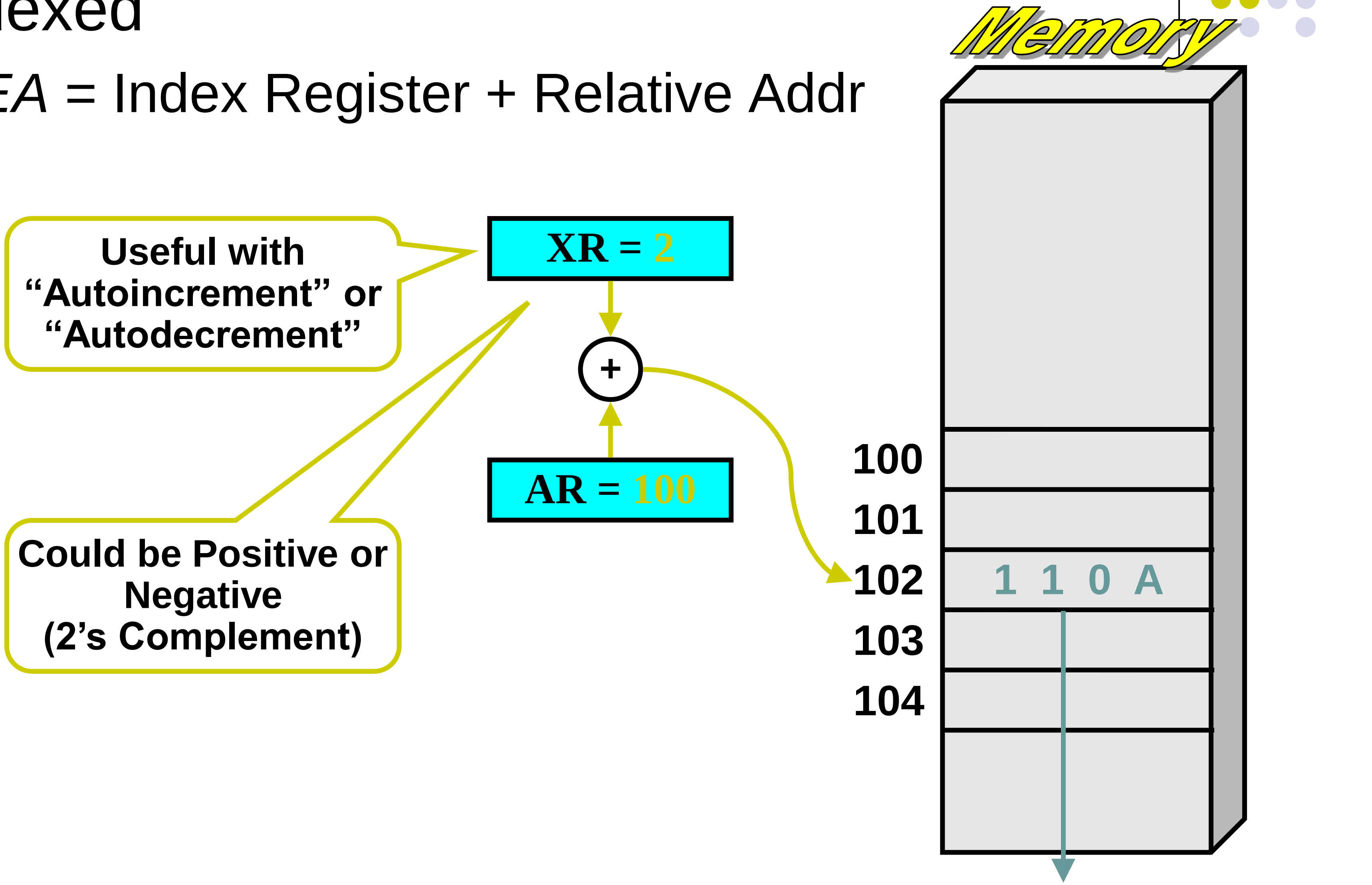
- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register
- $X(R_i)$: $EA = X + [R_i]$
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than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

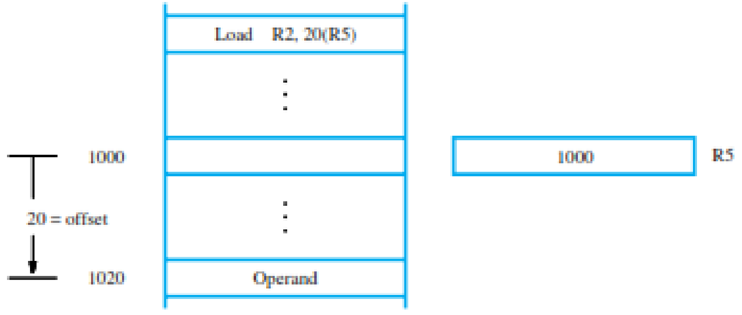
Indexing and Arrays

- 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): $EA = [R_i] + [R_j]$
  $X(R_i, R_j)$: $EA = X + [R_i] + [R_j]$

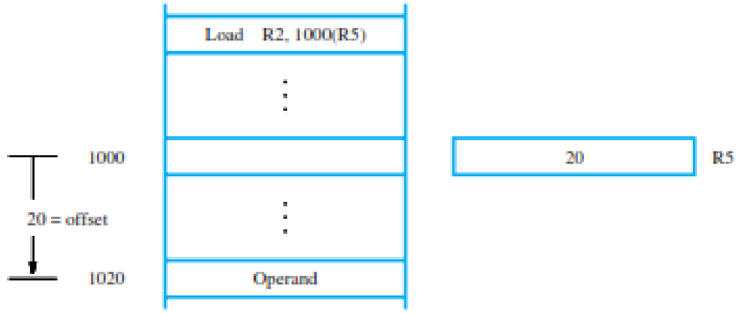
Addressing Modes

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


Memory
Useful with "Autoincrement" or "Autodecrement"
XR = 2
+
AR = 100
Could be Positive or Negative (2's Complement)
100
101
102
1 1 0 A
103
104



(a) Offset is given as a constant



(b) Offset is in the index register

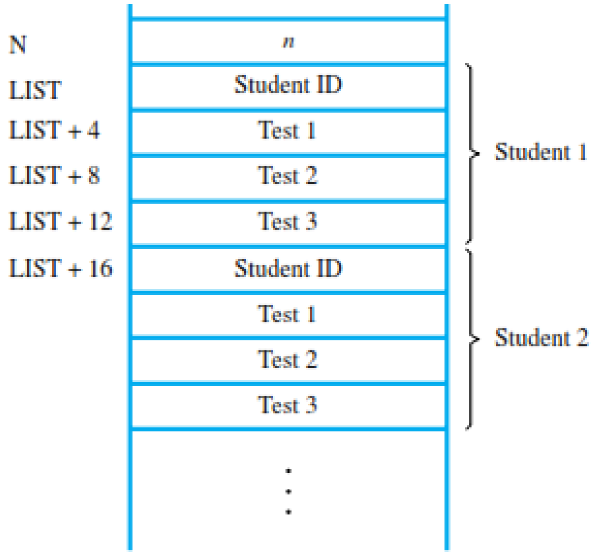
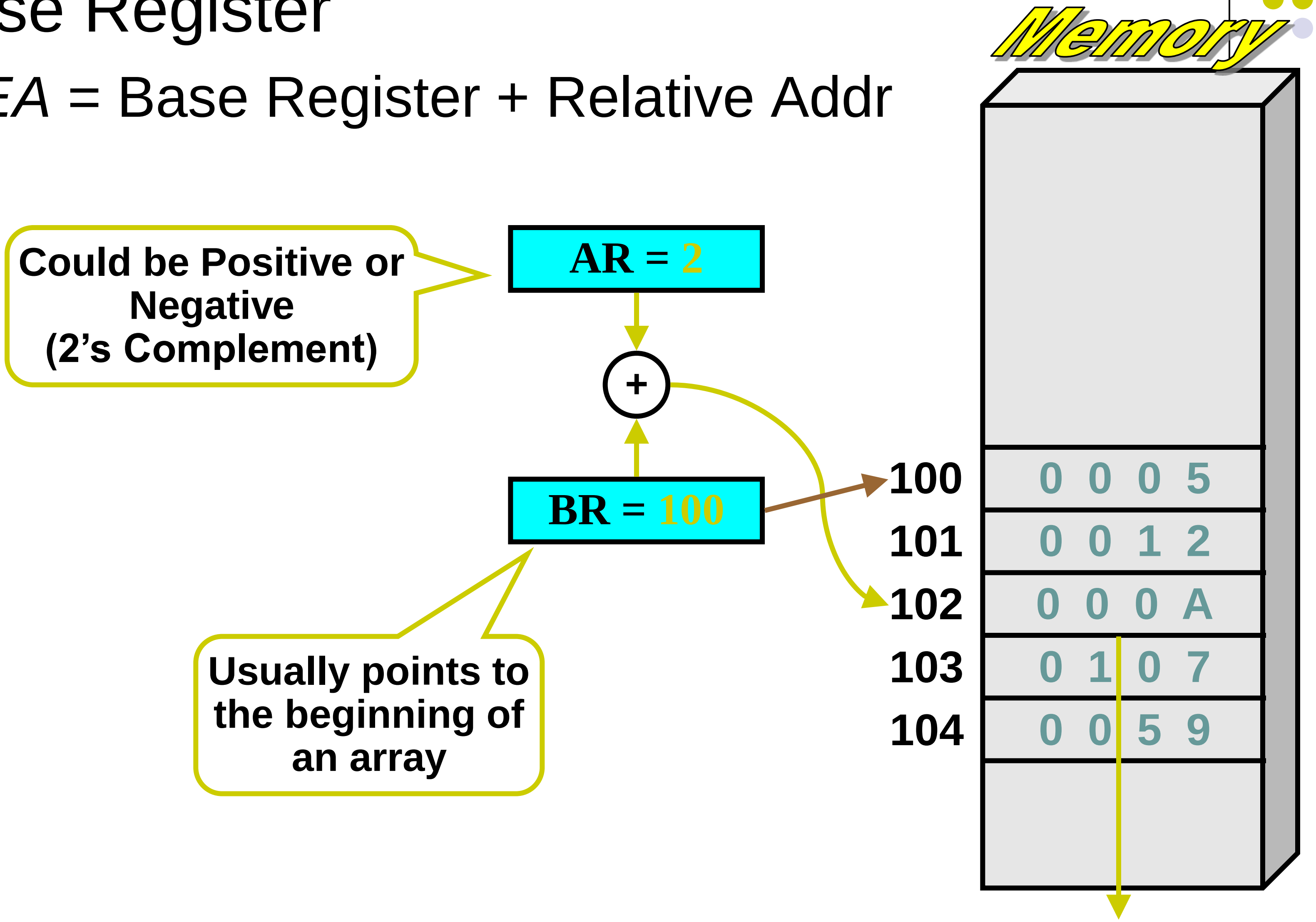


Figure 2.10 A list of students' marks.

```
         Move              R2, #LIST    Get the address LIST.
         Clear             R3
         Clear             R4
         Clear             R5
         Load              R6, N        Load the value n.
LOOP:    Load              R7, 4(R2)    Add the mark for next student's
         Add               R3, R3, R7     Test 1 to the partial sum.
         Load              R7, 8(R2)    Add the mark for that student's
         Add               R4, R4, R7     Test 2 to the partial sum.
         Load              R7, 12(R2)   Add the mark for that student's
         Add               R5, R5, R7     Test 3 to the partial sum.
         Add               R2, R2, #16  Increment the pointer.
         Subtract          R6, R6, #1   Decrement the counter.
         Branch_if_[R6]>0  LOOP         Branch back if not finished.
         Store             R3, SUM1     Store the total for Test 1.
         Store             R4, SUM2     Store the total for Test 2.
         Store             R5, SUM3     Store the total for Test 3.
```

Addressing Modes

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

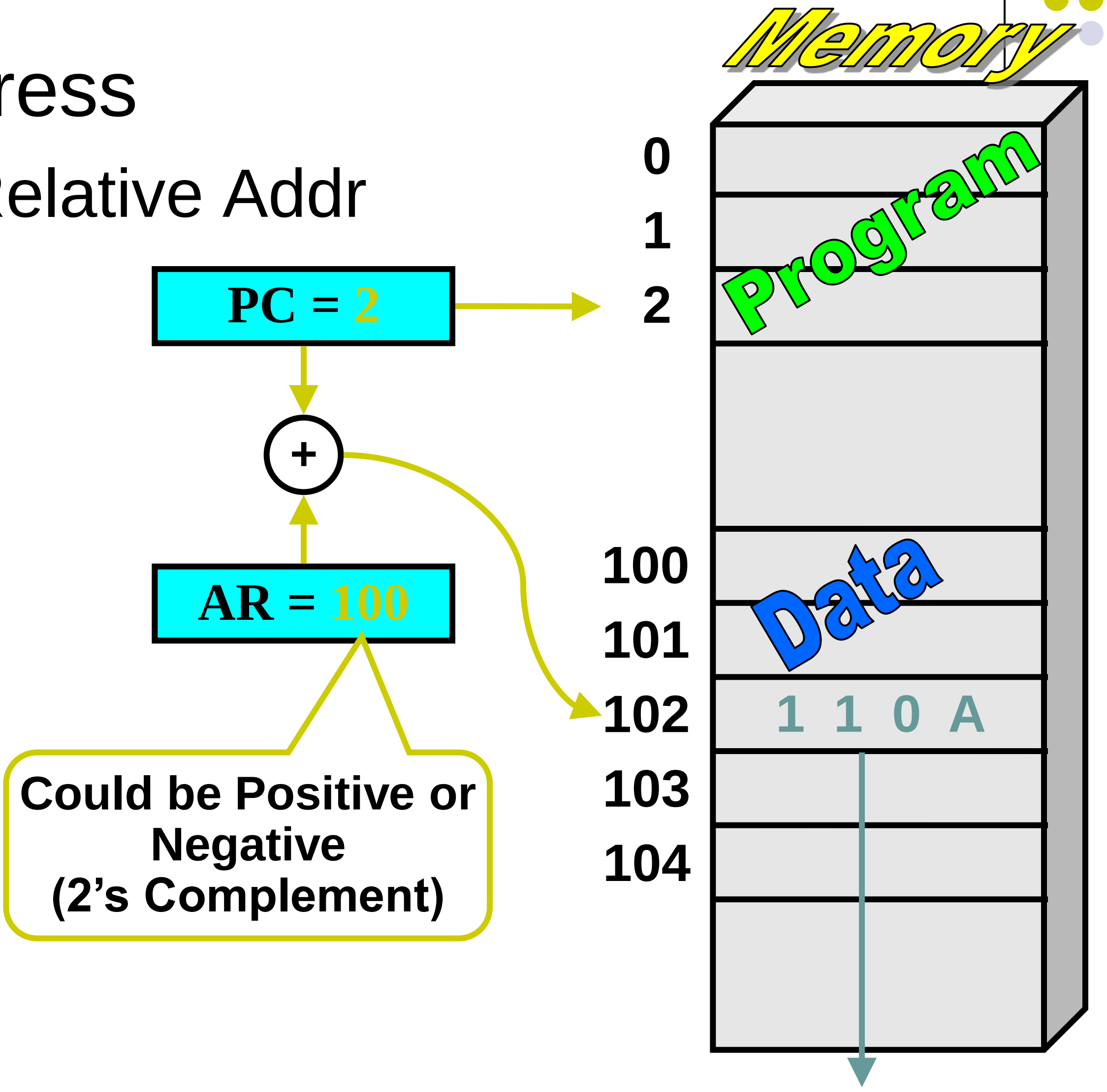

Memory
Could be Positive or Negative (2's Complement)
AR = 2
+
BR = 100
Usually points to the beginning of an array
100
101
102
103
104
0 0 0 5
0 0 1 2
0 0 0 A
0 1 0 7
0 0 5 9

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.

Addressing Modes

- Relative Address
  - *EA* = PC + 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

```
               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.

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**	**Addressingfunction**
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
Basewith index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Basewith 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*]

Subroutine Nesting and the Processor Stack

- A common programming practice, called *subroutine nesting, is to have one subroutine call* another.
- In this case, the return address of the second call is also stored in the link register, overwriting its previous contents.
- Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return address needed for this first return is the last one generated in the nested call sequence.
- That is, return addresses are generated and used in a last-in–first-out order.
- This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

- **Registers R4 and R5 contain the decimal numbers 2000 and 3000 before each of the following addressing modes is used to access a memory operand. What is the effective address (EA) in each case?**

(a) 12(R4)

(b) (R4,R5)

(c) 28(R4,R5)

(d) (R4)+

(e) -(R4)

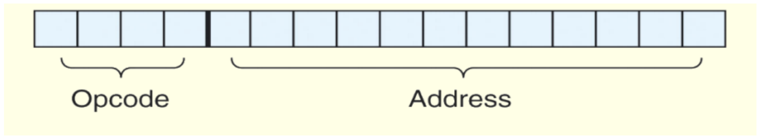
Expanding Opcode

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as HALT, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use **expanding opcodes**.

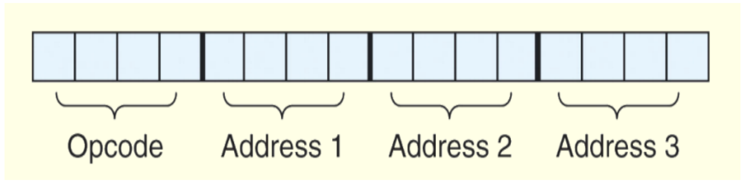
- The idea of **expanding opcodes** is to make some opcodes short, but have a means to provide longer ones when needed.
- When the opcode is short, a lot of bits are left to hold operands
  - So, we could have two or three operands per instruction
- If an instruction has no operands (such as Halt), all the bits can be used for the opcode
  - Many unique instructions are hence available
- In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands.

Example 1: Consider a machine with 16-bit instructions and 16 registers.

- The instruction format can have several structures:
  - Opcode + Memory address (such as MARIE):
    - If we have 4KB byte addressable memory we need 12 bits to specify an address location
    - The remaining 4 bits are used for the opcode: 16 instruction are hence available



  - Opcode + Registers Addresses
    - we need 4 bits to select one of the 16 available registers
    - Suppose we have 4 bits opcode, we could encode 16 different instructions with three operands each (3 x 4 bits = 12 bits).



- Is it possible to design an expanding opcode to allow the following to be encoded in a 12-bit instruction? Assume a register operand requires 3 bits and this instruction set does not allow memory addresses to be directly used in an instruction.
- 4 instructions with 3 registers
- 255 instructions with 1 register
- 16 instructions with 0 registers

- The first 4 instructions would account for $4 * 2^3 * 2^3 * 2^3 = 2^{11} =$ 2048 bit patterns.
- The next 255 instructions would account for $255 * 2^3 = 2040$ bit patterns.
- The last 16 instructions would account for 16 bit patterns.

- 12 bits allow for a total of $2^{12} = 4096$ bit patterns. If we add up what each instruction format requires, we get 2048 + 2040 + 16 = 4104. We need 4104 bit patterns to create this instruction set, but with 12 bits we only have 4096 bit patterns possible. Therefore, we cannot design an expanding opcode instruction set to meet the specified requirements.

- Given 8-bit instructions, it is possible to use expanding opcodes to allow the following to be encoded? If so, show the encoding.
- 3 instructions with two 3-bit operands
- 2 instructions with one 4-bit operand
- 4 instructions with one 3-bit operand

- First, we must determine if the encoding is possible.
  - $3 * 2^3 * 2^3 = 3 * 2^6 = 192$
  - $2 * 2^4 = 32$
  - $4 * 2^3 = 32$
- If we sum the required number of bit patterns, we get 192 + 32 + 32 = 256. 8 bits in the instruction means a total of 28 =256 bit patterns, so we have an exact match (which means the encoding is possible, but every bit pattern will be used in creating it).

- The encoding we can use is as follows:

  00 xxx xxx

  01 xxx xxx

  10 xxx xxx

  11 – escape opcode

  1100 xxxx

  1101 xxxx

  1110 – escape opcode

  1111 – escape opcode

  11100 xxx

  11101 xxx

  11110 xxx

  11111 xxx

- **Example 2:** Consider a machine with 16-bit instructions and 16 registers. And we wish to encode the following instructions:
  - 15 instructions with 3 addresses
  - 14 instructions with 2 addresses
  - 31 instructions with 1 address
  - 16 instructions with 0 addresses

Can we encode this instruction set in 16 bits?

- The first 15 instructions account for:

  $15x2^4x2^4x2^4 = 15 \times 2^{12} = 61440$ bit patterns
- The next 14 instructions account for:

  $14 \times 2^4 \times 2^4 = 15 \times 2^8 = 3584$ bit patterns
- The next 31 instructions account for:

  $31 \times 2^4 = 496$ bit patterns
- The last 16 instructions account for 16 bit patterns
- In total we need 61440 + 3584 + 496 + 16 = 65536 different bit patterns
- Having a total of 16 bits we can create $2^{16} = 65536$ bit patterns
- **We have an exact match with no wasted patterns. So our instruction set is possible.**