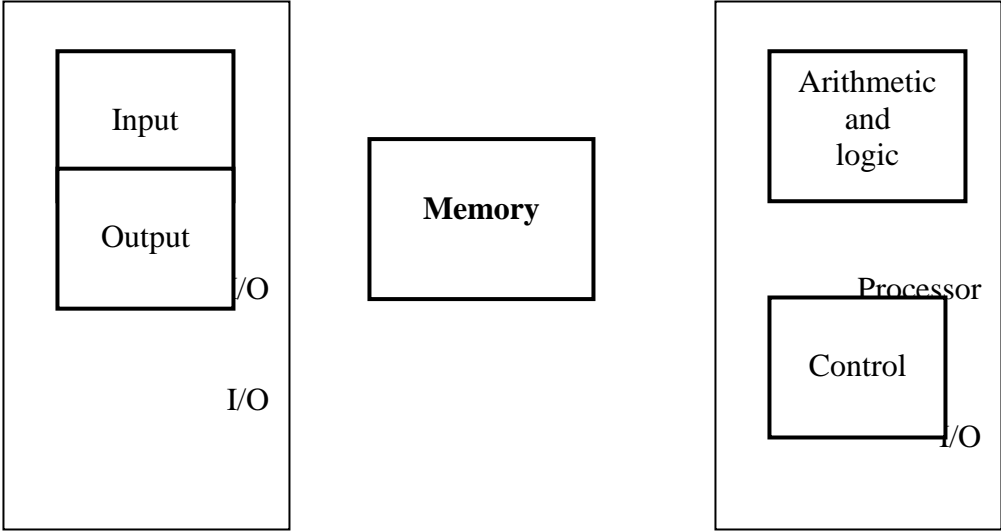


Scheme Of Evaluation
Internal Assessment Test I – September 2019

Sub:	Computer Organization and Architecture						Code:	18EC35	
Date:	09/09 /2019	Duration:	90mins	Max Marks:	50	Sem:	III	Branch:	ECE

Note: Answer Any Five Questions

Question #	Description	Marks Distribution	Max Marks
a)	Describe the basic functional units of a computer. <ul style="list-style-type: none"> • Diagram • Explanation 	3 4	7
1	<p>Functional Units</p> <p>A computer consists of five functionally independent main parts: input, memory, arithmetic and logic, output and control units as shown in fig 1.1.</p> <div style="text-align: center; margin: 20px 0;">  </div> <p style="text-align: center;">Fig 1.1 Basic functional units of a computer</p> <p>The input unit accepts coded information from human operators, from electromechanical devices such as keyboards or from other computers over digital communication line. The information received is either stored in the computer memory for later reference or immediately used by the arithmetic and logic circuitry to perform the desired operations. The processing steps are determined by the program stored in the memory. Finally the results are shown on the output unit. All of these actions are co-ordinated by the control unit. We refer to the arithmetic and logic circuits in conjunction to the control circuits as the processor and input and output units are referred to as input-output (I/O) unit.</p> <p><i>Instructions or machine instructions</i> are explicit commands that</p>		10

- Govern the transfer of information within a computer as well as between the computers and its I/O devices.
- Specify the arithmetic and logic operations to be performed.

A list of instructions that perform a task is called *program*. The computer is completely controlled by a stored program except for a possible external interruption by an operator or by I/O devices connected to the machine. Data is used to mean any digital information. Each number, character or instruction is encoded as a string of binary digits known as bits each having one of two possible values 0 or 1.

Input Unit

Computers accept the coded information through input units which read the data. The well known input device is Keyboard. When a key is pressed, the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or processor.

Memory Unit

The memory unit is used to store program and data. There are two classes of storage known as primary and secondary.

Primary memory is a fast storage that operates at electronic speeds. Programs are stored in the memory while they are executed. The memory contains large number of semiconductor storage cells each capable of storing one bit but instead are processed as groups of fixed size called words. The memory is organized so that a word can be stored or retrieved in one basic operation. A distinct address is associated to each word in the memory. Addresses are numbers that identify successive locations.

Programs must reside in the memory during execution. Instructions and data can be read out or written into the memory under the control of the processor. Memory in which any location can be reached in short and fixed amount of time after specifying its address is called random-access memory (RAM). The small, fast RAM units are called *caches*.

The additional cheaper secondary storage is used when large amount of data and many programs have to be stored particularly for information that is accessed infrequently.

Arithmetic and Logic Unit (ALU)

Any arithmetic and logic operation is initiated by bringing the required operands into the processor where the operation is performed by the ALU. When the operands are brought into the processor they are stored in high speed storage elements called *registers*. Access time to register is faster than access time to the fastest cache unit in the memory hierarchy. The control and the arithmetic logic units are many times faster than any other devices connected to a computer system.

Output Unit

The output unit is a counterpart of input unit. Its function is used the processed results to the outside world. The most familiar example of such a device is a printer.

	<p><u>Control Unit</u></p> <p>The control unit is a well defined physically separate unit that interacts with other parts of the machine. The control unit sends the control signals to other units and senses their states. Timing signals are generated by the control circuits that determine when a given action is to take place. Data transfer between memory and processor is also controlled unit through timing signals. A large set of control lines (wires) carries the signals used for timing and synchronization of events in all units.</p>			
b)	<p>Describe the basic performance equation of the computer processor.</p> <ul style="list-style-type: none"> • Equation • Explanation 	1 2	3	
	<p>The total time required to execute the program is known as <i>elapse time</i>. This is a measure of performance of entire computer system. The processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are connected by a bus</p> <p><u>Processor Clock</u></p> <p>Processor circuits are controlled by a timing signal called <i>clock</i>. The clock defines regular time intervals called <i>clock cycles</i>. To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps, such that each can be completed in one clock cycle. The length P of one clock cycle is an important parameter that affects the processor performance. Its inverse is the clock rate, $R = 1/P$ which is measured in cycles per second.</p> <p><u>Basic Performance Equation</u></p> <p>Let T be the processor time required to execute a program that has been prepared by some high level language. The compiler generates machine level object program that corresponds to source program. Assume that complete execution of the program requires the execution of N machine language instructions. Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock rate is R cycles per second, the program execution time is given by <i>basic performance equation</i>.</p> $T = \frac{N \times S}{R}$ <p>To achieve high performance, the value of T must be reduced which can be done by reducing N and S, and increasing R. The value of N is reduced if the source program is compiled in fewer machine instructions. The value of S is reduced if instructions have a smaller number of basic steps to perform or if the execution of instructions are overlapped. Using a higher-frequency clock increases the value of R which means the time required to complete a basic execution step is reduced.</p>			
2	<p>With an example and block diagram, discuss the basic operational concepts of computer.</p> <ul style="list-style-type: none"> • Diagram • Explanation 	4 6	1 0	10

Basic Operational Concepts

To perform a given 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 may be

Add LOCA, R0

This instruction adds the operand at memory location LOCA to the operand in a register in the processor, R0, and places the sum in the register R0. The original contents of location LOCA are preserved whereas those of R0 are overwritten. First the instruction is fetched from the memory into the processor. Next the operand at LOCA is fetched and added to the contents of R0. Finally the resulting sum is stored in register R0.

Transfers between memory and processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data is transferred to or from the memory. The memory and processor connection is shown in Fig 2.1

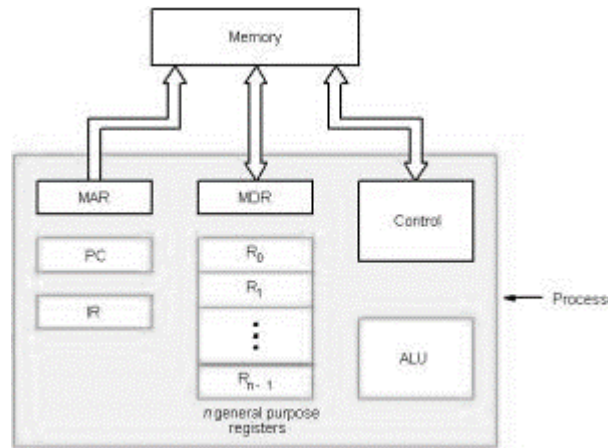


Fig 2.1 Connections between the processor and memory

The Instruction register (IR) holds the instruction that is currently being executed. Its output is available to control circuits which generate the timing signals that control various processing elements involved in executing the instruction.

The Program Counter (PC) holds the address of the next instruction to be fetched and executed. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed. MAR and MDR facilitate communication with the memory.

MAR (Memory Address Register) hold the address of the location to be accessed and MDR (Memory Data Register) contains data written into or read out of the addressed location.

If some devices require urgent servicing then they raise the interrupt signal interrupting the normal execution of the current program. The processor provides the requested service by executing the appropriate interrupt service routine.

3	a)	<p>Represent the following using binary 1's and 2's complement numbers: <i>i) (-3)₁₀, ii) (-8)₁₀, iii) (6)₁₀, iv) (-12)₁₀</i></p> <ul style="list-style-type: none"> • Conversion from decimal to 1'S AND 2's complement binary 1 mark for each. 	1 * 4	4	10
---	----	---	-------	---	----

	<p>i) -3</p> <p>1's complement:</p> <p>3 – 000011 -3 – 1111100</p> <p>2's complement:</p> <p>3 – 0000011 -3 – 1111101</p> <p>ii) -8</p> <p>1's complement:</p> <p>8 – 00001000 -8 – 11110111</p> <p>2's complement:</p> <p>8 – 00001000 -8 – 11111000</p> <p>ii)6</p> <p>1's complement:</p> <p>6 – 00000110</p> <p>2's complement:</p> <p>6- 00000110</p> <p>ii)-12</p> <p>1's complement:</p> <p>12- 00001100</p> <p>2's complement:</p> <p>12- 11110100</p>			
b)	<p>Convert the following decimal numbers to signed binary and find the result of the arithmetic operations and also comment on the status of the overflow flag:</p> <p><i>i) $(-14)_{10} - (11)_{10}$, <i>ii) $(-10)_{10} - (13)_{10}$</i></i></p> <ul style="list-style-type: none"> • Conversion from decimal to 2's complement binary. • Subtraction • Comments on Overflow flag 	0.5 2 0.5	6	
	<p><i>i) $(-14)_{10} - (11)_{10}$</i></p> <p>2's complement representation of -14</p> <p>14 – 01110 -14 – 10010</p> <p>11- 01011</p>			

	<p>Add subtrahend with 2's complement of minuend</p> $\begin{array}{r} -14 + \quad 10010 + \\ -11 \quad 10101 \\ \hline \end{array}$ <p>1] 00111 ----→ 7 The answer is incorrect, Overflow flag will be set.</p> <p>ii) $(-10)_{10} - (13)_{10}$ 2's complement representation of -10 10 - 01010 -10 - 10110</p> <p>13- 01101</p> <p>Add subtrahend with 2's complement of minuend</p> $\begin{array}{r} -10 + \quad 10110 + \\ -13 \quad 10011 \\ \hline \end{array}$ <p>1] 01001 ----→ 9 the answer is incorrect , Overflow flag will be set</p>												
4	<p>a) Explain the straight line sequencing of instructions execution of a program with an example.</p> <ul style="list-style-type: none"> • Diagram • Explanation with example 	2 4	6										
	<p><u>Instruction Execution and Straight-Line Sequence</u></p> <p>We assume computer allows one memory operand per instruction and has a number of processor registers. Fig 1.12 shows a program segment in the memory of a computer. The word length is 32 bits and the memory is byte addressable. Each instruction is 4 bytes long, the second and third instructions start at addresses $i + 4$ and $i + 8$.</p> <p>The Program Counter (PC) contains the address of the instruction to be executed next. To begin executing a program, the address of its first instruction must be placed in to PC. Then the processor control circuits use the information in the PC to fetch and execute the instructions, one at a time, in the order of increasing addresses. This is called <i>straight-line sequencing</i>.</p> <p>Executing a given instruction is a two phase-procedure. In the first phase called <i>instruction fetch</i>, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the <i>instruction register (IR)</i> in the processor. At the start of second phase called <i>instruction execute</i>, the instruction in the IR is examined to determine which operation is to be determined.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="text-align: left;">Address</th> <th style="text-align: left;">Contents</th> </tr> </thead> <tbody> <tr> <td style="text-align: left;">Begin execution here → i</td> <td style="text-align: center;"><i>Move A, R0</i></td> </tr> <tr> <td style="text-align: left;">$i + 4$</td> <td style="text-align: center;"><i>Add B, R0</i></td> </tr> <tr> <td style="text-align: left;">instruction</td> <td></td> </tr> <tr> <td style="text-align: left;">program</td> <td></td> </tr> </tbody> </table>	Address	Contents	Begin execution here → i	<i>Move A, R0</i>	$i + 4$	<i>Add B, R0</i>	instruction		program		3-	10
Address	Contents												
Begin execution here → i	<i>Move A, R0</i>												
$i + 4$	<i>Add B, R0</i>												
instruction													
program													

for the

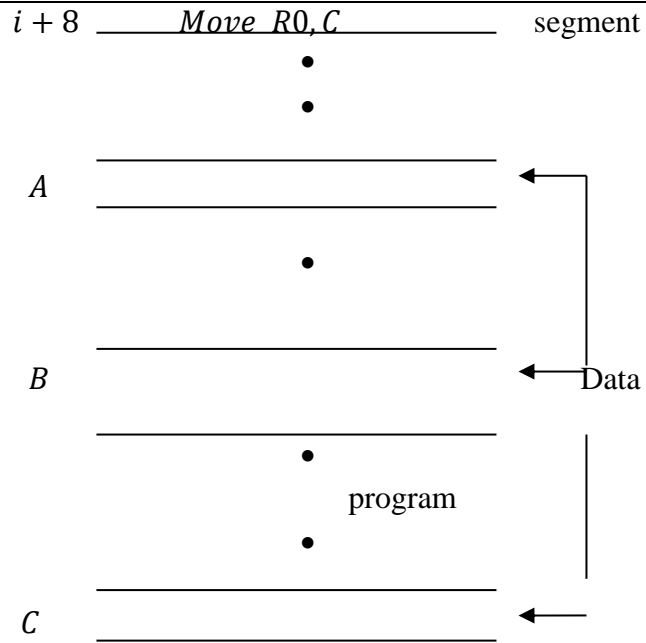


Fig 1.12: A program for $C \leftarrow [A] + [B]$

Branching

Consider a task of adding a list of n numbers. The address of the memory locations containing the n numbers are given as $NUM1, NUM2, \dots, NUMn$ and a separate *ADD* instruction is used to add each number to the contents of the register $R0$. After all numbers have been added, the result is placed in the memory location *SUM*.

Instead of using a long list of *Add* instructions, it is possible to place a single *Add* instruction in a program loop as shown in Fig 1.13. The *loop* is a straight line sequence of instructions executed as many times as needed. It starts at location *LOOP* and ends at the instruction *Branch>0*. $R1$ is used as a counter to determine the number of times loop is executed and holds the contents of the memory location N which contains the number of entries in the list n . Then, within the body of loop, the instruction

Decrement R1

Execution of the loop is repeated as long as the result of the decrement operation is greater than zero.

<i>Move N, R1</i>
<i>Clear R0</i>
Determine the address of "Next" number and add "Next" number to $R0$

	<p style="text-align: center;"> <i>Decrement R1</i> <hr/> <i>Branch > 0 LOOP</i> <hr/> <i>Move R0, SUM</i> <hr/> • <hr/> SUM N <i>n</i> <hr/> <hr/> NUM1 NUM2 <hr/> • • <hr/> NUMn <hr/> </p> <p style="text-align: center;">Fig 1.13: Using a loop to add n numbers</p> <p style="text-align: center;">A <i>conditional branch</i> instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in a normal way and the next instruction in sequential address order is fetched and executed.</p>			
b)	<p>Write an assembly language program to perform the following operation: $C = A + B$. Where C,A and B are memory operands.</p> <ul style="list-style-type: none"> • Logic • Assembly Language program 	1 3	4	
	<pre> MOVE A,R1 ADD B,R1 MOVE R1,C </pre>			
5	<p>Describe any five typical addressing modes used in a computer with examples.</p> <ul style="list-style-type: none"> • 2 marks for each addressing mode 	2 * 5	1 0	10
	<p><u>Addressing Modes</u></p> <p>The different ways in which the location of an operand is specified in an instruction is known as <i>addressing modes</i>. Variables and constants are the simplest</p>			

data types. In assembly language, a variable is represented by allocating a register or memory location to hold its value. Thus, the value can be changed as needed using appropriate instructions.

- *Register mode* – The operand is the contents of a processor register; the name of the register is given in the instruction.
- *Absolute mode* – The operand is in a memory location; the address of this location is given explicitly in the instruction.

The instruction *Move LOC, R2*

uses two modes. Processor registers are temporary storage locations where data in a register is accessed using the Register mode. Address and data constants can be represented in assembly language using the Immediate mode addressing where the operand is given explicitly in the instruction. For example, the instruction

Move 200_{immediate}, R0

Places the value 200 in register *R0*. A common convention is to use # in front of the immediate value to indicate that this value is to be used as an immediate operand. Hence we can write the instruction above in the form

Move #200, R0

Constant values are used frequently in high-level language programs. The statements $A = B + 6$ contains the constant 6. Assuming that *A* and *B* have been declared as variables and may be accessed using Absolute mode.

Move B, R1

Add #6, R1

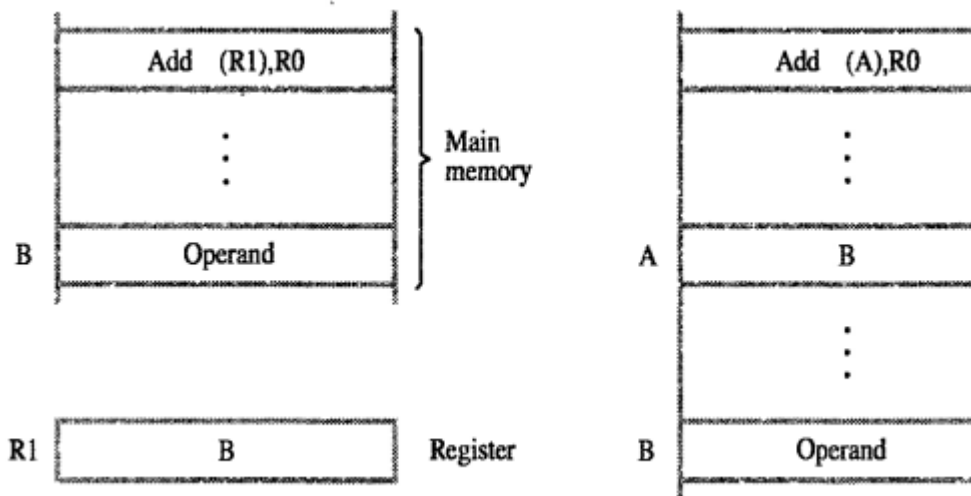
Move R1, A

Indirect Addressing mode

In indirect mode addressing, the instruction does not give the operand or the address explicitly. Instead it provides information from which the memory address of the operand can be determined. This address is referred to as *effective address* (EA) of the operand.

Indirect mode – The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

To execute the *Add* instruction in Fig 2.1a, the processor uses the value *B*, which is in the register *R1*, as the effective address of the operand. It requests a read operation from the memory to read the contents of location *B*. The value read is the desired operand, which the processor adds to the contents of register *R0*. Indirect addressing through a memory location is also possible as shown in Fig 2.1b. In this case, the processor first reads the contents of memory location *A*, then request the second read operation using the value *B* as a address to obtain the operand.



a) Through a general purpose register location

b) Through a memory location

Fig 2.1: Indirect addressing

The register or the memory location that contains the address of the operand is called a *pointer*.

Index Addressing Mode

This addressing mode provides flexibility for accessing operands and is useful in dealing with lists and arrays.

Index mode – The effective address of the operand is generated by adding a constant value to the contents of a register. This register is referred to as *index register*.

We indicate the Index mode symbolically as $X(Ri)$ where X denotes the constant value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by

$$EA = X + [Ri]$$

Fig 2.2 illustrates two ways of using Index mode. In Fig 2.2a, the index register `R1` contains the address of the memory location and the value X defines an *offset* or *displacement* from this address to the location where the operand is found.

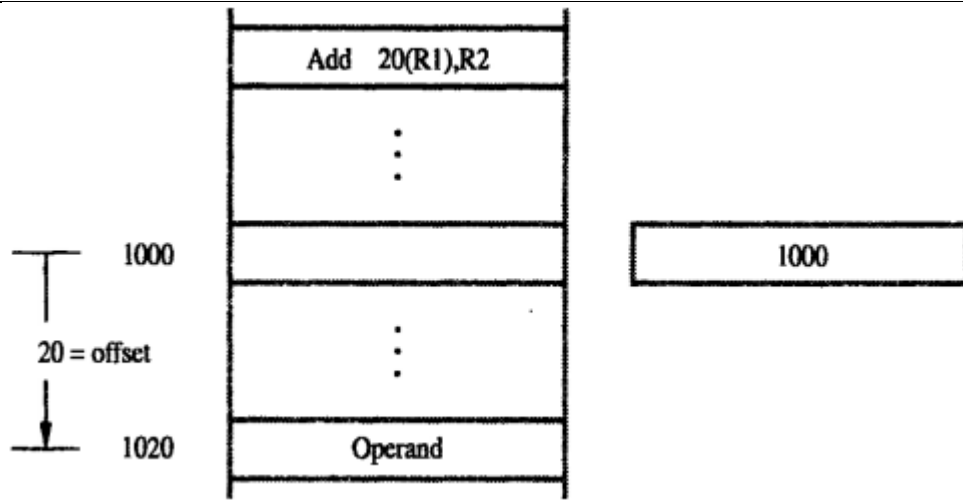


Fig 2.2 a: Offset is given as a constant

An alternate use is illustrated in Fig 2.2b. Here, the constant X corresponds to a memory address and the content of the index register defines the offset to the operand. In either case, the effective address is the sum of two values, one is given explicitly in the instruction and the other is stored in the register.

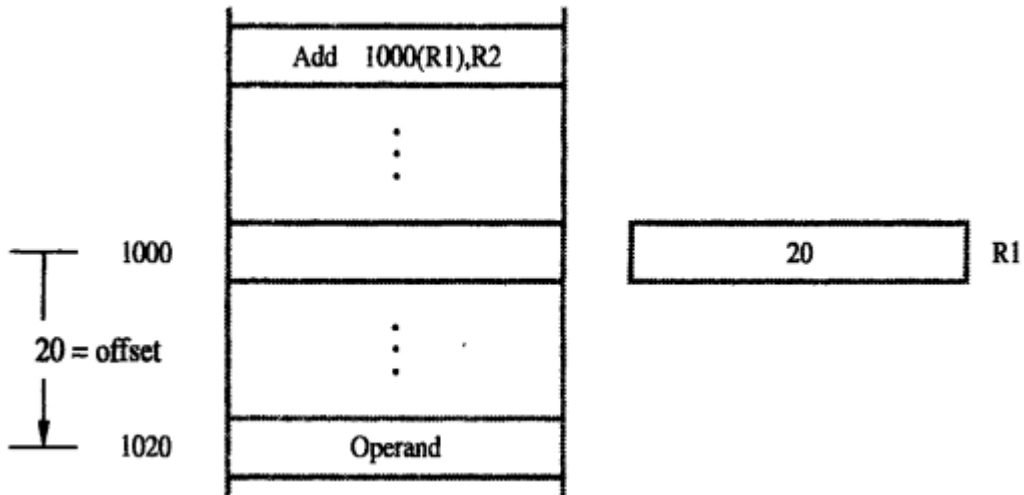


Fig 2.2b: Offset is in the register

	<p><u>Relative Addressing</u></p> <p>Here the Program Counter (PC) is used instead of a general purpose register. In <i>Relative mode</i>, the effective address is determined by the Index mode using program counter in place of general-purpose register <i>Ri</i>. It's most common use is to specify the target address in branch instructions. An instruction such as</p> <p style="text-align: center;"><i>Branch > 0 LOOP</i></p> <p>causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Suppose that Relative mode is used to generate the target branch address LOOP in the Branch instruction of the program</p> <p style="text-align: center;"> <i>LOOP: Add (R2),R0 Add #4,R2 Decrement R1 Branch > 0 LOOP</i> </p> <p>Assume that the four instructions of the loop body, starting at LOOP are located at memory locations 1000,1004,1008 and 1012. Hence the updated contents of the PC at the time of branch target address is generated will be 1016. To branch to location LOOP(1000), the offset needed is $X = -16$.</p>		
6	<p>a) Explain the following assembler directives with an example: i) EQU, ii) RESERVE, iii) DATAWORD</p> <ul style="list-style-type: none"> • Explanation for Directive one mark each <p>DATAWORD directive is used to inform the assembler to place the data in the address. N DATAWORD 30 This will allocate a Word size memory location and assign a value 30, the starting location is given the label N</p> <p>RESERVE directive declares a memory block and does not cause any data to be loaded in these locations.</p> <p>NUM RESERVE 100, this reserve 100 continuous byte locations and the starting label is NUM.</p> <p>EQU directive is used to assign a value to a label SUM EQU 200 It informs the assembler that the name SUM should be replaced by the value 200 wherever it appears in the program.</p> <p>b) Write an Assembly language program to add N numbers in an array. Use proper assembler directives to assign memory locations for program and data.</p> <ul style="list-style-type: none"> • Logic • Assembly Language program 	1 * 3	3
		2 5	7

	<pre> ORIGIN 1000 ; SOURCE DATA STARTS FROM ADDRESS 1000 SRC DATAWORD 100,120,20,30,.....,60 ; DECLARE WORD ARRAY ORIGIN 2000 ; DESTINATION AT 2000 SUM DATAWORD 0 N EQU 100 ; THE NUMBER OF ELEMENTS IN THE ARRAY ORIGIN 3000 ; PROGRAM IS STORED FROM LOCATION 3000 START MOVE #SRC,R1 ; INITIALISE POINTER MOVE N,R2 ; INITIALIZE COUNTER CLR R3 ; CLEAR TARGET REGISTER FOR RESULT BACK ADD (R1)+,R3 DEC R2 BGTZ BACK MOVE R3,SUM ; STORE THE RESULT END START </pre>		
--	---	--	--