**Q1.**

The Fetch-Execute cycle can be described in register-transfer language as follows:

MAR← [PC]

PC ←[PC] + 1; MBR ←[Memory]addressed

CIR← [MBR]

[CIR] decoded and executed

where [ ] means *contents of*.

Describe the Fetch-Execute cycle in your own words.

**(Total 6 marks)**

**Q2.**

(a)  The table lists five different quantities of memory, each measured using different units.

Place the quantities of memory into order by writing the numbers 1 to 5 in the **Position** column of the table, with 1 representing the smallest quantity and 5 representing the largest quantity.

|  |  |
| --- | --- |
| **Quantity** | **Position** |
| 3 kilobytes |   |
| 2 mebibytes |   |
| 2 bytes |   |
| 2 megabytes |   |
| 20 bits |   |

**(2)**

(b)  Convert the **hexadecimal** numbers 27 and C9 into **binary**. Then, in **binary**, add them together to work out the total. Finally, convert the total back into **hexadecimal** to give the answer.

You **must** show your working.

**(2)**

**(Total 4 marks)**

**Q3.**

**Figure 1** shows some of the internal components of a processor and how the processor is connected to the main memory. The internal connections within the processor are not shown.

**Figure 1**

****

Describe how an instruction is fetched from main memory during the **fetch stage** of the fetch-execute cycle.

Your description should cover the use of registers and buses, together with the role of main memory.

**(Total 4 marks)**

**Q4.**

**Table 1** shows the standard AQA assembly language instruction set that should be used to answer the question below.

**Table 1 – standard AQA assembly language instruction set**

|  |  |
| --- | --- |
| LDR Rd, <memory ref> | Load the value stored in the memory location specified by <memory ref> into register d. |
| STR Rd, <memory ref> | Store the value that is in register d into the memory location specified by <memory ref>. |
| ADD Rd, Rn, <operand2> | Add the value specified in <operand2> to the value in register n and store the result in register d. |
| SUB Rd, Rn, <operand2> | Subtract the value specified by <operand2> from the value in register n and store the result in register d. |
| MOV Rd, <operand2> | Copy the value specified by <operand2> into register d. |
| CMP Rn, <operand2> | Compare the value stored in register n with the value specified by <operand2>. |
| B <label> | Always branch to the instruction at position <label> in the program. |
| B<condition> <label> | Branch to the instruction at position <label> if the last comparison met the criterion specified by <condition>. Possible values for <condition> and their meanings are:    EQ: equal to        NE: not equal to    GT: greater than    LT: less than |
| AND Rd, Rn, <operand2> | Perform a bitwise logical AND operation between the value in register n and the value specified by <operand2> and store the result in register d. |
| ORR Rd, Rn, <operand2> | Perform a bitwise logical OR operation between the value in register n and the value specified by <operand2> and store the result in register d. |
| EOR Rd, Rn, <operand2> | Perform a bitwise logical XOR (exclusive or) operation between the value in register n and the value specified by <operand2> and store the result in register d. |
| MVN Rd, <operand2> | Perform a bitwise logical NOT operation on the value specified by <operand2> and store the result in register d. |
| LSL Rd, Rn, <operand2> | Logically shift left the value stored in register n by the number of bits specified by <operand2> and store the result in register d. |
| LSR Rd, Rn, <operand2> | Logically shift right the value stored in register n by the number of bits specified by <operand2> and store the result in register d. |
| HALT | Stops the execution of the program. |

**Labels**: A label is placed in the code by writing an identifier followed by a colon (:). To refer to a label the identifier of the label is placed after the branch instruction.

**Interpretation of <operand2>**

<operand2> can be interpreted in two different ways, depending on whether the first character is a # or an R:

•   # – use the decimal value specified after the #, eg #25 means use the decimal value 25

•   Rm – use the value stored in register m, eg R6 means use the value stored in register 6

The available general purpose registers that the programmer can use are numbered 0–12

Write an assembly language program to encrypt a single character using the Caesar cipher. The character to be encrypted is represented using a character set consisting of 26 characters with character codes 0–25. The output of the process should be the character code of the encrypted character.

The assembly language instruction set that you should use to write the program is listed in **Table 1**.

**Table 2** shows the character codes and the characters they represent.

**Table 2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code** | **Character** |   | **Code** | **Character** |   | **Code** | **Character** |
| 0 | A |   | 9 | J |   | 18 | S |
| 1 | B |   | 10 | K |   | 19 | T |
| 2 | C |   | 11 | L |   | 20 | U |
| 3 | D |   | 12 | M |   | 21 | V |
| 4 | E |   | 13 | N |   | 22 | W |
| 5 | F |   | 14 | O |   | 23 | X |
| 6 | G |   | 15 | P |   | 24 | Y |
| 7 | H |   | 16 | Q |   | 25 | Z |
| 8 | I |   | 17 | R |   |   |   |

•   Memory location 100 contains the character code to be encrypted, which is in the range

0–25

•   Memory location 101 contains an integer key to be used for encryption, which is in the range 0–25

•   The program should store the character code of the encrypted character in memory location 102

**(Total 4 marks)**

**Q5.**

The pseudo-code in **Figure 1** shows one method for carrying out encryption of a single character using the Caesar Cipher.

If the character to be encrypted is a capital letter, then the encrypted character will be shifted along the alphabet by the number of positions specified by the key. If the character is not a capital letter, then the encrypted character is set to be equal to the original character.

The pseudo-code assumes that the letter to encrypt is stored using the Unicode UTF-8 encoding method, for which the values of capital letters (in decimal) are shown in **Table 1**.

**Figure 1**

 IF characterCode >= 65 AND characterCode <= 90 THEN

    encryptedCode ← characterCode + keyValue

    IF encryptedCode > 90 THEN

       encryptedCode ← encryptedCode − 26

    ENDIF

 ELSE

    encryptedCode ← characterCode

 ENDIF

 **Table 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | 65 |   | N | 78 |
| B | 66 |   | O | 79 |
| C | 67 |   | P | 80 |
| D | 68 |   | Q | 81 |
| E | 69 |   | R | 82 |
| F | 70 |   | S | 83 |
| G | 71 |   | T | 84 |
| H | 72 |   | U | 85 |
| I | 73 |   | V | 86 |
| J | 74 |   | W | 87 |
| K | 75 |   | X | 88 |
| L | 76 |   | Y | 89 |
| M | 77 |   | Z | 90 |

**Figure 2** shows an incomplete assembly language program that has been written to implement the pseudo-code algorithm shown in **Figure 1**. The assembly language instruction set that has been used to write the program is listed in **Table 2** in part (c).

The symbols  and  indicate the positions of missing lines of code.



(a)  By analysing the assembly language program in **Figure 2**, explain the purpose for which the registers R1, R2 and R3 have been used.

|  |  |
| --- | --- |
| **Register** | **Purpose** |
| R1 |   |
| R2 |   |
| R3 |   |

**(2)**

(b)  On **Figure 2**, write the assembly language instruction that is missing from position .

**(1)**

(c)  On **Figure 2**, write the assembly language instructions that are missing from position .

**Table 2 – Standard AQA assembly language instruction set**

|  |  |
| --- | --- |
| LDR Rd, <memory ref> | Load the value stored in the memory location specified by <memory ref> into register d. |
| STR Rd, <memory ref> | Store the value that is in register d into the memory location specified by <memory ref>. |
| ADD Rd, Rn, <operand2> | Add the value specified in <operand2> to the value in register n and store the result in register d. |
| SUB Rd, Rn, <operand2> | Subtract the value specified by <operand2> from the value in register n and store the result in register d. |
| MOV Rd, <operand2> | Copy the value specified by <operand2> into register d. |
| CMP Rn, <operand2> | Compare the value stored in register n with the value specified by <operand2>. |
| B <label> | Always branch to the instruction at position <label> in the program. |
| B<condition> <label> | Branch to the instruction at position <label> if the last comparison met the criterion specified by <condition>. Possible values for <condition> and their meanings are:  EQ: equal to    NE: not equal to  GT: greater than   LT: less than |
| AND Rd, Rn, <operand2> | Perform a bitwise logical AND operation between the value in register n and the value specified by <operand2> and store the result in register d. |
| ORR Rd, Rn, <operand2> | Perform a bitwise logical OR operation between the value in register n and the value specified by <operand2> and store the result in register d. |
| EOR Rd, Rn, <operand2> | Perform a bitwise logical XOR (exclusive or) operation between the value in register n and the value specified by <operand2> and store the result in register d. |
| MVN Rd, <operand2> | Perform a bitwise logical NOT operation on the value specified by <operand2> and store the result in register d. |
| LSL Rd, Rn, <operand2> | Logically shift left the value stored in register n by the number of bits specified by <operand2> and store the result in register d. |
| LSR Rd, Rn, <operand2> | Logically shift right the value stored in register n by the number of bits specified by <operand2> and store the result in register d. |
| HALT | Stops the execution of the program. |

**Labels**: A label is placed in the code by writing an identifier followed by a colon (:). To refer to a label, the identifier of the label is placed after the branch instruction.

**Interpretation of <operand2>**

<operand2> can be interpreted in two different ways, depending on whether the first character is a # or an R:

•   # – Use the decimal value specified after the #, eg #25 means use the decimal value 25.

•   Rm – Use the value stored in register m, eg R6 means use the value stored in register 6.

The available general purpose registers that the programmer can use are numbered 0 to 12.

**(3)**

**(Total 6 marks)**

**Q6.**

Some of the assembly language instructions supported by a simple microprocessor are:

|  |
| --- |
| **Assembly Language** |
| STORE |
| LOAD |
| ADD |

Examples of the use of these assembly language instructions are:

|  |  |
| --- | --- |
| STORE  5 | Copy the contents of the accumulator into memory location 5 |
| LOAD   5 | Copy the contents of memory location 5 into the accumulator |
| ADD    2 | Add the contents of memory location 2 to the current contents of the accumulator, storing the result in the accumulator |

(a)     Write into the table below the opcode and the operand parts of the following instruction.

LOAD    5

|  |  |
| --- | --- |
| **Operand** |   |
| **Opcode** |   |

**(1)**

(b)     Write an assembly language program, using the instructions given above, that adds the contents of memory locations 7, 8 and 3, storing the answer in memory location 21.

**(3)**

**(Total 4 marks)**

**Q7.**

Describe the purpose of start and stop bits in asynchronous data transfer.

**(Total 2 marks)**