AQA

AS
COMPUTER SCIENCE
7516/1

Paper 1

Mark scheme

June 2023

Version: 1:0 Final

236A7516/1/MS

Mark schemes are prepared by the Lead Assessment Writer and considered, together with the relevant questions, by a panel of subject teachers. This mark scheme includes any amendments made at the standardisation events which all associates participate in and is the scheme which was used by them in this examination. The standardisation process ensures that the mark scheme covers the students' responses to questions and that every associate understands and applies it in the same correct way. As preparation for standardisation each associate analyses a number of students' scripts. Alternative answers not already covered by the mark scheme are discussed and legislated for. If, after the standardisation process, associates encounter unusual answers which have not been raised they are required to refer these to the Lead Examiner.

It must be stressed that a mark scheme is a working document, in many cases further developed and expanded on the basis of students' reactions to a particular paper. Assumptions about future mark schemes on the basis of one year's document should be avoided; whilst the guiding principles of assessment remain constant, details will change, depending on the content of a particular examination paper.

Further copies of this mark scheme are available from aqa.org.uk

The following annotation is used in the mark scheme:

; - means a single mark
// - means alternative response
/ - means an alternative word or sub-phrase
A. - means acceptable creditworthy answer
R. - means reject answer as not creditworthy
NE. - means not enough
I. - means ignore
DPT. - means "Don't penalise twice". In some questions a specific error made by a candidate, if repeated, could result in the loss of more than one mark. The **DPT** label indicates that this mistake should only result in a candidate losing one mark, on the first occasion that the error is made. Provided that the answer remains understandable, subsequent marks should be awarded as if the error was not being repeated.

Level of response marking instructions

Level of response mark schemes are broken down into levels, each of which has a descriptor. The descriptor for the level shows the average performance for the level. There are marks in each level.

Before you apply the mark scheme to a student's answer read through the answer and annotate it (as instructed) to show the qualities that are being looked for. You can then apply the mark scheme.

Step 1 Determine a level

Start at the lowest level of the mark scheme and use it as a ladder to see whether the answer meets the descriptor for that level. The descriptor for the level indicates the different qualities that might be seen in the student's answer for that level. If it meets the lowest level then go to the next one and decide if it meets this level, and so on, until you have a match between the level descriptor and the answer. With practice and familiarity you will find that for better answers you will be able to quickly skip through the lower levels of the mark scheme.

When assigning a level you should look at the overall quality of the answer and not look to pick holes in small and specific parts of the answer where the student has not performed quite as well as the rest. If the answer covers different aspects of different levels of the mark scheme you should use a best fit approach for defining the level and then use the variability of the response to help decide the mark within the level, ie if the response is predominantly level 3 with a small amount of level 4 material it would be placed in level 3 but be awarded a mark near the top of the level because of the level 4 content.

Step 2 Determine a mark

Once you have assigned a level you need to decide on the mark. The descriptors on how to allocate marks can help with this. The exemplar materials used during standardisation will help. There will be an answer in the standardising materials which will correspond with each level of the mark scheme. This answer will have been awarded a mark by the Lead Examiner. You can compare the student's answer with the example to determine if it is the same standard, better or worse than the example. You can then use this to allocate a mark for the answer based on the Lead Examiner's mark on the example.

You may well need to read back through the answer as you apply the mark scheme to clarify points and assure yourself that the level and the mark are appropriate.

Indicative content in the mark scheme is provided as a guide for examiners. It is not intended to be exhaustive and you must credit other valid points. Students do not have to cover all of the points mentioned in the Indicative content to reach the highest level of the mark scheme.

An answer which contains nothing of relevance to the question must be awarded no marks.

Examiners are required to assign each of the candidates' responses to the most appropriate level according to **its overall quality**, then allocate a single mark within the level. When deciding upon a mark in a level examiners should bear in mind the relative weightings of the assessment objectives

eg
In question **15.1**, the marks available for the AO3 elements are as follows:

AO3 (design) – 3 marks
AO3 (programming) – 9 marks
Where a candidate's answer only reflects one element of the AO, the maximum mark they can receive will be restricted accordingly.

Section A

<table>
<tr><th colspan="2">Qu</th><th colspan="2">Marks</th></tr>
<tr><td>01</td><td></td><td colspan="2">6 marks for AO2 (analyse)
<table>
<tr><th>Event / State</th><th>Label(s): (A) to (I), (X) to (Z)</th></tr>
<tr><td>Alarm bell ringing mode</td><td>Y</td></tr>
<tr><td>Alert mode</td><td>Z</td></tr>
<tr><td>Detect movement</td><td>F</td></tr>
<tr><td>Enter correct code</td><td>C, D, E</td></tr>
<tr><td>Enter incorrect code</td><td>B, H, G</td></tr>
<tr><td>Sensing mode</td><td>X</td></tr>
<tr><td>Switch on</td><td>A</td></tr>
<tr><td>10 second delay elapsed</td><td>I</td></tr>
</table>
1 mark per two correct labels

R. any labels used more than once
R. more than 3 labels in “Enter correct code” or “Enter incorrect code”

Max 5 if any errors</td><td>6</td></tr>
</table>

02		**5 marks for AO2 (apply)** **1 mark** for area A correct; **1 mark** for area B correct; **1 mark** for area C correct; **1 mark** for area D correct; **1 mark** for area E correct; Award a mark if the values in an area are correct regardless of which row they are on so long as they are in the correct overall sequence in a column. **I.** duplicated values instead of blanks **Max 4** if any errors	**5**

X	Y	N	Numbers			
			[0]	[1]	[2]	[3]
			45	19	62	12
1	0	19				
				45		**A**
	-1		19			**B**
2	1	62				**C**
3	2	12				62
	1				45	**D**
	0			19		
	-1		12			**E**

03		**2 marks for AO1 (knowledge)** Global variable can be used anywhere in program // global variable is declared in outmost block / outside subroutines; Local variable can only be used in the block / subroutine in which it is declared; ***Alternative answer 1:*** Local variables only use memory when the program block they are in is executing; global variables use memory the entire time the program is executing; ***Alternative answer 2:*** Local variables only exist when the program block they are in is executing; global variables exist the entire time the program is running; **A.** local variables are stored on a stack / in a stack frame; global variables are generally stored elsewhere in the memory;	**2**
04		**2 marks for AO1 (understand)** a sequence of steps (to complete a task); **R**. set that always terminates / runs in finite time;	**2**
05	**1**	**9 marks for AO3 (programming)** **Mark as follows:** 1) Correct variable declarations for `Number, X, Count, Multi`; **Note to examiners** If a language allows variables to be used without explicit declaration (eg Python) then this mark should be awarded if the correct variables exist in the program code and the first value they are assigned is of the correct data type. 2) Correct prompt `"Enter an integer greater than 1: "` and `Number` assigned integer value entered by user; 3) Correct initialisation of `X` and `Count` before outer `WHILE` loop; 4) Correct outer `WHILE` loop with syntax allowed by the programming language and correct condition for termination of the outer loop; 5) Correct assignment of `Multi` in outer loop; 6) Correct inner `WHILE` loop syntax allowed by the programming language and correct condition for termination of the loop; 7) `IF` statement with correct condition and output within inner loop; 8) Correct incrementation of `Count` and correct assignment to `Multi` and `Number` within inner `WHILE` loop; 9) Correct assignments of `X` in outer loop; **I.** minor differences in case and spelling **Max 8** if code does not function correctly	**9**

05	**2**	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		Must match code from **05.1**, including prompts on screen capture matching those in code.
		Code for **05.1** must be sensible.
		
		Screen capture showing:
		'23' being entered and '23' displayed followed, by '1'
		'25' being entered and the message '5' displayed, followed by '2'
		'1260' being entered and '2 3 5 7' displayed, followed by '6'
		
		(Accept on same or separate lines)
		
		`Enter a number greater than 1: 23`
		`23`
		`1`
		`>>>`
		`Enter a number greater than 1: 25`
		`5`
		`2`
		`>>>`
		`Enter a number greater than 1: 1260`
		`2`
		`3`
		`5`
		`7`
		`6`
		`>>>`
		
		Alternative:
		
		`Enter a number greater than 1: 23`
		`23`
		`1`
		`>>>`
		`Enter a number greater than 1: 25`
		`5`
		`2`
		`>>>`
		`Enter a number greater than 1: 1260`
		`2  3  5  7`
		`6`
		`>>>`
			1

Section B

Qu		Marks	
06	**1**	**Mark is for AO1 (understand)** `FileExists / Finished`; **A.** `OpCodeExists`, `Found` (Pascal only) **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	**1**
06	**2**	**Mark is for AO1 (understand)** `ConvertToDecimal`; **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	**1**
06	**3**	**Mark is for AO1 (understand)** `GetMenuOption`; **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	**1**
06	**4**	**Mark is for AO1 (understand)** `LoadFile / PassTwo`; **A.** `Readline` (Java only) **R.** if any additional code **R.** if spelt incorrectly **I.** case & spacing	**1**

<table>
<tr><td>07</td><td>1</td><td>Mark is for AO1 (understand)

<code>AssemblerInstruction</code>;

A. <code>Memory</code> / <code>SymbolTable</code>;
R. if any additional code
R. if spelt incorrectly
I. case & spacing

Max 1</td><td>1</td></tr>
<tr><td>07</td><td>2</td><td>Mark is for AO1 (understand)

<code>SourceCode</code> / <code>Registers</code> / <code>OpCodeValues</code>;

A. <code>Memory</code> / <code>SymbolTable</code>; (if neither given in 07.1)
R. if any additional code
R. if spelt incorrectly
I. case & spacing

Max 1</td><td>1</td></tr>
<tr><td>08</td><td></td><td>Mark is for AO1 (knowledge)

Representational Abstraction;

A. Abstraction</td><td>1</td></tr>
<tr><td>09</td><td></td><td>5 marks are for AO2 (analyse)

<table>
<tr><th>Proposed error message</th><th>Error code</th></tr>
<tr><td>Duplicate label found</td><td>3</td></tr>
<tr><td>File not found</td><td>1</td></tr>
<tr><td>No assembled code to run</td><td>10</td></tr>
<tr><td>No source code to display</td><td>7</td></tr>
<tr><td>Unknown opcode</td><td>5</td></tr>
</table>

1 mark for each correct error code
R. if more than one error code assigned to an error message</td><td>5</td></tr>
</table>

10		**2 marks for AO2 (analyse)** `Operand` is not empty / an empty string; `Operand` is not an existing label // `Operand` is not in the symbol table; **NE**. invalid operand `Operand` is not (the string representation of) a number; **Max 2**	**2**

11	**1**	**2 marks for AO2 (analyse)** Check each character in the line / operand; **A**. iterate over each character in the line / operand Store the position of an '*' // store position of comment symbol if there is one // store position of the start of a comment;	**2**

11	**2**	**2 marks for AO2 (analyse)** If there is an '*' // if there is a comment (symbol); Only use the characters to left of '*' // remove the comment (from the string);	**2**

11	**3**	**2 marks for AO2 (analyse)** When code encounters a '*' // When code encounters a comment (symbol); Can **stop** iteration when */comment is encountered // no need to examine to the end of the string; ***Alternative answer:*** If there are multiple asterisks, it finds the first one (rather than the last one); Additional asterisks within the comment will not affect assembly; **R.** a variable (`ThisPosition`) to hold the position wouldn't be needed	**2**

Section C

Qu		Marks	
12	**1**	**4 marks for AO3 (programming)** **Mark as follows:** 1) Add required parameter to subroutine call in `Execute`; 2) Add required parameter to `ExecuteSKP` definition; 3) Add `1` to `Registers[ACC]`; 4) Update status register (by calling `SetFlags` with correct parameters); **Max 3** if any errors	**4**

12	2	**Mark is for AO3 (evaluate)** ****** SCREEN CAPTURE ****** Must match code from **12.1**, including prompts on screen capture matching those in code. Code for **12.1** must be sensible. Screen capture showing (values changing from Frame 0 to Frame 5 shown highlighted): `Enter your choice: R` `****** Frame 0 ************************************************` `*` `*  Memory     Location  Label  Op   Operand Comment` `*  Contents                    Code` `* JMP 1	0 |` `* LDA# 3	1	LDA# 3 * test negative` `* SUB 10	2	SUB NUM1` `* SKP 0	3	SKP` `* STA 11	4	STA FINAL` `* HLT 0	5	HLT` `* 0	6 |` `* 0	7 |` `* 0	8 |` `* 0	9 |` `* 5	10	NUM1: 5` `* 0	11	FINAL: 0` `*` `* PC: 0 ACC: 0 TOS: 20` `* Status Register: ZNV` `* 100` `**`  `****** Frame 5 **` `* Current Instruction Register: STA 11` `*` `* Memory Location Label Op Operand Comment` `* Contents Code` `* JMP 1	0 |` `* LDA# 3	1	LDA# 3 * test negative` `* SUB 10	2	SUB NUM1` `* SKP 0	3	SKP` `* STA 11	4	STA FINAL` `* HLT 0	5	HLT` `* 0	6 |` `* 0	7 |` `* 0	8 |` `* 0	9 |` `* 5	10	NUM1: 5` `* -1	11	FINAL: 0` `*` `* PC: 5 ACC: -1 TOS: 20` `* Status Register: ZNV` `* 010` `**` `Execution terminated`	1

Qu		Marks	
13	1	**5 marks for AO3 (programming)** **Mark as follows:** 1) Check for non-integer input; 2) Check within valid lower boundary; 3) Check within valid upper boundary; 4) At least 2 correct checks will be repeated until valid data is input at which point the loop exits; 5) Output suitable error message(s) under appropriate circumstances based upon at least 2 correct checks; **R.** if message is displayed when it should not be **Max 4** if any errors	**5**

13	2	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		Must match code from **13.1**, including prompts on screen capture matching those in code.
		Code for **13.1** must be sensible.
		
		Screen capture showing:
		
		`Enter your choice: E`
		`Enter line number of code to edit: Q`
		`Not a valid number`
		`Enter line number of code to edit: 22`
		`Not a valid line number`
		`Enter line number of code to edit: 0`
		`Not a valid line number`
		`Enter line number of code to edit: 2`
		`  SUB  NUM1`
		`E - Edit this line`
		`C - Cancel edit`
		`Enter your choice:`
			1

14	**1**	**2 marks for AO3 (design) and 2 marks for AO3 (programming)** **Marking guidance:** **Evidence of AO3 design – 2 points:** Evidence of design to look for in response: 1) Check in `ExecuteJSR` that stack does not overwrite instruction / data; 2) Recognise that instructions for `JSR` should only be executed if no error; **Note:** AO3 (design) points are for selecting appropriate techniques to use to solve the problem, so should be credited whether the syntax of programming language statements is correct or not and regardless of whether the solution works. **Evidence of AO3 programming – 2 points:** Evidence of programming to look for in response: 3) Correct value for number of program lines passed into subroutine // check that memory location pointed to by `TOS` is empty; 4) `ReportRunTimeError` called with suitable message in appropriate place; **Max 3** if code does not function correctly	**4**

14	**2**	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		Must match code from **14.1**, including prompts on screen capture matching those in code.
		Code for **14.1** must be sensible.
		
		`****** Frame 7 **********`
		`*  Current Instruction Register:  JSR 7`
		`Run time error:  Memory Address Error`
		`Stack contents:`
		` ----`
		`|  3 |`
		`| 14 |`
		` ----`
		`Execution terminated`
			1

15	**1**	**3 marks for AO3 (design) and 9 marks for AO3 (programming)**	**12**

Level	Description	Mark Range
3	A line of reasoning has been followed to arrive at a logically structured working or almost fully working programmed solution. All of the appropriate design decisions have been taken. The last line of source code may not be displayed correctly (if last line not moved due to exclusive boundary).	9–12
2	There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work. The subroutine `EditSourceCode` has been amended and has some added functionality.	5–8
1	An attempt has been made to amend the subroutine `EditSourceCode`. Some appropriate programming statements have been written. There is little evidence to suggest that a line of reasoning has been followed or that the solution has been designed. The statements written may or may not be syntactically correct and the subroutines will have very little or none of the extra required functionality. It is unlikely that any of the key design elements of the task have been recognised.	1–4

Marking guidance:

Evidence of AO3 design – 3 marks:

Evidence of design to look for in response:

1) Adjust the number of lines stored in `SourceCode` (ie update `SourceCode[0]`)
2) Loop through program lines consecutively or equivalent
3) Move program lines after specified location in `SourceCode`

Note: AO3 (design) points are for selecting appropriate techniques to use to solve the problem, so should be credited whether the syntax of programming language statements is correct or not and regardless of whether the solution works.

A. design evidence in option `D` or `I` code.

Evidence of AO3 programming – 9 marks:

Evidence of programming to look for in response:

4) Insert `D` / `I` / both in addition to existing options in the menu and add `D` / `I` / both to conditions of `WHILE` loop
5) Add selection to test for option `D` / `I` / both after `WHILE` loop
6) Use correct range to loop through program lines in both options, `D` and `I`
7) Correctly adjust the number of lines stored in `SourceCode` in both options, `D` and `I`

		8) Within loop, move line referenced by loop counter one location in correct direction in option `D` 9) Within loop, move line referenced by loop counter one location in correct direction in option `I` 10) For option `I` get user input of new line 11) For option `I` insert new line if there is space, otherwise display error message 12) Insert line entered by user in correct row of `SourceCode` **Max 11** if code does not function correctly	

15	**2**	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		Must match code from **15.1**, including prompts on screen capture matching those in code.
		Code for **15.1** must be sensible.
		
		Screen capture showing (for ease of reference inserted line highlighted):
		
		`Enter your choice: E`
		`Enter line number of code to edit: 10`
		`SKP`
		`E - Edit this line`
		`D - Delete this line`
		`I - Insert a new line above this line`
		`C - Cancel edit`
		`Enter your choice: D`
		
		`0 12`
		`1 NUM1: 2`
		`2 NUM2: 5`
		`3 NUM3: -1`
		`4 NUM4: 125`
		`5 START: LDA NUM1 * test while loop`
		`6 WHILE: CMP# 12`
		`7 BEQ WEND`
		`8 ADD NUM2`
		`9 JMP WHILE`
		`10 WEND: STA NUM3`
		`11 ADD NUM4`
		`12 HLT`
			1

15	**3**	**Mark is for AO3 (evaluate)**
		
		****** SCREEN CAPTURE ******
		Must match code from **15.1**, including prompts on screen capture matching those in code.
		Code for **15.1** must be sensible.
		
		Screen capture showing (for ease of reference inserted line highlighted):
		
		`Enter your choice: E`
		`Enter line number of code to edit: 4`
		`       STA  FINAL`
		`E - Edit this line`
		`D - Delete this line`
		`I - Insert a new line above this line`
		`C - Cancel edit`
		`Enter your choice: I`
		`Enter the new line:`
		`LABEL: SKP`
		
		`0 12`
		`1         LDA# 3       * test negative`
		`2         SUB  NUM1`
		`3         SKP`
		`4 LABEL: SKP`
		`5         STA  FINAL`
		`6         HLT`
		`7`
		`8`
		`9`
		`10`
		`11  NUM1:      5`
		`12 FINAL:      0`
			1

VB.Net

05 | **1** | **9**

```
Console.Write("Enter an integer greater than 1: ") 'MP2
Dim Number As Integer = Console.ReadLine()
Dim X As Integer = 2          'MP1
Dim Count As Integer = 0      'MP3
Dim Multi As Boolean
While Number > 1              'MP4
  Multi = False               'MP5
  While (Number Mod X) = 0    'MP6
    If Not Multi Then         'MP7
      Console.WriteLine(X)
    End If
    Count = Count + 1
    Multi = True
    Number = Number \ X       'MP8
  End While
  X = X + 1                   'MP9
End While
Console.WriteLine(Count)
```

12 | **1** | **4**

```
Sub ExecuteSKP(Registers() As Integer) ' MP2
  Registers(ACC) += 1    'MP3
  SetFlags(Registers(ACC), Registers)  'MP4
End Sub

Sub Execute(ByVal SourceCode() As String, ByVal Memory() As
AssemblerInstruction)
    Dim Registers() As Integer = {0, 0, 0, 0, 0}
    SetFlags(Registers(ACC), Registers)
    Registers(PC) = 0
    Registers(TOS) = HI_MEM
    Dim FrameNumber As Integer = 0
    DisplayFrameDelimiter(FrameNumber)
    DisplayCurrentState(SourceCode, Memory, Registers)
    Dim OpCode As String = Memory(Registers(PC)).OpCode
    While OpCode <> "HLT"
      FrameNumber += 1
      Console.WriteLine()
      DisplayFrameDelimiter(FrameNumber)
      Dim Operand As Integer = Memory(Registers(PC)).OperandValue
      Console.WriteLine($"*  Current Instruction Register:  {OpCode}
{Operand}")
      Registers(PC) += 1
      Select Case OpCode
        Case "LDA"
          ExecuteLDA(Memory, Registers, Operand)
        Case "STA"
          ExecuteSTA(Memory, Registers, Operand)
        Case "LDA#"
          ExecuteLDAimm(Registers, Operand)
        Case "ADD"
          ExecuteADD(Memory, Registers, Operand)
        Case "JMP"
          ExecuteJMP(Registers, Operand)
        Case "JSR"
          ExecuteJSR(Memory, Registers, Operand)
        Case "CMP#"
          ExecuteCMPimm(Registers, Operand)
        Case "BEQ"
          ExecuteBEQ(Registers, Operand)
```

		Case "SUB"
		ExecuteSUB(Memory, Registers, Operand)
		Case "SKP"
		ExecuteSKP(**Registers**) 'MP1
		Case "RTN"
		ExecuteRTN(Memory, Registers)
		End Select
		If Registers(ERR) = 0 Then
		OpCode = Memory(Registers(PC)).OpCode
		DisplayCurrentState(SourceCode, Memory, Registers)
		Else
		OpCode = "HLT"
		End If
		End While
		Console.WriteLine("Execution terminated")
		End Sub	

13	**1**	Sub EditSourceCode(ByRef SourceCode() As String)
		Dim Choice As String = EMPTY_STRING
		Dim LineNumber As Integer
		Do
		Console.Write("Enter line number of code to edit: ")
		Dim Temp As String = Console.ReadLine ' MP1
		If Not Integer.TryParse(Temp, LineNumber) Then
		LineNumber = -1
		Console.WriteLine("Not a valid number") ' MP5
		End If
		Loop Until LineNumber > 0 And LineNumber <= Convert.ToInt32(SourceCode(0)) ' MP2, MP3, MP4
		
		Console.WriteLine(SourceCode(LineNumber))
		While Choice <> "C"
		Choice = EMPTY_STRING
		While Choice <> "E" And Choice <> "C"
		Console.WriteLine("E - Edit this line")
		Console.WriteLine("C - Cancel edit")
		Console.Write("Enter your choice: ")
		Choice = Console.ReadLine()
		End While
		If Choice = "E" Then
		Console.Write("Enter the new line: ")
		SourceCode(LineNumber) = Console.ReadLine()
		End If
		DisplaySourceCode(SourceCode)
		End While
		End Sub
			5

14	**1**	Sub ExecuteJSR(ByRef Memory() As AssemblerInstruction, ByRef Registers() As Integer, ByVal Address As Integer, **ByVal MaxLines As Integer**) Dim StackPointer As Integer = Registers(TOS) - 1 **If StackPointer <= MaxLines Then** 'MP1 **ReportRunTimeError("Memory Address Error", Registers)** 'MP4 **Else** 'MP2 Memory(StackPointer).OperandValue = Registers(PC) Registers(PC) = Address Registers(TOS) = StackPointer **End If** DisplayStack(Memory, Registers)	**4**

```
End Sub

  Sub Execute(ByVal SourceCode() As String, ByVal Memory() As
AssemblerInstruction)
    Dim Registers() As Integer = {0, 0, 0, 0, 0}
    SetFlags(Registers(ACC), Registers)
    Registers(PC) = 0
    Registers(TOS) = HI_MEM
    Dim FrameNumber As Integer = 0
    DisplayFrameDelimiter(FrameNumber)
    DisplayCurrentState(SourceCode, Memory, Registers)
    Dim OpCode As String = Memory(Registers(PC)).OpCode
    While OpCode <> "HLT"
      FrameNumber += 1
      Console.WriteLine()
      DisplayFrameDelimiter(FrameNumber)
      Dim Operand As Integer = Memory(Registers(PC)).OperandValue
      Console.WriteLine($"*  Current Instruction Register:  {OpCode}
{Operand}")
      Registers(PC) += 1
      Select Case OpCode
        Case "LDA"
          ExecuteLDA(Memory, Registers, Operand)
        Case "STA"
          ExecuteSTA(Memory, Registers, Operand)
        Case "LDA#"
          ExecuteLDAimm(Registers, Operand)
        Case "ADD"
          ExecuteADD(Memory, Registers, Operand)
        Case "JMP"
          ExecuteJMP(Registers, Operand)
        Case "JSR"
          ExecuteJSR(Memory, Registers, Operand, Int(SourceCode(0)))
'MP3
        Case "CMP#"
          ExecuteCMPimm(Registers, Operand)
        Case "BEQ"
          ExecuteBEQ(Registers, Operand)
        Case "SUB"
          ExecuteSUB(Memory, Registers, Operand)
        Case "SKP"
          ExecuteSKP()
        Case "RTN"
          ExecuteRTN(Memory, Registers)
      End Select
      If Registers(ERR) = 0 Then
        OpCode = Memory(Registers(PC)).OpCode
        DisplayCurrentState(SourceCode, Memory, Registers)
      Else
        OpCode = "HLT"
      End If
    End While
    Console.WriteLine("Execution terminated")
  End Sub
```

15 **1**

```
Sub EditSourceCode(ByRef SourceCode() As String)
  Dim LineNumber As Integer
  ...
  While Choice <> "C"
    Choice = Empty_String
```

12

```
    While Choice <> "E" And Choice <> "C" And Choice <> "D" And Choice
<> "I"
      Console.WriteLine("E - Edit this line")
      Console.WriteLine("C - Cancel edit")
      Console.WriteLine("D - Delete this line")                    'MP4
      Console.WriteLine("I - Insert a new line above this line")
      Console.Write("Enter your choice: ")
      Choice = Console.ReadLine()
    End While
    If Choice = "E" Then
      ...
    End If
    If Choice = "D" Then 'MP5
      Dim NumberOfLines As Integer = Convert.ToInt32(SourceCode(0))
      NumberOfLines -= 1
      SourceCode(0) = Convert.ToString(NumberOfLines) 'MP1, MP7 part
      For ThisLine = LineNumber To NumberOfLines  'MP2, MP6 part
        SourceCode(ThisLine) = SourceCode(ThisLine + 1)  'MP3, MP8
      Next
    End If
    If Choice = "I" Then  'MP5 alt.
      If Convert.ToInt32(SourceCode(0)) < HI_MEM - 1 Then  'MP11 part
        Console.Write("Enter the new line:")
        Dim NewLine As String = Console.ReadLine()'MP10
        Dim NumberOfLines As Integer = Convert.ToInt32(SourceCode(0))
        NumberOfLines += 1
        SourceCode(0) = Convert.ToString(NumberOfLines) 'MP1 alt. / MP7
part
        For ThisLine = NumberOfLines To LineNumber + 1 Step -1  'MP2
alt., MP6 part
          SourceCode(ThisLine) = SourceCode(ThisLine - 1) 'MP3 alt.,
MP9
        Next
        SourceCode(LineNumber) = NewLine 'MP12
      Else
        Console.WriteLine("Error - can't add a new line. Program is at
maximum length")  'MP11 part
      End If
    End If
    DisplaySourceCode(SourceCode)
  End While
End Sub
```

Python 3

05 **1**

```
Number = int(input("Enter a number greater than 1: ")) # MP2
X = 2                                                  # MP1
Count = 0                                      # MP3
while Number > 1:                              # MP4
  Multi = False                                # MP5
  while Number % X == 0:                       # MP6
    if not Multi:                              # MP7
      print(X, end = '  ')
    Count += 1                                 #
    Multi = True                               #
    Number = Number // X                       # MP8
  X = X + 1                                    # MP9
print(Count)
```

9

12 **1**

```
def ExecuteSKP(Registers): # MP2
  Registers[ACC] = Registers[ACC] + 1 # MP3
  Registers = SetFlags(Registers[ACC], Registers) # MP4

def Execute(SourceCode, Memory):
  Registers = [0, 0, 0, 0, 0]
  Registers = SetFlags(Registers[ACC], Registers)
  Registers[PC] = 0
  Registers[TOS] = HI_MEM
  FrameNumber = 0
  DisplayFrameDelimiter(FrameNumber)
  DisplayCurrentState(SourceCode, Memory, Registers)
  OpCode = Memory[Registers[PC]].OpCode
  while OpCode != "HLT":
    FrameNumber += 1
    print()
    DisplayFrameDelimiter(FrameNumber)
    Operand = Memory[Registers[PC]].OperandValue
    print("*  Current Instruction Register: ", OpCode, Operand)
    Registers[PC] = Registers[PC] + 1
    if OpCode == "LDA":
      Registers = ExecuteLDA(Memory, Registers, Operand)
    elif OpCode == "STA":
      Memory = ExecuteSTA(Memory, Registers, Operand)
    elif OpCode == "LDA#":
      Registers = ExecuteLDAimm(Registers, Operand)
    elif OpCode == "ADD":
      Registers = ExecuteADD(Memory, Registers, Operand)
    elif OpCode == "JMP":
      Registers = ExecuteJMP(Registers, Operand)
    elif OpCode == "JSR":
      Memory, Registers = ExecuteJSR(Memory, Registers, Operand)
    elif OpCode == "CMP#":
      Registers = ExecuteCMPimm(Registers, Operand)
    elif OpCode == "BEQ":
      Registers = ExecuteBEQ(Registers, Operand)
    elif OpCode == "SUB":
      Registers = ExecuteSUB(Memory, Registers, Operand)
    elif OpCode == "SKP":
      ExecuteSKP(Registers) # MP1
    elif OpCode == "RTN":
      Registers = ExecuteRTN(Memory, Registers)
    if Registers[ERR] == 0:
      OpCode = Memory[Registers[PC]].OpCode
      DisplayCurrentState(SourceCode, Memory, Registers)
```

4

```
    else:
      OpCode = "HLT"
  print("Execution terminated")
```

13 **1**

```
def EditSourceCode(SourceCode):
  LineNumber = 0
  NumberOfLines = int(SourceCode[0])
  while LineNumber  not in range(1, NumberOfLines + 1): # MP2, MP3, MP4
    try: # MP1
      LineNumber = int(input("Enter line number of code to edit: "))
      if LineNumber not in range(1, NumberOfLines + 1):
        print("Not a valid line number") # MP5
    except:
      print("Not a valid number")
  print(SourceCode[LineNumber])
  Choice = EMPTY_STRING
  while Choice != "C":
    Choice = EMPTY_STRING
    while Choice != "E" and Choice != "C":
      print("E - Edit this line")
      print("C - Cancel edit")
      Choice = input("Enter your choice: ")
    if Choice == "E":
      SourceCode[LineNumber] = input("Enter the new line:")
    DisplaySourceCode(SourceCode)
  return SourceCode
```

Alternative method:

```
def EditSourceCode(SourceCode):
  LineNumberStr = input("Enter line number of code to edit: ")
  Valid = False
  while not Valid:
    if LineNumberStr.isdigit() and int(LineNumberStr) > 0 and
int(LineNumberStr) <= int(SourceCode[0]): # MP1, MP2, MP3, MP4
      Valid = True
      LineNumber = int(LineNumberStr)
    else:
      print("Not a valid line number") # MP5
      LineNumberStr = input("Enter line number of code to edit: ")
  print(SourceCode [LineNumber])
  Choice = EMPTY_STRING
  while Choice != "C":
    Choice = EMPTY_STRING
    while Choice != "E" and Choice != "C":
      print("E - Edit this line")
      print("C - Cancel edit")
      Choice = input("Enter your choice: ")
    if Choice == "E":
      SourceCode[LineNumber] = input("Enter the new line:")
    DisplaySourceCode(SourceCode)
  return SourceCode
```

5

14	**1**	def ExecuteJSR(Memory, Registers, Address, **MaxLines**):
		  StackPointer = Registers[TOS] - 1
		  if StackPointer <= MaxLines: # MP1
		  ReportRunTimeError("Memory Address Error", Registers) # MP4
		  else: # MP2
		  Memory[StackPointer].OperandValue = Registers[PC]
		  Registers[PC] = Address
		  Registers[TOS] = StackPointer
		  DisplayStack(Memory, Registers)
		  return Memory, Registers
		
		def Execute(SourceCode, Memory):
		  Registers = [0, 0, 0, 0, 0]
		  Registers = SetFlags(Registers[ACC], Registers)
		  Registers[PC] = 0
		  Registers[TOS] = HI_MEM
		  FrameNumber = 0
		  DisplayFrameDelimiter(FrameNumber)
		  DisplayCurrentState(SourceCode, Memory, Registers)
		  OpCode = Memory[Registers[PC]].OpCode
		  while OpCode != "HLT":
		  FrameNumber += 1
		  print()
		  DisplayFrameDelimiter(FrameNumber)
		  Operand = Memory[Registers[PC]].OperandValue
		  print("* Current Instruction Register: ", OpCode, Operand)
		  Registers[PC] = Registers[PC] + 1
		  if OpCode == "LDA":
		  Registers = ExecuteLDA(Memory, Registers, Operand)
		  elif OpCode == "STA":
		  Memory = ExecuteSTA(Memory, Registers, Operand)
		  elif OpCode == "LDA#":
		  Registers = ExecuteLDAimm(Registers, Operand)
		  elif OpCode == "ADD":
		  Registers = ExecuteADD(Memory, Registers, Operand)
		  elif OpCode == "JMP":
		  Registers = ExecuteJMP(Registers, Operand)
		  elif OpCode == "JSR":
		  Memory, Registers = ExecuteJSR(Memory, Registers, Operand, **int(SourceCode[0])**) **# MP3**
		  elif OpCode == "CMP#":
		  Registers = ExecuteCMPimm(Registers, Operand)
		  elif OpCode == "BEQ":
		  Registers = ExecuteBEQ(Registers, Operand)
		  elif OpCode == "SUB":
		  Registers = ExecuteSUB(Memory, Registers, Operand)
		  elif OpCode == "SKP":
		  ExecuteSKP()
		  elif OpCode == "RTN":
		  Registers = ExecuteRTN(Memory, Registers)
		  if Registers[ERR] == 0:
		  OpCode = Memory[Registers[PC]].OpCode
		  DisplayCurrentState(SourceCode, Memory, Registers)
		  else:
		  OpCode = "HLT"
		  print("Execution terminated")
			4

Alternative method:

```
def ExecuteJSR(Memory, Registers, Address):
  StackPointer = Registers[TOS] - 1
  if Memory[StackPointer].OperandString != "" or
Memory[StackPointer].OpCode != "": # MP1, MP3
    ReportRunTimeError("Memory Address Error", Registers) # MP4
  else: #MP2
    Memory[StackPointer].OperandValue = Registers[PC]
    Registers[PC] = Address
    Registers[TOS] = StackPointer
  DisplayStack(Memory, Registers)
  return Memory, Registers
```

15 **1**

```
def EditSourceCode(SourceCode):
  LineNumber = int(input("Enter line number of code to edit: "))
  print(SourceCode[LineNumber])
  Choice = EMPTY_STRING
  while Choice != "C":
    Choice = EMPTY_STRING
    while Choice != "E" and Choice != "D" and Choice != "I" and Choice
!= "C":
      print("E - Edit this line")
      print("D - Delete this line")                                  # MP4
      print("I - Insert a new line above this line")
      print("C - Cancel edit")
      Choice = input("Enter your choice: ")
    if Choice == "E":
      SourceCode[LineNumber] = input("Enter the new line:")
    if Choice == "D": # MP5
      NumberOfLines = int(SourceCode[0])  #
      NumberOfLines -= 1                  #
      SourceCode[0] = str(NumberOfLines)  # MP1, MP7 part
      for ThisLine in range(LineNumber, NumberOfLines + 1): # MP2, MP6
part
        SourceCode[ThisLine] = SourceCode[ThisLine + 1] # MP3, MP8
    if Choice == "I": # MP5 alt
      if int(SourceCode[0]) < HI_MEM - 1: # MP11 part
        print("Enter the new line:") #
        NewLine = input()           # MP10
        NumberOfLines = int(SourceCode[0])  #
        NumberOfLines += 1                  #
        SourceCode[0] = str(NumberOfLines)  # MP1 alt./MP7 part
        for ThisLine in range(NumberOfLines, LineNumber, - 1): # MP2
alt., MP6 part
          SourceCode[ThisLine] = SourceCode[ThisLine - 1] #  MP3 alt.,
MP9
        SourceCode[LineNumber] = NewLine # MP12
      else:
        print("Error - can't add a new line. Program is at maximum
length") # MP11 part
    DisplaySourceCode(SourceCode)
  return SourceCode
```

12

Python 2

05	**1**	`Number = int(raw_input("Enter a number greater than 1: ")) # MP2`
		`X = 2 # MP1`
		`Count = 0 # MP3`
		`while Number > 1: # MP4`
		`  Multi = False # MP5`
		`  while Number % X == 0: # MP6`
		`  if not Multi: # MP7`
		`  print X`
		`  Count += 1 #`
		`  Multi = True #`
		`  Number = Number // X # MP8`
		`  X = X + 1 # MP9`
		`print Count`
			9

12	**1**	`def ExecuteSKP(Registers): # MP2`
		`  Registers[ACC] = Registers[ACC] + 1 # MP3`
		`  Registers = SetFlags(Registers[ACC], Registers) # MP4`
		
		`def Execute(SourceCode, Memory):`
		`  Registers = [0, 0, 0, 0, 0]`
		`  Registers = SetFlags(Registers[ACC], Registers)`
		`  Registers[PC] = 0`
		`  Registers[TOS] = HI_MEM`
		`  FrameNumber = 0`
		`  DisplayFrameDelimiter(FrameNumber)`
		`  DisplayCurrentState(SourceCode, Memory, Registers)`
		`  OpCode = Memory[Registers[PC]].OpCode`
		`  while OpCode != "HLT":`
		`  FrameNumber += 1`
		`  print`
		`  DisplayFrameDelimiter(FrameNumber)`
		`  Operand = Memory[Registers[PC]].OperandValue`
		`  print "*  Current Instruction Register: ", OpCode, Operand`
		`  Registers[PC] = Registers[PC] + 1`
		`  if OpCode == "LDA":`
		`  Registers = ExecuteLDA(Memory, Registers, Operand)`
		`  elif OpCode == "STA":`
		`  Memory = ExecuteSTA(Memory, Registers, Operand)`
		`  elif OpCode == "LDA#":`
		`  Registers = ExecuteLDAimm(Registers, Operand)`
		`  elif OpCode == "ADD":`
		`  Registers = ExecuteADD(Memory, Registers, Operand)`
		`  elif OpCode == "JMP":`
		`  Registers = ExecuteJMP(Registers, Operand)`
		`  elif OpCode == "JSR":`
		`  Memory, Registers = ExecuteJSR(Memory, Registers, Operand)`
		`  elif OpCode == "CMP#":`
		`  Registers = ExecuteCMPimm(Registers, Operand)`
		`  elif OpCode == "BEQ":`
		`  Registers = ExecuteBEQ(Registers, Operand)`
		`  elif OpCode == "SUB":`
		`  Registers = ExecuteSUB(Memory, Registers, Operand)`
		`  elif OpCode == "SKP":`
		`  ExecuteSKP(Registers) # MP1`
		`  elif OpCode == "RTN":`
			4

```
      Registers = ExecuteRTN(Memory, Registers)
    if Registers[ERR] == 0:
      OpCode = Memory[Registers[PC]].OpCode
      DisplayCurrentState(SourceCode, Memory, Registers)
    else:
      OpCode = "HLT"
  print "Execution terminated"
```

13 **1** **5**

```
def EditSourceCode(SourceCode):
  LineNumber = 0
  NumberOfLines = int(SourceCode[0])
  while LineNumber  not in range(1, NumberOfLines + 1): # MP2, MP3, MP4
    try: # MP1
      LineNumber = int(raw_input("Enter line number of code to edit: ") .replace("\r", ""))
      if LineNumber not in range(1, NumberOfLines + 1):
        print "Not a valid line number" # MP5
    except:
      print "Not a valid number"
  print SourceCode[LineNumber]
  Choice = EMPTY_STRING
  while Choice != "C":
    Choice = EMPTY_STRING
    while Choice != "E" and Choice != "C":
      print "E - Edit this line"
      print "C - Cancel edit"
      Choice = raw_input("Enter your choice: ") .replace("\r", "")
    if Choice == "E":
      SourceCode[LineNumber] = raw_input("Enter the new line: ") .replace("\r", "")
    DisplaySourceCode(SourceCode)
  return SourceCode
```

Alternative method:

```
def EditSourceCode(SourceCode):
  LineNumberStr = raw_input("Enter line number of code to edit: ") .replace("\r", "")
  Valid = False
  while not Valid:
    if LineNumberStr.isdigit() and int(LineNumberStr) > 0 and int(LineNumberStr) <= int(SourceCode[0]): # MP1, MP2, MP3, MP4
      Valid = True
      LineNumber = int(LineNumberStr)
    else:
      print "Not a valid line number" # MP5
      LineNumberStr = raw_input("Enter line number of code to edit: ") .replace("\r", "")
  print SourceCode [LineNumber]
  Choice = EMPTY_STRING
  while Choice != "C":
    Choice = EMPTY_STRING
    while Choice != "E" and Choice != "C":
      print "E - Edit this line"
      print "C - Cancel edit"
```

```
      Choice = raw_input("Enter your choice: ") .replace("\r",
"")
    if Choice == "E":
      SourceCode[LineNumber] = raw_input("Enter the new line:")
.replace("\r", "")
    DisplaySourceCode(SourceCode)
  return SourceCode
```

14 **1**

```
def ExecuteJSR(Memory, Registers, Address, MaxLines):
  StackPointer = Registers[TOS] - 1
  if StackPointer <= MaxLines: # MP1
    ReportRunTimeError("Memory Address Error", Registers) # MP4
  else: # MP2
    Memory[StackPointer].OperandValue = Registers[PC]
    Registers[PC] = Address
    Registers[TOS] = StackPointer
  DisplayStack(Memory, Registers)
  return Memory, Registers

def Execute(SourceCode, Memory):
  Registers = [0, 0, 0, 0, 0]
  Registers = SetFlags(Registers[ACC], Registers)
  Registers[PC] = 0
  Registers[TOS] = HI_MEM
  FrameNumber = 0
  DisplayFrameDelimiter(FrameNumber)
  DisplayCurrentState(SourceCode, Memory, Registers)
  OpCode = Memory[Registers[PC]].OpCode
  while OpCode != "HLT":
    FrameNumber += 1
    print
    DisplayFrameDelimiter(FrameNumber)
    Operand = Memory[Registers[PC]].OperandValue
    print "*  Current Instruction Register: ", OpCode, Operand
    Registers[PC] = Registers[PC] + 1
    if OpCode == "LDA":
      Registers = ExecuteLDA(Memory, Registers, Operand)
    elif OpCode == "STA":
      Memory = ExecuteSTA(Memory, Registers, Operand)
    elif OpCode == "LDA#":
      Registers = ExecuteLDAimm(Registers, Operand)
    elif OpCode == "ADD":
      Registers = ExecuteADD(Memory, Registers, Operand)
    elif OpCode == "JMP":
      Registers = ExecuteJMP(Registers, Operand)
    elif OpCode == "JSR":
      Memory, Registers = ExecuteJSR(Memory, Registers, Operand,
int(SourceCode[0])) # MP3
    elif OpCode == "CMP#":
      Registers = ExecuteCMPimm(Registers, Operand)
    elif OpCode == "BEQ":
      Registers = ExecuteBEQ(Registers, Operand)
    elif OpCode == "SUB":
      Registers = ExecuteSUB(Memory, Registers, Operand)
    elif OpCode == "SKP":
      ExecuteSKP()
    elif OpCode == "RTN":
```

4

```
      Registers = ExecuteRTN(Memory, Registers)
    if Registers[ERR] == 0:
      OpCode = Memory[Registers[PC]].OpCode
      DisplayCurrentState(SourceCode, Memory, Registers)
    else:
      OpCode = "HLT"
  print "Execution terminated"
```

Alternative method:

```
def ExecuteJSR(Memory, Registers, Address):
  StackPointer = Registers[TOS] - 1
  if Memory[StackPointer].OperandString != "" or
Memory[StackPointer].OpCode != "": # MP1, MP3
    ReportRunTimeError("Memory Address Error", Registers) # MP4
  else: #MP2
    Memory[StackPointer].OperandValue = Registers[PC]
    Registers[PC] = Address
    Registers[TOS] = StackPointer
  DisplayStack(Memory, Registers)
  return Memory, Registers
```

15 **1** **12**

```
def EditSourceCode(SourceCode):
  LineNumber = int(raw_input("Enter line number of code to edit:
") .replace("\r", ""))
  print SourceCode[LineNumber]
  Choice = EMPTY_STRING
  while Choice != "C":
    Choice = EMPTY_STRING
    while Choice != "E" and Choice != "D" and Choice != "I" and
Choice != "C": #
      print "E - Edit this line"
      print "D - Delete this line"
# MP4
      print "I - Insert a new line above this line"
#
      print "C - Cancel edit"
      Choice = raw_input("Enter your choice: ") .replace("\r",
"")
    if Choice == "E":
      SourceCode[LineNumber] = raw_input("Enter the new line:")
.replace("\r", "")
    if Choice == "D": # MP5
      NumberOfLines = int(SourceCode[0])  #
      NumberOfLines -= 1                  #
      SourceCode[0] = str(NumberOfLines)  # MP1, MP7 part
      for ThisLine in range(LineNumber, NumberOfLines + 1): #
MP2, MP6 part
        SourceCode[ThisLine] = SourceCode[ThisLine + 1] # MP3,
MP8
    if Choice == "I": # MP5 alt.
      if int(SourceCode[0]) < HI_MEM - 1: # MP11 part
        print "Enter the new line:"  #
        NewLine = raw_input()             # MP10
        NumberOfLines = int(SourceCode[0])  #
        NumberOfLines += 1                  #
        SourceCode[0] = str(NumberOfLines)  # MP1 alt./MP7 part
```

```
        for ThisLine in range(NumberOfLines, LineNumber, - 1): # MP2 alt., MP6 part
          SourceCode[ThisLine] = SourceCode[ThisLine - 1] #  MP3 alt., MP9
        SourceCode[LineNumber] = NewLine # MP12
      else:
        print "Error - can't add a new line. Program is at maximum length" # MP11 part
    DisplaySourceCode(SourceCode)
  return SourceCode
```

Pascal

05 **1** **9**

```
var
  Number, X, Count : integer; // MP1
  Multi : boolean;

begin
  write('Enter a number greater than 1: ');
  readln(Number); // MP2
  X := 2;
  Count := 0;  // MP3
  while Number > 1 do // MP4
  begin
    Multi := false; // MP5
    while Number Mod X = 0 do // MP6
    begin
      if not Multi // MP7
        then
          writeln(X);
      Count := Count + 1;
      Multi := true;
      Number := Number DIV X; // MP8
   end;
   X := X + 1; // MP9
  end;
  writeln(Count);
  readln;
end.
```

12 **1** **4**

```
procedure ExecuteSKP(var Registers: TRegisters); // MP2
begin
  Registers[ACC] := Registers[ACC] + 1; // MP3
  SetFlags(Registers[ACC], Registers);  // MP4
end;

procedure Execute(var SourceCode: TSourceCode; var Memory:
TMemory);
var
  Registers: TRegisters;
  OpCode: string;
  Operand, FrameNumber: integer;
begin
  Registers := TRegisters.create(0, 0, 0, 0, 0);
  SetFlags(Registers[ACC], Registers);
  Registers[PC] := 0;
  Registers[TOS] := HI_MEM;
  FrameNumber := 0;
  DisplayFrameDelimiter(FrameNumber);
  DisplayCurrentState(SourceCode, Memory, Registers);
  OpCode := Memory[Registers[PC]].OpCode;
  while OpCode <> 'HLT' do
  begin
    FrameNumber := FrameNumber + 1;
    writeln;
   DisplayFrameDelimiter(FrameNumber);
    Operand := Memory[Registers[PC]].OperandValue;
```

```
    writeln('*  Current Instruction Register:  ', OpCode, ' ',
Operand);
    Registers[PC] := Registers[PC] + 1;
    if OpCode = 'LDA' then ExecuteLDA(Memory, Registers,
Operand);
    if OpCode = 'STA' then ExecuteSTA(Memory, Registers,
Operand);
    if OpCode = 'LDA#' then ExecuteLDAimm(Registers, Operand);
    if OpCode = 'ADD' then ExecuteADD(Memory, Registers,
Operand);
    if OpCode = 'JMP' then ExecuteJMP(Registers, Operand);
    if OpCode = 'JSR' then ExecuteJSR(Memory, Registers,
Operand);
    if OpCode = 'CMP#' then ExecuteCMPimm( Registers, Operand);
    if OpCode = 'BEQ' then ExecuteBEQ(Registers, Operand);
    if OpCode = 'SUB' then ExecuteSUB(Memory, Registers,
Operand);
    if OpCode = 'SKP' then ExecuteSKP(Registers); // MP1
    if OpCode = 'RTN' then ExecuteRTN(Memory, Registers);
    if Registers[ERR] = 0
    then
      begin
        OpCode := Memory[Registers[PC]].OpCode;
        DisplayCurrentState(SourceCode, Memory, Registers);
      end
    else
      OpCode := 'HLT';
  end;
  writeln('Execution terminated');
end;
```

13 **1**

```
procedure EditSourceCode(var SourceCode: TSourceCode);
var
  LineNumber, NumberOfLines: integer;
  Choice: string;
begin
  LineNumber := 0;
  NumberOfLines := StrToInt(SourceCode[0]);
  while (LineNumber < 1) or (LineNumber >= NumberOfLines) do
// MP2, MP3, MP4
  begin
    try   // MP1
      write('Enter line number of code to edit: ');
      readln(LineNumber);
      if (LineNumber < 1) or (LineNumber >= NumberOfLines)
      then
        writeln('Not a valid line number'); // MP5
    except
      writeln('Not a valid number');
    end;
  end;
  writeln(SourceCode[LineNumber]);
  Choice := EMPTY_STRING;
  while Choice <> 'C' do
  begin
    Choice := EMPTY_STRING;
```

5

```
    while (Choice <> 'E') and (Choice <> 'C') do
    begin
      writeln('E - Edit this line');
      writeln('C - Cancel edit');
      write('Enter your choice: ');
      readln(Choice);
    end;
    if Choice = 'E'
    then
      begin
        writeln('Enter the new line:');
        readln(SourceCode[LineNumber]);
      end;
    DisplaySourceCode(SourceCode);
  end;
end;
```

14 **1**

```
procedure ExecuteJSR(var Memory: TMemory; var Registers:
TRegisters; Address: integer; MaxLines: integer);
var
  StackPointer: integer;
begin
  StackPointer := Registers[TOS] - 1;
  if StackPointer <= MaxLines  // MP1
  then
    ReportRunTimeError('Memory Address Error', Registers) // MP4
  else // MP2
    begin
      Memory[StackPointer].OperandValue := Registers[PC];
      Registers[PC] := Address;
      Registers[TOS] := StackPointer;
    end;
  DisplayStack(Memory, Registers)
end;

procedure Execute(var SourceCode: TSourceCode; var Memory:
TMemory);
var
  Registers: TRegisters;
  OpCode: string;
  Operand, FrameNumber: integer;
begin
  Registers := TRegisters.create(0, 0, 0, 0, 0);
  SetFlags(Registers[ACC], Registers);
  Registers[PC] := 0;
  Registers[TOS] := HI_MEM;
  FrameNumber := 0;
  DisplayFrameDelimiter(FrameNumber);
  DisplayCurrentState(SourceCode, Memory, Registers);
  OpCode := Memory[Registers[PC]].OpCode;
  while OpCode <> 'HLT' do
  begin
    FrameNumber := FrameNumber + 1;
    writeln;
   DisplayFrameDelimiter(FrameNumber);
    Operand := Memory[Registers[PC]].OperandValue;
```

4

```
    writeln('*  Current Instruction Register:  ', OpCode, ' ',
Operand);
    Registers[PC] := Registers[PC] + 1;
    if OpCode = 'LDA' then ExecuteLDA(Memory, Registers,
Operand);
    if OpCode = 'STA' then ExecuteSTA(Memory, Registers,
Operand);
    if OpCode = 'LDA#' then ExecuteLDAimm(Registers, Operand);
    if OpCode = 'ADD' then ExecuteADD(Memory, Registers,
Operand);
    if OpCode = 'JMP' then ExecuteJMP(Registers, Operand);
    if OpCode = 'JSR' then ExecuteJSR(Memory, Registers,
Operand, StrToInt(SourceCode[0])); // MP3
    if OpCode = 'CMP#' then ExecuteCMPimm( Registers, Operand);
    if OpCode = 'BEQ' then ExecuteBEQ(Registers, Operand);
    if OpCode = 'SUB' then ExecuteSUB(Memory, Registers,
Operand);
    if OpCode = 'SKP' then ExecuteSKP();
    if OpCode = 'RTN' then ExecuteRTN(Memory, Registers);
    if Registers[ERR] = 0
    then
      begin
        OpCode := Memory[Registers[PC]].OpCode;
        DisplayCurrentState(SourceCode, Memory, Registers);
      end
    else
      OpCode := 'HLT';
  end;
  writeln('Execution terminated');
end;
```

Alternative method:

```
procedure ExecuteJSR(var Memory: TMemory; var Registers:
TRegisters; Address: integer);
var
  StackPointer: integer;
begin
  StackPointer := Registers[TOS] - 1;
 if (Memory[StackPointer].OperandString <> EMPTY_STRING) or
     Memory[StackPointer].OpCode <> EMPTY_STRING // MP1, MP3
 then
    ReportRunTimeError('Memory Address Error', Registers) // MP4
 else // MP2
  begin
    Memory[StackPointer].OperandValue := Registers[PC];
    Registers[PC] := Address;
    Registers[TOS] := StackPointer;
  end;
  DisplayStack(Memory, Registers)
end;
```

15	**1**	`procedure EditSourceCode(var SourceCode: TSourceCode);`
		`var`
		`  LineNumber, NumberOfLines, ThisLine: integer;`
		`  Choice: string;`
		`  NewLine: string;`
		`begin`
			12

```
  write('Enter line number of code to edit: ');
  readln(LineNumber);
  writeln(SourceCode[LineNumber]);
  Choice := EMPTY_STRING;
  while Choice <> 'C' do
    begin
      Choice := EMPTY_STRING;
      while (Choice <> 'E') and (Choice <> 'D') and (Choice <>
'I') and (Choice <> 'C') do //
        begin
          writeln('E - Edit this line');
          writeln('D - Delete this line');                  // MP4
          writeln('I - Insert a new line above this line');//
          writeln('C - Cancel edit');
          write('Enter your choice: ');
          readln(Choice);
        end;
      if Choice = 'E'
      then
        begin
          writeln('Enter the new line:');
          readln(SourceCode[LineNumber]);
        end;
      if Choice = 'D'  // MP5
      then
        begin
          NumberOfLines := StrToInt(SourceCode[0]); //
          NumberOfLines := NumberOfLines - 1;        //
          SourceCode[0] := IntToStr(NumberOfLines); // MP1, MP7
part
          for ThisLine := LineNumber to NumberOfLines do  //
MP2, MP6 part
            SourceCode[ThisLine] := SourceCode[ThisLine + 1]; //
MP3 alt., MP8
        end;
      if Choice = 'I'  // MP5 alt.
      then
        begin
          if StrToInt(SourceCode[0]) < HI_MEM - 1   // MP11 part
          then
            begin
              writeln('Enter the new line:'); //
              readln(NewLine);                // MP10
              NumberOfLines := StrToInt(SourceCode[0]); //
              NumberOfLines := NumberOfLines + 1;        //
              SourceCode[0] := IntToStr(NumberOfLines); // MP1
alt. MP7 part
              for ThisLine := NumberOfLines downto LineNumber +
1 do   // MP2 alt., MP6 part
              SourceCode[ThisLine] := SourceCode[ThisLine - 1];
// MP3 alt., MP9
              SourceCode[LineNumber] := NewLine;  // MP12
            end
          else
            writeln('Error - can''t add a new line as program is
at maximum length');
```

```
            end; // MP11 part
        DisplaySourceCode(SourceCode);
    end;
end;
```

C#

05 **1**

```
bool multi;
Console.Write("Enter an integer greater than 1: ");
int number = Convert.ToInt32(Console.ReadLine());   // MP2
int x = 2;                              // MP1
int count = 0;                          // MP3
while (number > 1)                      // MP4
{
    multi = false;                      // MP5
    while (number % x == 0)             // MP6
    {
        if (!multi)                     // MP7
        {
            Console.WriteLine(x);
        }
        count++;
        multi = true;
        number = number / x;            // MP8
    }
    x++;                                // MP9
}
Console.WriteLine(count);
```

9

12 **1**

```
private static void ExecuteSKP(int[] registers)  // MP2
{
    registers[ACC] = registers[ACC] + 1;  // MP3
    SetFlags(registers[ACC], registers);  // MP4
}

private static void Execute(string[] sourceCode, AssemblerInstruction[]
memory)
    {
      int[] registers = new int[] { 0, 0, 0, 0, 0 };
      int frameNumber = 0, operand = 0;
      SetFlags(registers[ACC], registers);
      registers[PC] = 0;
      registers[TOS] = HI_MEM;
      DisplayFrameDelimiter(frameNumber);
      DisplayCurrentState(sourceCode, memory, registers);
      string opCode = memory[registers[PC]].opCode;
      while (opCode != "HLT")
      {
        frameNumber++;
        Console.WriteLine();
        DisplayFrameDelimiter(frameNumber);
        operand = memory[registers[PC]].operandValue;
        Console.WriteLine($"*  Current Instruction Register:  {opCode}
{operand}");
        registers[PC] = registers[PC] + 1;
        switch (opCode)
        {
          case "LDA":
            ExecuteLDA(memory, registers, operand); break;
          case "STA":
            ExecuteSTA(memory, registers, operand); break;
          case "LDA#":
            ExecuteLDAimm(registers, operand); break;
          case "ADD":
            ExecuteADD(memory, registers, operand); break;
          case "JMP":
            ExecuteJMP(registers, operand); break;
```

4

```
            case "JSR":
              ExecuteJSR(memory, registers, operand); break;
            case "CMP#":
              ExecuteCMPimm(registers, operand); break;
            case "BEQ":
              ExecuteBEQ(registers, operand); break;
            case "SUB":
              ExecuteSUB(memory, registers, operand); break;
            case "SKP":
              ExecuteSKP(registers); break; // MP1
            case "RTN":
              ExecuteRTN(memory, registers); break;
          }
          if (registers[ERR] == 0)
          {
            opCode = memory[registers[PC]].opCode;
            DisplayCurrentState(sourceCode, memory, registers);
          }
          else
          {
            opCode = "HLT";
          }
        }
        Console.WriteLine("Execution terminated");
      }
```

13 **1**

```
private static void EditSourceCode(string[] sourceCode)
{
    int lineNumber = 0;
    int numberOfLines = Convert.ToInt32(sourceCode[0]);
    while (lineNumber < 1 || lineNumber >= numberOfLines) // MP2, MP3,
MP4
    {
        Try  // MP1
        {
            Console.Write("Enter line number of code to edit: ");
            lineNumber = Convert.ToInt32(Console.ReadLine());
            if (lineNumber < 1 || lineNumber >= numberOfLines)
            {
                Console.WriteLine("Not a valid line number"); // MP5
            }
        }
        catch (Exception)
        {
            Console.WriteLine("Not a valid number");
        }
    }
    Console.WriteLine(sourceCode[lineNumber]);
    string choice = EMPTY_STRING;
    while (choice != "C")
    {
        choice = EMPTY_STRING;
        while (choice != "E" && choice != "C")
        {
            Console.WriteLine("E - Edit this line");
            Console.WriteLine("C - Cancel edit");
            Console.Write("Enter your choice: ");
            choice = Console.ReadLine();
        }
        if (choice == "E")
        {
```

5

```
            Console.WriteLine("Enter the new line:");
            sourceCode[lineNumber] = Console.ReadLine();
        }
        DisplaySourceCode(sourceCode);
    }
}
```

14 **1**

```
private static void ExecuteJSR(AssemblerInstruction[] memory, int[]
registers, int address, int maxLines)
{
    int stackPointer = registers[TOS] - 1;
    if (stackPointer <= maxLines) // MP1
    {
        ReportRunTimeError("Memory Address Error", registers); // MP4
    }
    else // MP2
    {
        memory[stackPointer].operandValue = registers[PC];
        registers[PC] = address;
        registers[TOS] = stackPointer;
    }
    DisplayStack(memory, registers);
}

    private static void Execute(string[] sourceCode,
AssemblerInstruction[] memory)
    {
      int[] registers = new int[] { 0, 0, 0, 0, 0 };
      int frameNumber = 0, operand = 0;
      SetFlags(registers[ACC], registers);
      registers[PC] = 0;
      registers[TOS] = HI_MEM;
      DisplayFrameDelimiter(frameNumber);
      DisplayCurrentState(sourceCode, memory, registers);
      string opCode = memory[registers[PC]].opCode;
      while (opCode != "HLT")
      {
        frameNumber++;
        Console.WriteLine();
        DisplayFrameDelimiter(frameNumber);
        operand = memory[registers[PC]].operandValue;
        Console.WriteLine($"*  Current Instruction Register:  {opCode}
{operand}");
        registers[PC] = registers[PC] + 1;
        switch (opCode)
        {
          case "LDA":
            ExecuteLDA(memory, registers, operand); break;
          case "STA":
            ExecuteSTA(memory, registers, operand); break;
          case "LDA#":
            ExecuteLDAimm(registers, operand); break;
          case "ADD":
            ExecuteADD(memory, registers, operand); break;
          case "JMP":
            ExecuteJMP(registers, operand); break;
          case "JSR":
            ExecuteJSR(memory, registers, operand,
Convert.ToInt32(sourceCode[0])); break;  // MP3
          case "CMP#":
            ExecuteCMPimm(registers, operand); break;
```

4

```
            case "BEQ":
              ExecuteBEQ(registers, operand); break;
            case "SUB":
              ExecuteSUB(memory, registers, operand); break;
            case "SKP":
              ExecuteSKP(); break;
            case "RTN":
              ExecuteRTN(memory, registers); break;
          }
          if (registers[ERR] == 0)
          {
            opCode = memory[registers[PC]].opCode;
            DisplayCurrentState(sourceCode, memory, registers);
          }
          else
          {
            opCode = "HLT";
          }
        }
        Console.WriteLine("Execution terminated");
      }
```

Alternative method:

```
private static void ExecuteJSR(AssemblerInstruction[] memory, int[]
registers, int address)
{
    int stackPointer = registers[TOS] - 1;
    if (memory[stackPointer].operandString != "") // MP1, MP3
    {
        ReportRunTimeError("Memory Address Error", registers); // MP4
    }
    else // MP2
    {
        memory[stackPointer].operandValue = registers[PC];
        registers[PC] = address;
        registers[TOS] = stackPointer;
    }
    DisplayStack(memory, registers);
}
```

15 **1**

```
private static void EditSourceCode(string[] sourceCode)
{
    int lineNumber = 0;
    int numberOfLines = Convert.ToInt32(sourceCode[0]);
    while (lineNumber < 1 || lineNumber >= numberOfLines)
    {
        try
        {
            Console.Write("Enter line number of code to edit: ");
            lineNumber = Convert.ToInt32(Console.ReadLine());
            if (lineNumber < 1 || lineNumber >= numberOfLines)
            {
                Console.WriteLine("Not a valid line number");
            }
        }
        catch (Exception)
        {
            Console.WriteLine("Not a valid number");
        }
    }
```

12

```
    Console.WriteLine(sourceCode[lineNumber]);
    string choice = EMPTY_STRING;
    while (choice != "C")
    {
        choice = EMPTY_STRING;
        while (choice != "E" && choice != "C" && choice != "D" &&
choice != "I")
        {
            Console.WriteLine("E - Edit this line");
            Console.WriteLine("D - Delete this line"); // MP4
            Console.WriteLine("C - Cancel edit");
            Console.WriteLine("I - Insert a new line above this line");
            Console.Write("Enter your choice: ");
            choice = Console.ReadLine();
        }
        if (choice == "E")
        {
            Console.WriteLine("Enter the new line:");
            sourceCode[lineNumber] = Console.ReadLine();
        }
        if (choice == "D") // MP5
        {
            numberOfLines--;
            sourceCode[0] = numberOfLines.ToString();// MP1, MP7 part
            for (int thisLine = lineNumber; thisLine < numberOfLines +
1; thisLine++) // MP2, MP6 part
            {
                sourceCode[thisLine] = sourceCode[thisLine + 1]; //
MP3, MP8
            }
        }
        if (choice == "I") // MP5 alt.
        {
            if (Convert.ToInt32(sourceCode[0]) < HI_MEM - 1) // MP11
part
            {
                Console.WriteLine("Enter the new line:");
                string newLine = Console.ReadLine();// MP10
                numberOfLines = Convert.ToInt32(sourceCode[0]);
                numberOfLines++;
                sourceCode[0] = numberOfLines.ToString();// MP1 alt./
MP7 part
                for (int thisLine = numberOfLines; thisLine >
lineNumber; thisLine--)//MP2 alt / MP6 part
                {
                    sourceCode[thisLine] = sourceCode[thisLine - 1];
                }   // MP3 alt. MP9
                sourceCode[lineNumber] = newLine;  // MP12
            }
            else
            {
                Console.WriteLine("Error - can't add a new line as
program is at maximum length");  // MP11 part
            }
        }
        DisplaySourceCode(sourceCode);
    }
}
```

Java

05 | **1** | **9**

```
Console.println("Enter an integer greater than 1: ");
int number = Integer.parseInt(Console.readLine());// MP2
int x = 2;                                  // MP1
int count = 0;                              // MP3
while (number > 1) {                        // MP4
    boolean multi = false;                  // MP5
    while (number % x == 0) {               // MP6
        if (!multi) {                       // MP7
            Console.writeLine(x);
        }
        count += 1;
        multi = true;
        number = number / x;                // MP8
    }
    x += 1;                                 // MP9
}
Console.writeLine(count);
```

12 | **1** | **4**

```
void executeSKP(int[] registers) {   // MP4
        registers[ACC] = registers[ACC] + 1;   // MP2
        setFlags(registers[ACC], registers);   // MP3
}

void execute(String[] sourceCode, AssemblerInstruction[] memory)
{
        int[] registers = {0, 0, 0, 0, 0} ;
        setFlags(registers[ACC], registers);
        registers[PC] = 0;
        registers[TOS] = HI_MEM;
        int frameNumber = 0;
        displayFrameDelimiter(frameNumber);
        displayCurrentState(sourceCode, memory, registers);
        String opCode = memory[registers[PC]].opCode;
        while (!opCode.equals("HLT")) {
            frameNumber += 1;
            Console.println();
            displayFrameDelimiter(frameNumber);
            int operand = memory[registers[PC]].operandValue;
            Console.println("*  Current instruction Register:  "
+ opCode + " " + operand);
            registers[PC] = registers[PC] + 1;
            switch (opCode) {
                case "LDA":
                    executeLDA(memory, registers, operand);
break;
                case "STA":
                    executeSTA(memory, registers, operand);
break;
                case "LDA#":
                    executeLDAimm(registers, operand); break;
                case "ADD":
                    executeADD(memory, registers, operand);
break;
                case "JMP":
                    executeJMP(registers, operand); break;
```

```
                    case "JSR":
                        executeJSR(memory, registers, operand);
break;
                    case "CMP#":
                        executeCMPimm(registers, operand); break;
                    case "BEQ":
                        executeBEQ(registers, operand); break;
                    case "SUB":
                        executeSUB(memory, registers, operand);
break;
                    case "SKP":
                        executeSKP(registers); break;  // MP1
                    case "RTN":
                        executeRTN(memory, registers); break;
                    default:
                        break;
                }
                if (registers[ERR] == 0) {
                    opCode = memory[registers[PC]].opCode;
                    displayCurrentState(sourceCode, memory,
registers);
                } else {
                    opCode = "HLT";
                }
            }
            Console.println("Execution terminated");
        }
```

13 **1**

```
void editSourceCode(String[] sourceCode) {
    int lineNumber;
    do {
        Console.print("Enter line number of code to edit: ");
        try { // MP1
            lineNumber = Integer.parseInt(Console.readLine());
        } catch (Exception e) {
            lineNumber = 0;
        }
        if (lineNumber < 1 || lineNumber > sourceCode.length +
1) {
            Console.println("Invalid line number please try
again.");  // MP5
        }
    } while (lineNumber < 1 || lineNumber > sourceCode.length +
1);  // MP2, MP3, MP4
    Console.println(sourceCode[lineNumber]);
    String choice = EMPTY_STRING;
    while (!choice.equals("C")) {
        choice = EMPTY_STRING;
        while (!choice.equals("E") && !choice.equals("C")) {
            Console.println("E - Edit this line");
            Console.println("C - Cancel edit");
            Console.print("Enter your choice: ");
            choice = Console.readLine();
        }
        if (choice.equals("E")) {
            Console.print("Enter the new line: ");
```

5

```
                sourceCode[lineNumber] = Console.readLine();
                displaySourceCode(sourceCode);
            }
        }
    }
```

14 **1**

```
void executeJSR(AssemblerInstruction[] memory, int[] registers,
int address, int maxLines) {
    int stackPointer = registers[TOS] - 1;
    if (stackPointer <= maxLines) { // MP1
        reportRunTimeError("Memory Address Error", registers); //
MP4
    } else { // MP2
        memory[stackPointer].operandValue = registers[PC] ;
        registers[PC] = address;
        registers[TOS] = stackPointer;
    }
    displayStack(memory, registers);

}

   void execute(String[] sourceCode, AssemblerInstruction[]
memory) {
        int[] registers = {0, 0, 0, 0, 0} ;
        setFlags(registers[ACC], registers);
        registers[PC] = 0;
        registers[TOS] = HI_MEM;
        int frameNumber = 0;
        displayFrameDelimiter(frameNumber);
        displayCurrentState(sourceCode, memory, registers);
        String opCode = memory[registers[PC]].opCode;
        while (!opCode.equals("HLT")) {
            frameNumber += 1;
            Console.println();
            displayFrameDelimiter(frameNumber);
            int operand = memory[registers[PC]].operandValue;
            Console.println("*  Current instruction Register:  "
+ opCode + " " + operand);
            registers[PC] = registers[PC] + 1;
            switch (opCode) {
                case "LDA":
                    executeLDA(memory, registers, operand);
break;
                case "STA":
                    executeSTA(memory, registers, operand);
break;
                case "LDA#":
                    executeLDAimm(registers, operand); break;
                case "ADD":
                    executeADD(memory, registers, operand);
break;
                case "JMP":
                    executeJMP(registers, operand); break;
                case "JSR":
                    executeJSR(memory, registers, operand,
Integer.parseInt(sourceCode[0])); // MP3
                break;
```

4

```
                    case "CMP#":
                        executeCMPimm(registers, operand); break;
                    case "BEQ":
                        executeBEQ(registers, operand); break;
                    case "SUB":
                        executeSUB(memory, registers, operand);
break;
                    case "SKP":
                        executeSKP(); break;
                    case "RTN":
                        executeRTN(memory, registers); break;
                    default:
                        break;
                }
                if (registers[ERR] == 0) {
                    opCode = memory[registers[PC]].opCode;
                    displayCurrentState(sourceCode, memory,
registers);
                } else {
                    opCode = "HLT";
                }
            }
            Console.println("Execution terminated");
        }

Alternative Answer:
void executeJSR(AssemblerInstruction[] memory, int[] registers,
int address) {
    int stackPointer = registers[TOS] - 1;
    if
(!memory[stackPointer].operandString.equals(EMPTY_STRING)||
            !memory[stackPointer].opCode.equals(EMPTY_STRING)) //
MP1, MP3{
        reportRunTimeError("Memory Address Error", registers); //
MP4
    } else { // MP2
        memory[stackPointer].operandValue = registers[PC] ;
        registers[PC] = address;
        registers[TOS] = stackPointer;
    }
    displayStack(memory, registers);
}
```

15	**1**	`void editsourceCode(String[] sourceCode) {`
		`    int lineNumber;`
		`    do {`
		`        Console.print("Enter line number of code to edit: ");`
		`        try {`
		`            lineNumber = Integer.parseInt(Console.readLine());`
		`        } catch (Exception e) {`
		`            lineNumber = 0;`
		`        }`
		`        if (lineNumber < 1 || lineNumber > sourceCode.length + 1) {`
		`            Console.println("Invalid line number please try again.");`
			12

```
        }
    } while (lineNumber < 1 || lineNumber > sourceCode.length +
1);
    Console.println(sourceCode[lineNumber]);
    String choice = EMPTY_STRING;
    while (!choice.equals("C")) {
        choice = EMPTY_STRING;
        while (!choice.equals("E") && !choice.equals("C") &&
                !choice.equals("D") && !choice.equals("I")) {
            Console.println("E - Edit this line");
            Console.println("D - Delete the current line");// MP4
            Console.println("I - Insert a new line above this
line");
            Console.println("C - Cancel edit");
            Console.print("Enter your choice: ");
            choice = Console.readLine();
        }
        if (choice.equals("E")) {
            Console.print("Enter the new line: ");
            sourceCode[lineNumber] = Console.readLine();
        } else if (choice.equals("D")) { // MP5
            for (int line = lineNumber; line <
sourceCode.length-1; line++) // MP2, MP6 part{
                sourceCode[line] = sourceCode[line+1]; // MP3,
MP8
            sourceCode[0] =
Integer.toString(Integer.parseInt(sourceCode[0])-1); // MP1 / MP7
part

            }
        } else if (choice.equals("I")) { // MP5 alt.
            if (Integer.parseInt(sourceCode[0]) < HI_MEM - 1) //
MP11 part {
                Console.println("Enter the new line:");
                String newLine = Console.readLine();// MP10
                for (int line = sourceCode.length-1; line >
lineNumber; line--)  // MP2 alt / MP6 part{
                    sourceCode[line] = sourceCode[line-1]; // MP3
alt MP9
                }
                sourceCode[lineNumber] = newLine; // MP12
                sourceCode[0] =
Integer.toString(Integer.parseInt(sourceCode[0])+1); // MP1 alt /
MP7 part
            } else {
                Console.println("Error - program is at maximum
length"); // MP11 part
            }
        }
        displaySourceCode(sourceCode);
    }
}
```