

**AS  
COMPUTER SCIENCE  
7516/1**

Paper 1

---

Mark scheme

June 2023

---

Version: 1:0 Final



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](http://aqa.org.uk)

#### **Copyright information**

AQA retains the copyright on all its publications. However, registered schools/colleges for AQA are permitted to copy material from this booklet for their own internal use, with the following important exception: AQA cannot give permission to schools/colleges to photocopy any material that is acknowledged to a third party even for internal use within the centre.

Copyright © 2023 AQA and its licensors. All rights reserved.

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

Qu	Marks																		
01	<p><b>6 marks for AO2 (analyse)</b></p> <table border="1" data-bbox="408 443 1251 1066"> <thead> <tr> <th data-bbox="408 443 796 510">Event / State</th> <th data-bbox="796 443 1251 510">Label(s): (A) to (I), (X) to (Z)</th> </tr> </thead> <tbody> <tr> <td data-bbox="408 510 796 577">Alarm bell ringing mode</td> <td data-bbox="796 510 1251 577">Y</td> </tr> <tr> <td data-bbox="408 577 796 645">Alert mode</td> <td data-bbox="796 577 1251 645">Z</td> </tr> <tr> <td data-bbox="408 645 796 712">Detect movement</td> <td data-bbox="796 645 1251 712">F</td> </tr> <tr> <td data-bbox="408 712 796 779">Enter correct code</td> <td data-bbox="796 712 1251 779">C, D, E</td> </tr> <tr> <td data-bbox="408 779 796 846">Enter incorrect code</td> <td data-bbox="796 779 1251 846">B, H, G</td> </tr> <tr> <td data-bbox="408 846 796 913">Sensing mode</td> <td data-bbox="796 846 1251 913">X</td> </tr> <tr> <td data-bbox="408 913 796 981">Switch on</td> <td data-bbox="796 913 1251 981">A</td> </tr> <tr> <td data-bbox="408 981 796 1066">10 second delay elapsed</td> <td data-bbox="796 981 1251 1066">I</td> </tr> </tbody> </table> <p data-bbox="256 1106 639 1137"><b>1 mark</b> per two correct labels</p> <p data-bbox="256 1173 711 1205"><b>R.</b> any labels used more than once</p> <p data-bbox="256 1209 1161 1240"><b>R.</b> more than 3 labels in “Enter correct code” or “Enter incorrect code”</p> <p data-bbox="256 1276 504 1308"><b>Max 5</b> if any errors</p>	Event / State	Label(s): (A) to (I), (X) to (Z)	Alarm bell ringing mode	Y	Alert mode	Z	Detect movement	F	Enter correct code	C, D, E	Enter incorrect code	B, H, G	Sensing mode	X	Switch on	A	10 second delay elapsed	I
Event / State	Label(s): (A) to (I), (X) to (Z)																		
Alarm bell ringing mode	Y																		
Alert mode	Z																		
Detect movement	F																		
Enter correct code	C, D, E																		
Enter incorrect code	B, H, G																		
Sensing mode	X																		
Switch on	A																		
10 second delay elapsed	I																		
	<b>6</b>																		

02

5 marks for AO2 (apply)

5

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

- 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

03		<p><b>2 marks for AO1 (knowledge)</b></p> <p>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;</p> <p><b>Alternative answer 1:</b></p> <p>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;</p> <p><b>Alternative answer 2:</b></p> <p>Local variables only exist when the program block they are in is executing; global variables exist the entire time the program is running;</p> <p><b>A.</b> local variables are stored on a stack / in a stack frame; global variables are generally stored elsewhere in the memory;</p>	2
04		<p><b>2 marks for AO1 (understand)</b></p> <p>a sequence of steps (to complete a task); <b>R.</b> set that always terminates / runs in finite time;</p>	2
05	1	<p><b>9 marks for AO3 (programming)</b></p> <p><b>Mark as follows:</b></p> <p>1) Correct variable declarations for <code>Number</code>, <code>X</code>, <code>Count</code>, <code>Multi</code>;</p> <p><b>Note to examiners</b> 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.</p> <p>2) Correct prompt <code>"Enter an integer greater than 1: "</code> and <code>Number</code> assigned integer value entered by user;</p> <p>3) Correct initialisation of <code>X</code> and <code>Count</code> before outer <code>WHILE</code> loop;</p> <p>4) Correct outer <code>WHILE</code> loop with syntax allowed by the programming language and correct condition for termination of the outer loop;</p> <p>5) Correct assignment of <code>Multi</code> in outer loop;</p> <p>6) Correct inner <code>WHILE</code> loop syntax allowed by the programming language and correct condition for termination of the loop;</p> <p>7) <code>IF</code> statement with correct condition and output within inner loop;</p> <p>8) Correct incrementation of <code>Count</code> and correct assignment to <code>Multi</code> and <code>Number</code> within inner <code>WHILE</code> loop;</p> <p>9) Correct assignments of <code>X</code> in outer loop;</p> <p><b>I.</b> minor differences in case and spelling</p> <p><b>Max 8</b> if code does not function correctly</p>	9

<p><b>05</b></p>	<p><b>2</b></p>	<p><b>Mark is for AO3 (evaluate)</b></p> <p><b>**** SCREEN CAPTURE ****</b></p> <p>Must match code from <b>05.1</b>, including prompts on screen capture matching those in code. Code for <b>05.1</b> must be sensible.</p> <p>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'</p> <p>(Accept on same or separate lines)</p> <pre> Enter a number greater than 1: 23 23 1 &gt;&gt;&gt; Enter a number greater than 1: 25 5 2 &gt;&gt;&gt; Enter a number greater than 1: 1260 2 3 5 7 6 &gt;&gt;&gt;  <b>Alternative:</b>  Enter a number greater than 1: 23 23 1 &gt;&gt;&gt; Enter a number greater than 1: 25 5 2 &gt;&gt;&gt; Enter a number greater than 1: 1260 2 3 5 7 6 &gt;&gt;&gt;                 </pre>	<p><b>1</b></p>
------------------	-----------------	--	-----------------



**Section B**

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

07	1	<p><b>Mark is for AO1 (understand)</b></p> <p>AssemblerInstruction;</p> <p><b>A.</b> Memory / SymbolTable;  <b>R.</b> if any additional code  <b>R.</b> if spelt incorrectly  <b>I.</b> case &amp; spacing</p> <p><b>Max 1</b></p>	1												
07	2	<p><b>Mark is for AO1 (understand)</b></p> <p>SourceCode / Registers / OpCodeValues;</p> <p><b>A.</b> Memory / SymbolTable; <b>(if neither given in 07.1)</b>  <b>R.</b> if any additional code  <b>R.</b> if spelt incorrectly  <b>I.</b> case &amp; spacing</p> <p><b>Max 1</b></p>	1												
08		<p><b>Mark is for AO1 (knowledge)</b></p> <p>Representational Abstraction;</p> <p><b>A.</b> Abstraction</p>	1												
09		<p><b>5 marks are for AO2 (analyse)</b></p> <table border="1" data-bbox="432 1361 1225 1818"> <thead> <tr> <th data-bbox="432 1361 997 1440">Proposed error message</th> <th data-bbox="997 1361 1225 1440">Error code</th> </tr> </thead> <tbody> <tr> <td data-bbox="432 1440 997 1518">Duplicate label found</td> <td data-bbox="997 1440 1225 1518">3</td> </tr> <tr> <td data-bbox="432 1518 997 1597">File not found</td> <td data-bbox="997 1518 1225 1597">1</td> </tr> <tr> <td data-bbox="432 1597 997 1675">No assembled code to run</td> <td data-bbox="997 1597 1225 1675">10</td> </tr> <tr> <td data-bbox="432 1675 997 1753">No source code to display</td> <td data-bbox="997 1675 1225 1753">7</td> </tr> <tr> <td data-bbox="432 1753 997 1818">Unknown opcode</td> <td data-bbox="997 1753 1225 1818">5</td> </tr> </tbody> </table> <p><b>1 mark</b> for each correct error code  <b>R.</b> if more than one error code assigned to an error message</p>	Proposed error message	Error code	Duplicate label found	3	File not found	1	No assembled code to run	10	No source code to display	7	Unknown opcode	5	5
Proposed error message	Error code														
Duplicate label found	3														
File not found	1														
No assembled code to run	10														
No source code to display	7														
Unknown opcode	5														

10		<p><b>2 marks for AO2 (analyse)</b></p> <p>Operand is not empty / an empty string;          Operand is not an existing label // Operand is not in the symbol table; <b>NE.</b> invalid operand          Operand is not (the string representation of) a number;</p> <p><b>Max 2</b></p>	2
11	1	<p><b>2 marks for AO2 (analyse)</b></p> <p>Check each character in the line / operand; <b>A.</b> 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;</p>	2
11	2	<p><b>2 marks for AO2 (analyse)</b></p> <p>If there is an '*' // if there is a comment (symbol);          Only use the characters to left of '*' // remove the comment (from the string);</p>	2
11	3	<p><b>2 marks for AO2 (analyse)</b></p> <p>When code encounters a '*' // When code encounters a comment (symbol);          Can <b>stop</b> iteration when */comment is encountered // no need to examine to the end of the string;</p> <p><b>Alternative answer:</b></p> <p>If there are multiple asterisks, it finds the first one (rather than the last one);          Additional asterisks within the comment will not affect assembly;</p> <p><b>R.</b> a variable (<code>ThisPosition</code>) to hold the position wouldn't be needed</p>	2

## Section C

Qu		Marks
12	1	<p><b>4 marks for AO3 (programming)</b></p> <p><b>Mark as follows:</b></p> <ol style="list-style-type: none"><li>1) Add required parameter to subroutine call in <code>Execute</code>;</li><li>2) Add required parameter to <code>ExecuteSKP</code> definition;</li><li>3) Add 1 to <code>Registers[ACC]</code>;</li><li>4) Update status register (by calling <code>SetFlags</code> with correct parameters);</li></ol> <p><b>Max 3</b> if any errors</p>

<b>12</b>	<b>2</b>	<p><b>Mark is for AO3 (evaluate)</b></p> <p><b>**** SCREEN CAPTURE ****</b></p> <p>Must match code from <b>12.1</b>, including prompts on screen capture matching those in code. Code for <b>12.1</b> must be sensible.</p> <p>Screen capture showing (values changing from Frame 0 to Frame 5 shown highlighted):</p> <pre> Enter your choice: R ***** Frame 0 ***** * * Memory      Location  Label  Op   Operand Comment * Contents * 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 * 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                 </pre>	<b>1</b>
-----------	----------	--	----------

Qu		Marks
13	1	<p><b>5 marks for AO3 (programming)</b></p> <p><b>Mark as follows:</b></p> <ol style="list-style-type: none"> <li>1) Check for non-integer input;</li> <li>2) Check within valid lower boundary;</li> <li>3) Check within valid upper boundary;</li> <li>4) At least 2 correct checks will be repeated until valid data is input at which point the loop exits;</li> <li>5) Output suitable error message(s) under appropriate circumstances based upon at least 2 correct checks; <b>R.</b> if message is displayed when it should not be</li> </ol> <p><b>Max 4</b> if any errors</p>
13	2	<p><b>Mark is for AO3 (evaluate)</b></p> <p><b>**** SCREEN CAPTURE ****</b></p> <p>Must match code from <b>13.1</b>, including prompts on screen capture matching those in code. Code for <b>13.1</b> must be sensible.</p> <p>Screen capture showing:</p> <pre> 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:                     </pre>

14	1	<p><b>2 marks for AO3 (design) and 2 marks for AO3 (programming)</b></p> <p><b>Marking guidance:</b></p> <p><b>Evidence of AO3 design – 2 points:</b></p> <p>Evidence of design to look for in response:</p> <ol style="list-style-type: none"> <li>1) Check in <code>ExecuteJSR</code> that stack does not overwrite instruction / data;</li> <li>2) Recognise that instructions for <code>JSR</code> should only be executed if no error;</li> </ol> <p><b>Note:</b> 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.</p> <p><b>Evidence of AO3 programming – 2 points:</b></p> <p>Evidence of programming to look for in response:</p> <ol style="list-style-type: none"> <li>3) Correct value for number of program lines passed into subroutine // check that memory location pointed to by <code>TOS</code> is empty;</li> <li>4) <code>ReportRunTimeError</code> called with suitable message in appropriate place;</li> </ol> <p><b>Max 3</b> if code does not function correctly</p>	4
14	2	<p><b>Mark is for AO3 (evaluate)</b></p> <p><b>**** SCREEN CAPTURE ****</b></p> <p>Must match code from <b>14.1</b>, including prompts on screen capture matching those in code. Code for <b>14.1</b> must be sensible.</p> <pre> ***** Frame 7 ***** * Current Instruction Register: JSR 7 Run time error: Memory Address Error Stack contents: ----   3     14   ---- Execution terminated </pre>	1

<b>15</b>	<b>1</b>	<b>3 marks for AO3 (design) and 9 marks for AO3 (programming)</b>	<b>12</b>												
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 10%;">Level</th> <th style="width: 60%;">Description</th> <th style="width: 30%;">Mark Range</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">3</td> <td>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).</td> <td style="text-align: center;">9–12</td> </tr> <tr> <td style="text-align: center;">2</td> <td>There is evidence that a line of reasoning has been partially followed. There is evidence of some appropriate design work. The subroutine <code>EditSourceCode</code> has been amended and has some added functionality.</td> <td style="text-align: center;">5–8</td> </tr> <tr> <td style="text-align: center;">1</td> <td>An attempt has been made to amend the subroutine <code>EditSourceCode</code>. 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.</td> <td style="text-align: center;">1–4</td> </tr> </tbody> </table>				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 <code>EditSourceCode</code> has been amended and has some added functionality.	5–8	1	An attempt has been made to amend the subroutine <code>EditSourceCode</code> . 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
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 <code>EditSourceCode</code> has been amended and has some added functionality.	5–8													
1	An attempt has been made to amend the subroutine <code>EditSourceCode</code> . 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													
<p><b>Marking guidance:</b></p> <p><b>Evidence of AO3 design – 3 marks:</b></p> <p>Evidence of design to look for in response:</p> <ol style="list-style-type: none"> <li>1) Adjust the number of lines stored in <code>SourceCode</code> (ie update <code>SourceCode[0]</code>)</li> <li>2) Loop through program lines consecutively or equivalent</li> <li>3) Move program lines after specified location in <code>SourceCode</code></li> </ol> <p><b>Note:</b> 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.</p> <p><b>A.</b> design evidence in option D or I code.</p> <p><b>Evidence of AO3 programming – 9 marks:</b></p> <p>Evidence of programming to look for in response:</p> <ol style="list-style-type: none"> <li>4) Insert <code>D / I / both</code> in addition to existing options in the menu and add <code>D / I / both</code> to conditions of <code>WHILE</code> loop</li> <li>5) Add selection to test for option <code>D / I / both</code> after <code>WHILE</code> loop</li> <li>6) Use correct range to loop through program lines in both options, D and I</li> <li>7) Correctly adjust the number of lines stored in <code>SourceCode</code> in both options, D and I</li> </ol>															



	<p>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 <code>SourceCode</code></p> <p><b>Max 11</b> if code does not function correctly</p>	
--	---	--

<p><b>15</b></p>	<p><b>2</b> <b>Mark is for AO3 (evaluate)</b></p> <p><b>**** SCREEN CAPTURE ****</b>                  Must match code from <b>15.1</b>, including prompts on screen capture matching those in code.                  Code for <b>15.1</b> must be sensible.</p> <p>Screen capture showing (for ease of reference inserted line highlighted):</p> <pre> 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                 </pre>	<p><b>1</b></p>
------------------	---	-----------------

15	3	<p><b>Mark is for AO3 (evaluate)</b></p> <p><b>**** SCREEN CAPTURE ****</b></p> <p>Must match code from <b>15.1</b>, including prompts on screen capture matching those in code. Code for <b>15.1</b> must be sensible.</p> <p>Screen capture showing (for ease of reference inserted line highlighted):</p> <p>Enter your choice: E Enter line number of code to edit: 4  <code>STA FINAL</code>  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:  <code>LABEL: SKP</code></p> <pre> 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 </pre>	1
----	---	--	---

VB.Net

<p>05</p>	<p>1</p>	<pre> Console.WriteLine("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 &gt; 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)                 </pre>	<p>9</p>
<p>12</p>	<p>1</p>	<pre> 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 &lt;&gt; "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)         End Select     End While End Sub                 </pre>	<p>4</p>

		<pre> 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 </pre>	
13	1	<pre> Sub EditSourceCode (ByRef SourceCode () As String)     Dim Choice As String = EMPTY_STRING     Dim LineNumber As Integer     Do         Console.WriteLine ("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 &gt; 0 And LineNumber &lt;= Convert.ToInt32 (SourceCode (0)) ' MP2, MP3, MP4          Console.WriteLine (SourceCode (LineNumber))         While Choice &lt;&gt; "C"             Choice = EMPTY_STRING             While Choice &lt;&gt; "E" And Choice &lt;&gt; "C"                 Console.WriteLine ("E - Edit this line")                 Console.WriteLine ("C - Cancel edit")                 Console.WriteLine ("Enter your choice: ")                 Choice = Console.ReadLine ()             End While             If Choice = "E" Then                 Console.WriteLine ("Enter the new line: ")                 SourceCode (LineNumber) = Console.ReadLine ()             End If             DisplaySourceCode (SourceCode)         End While     End Sub </pre>	5
14	1	<pre> 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 &lt;= 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) </pre>	4

		<pre> 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 &lt;&gt; "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, <b>Int(SourceCode(0))</b>) '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 </pre>	
<p><b>15</b></p>	<p><b>1</b></p>	<pre> Sub EditSourceCode(ByRef SourceCode() As String)     Dim LineNumber As Integer     ...     While Choice &lt;&gt; "C"         Choice = Empty_String </pre>	<p><b>12</b></p>

```

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

<p>05</p>	<p>1</p>	<pre> Number = int(input("Enter a number greater than 1: ")) # MP2 X = 2 # MP1 Count = 0 # MP3 while Number &gt; 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) </pre>	<p>9</p>
<p>12</p>	<p>1</p>	<pre> 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) </pre>	<p>4</p>

		<pre> else:     OpCode = "HLT"     print("Execution terminated") </pre>	
13	1	<pre> 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) &gt; 0 and int(LineNumberStr) &lt;= 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 </pre>	5



14	1	<pre> def ExecuteJSR(Memory, Registers, Address, MaxLines):     StackPointer = Registers[TOS] - 1     if StackPointer &lt;= 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") </pre>	4
----	---	---	---

	<p><b>Alternative method:</b></p> <pre>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</pre>	
<p>15 1</p>	<pre>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]) &lt; 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</pre>	<p>12</p>

Python 2

<p>05</p>	<p>1</p>	<pre> Number = int(raw_input("Enter a number greater than 1: ")) # MP2 X = 2 # MP1 Count = 0 # MP3 while Number &gt; 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                     </pre>	<p>9</p>
<p>12</p>	<p>1</p>	<pre> 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":                     </pre>	<p>4</p>

		<pre> Registers = ExecuteRTN(Memory, Registers) if Registers[ERR] == 0:     OpCode = Memory[Registers[PC]].OpCode     DisplayCurrentState(SourceCode, Memory, Registers) else:     OpCode = "HLT" print "Execution terminated" </pre>	
13	1	<pre> def EditSourceCode(SourceCode):     <b>LineNumber = 0</b>     <b>NumberOfLines = int(SourceCode[0])</b>     <b>while LineNumber not in range(1, NumberOfLines + 1): # MP2,</b> <b>MP3, MP4</b>         <b>try: # MP1</b>             LineNumber = int(raw_input("Enter line number of code to edit: ").replace("\r", ""))             <b>if LineNumber not in range(1, NumberOfLines + 1):</b>                 <b>print "Not a valid line number" # MP5</b>             <b>except:</b>                 <b>print "Not a valid number"</b>         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  <b>Alternative method:</b> def EditSourceCode(SourceCode):     LineNumberStr = raw_input("Enter line number of code to edit: ").replace("\r", "")     <b>Valid = False</b>     <b>while not Valid:</b>         <b>if LineNumberStr.isdigit() and int(LineNumberStr) &gt; 0 and</b> <b>int(LineNumberStr) &lt;= int(SourceCode[0]): # MP1, MP2, MP3, MP4</b>             <b>Valid = True</b>             <b>LineNumber = int(LineNumberStr)</b>         <b>else:</b>             <b>print "Not a valid line number" # MP5</b>             <b>LineNumberStr = raw_input("Enter line number of code to</b> <b>edit: ").replace("\r", "")</b>     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" </pre>	5

		<pre> 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 </pre>	
14	1	<pre> def ExecuteJSR(Memory, Registers, Address, MaxLines):     StackPointer = Registers[TOS] - 1     if StackPointer &lt;= 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": </pre>	4

	<pre> Registers = ExecuteRTN(Memory, Registers) if Registers[ERR] == 0:     OpCode = Memory[Registers[PC]].OpCode     DisplayCurrentState(SourceCode, Memory, Registers) else:     OpCode = "HLT" print "Execution terminated"  <b>Alternative method:</b> def ExecuteJSR(Memory, Registers, Address):     StackPointer = Registers[TOS] - 1     <b>if Memory[StackPointer].OperandString != "" or</b> <b>Memory[StackPointer].OpCode != "": # MP1, MP3</b>         <b>ReportRuntimeError("Memory Address Error", Registers) # MP4</b>     <b>else: #MP2</b>         Memory[StackPointer].OperandValue = Registers[PC]         Registers[PC] = Address         Registers[TOS] = StackPointer         DisplayStack(Memory, Registers)     return Memory, Registers </pre>	
<p>15 1</p>	<pre> 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"             <b>print "D - Delete this line"</b>         <b># MP4</b>             <b>print "I - Insert a new line above this line"</b>         #             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", "")             <b>if Choice == "D": # MP5</b>                 <b>NumberOfLines = int(SourceCode[0]) #</b>                 <b>NumberOfLines -= 1 #</b>                 <b>SourceCode[0] = str(NumberOfLines) # MP1, MP7 part</b>                 <b>for ThisLine in range(LineNumber, NumberOfLines + 1): #</b> <b>MP2, MP6 part</b>                     <b>SourceCode[ThisLine] = SourceCode[ThisLine + 1] # MP3,</b> <b>MP8</b>                 <b>if Choice == "I": # MP5 alt.</b>                     <b>if int(SourceCode[0]) &lt; HI_MEM - 1: # MP11 part</b>                         <b>print "Enter the new line:" #</b>                         <b>NewLine = raw_input() # MP10</b>                         <b>NumberOfLines = int(SourceCode[0]) #</b>                         <b>NumberOfLines += 1 #</b>                         <b>SourceCode[0] = str(NumberOfLines) # MP1 alt./MP7 part</b> </pre>	<p>12</p>

	<pre>        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</pre>	
--	---	--

Pascal

<p>05 1</p>	<pre> 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 &gt; 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. </pre>	<p>9</p>
<p>12 1</p>	<pre> 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 &lt;&gt; 'HLT' do   begin     FrameNumber := FrameNumber + 1;     writeln;     DisplayFrameDelimiter(FrameNumber);     Operand := Memory[Registers[PC]].OperandValue; </pre>	<p>4</p>



	<pre> 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; </pre>	
<p>13 1</p>	<pre> procedure EditSourceCode(var SourceCode: TSourceCode); var   LineNumber, NumberOfLines: integer;   Choice: string; begin   LineNumber := 0;   NumberOfLines := StrToInt(SourceCode[0]);   while (LineNumber &lt; 1) or (LineNumber &gt;= NumberOfLines) do   // MP2, MP3, MP4   begin     try // MP1       write('Enter line number of code to edit: ');       readln(LineNumber);       if (LineNumber &lt; 1) or (LineNumber &gt;= 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 &lt;&gt; 'C' do       begin         Choice := EMPTY_STRING; </pre>	<p>5</p>

	<pre> while (Choice &lt;&gt; 'E') and (Choice &lt;&gt; '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; </pre>	
<p><b>14</b> <b>1</b></p>	<pre> <b>procedure</b> ExecuteJSR(<b>var</b> Memory: TMemory; <b>var</b> Registers: TRegisters; Address: integer; <b>MaxLines</b>: integer); <b>var</b>   StackPointer: integer; <b>begin</b>   StackPointer := Registers[TOS] - 1;   <b>if</b> StackPointer &lt;= MaxLines // MP1   <b>then</b>     ReportRunTimeError('Memory Address Error', Registers) // MP4   <b>else</b> // MP2     <b>begin</b>       Memory[StackPointer].OperandValue := Registers[PC];       Registers[PC] := Address;       Registers[TOS] := StackPointer;     <b>end</b>;     DisplayStack(Memory, Registers)   <b>end</b>;  <b>procedure</b> Execute(<b>var</b> SourceCode: TSourceCode; <b>var</b> Memory: TMemory); <b>var</b>   Registers: TRegisters;   OpCode: string;   Operand, FrameNumber: integer; <b>begin</b>   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;   <b>while</b> OpCode &lt;&gt; 'HLT' <b>do</b>   <b>begin</b>     FrameNumber := FrameNumber + 1;     writeln;     DisplayFrameDelimiter(FrameNumber);     Operand := Memory[Registers[PC]].OperandValue; </pre>	<p><b>4</b></p>

	<pre>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, <b>StrToInt(SourceCode[0])</b>); // <b>MP3</b> 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;</pre> <p><b>Alternative method:</b></p> <pre>procedure ExecuteJSR(var Memory: TMemory; var Registers: TRegisters; Address: integer); var StackPointer: integer; begin StackPointer := Registers[TOS] - 1; <b>if (Memory[StackPointer].OperandString &lt;&gt; EMPTY_STRING) or</b> <b>Memory[StackPointer].OpCode &lt;&gt; EMPTY_STRING // MP1, MP3</b> <b>then</b> <b>ReportRunTimeError('Memory Address Error', Registers) // MP4</b> <b>else // MP2</b> begin Memory[StackPointer].OperandValue := Registers[PC]; Registers[PC] := Address; Registers[TOS] := StackPointer; end; DisplayStack(Memory, Registers) end;</pre>	
<p><b>15</b> <b>1</b></p>	<pre>procedure EditSourceCode(var SourceCode: TSourceCode); var LineNumber, NumberOfLines, ThisLine: integer; Choice: string; NewLine: string; begin</pre>	<p><b>12</b></p>

```

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');

```

	<pre>end; // MP11 part DisplaySourceCode (SourceCode); end; end;</pre>	
--	--	--

C#

<p>05</p>	<p>1</p>	<pre> bool multi; Console.WriteLine("Enter an integer greater than 1: "); int number = Convert.ToInt32(Console.ReadLine()); // MP2 int x = 2; // MP1 int count = 0; // MP3 while (number &gt; 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);                 </pre>	<p>9</p>
<p>12</p>	<p>1</p>	<pre> 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;         }     } }                 </pre>	<p>4</p>

		<pre>         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(<b>registers</b>); 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"); } </pre>	
13	1	<pre> private static void EditSourceCode(string[] sourceCode) {     int lineNumber = 0;     <b>int numberOfLines = Convert.ToInt32(sourceCode[0]);</b>     <b>while (lineNumber &lt; 1    lineNumber &gt;= numberOfLines) // MP2, MP3,</b> <b>MP4</b>     {         <b>Try // MP1</b>         {             Console.Write("Enter line number of code to edit: ");             lineNumber = Convert.ToInt32(Console.ReadLine());             <b>if (lineNumber &lt; 1    lineNumber &gt;= numberOfLines)</b>             {                 <b>Console.WriteLine("Not a valid line number"); // MP5</b>             }         }         <b>catch (Exception)</b>         {             <b>Console.WriteLine("Not a valid number");</b>         }     }     Console.WriteLine(sourceCode[lineNumber]);     string choice = EMPTY_STRING;     while (choice != "C")     {         choice = EMPTY_STRING;         while (choice != "E" &amp;&amp; choice != "C")         {             Console.WriteLine("E - Edit this line");             Console.WriteLine("C - Cancel edit");             Console.Write("Enter your choice: ");             choice = Console.ReadLine();         }     }     if (choice == "E")     { </pre>	5

		<pre>                 Console.WriteLine("Enter the new line:");                 sourceCode[lineNumber] = Console.ReadLine();             }             DisplaySourceCode(sourceCode);         }     } </pre>	
14	1	<pre> private static void ExecuteJSR(AssemblerInstruction[] memory, int[] registers, int address, int maxLines) {     int stackPointer = registers[TOS] - 1;     if (stackPointer &lt;= 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; </pre>	4



	<pre>         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"); } </pre> <p><b>Alternative method:</b></p> <pre> 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); } </pre>	
<p><b>15</b></p>	<p><b>1</b></p> <pre> private static void EditSourceCode(string[] sourceCode) {     int lineNumber = 0;     int numberOfLines = Convert.ToInt32(sourceCode[0]);     while (lineNumber &lt; 1    lineNumber &gt;= numberOfLines)     {         try         {             Console.Write("Enter line number of code to edit: ");             lineNumber = Convert.ToInt32(Console.ReadLine());             if (lineNumber &lt; 1    lineNumber &gt;= numberOfLines)             {                 Console.WriteLine("Not a valid line number");             }         }         catch (Exception)         {             Console.WriteLine("Not a valid number");         }     } } </pre>	<p><b>12</b></p>

```

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	<pre> 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 &gt; 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); </pre>	9
12	1	<pre> 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; </pre>	4

	<pre>                 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(<b>registers</b>); 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");     } </pre>	
<p><b>13</b> <b>1</b></p>	<pre> void editSourceCode(String[] sourceCode) {     <b>int</b> lineNumber;     <b>do</b> {         Console.print("Enter line number of code to edit: ");         <b>try</b> { // MP1             lineNumber = Integer.parseInt(Console.readLine());         } <b>catch</b> (Exception e) {             lineNumber = 0;         }         <b>if</b> (lineNumber &lt; 1    lineNumber &gt; sourceCode.length + 1) {             Console.println("Invalid line number please try again."); // MP5         }         } <b>while</b> (lineNumber &lt; 1    lineNumber &gt; sourceCode.length + 1); // MP2, MP3, MP4         Console.println(sourceCode[lineNumber]);         String choice = EMPTY_STRING;         <b>while</b> (!choice.equals("C")) {             choice = EMPTY_STRING;             <b>while</b> (!choice.equals("E") &amp;&amp; !choice.equals("C")) {                 Console.println("E - Edit this line");                 Console.println("C - Cancel edit");                 Console.print("Enter your choice: ");                 choice = Console.readLine();             }         }         <b>if</b> (choice.equals("E")) {             Console.print("Enter the new line: "); </pre>	<p><b>5</b></p>

		<pre> sourceCode[lineNumber] = Console.readLine(); displaySourceCode(sourceCode);     } } </pre>	
14	1	<pre> void executeJSR(AssemblerInstruction[] memory, int[] registers, int address, int maxLines) {     int stackPointer = registers[TOS] - 1;     if (stackPointer &lt;= 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; </pre>	4

	<pre> 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"); }  <b>Alternative Answer:</b> 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); } </pre>	
<p><b>15</b>   <b>1</b></p>	<pre> 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 &lt; 1    lineNumber &gt; sourceCode.length + 1) {             Console.println("Invalid line number please try again."); } } </pre>	<p><b>12</b></p>

```

    }
    } 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);
    }
}

```