

# THE LITTLE MAN COMPUTER

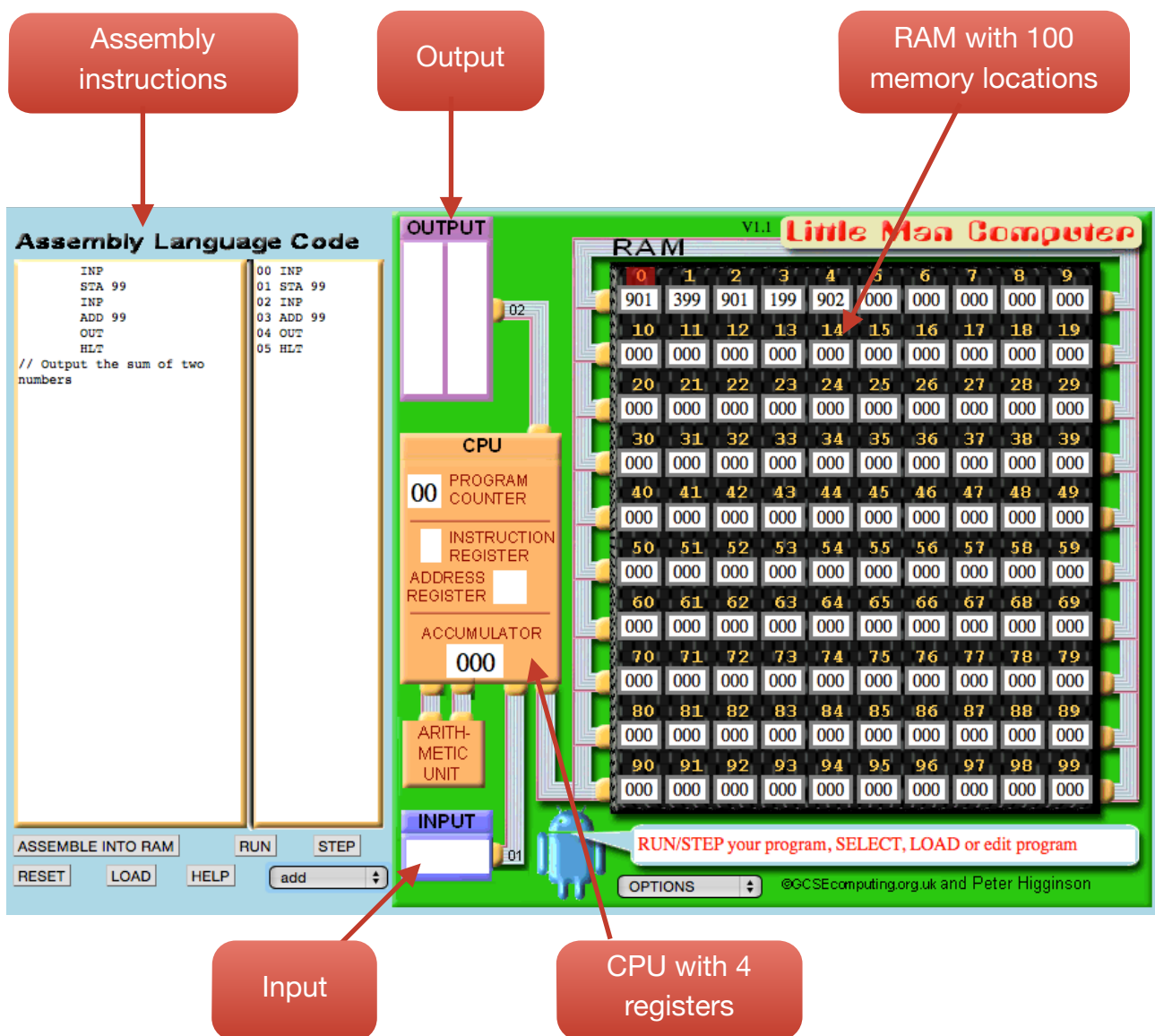
Name: \_\_\_\_\_

Group: \_\_\_\_\_

# INTRODUCTION

Most modern computers have a processor which executes instructions and memory which stores both the instructions and any data the processor needs to use. The computer has ways of getting data from the user and ways of giving the results of any processing to the user as outputs.

Modern computers are very complex machines but we can work with a simple version of a computer. This will teach us a great deal about how the computer actually works, while keeping the details quite simple to deal with. The **Little Man Computer** is a simulation of a modern computer system.



The image above is an example Little Man Computer (LMC). The LMC has 100 memory addresses or mailboxes. These can store either an instruction or data. The LMC only deals with whole number instructions or data from -999 to 999. The memory locations are numbered from 0 to 99. The processor can access the memory locations based on the instructions it has.

It has a Processor (CPU) which has 4 special memory locations called registers. These store data or instructions that the processor is currently dealing with.

CPU register	Function
<b>Accumulator</b>	This stores data that is being used in calculations. It can perform simple addition and subtraction.
<b>Program Counter</b>	This contains the memory address of the next instruction to be loaded. This automatically ticks to the next memory address when an instruction is loaded. It can be altered during the running of the program depending on the state of the accumulator.
<b>Instruction Register</b>	An Instruction Register to hold the top digit of the instruction read from memory.
<b>Address register</b>	An Address Register to hold the bottom two digits of the instruction read from memory.
<b>Input</b>	This registers allows the user to input numerical data to the LMC.
<b>Output</b>	This shows the data to output to the user.

## HOW THE LMC WORKS

Modern computers work by **fetching** an instruction from the memory. It then **decodes** the instruction so that it knows what to do. It then **executes** the instruction, carrying out the commands, before starting all over again. This is called the **Fetch-decode-execute cycle**.

The LMC understands a set of instructions and know what to do when these instructions are decoded. The LMC only understands a very limited set of instructions to show how a real processor works without becoming too complex. The list of instructions we can use is known as an **instruction set**.

The LMC will start to load the instruction from the memory address in the program counter. When the LMC first loads up this will set at zero. This memory location needs to be an instruction and will be dealt with as such. When the data is loaded the program counter is incremented to the next memory location. The memory location register will be updated with the address being accessed and the data retrieved will be stored in the memory data register.

The instructions and data exist in the memory together. It is important however that they are kept separate. The instructions of a program take up the first set of memory addresses and the data needed takes up the next set of memory addresses.

If the LMC loads data from a memory location and treats it as an instruction, chances are that it will not understand it and not function correctly or crash. It is possible to program the LMC to write over an instruction with data. This will probably cause the LMC to crash or function incorrectly.

# PROGRAMMING THE LMC

The LMC needs to have a computer program. The LMC only understands numbers. This can make it difficult for humans to deal with, so there are some 3 letter keywords that can be used to write a program. Each instruction is then compiled into computer code.

Mnemonic code	Instruction	Numeric code	Description
ADD	ADD	1xx	Add the value stored in mailbox xx to whatever value is currently on the accumulator (calculator). Note: the contents of the mailbox are not changed, and the actions of the accumulator (calculator) are not defined for add instructions that cause sums larger than 3 digits.
SUB	SUBTRACT	2xx	Subtract the value stored in mailbox xx from whatever value is currently on the accumulator (calculator). Note: the contents of the mailbox are not changed, and the actions of the accumulator are not defined for subtract instructions that cause negative results - however, a negative flag will be set so that <b>8xx (BRP)</b> can be used properly.
STA	STORE	3xx	Store the contents of the accumulator in mailbox xx (destructive). Note: the contents of the accumulator (calculator) are not changed (non-destructive), but contents of mailbox are replaced regardless of what was in there (destructive)
LDA	LOAD	5xx	Load the value from mailbox xx (non-destructive) and enter it in the accumulator (destructive).
BRA	BRANCH (unconditional)	6xx	Set the program counter to the given address (value xx). That is, value xx will be the next instruction executed.
BRZ	BRANCH IF ZERO (conditional)	7xx	If the accumulator (calculator) contains the value 000, set the program counter to the value xx. Otherwise, do nothing. Note: since the program is stored in memory, data and program instructions all have the same address/location format.
BRP	BRANCH IF POSITIVE (conditional)	8xx	If the accumulator (calculator) is 0 or positive, set the program counter to the value xx. Otherwise, do nothing. Note: since the program is stored in memory, data and program instructions all have the same address/location format.
INP	INPUT	901	Go to the INBOX, fetch the value from the user, and put it in the accumulator (calculator) Note: this will overwrite whatever value was in the accumulator (destructive)
OUT	OUTPUT	902	Copy the value from the accumulator (calculator) to the OUTBOX. Note: the contents of the accumulator are not changed (non-destructive).
HLT	HALT	0	Stop working.
DAT	DATA		This is an assembler instruction which simply loads the value into the next available mailbox. DAT can also be used in conjunction with labels to declare variables. For example, DAT 984 will store the value 984 into a mailbox at the address of the DAT instruction.

# Examples

## INPUT/OUTPUT

This program simply allows the user to input a number and then it is output back to the user. Complete the table converting each instruction into its numeric equivalent.

Instruction no	Assembly code	Numeric Code
0	INP	
1	OUT	
2	HLT	

Write the assembly code into the the message box so each line instruction is on a line of its own. Then click on the compile button. This converts the codes into the numeric equivalents and loads them into memory starting at location 00.

Run the program.

- INP will ask you to input a number. This will be copied into the In-box register and then copied into the accumulator.
- OUT will take the contents of the accumulator and place them in the Out-box.
- HLT will end the program.

## USING MEMORY

This program show how memory is used. The user inputs a value and then stores the value into memory. It then inputs another value and saves that into memory. The first value is then loaded and then placed into the outbox, followed by the second number.

Instruction no	Assembly code	Numeric Code
0	INP	
1	STA FIRST	
2	INP	
3	STA SECOND	
4	LDA FIRST	
5	OUT	
6	LDA SECOND	
7	OUT	
8	HLT	
9	FIRST DAT	
10	SECOND DAT	

The data FIRST and SECOND need to be converted into a memory address. The compiler looks at the line the FIRST DAT is located and uses that to calculate the memory address. So FIRST will be at address 08 and SECOND will be at address 09. It then replaces the labels in the program with these locations.

Complete the table filling in the numeric code.

Step through the program one step at a time. See how each instruction is loaded and executed.

## ADDING THREE NUMBERS

This program takes three numbers from the user and adds them together before giving the result to the user. The first two numbers need to be stored as inputting a number wipes the contents of the accumulator. Once the last number has been inputted the program adds the first and second numbers to the contents of the accumulator.

Instruction no	Assembly code	Numeric Code
0	INP	
1	STA first	
2	INP	
3	STA second	
4	INP	
5	STA third	
6	ADD first	
7	ADD second	
8	OUT	
9	HLT	
10	first DAT	
11	second DAT	
12	third DAT	

Complete the table.

Enter the code into the message box and compile the program.

Step through the program checking what each instruction actually does.

# BIGGEST

This program introduces branching. This is a way to alter the flow of the program based on the value stored in the accumulator. The program asks for two numbers, storing each into memory. The second number is subtracted from the first. If the number is positive then the first number is loaded and sent to the outbox. If the accumulator is negative then the second number is shown. If they are the same then zero will be shown to the user.

Instruction no	Assembly code	Numeric Code
0	INP	
1	STA first	
2	INP	
3	STA second	
4	LDA first	
5	SUB second	
6	BRP FIRSTBIG	
7	LDA second	
8	OUT	
9	HLT	
10	FIRSTBIG BRZ SAME	
11	LDA first	
12	OUT	
13	HLT	
14	SAME LDA zero	
15	OUT	
16	HLT	
17	first DAT	
18	second DAT	
19	zero DAT 0	

Again the labels SAME and FIRSTBIG refer to memory locations that need to be calculated. The instruction numbers are used to calculate these. So the label FIRSTBIG is instruction no 10 and so it is replaced in all those locations where it is used. SAME is in location no 14 and so that location is used when the label SAME is found.

Branch Positive (BRP) will jump to the memory location SAME if the value in the accumulator is zero or positive. Branch zero (BRZ) will jump to the memory location SAME if the value in the accumulator is zero.

Run this program three times.

- The first time enter 7 followed by 5. Step through the program and follow the instructions so you understand what is happening.
- The second time enter 8 followed by 10. Step through the program and follow the instructions so you understand what is happening.
- The third time enter the 9 followed by 9. Step through the program and follow the instructions so you understand what is happening.

Enter the program and step through the instructions. Enter a low value for the second number (no higher than 5) or the program will take some time to complete.




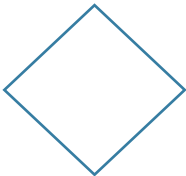
## HOW TO WRITE AN LMC PROGRAM

Writing an LMC program can be quite a challenge. As the instruction set is very limited we often need to perform what seems to us to be a very simple task in an even simpler way.

For example, looking at the program Biggest we want to compare to numbers and output the largest. However we do not have an instruction to compare numbers. We do however have a subtract instruction. If we were to enter two numbers and subtract the second from the first, if the result was positive that would tell us that the first number was the largest. If it was negative then the second number was largest. We can combine this with a Branch if Positive (BRP) instruction to alter the flow of our program.

Using a Flow chart to help write the program is very helpful. When the flow chart is created we can simply look at each shape on the chart and think what instructions would we need to have for that shape. These will often be no more than a couple of lines of LMC code.

In flow charts there are 4 symbols that we commonly use.

Symbol	Meaning	LMC instructions
	Start / Stop	Start has no instruction but Stop is HLT.
	Input / output	Any inputs will that need to be saved will be INP followed by an STA command to store the value. OUT is the output command. It may need to be
	Process	This could be a DAT command where we see variables initialised (e.g. counter = 0). addition and subtraction commands fit into this. A process such as $X = X + Y$ would need to be done in the correct order. So we would Load X, Add Y and then store the result as X. This would be <b>LDA X, ADD Y, STA X</b>
	Decision	There are only two instructions that can have two alternatives. Branch if Positive and Branch if Zero. If the test is true then the program can branch to another part of the program. If not the program carries on.



## WORKED EXAMPLE

We want a program to calculate averages. We want to be able to keep entering values until we enter a zero. The average is then calculated and displayed.

First thing create a flow chart to show what needs to be done.

Be as detailed as you can be.

Note any values you need to remember. These will be the variables. In LMC code they will become the DAT commands. Note if they have a start value.

We have 4 variables

- **total DAT 0**
- **count DAT 0**
- **result DAT 0**
- **num DAT**

If we need to add on or subtract a specific value we need to be able to store that too. We need to be able to add 1 so we can do this by having **one DAT 1**.

Now start at the top and write down the commands for the instructions for the flow chart. Assigning values can be ignored so the first command in Input number The LMC command is INP. If we need to store that we need to follow this with a store command and save it to memory using the DAT label we created. So we can turn Input num into **INP**

**STA num**

Next we see if the user has entered a zero. We can use Branch Zero to do this. If the accumulator is zero we will jump to another section of the code. We need to give this section a label. I will call this section **CALCULATE**. I will need to do that code later.

So our next command is

**BRZ CALCULATE**

The next section of code happens if num does NOT equal zero. Now I need to add the num to the total. I will load the total and then add the num.

**LDA total**

**ADD num**

This then needs to be saved back as the total.

**STA total**

Now I need to add one to the count. So I need to load count and then add one.

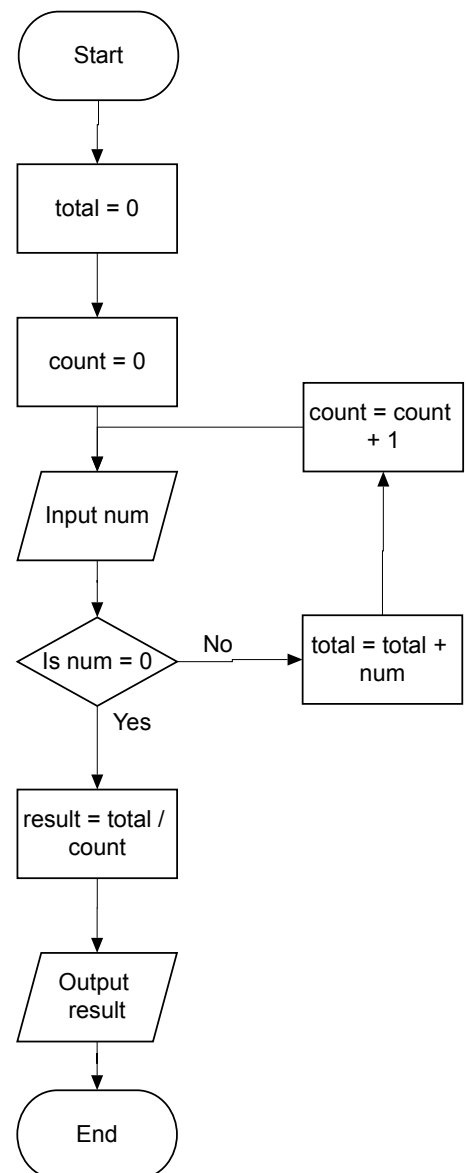
The result needs to be saved as count. So the next commands are:

**LDA count**

**ADD one**

**STA count**

The code now loops back to the Input command. We can use a Branch always command to do this. We need to label where we want the BRA command to jump to. I will call it **LOOPTOP**. I need to add this label to the INP command and use it in the BRA command.



The code we have so far is:

```
LOOPTOP INP  
STA num  
BRZ CALCULATE  
LDA total  
ADD num  
STA total  
LDA count  
ADD one  
STA count  
BRA LOOPTOP
```

Now we need to go back to our **CALCULATE** code. This code performs a division. LMC does not have a divide command. We can perform a divide by repeatedly subtracting the count from the total until we get to zero. Keeping a count of how many times we successfully subtract the count will be the same as dividing. The result will store this count. This code will run in a loop. We need to load total and then subtract the count. We then need to see if the count is below zero. If it is not we will add one to the result and then loop around. So the first command is to load the total and subtract the count.

```
LDA total  
SUB count
```

The total needs to be stored again. Then we Branch if the result is zero or higher, so we need BRP in order to add one to the result. I will give it the label **DIVIDE** and deal with that later. So we add

```
STA total  
BRP DIVIDE
```

If the value in the accumulator is negative then the BRP does not run. We now need to load the result and output it to the user. Once we do that the program is done.

```
LDA RESULT  
OUT  
HLT
```

Now we need to go back to what happens if the total - count is positive. Remember we jumped to a label called **DIVIDE**. We need to add one to the result and then start the loop again. We can use Branch always to jump back to the top of our loop. The loop already has a label, so we can use that.

```
DIVIDE LDA result  
ADD one  
STA result  
BRA CALCULATE
```

All that remains is to add the DAT commands to the end of our program.

```
total DAT 0  
count DAT 0  
num DAT  
result DAT 0  
one DAT 1
```

The completed program is therefore:

```

LOOPTOP INP
STA num
BRZ CALCULATE
LDA total
ADD num
STA total
LDA count
ADD one
STA count
BRA LOOPTOP
CALCULATE LDA total
SUB count
STA total
BRP DIVIDE
LDA RESULT
OUT
HLT
DIVIDE LDA result
ADD one
STA result
BRA CALCULATE
total DAT 0
count DAT 0
num DAT
result DAT 0
one DAT 1
```

# CHALLENGES

## SIMPLE

1. Write a program that enters 3 numbers such as 5, 8, 3 and then outputs them in reverse as in 3, 8, 5.
2. Write a program that asks for two numbers. It then outputs the first number subtracting the second number and then outputs the second number subtracting the first number. Eg if 3 and 5 were entered the first sum would be 3-5 and the second sum would be 5-3.
3. Write a program that outputs the negative of a positive number. So if 7 is entered -7 is output.

## INTERMEDIATE

1. Ask the user for 2 numbers. If they are the same then double the number and print it out. If they are different then print them both out individually.
2. Ask the user for 2 numbers. Print out biggest, then the smallest.
3. Ask the user for 2 numbers, print out the result of the biggest number minus the smallest number

## ADVANCED

1. Ask the user for a big number, then a small number. Using only a BRP to loop round, keep subtracting the smaller number until you get past zero, then output the result.
2. You can declare a constant at the end of the program like this: one DAT 1 (this will give the variable 'one' the value 1). Using this, add to your previous program to count the number of times you can successfully subtract the smaller number.

# PREPARING FOR THE A452 COURSEWORK

This coursework is an investigation. This means that you will be given a series of tasks and challenges that you need to work through. It is likely at first that you will come across things that you don't know. Part of the task is documenting how you find out the answers. You may use the Internet but you need to reference any sites you have used.

There is a practical investigation. Here you will need to explain the following as you work through the tasks.

- Follow instructions to the letter, document each step: use task headings and annotated screenshots of your activities
- Document your planning for each section with reasons: "I will find you how to [...] by visiting the following sites: [...]. This will help me because [...]"
- Report any decision processes you have made: "I did consider using (...) to solve this problem, but decided to use (...) instead, because (...)"
- Show that you have experimented with different ways of solving the task: use annotated screenshots
- Show that you have learnt something new: "I was unaware that (...), now I know that (...)"

When you code your solutions make sure that they are NOT too complex and they are well structured. Make sure you explain your solutions and screenshot evidence.

You need to show that you are acquiring technical understanding.

- Point out understanding of programming principles and techniques: "I am using (...), which means (...), because (...)"
- Use technical terms where possible: produce a glossary of terms or use footnotes
- Use naming conventions, e.g. camel code
- Compare the task to 'real world' software development: "I have found (no) evidence that the technology of (...), is used in the industry. For example (...)" - naming your sources!
- Show that you know more than the task asks for!

At the end you will need to evaluate what you have done and what you have learnt.

- Be critical and honest
- Write clearly and at length, showing excellent written communication skills
- Be specific and detailed in your reasoning, referring to examples from your activities and solutions
- Make reference to relevant programming methods
- Make reference to any relevant organisations
- Make reference to past, current and future developments