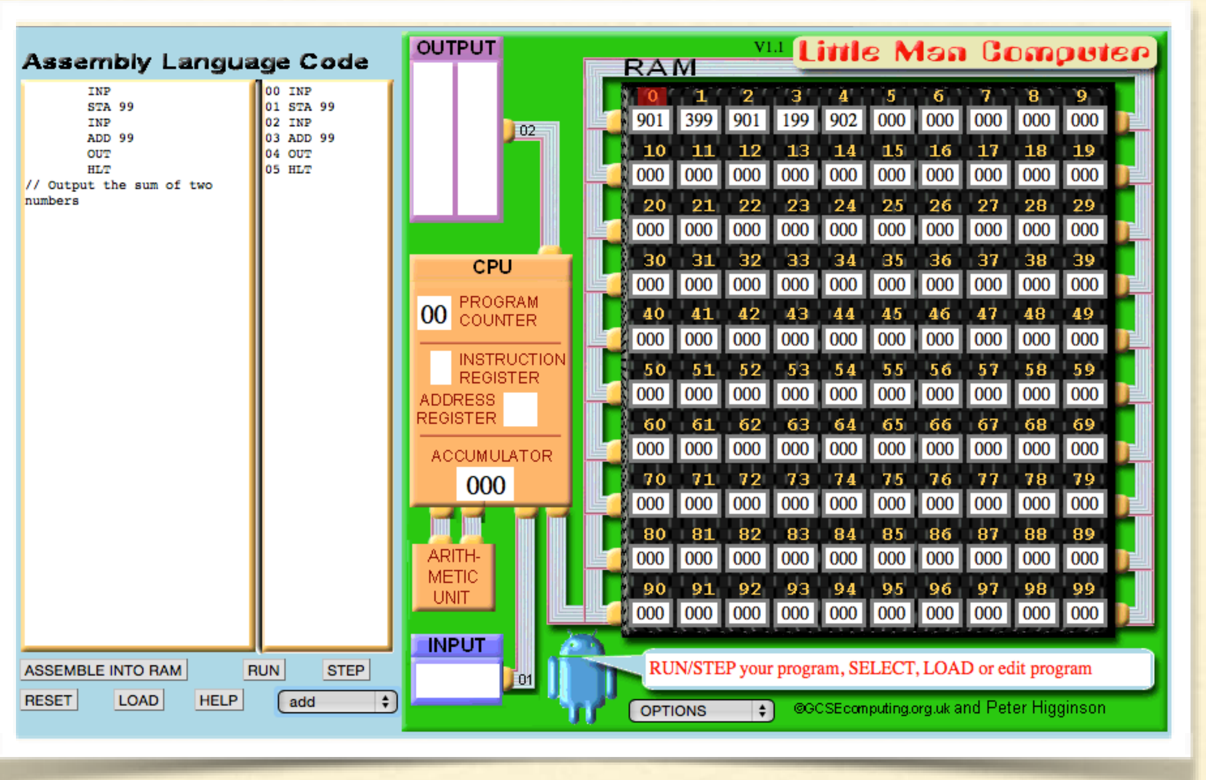
THE LITTLE MAN COMPUTER

WHAT IS THE LITTLE MAN COMPUTER?

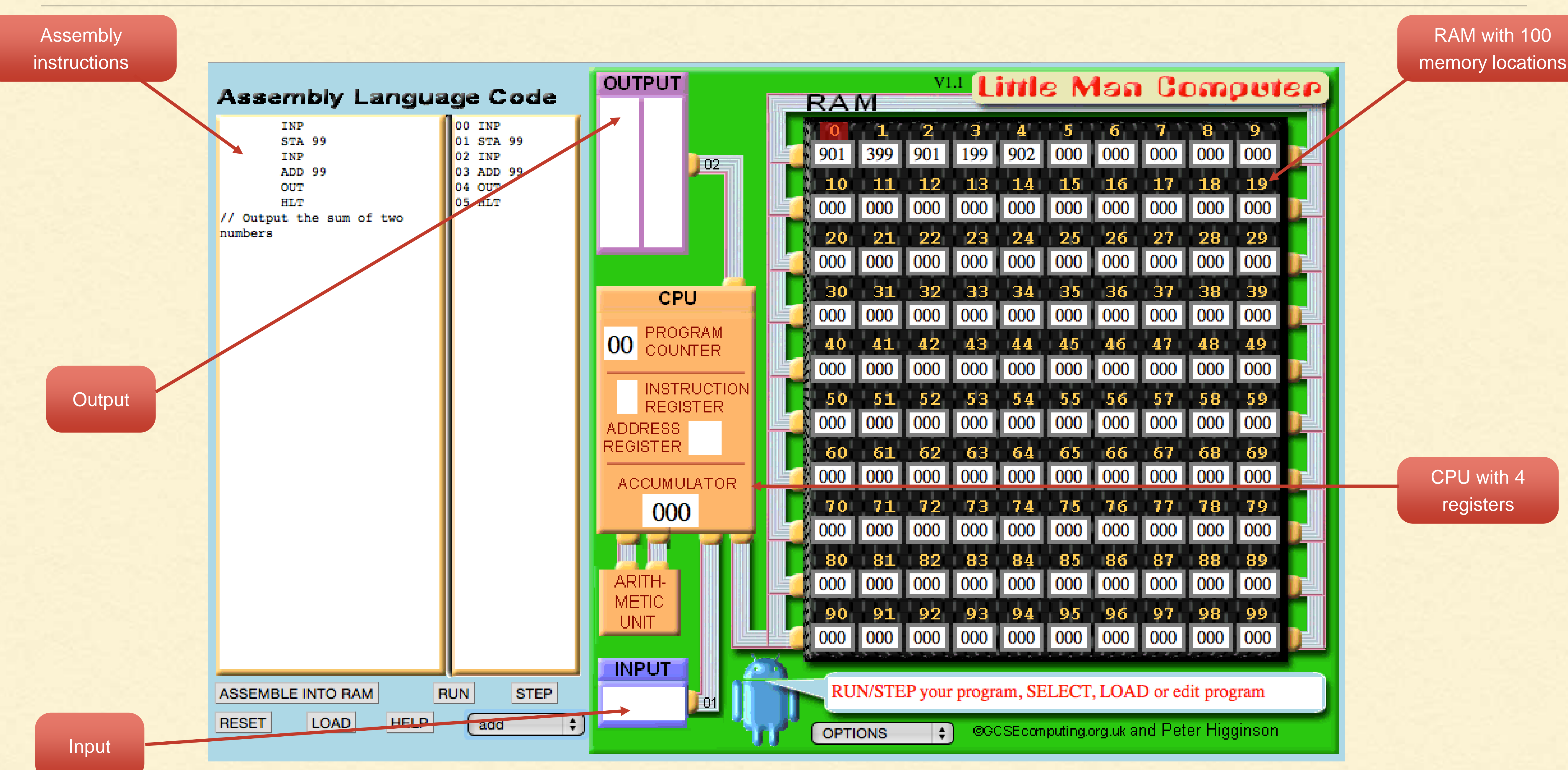
- Most modern computers have a processor which executes instructions and memory which stores both the instructions and any data the processor needs to use.
- The computer has ways of getting data from the user and ways of giving the results of any processing to the user as outputs.
- Modern computers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

WHICH LMC?



- There are many implementations of the Little Man Computer (LMC).
- We will use the excellent web based one that can be found here: http://peterhigginson.co.uk/LMC/

PARTS OF THE LMC



PARTS OF THE CPU

CPU REGISTER	FUNCTION
Accumulator	This stores data that is being used in calculations. It can perform simple addition and subtraction.
Program Counter	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.
Instruction Register	An Instruction Register to hold the top digit of the instruction read from memory.
Address register	An Address Register to hold the bottom two digits of the instruction read from memory.
Input	This registers allows the user to input numerical data to the LMC.
Output	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.

HOW THE LMC WORKS

- 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.

LMC INSTRUCTION SET

- The LMC processor understands 10 basic commands (plus 1 instruction to label data).
- The LMC only understands these instructions in a numerical form.
- This can be difficult for us to program in so there is a set of mnemonics we can use instead. This is known as assembly language. This will be converted into the LMC code before the program can run.

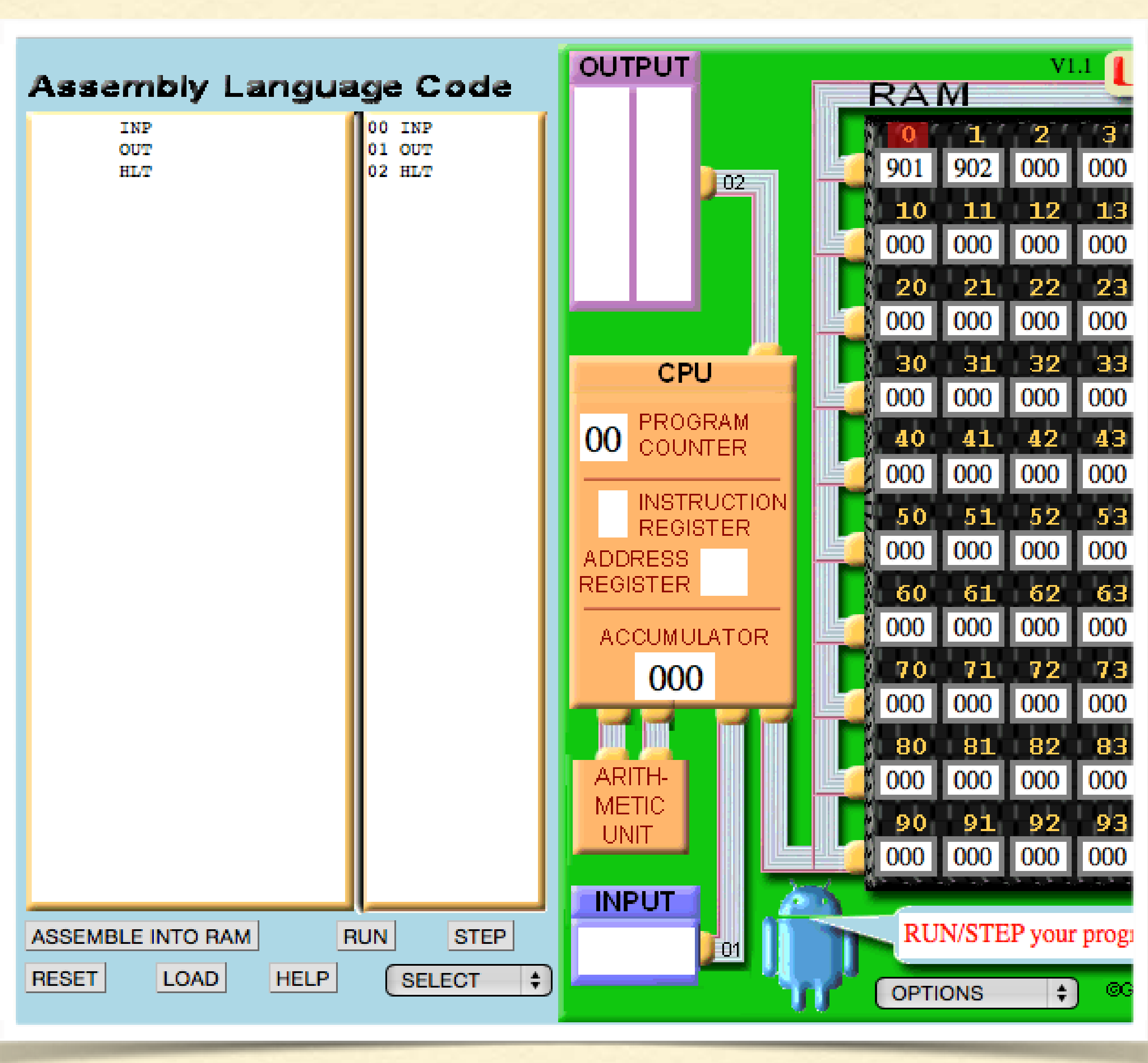
LMC INSTRUCTION SET

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 8xx (BRP) 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).
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).

LMC INSTRUCTION SET

MNEMONIC CODE	INSTRUCTION	NUMERIC CODE	DESCRIPTION
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.
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 asks the user for an input and then outputs what was input.
- The program has been assembled into RAM and you can see the numeric codes for the instructions in the first three memory locations.

USING MEMORY

```
        INP
        STA FIRST
        INP
        STA SECOND
        LDA FIRST
        OUT
        LDA SECOND
        OUT
        HLT
FIRST   DAT
SECOND  DAT
```

- This program asks the user to input a number.
- This is stored in a memory location defined by the DAT label.
- A second number is asked for and stored.
- These numbers are then loaded and output in order.

BIGGER

```
        INP
        STA FIRST
        INP
        STA SECOND
        SUB FIRST
        BRP FIRSTBIG
        LDA SECOND
        OUT
        HLT
FIRSTBIG  BRZ SAME
        LDA FIRST
        OUT
        HLT
SAME      LDA ZERO
        OUT
        HLT
FIRST     DAT
SECOND    DAT
ZERO      DAT 0
```

- This program uses two branch commands to alter the path of the program.
- There is no greater than or less than command so we simply subtract the second number from the first.
- If it is positive then the first number must have been bigger so we branch if positive.
- The two numbers could be the same however so we need to check to see if the result is zero. We branch if it is.
- The biggest number is output or zero if they are both the same.

POINTS TO NOTE

- The instruction set is very limited so you often need to come up with a different way to perform things like multiplication or comparing two numbers.
- The LMC does not store decimals.
- The LMC does not have a loop structure but you can use a Branch Always command to redirect the code to an earlier command.

HOW TO WRITE A LITTLE MAN COMPUTER PROGRAM

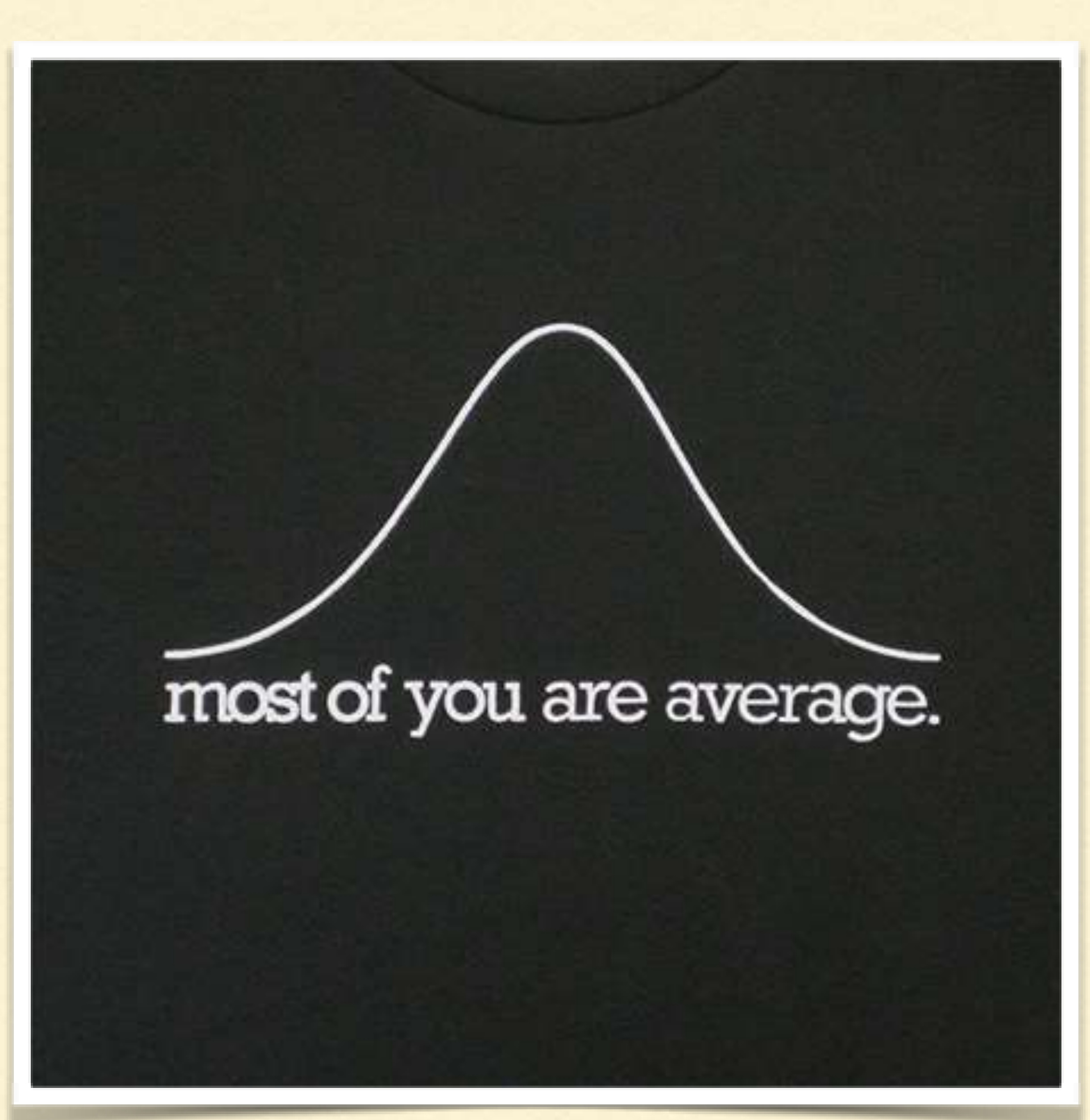
- Writing an LMC program can be quite a challenge. As the instruction set is very limited we often need to perform what seems to us to be a very simple task in an even simpler way.
- Using a Flow chart to help write the program is very helpful.
- When the flow chart is created we can simply look at each shape on the chart and think what instructions would we need to have for that shape. These will often be no more that a couple of lines of LMC code.

HOW TO WRITE A LITTLE MAN COMPUTER PROGRAM

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

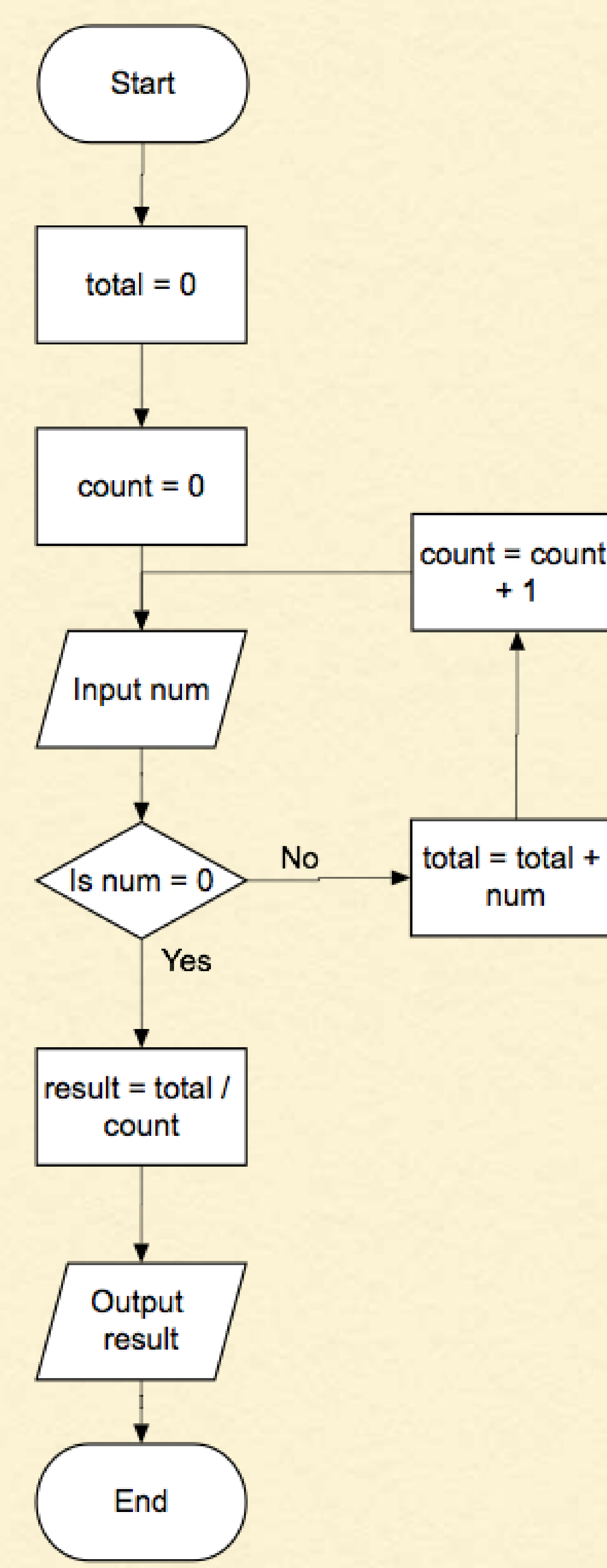
SYMBOL	MEANING	LMC INSTUCTIONS
	Start / Stop	Start has no instruction but Stop is HLT.
	Input / output	Any inputs will that need to be saved will be INP followed by an STA command to store the value. OUT is the output command. It may need to be
	Process	This could be a DAT command where we see variables 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 **LDA X, ADD Y, STA X**
	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.

EXAMPLE - THE PROBLEM

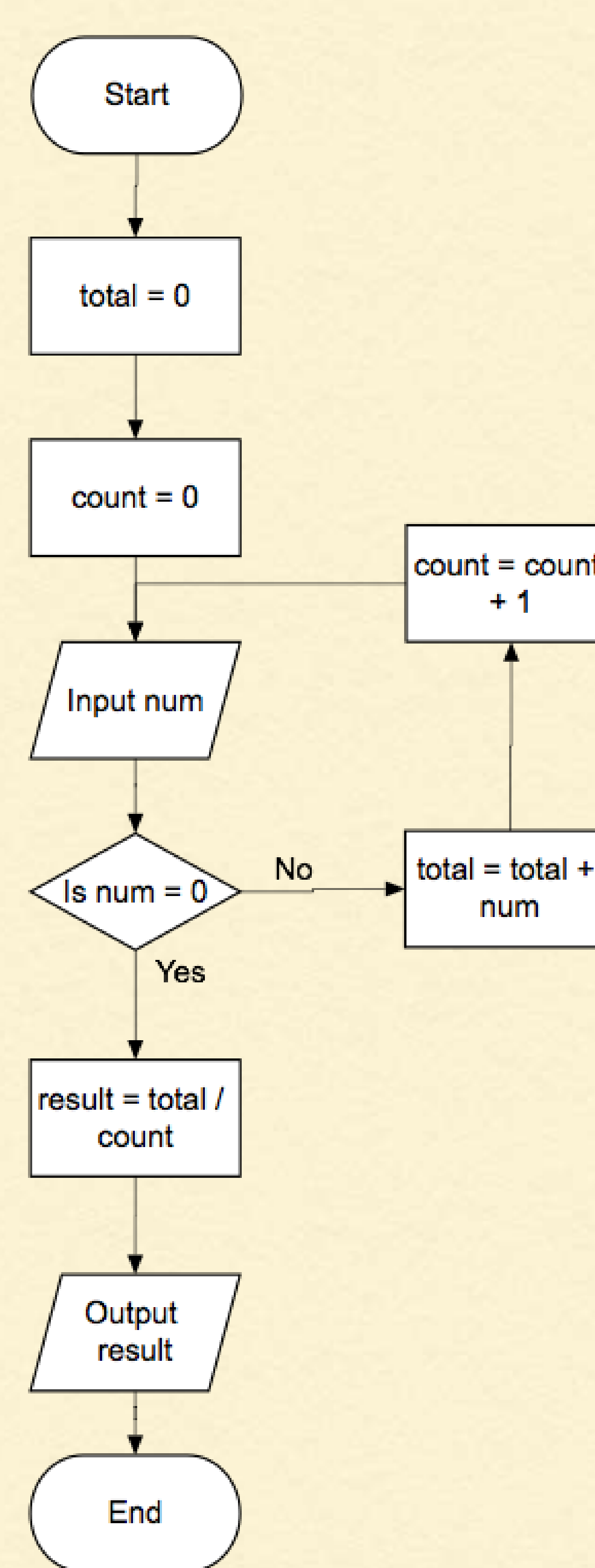


- We want a program to calculate averages.
- We want to be able to keep entering values until we enter a zero.
- The average is then calculated and displayed.

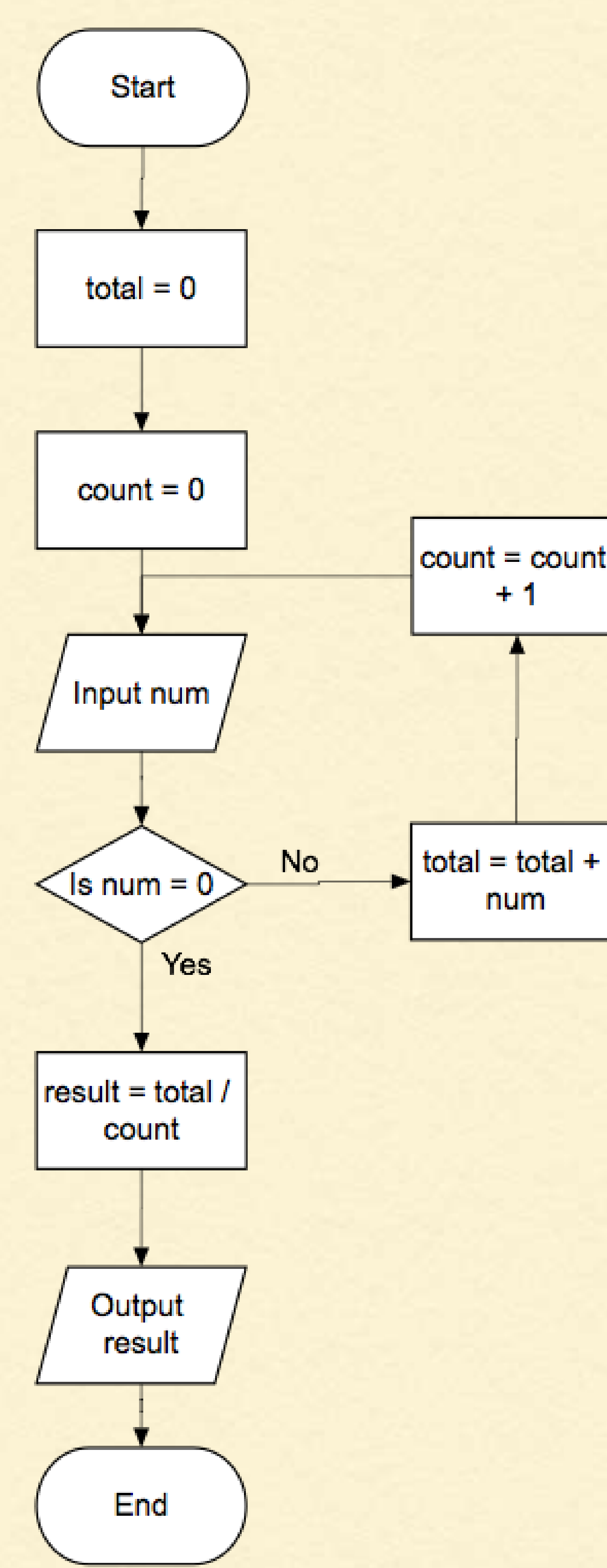
- First thing - create a flow chart to show what needs to be done.
- Be as detailed as you can be.



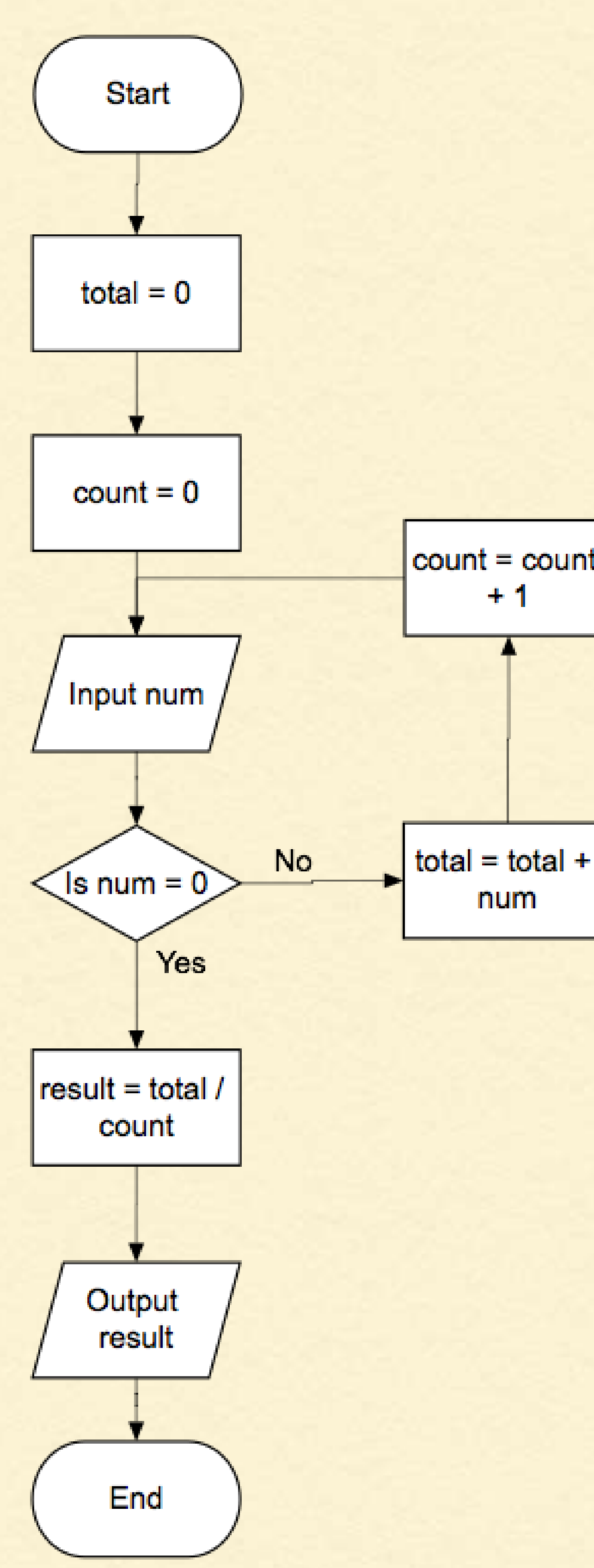
- Note any values you need to remember.
- These will be the variables.
- In LMC code they will become the DAT commands.
- Note if they have a start value.



- We have 4 variables
- total DAT 0
- count DAT 0
- result DAT 0
- num DAT

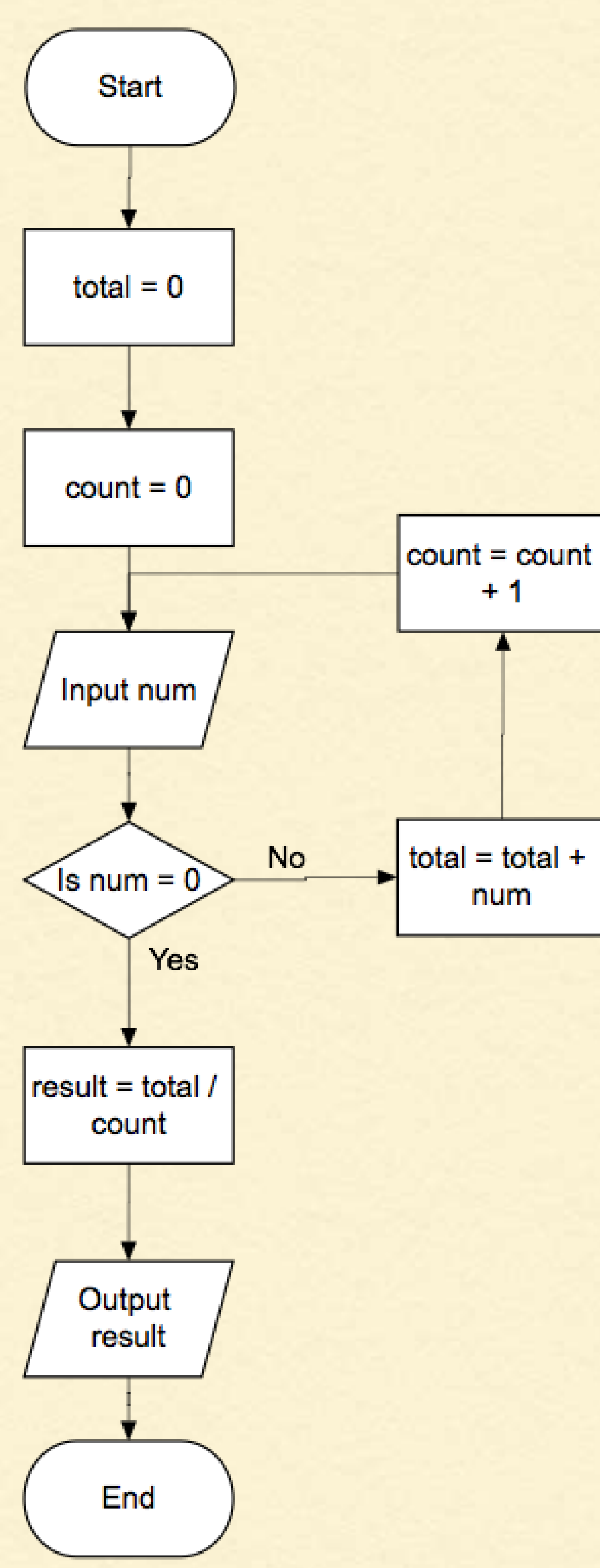


- If we need to add on or subtract a specific value we need to be able to store that too.
- We need to be able to add 1 do we can do this by having
- one DAT 1



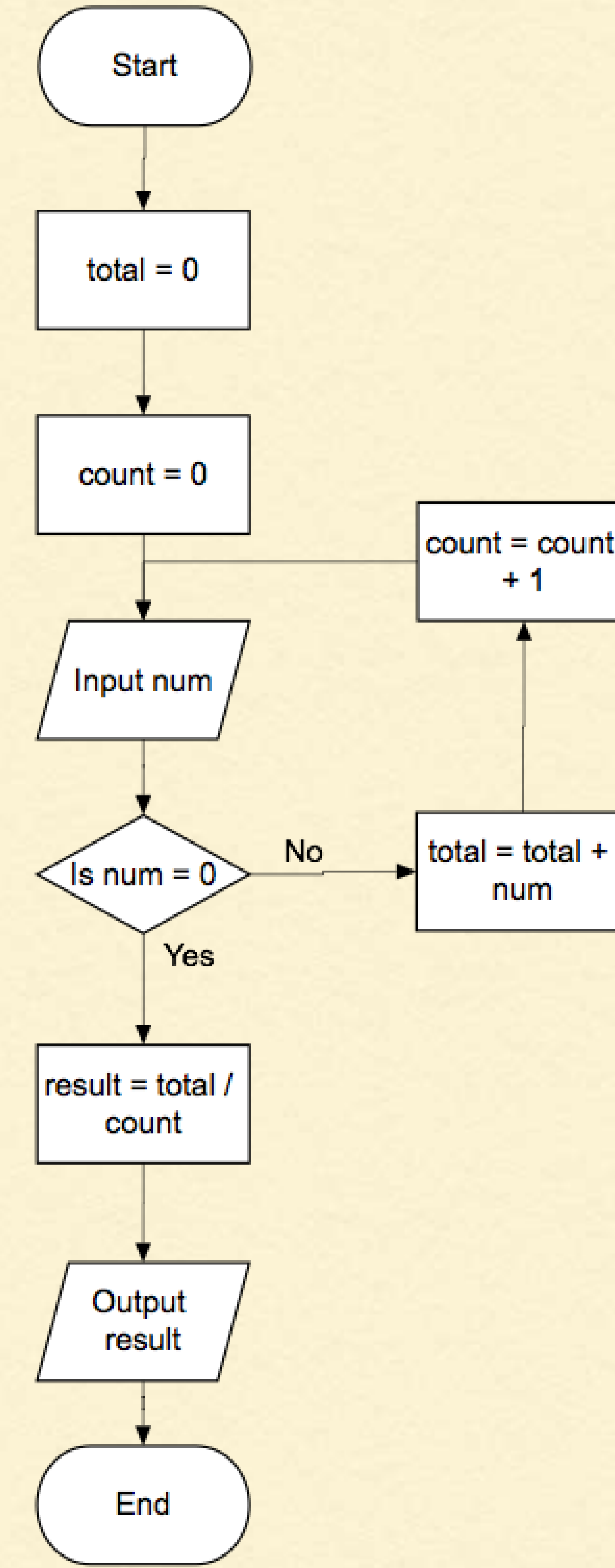
- Now start at the top and write down the commands for the instructions for the flow chart.
- Assigning values can be ignored so the first command in Input number
- The LMC command is INP
- If we need to store that we need to follow this with a store command and save it to memory using the DAT label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

INP
STA num



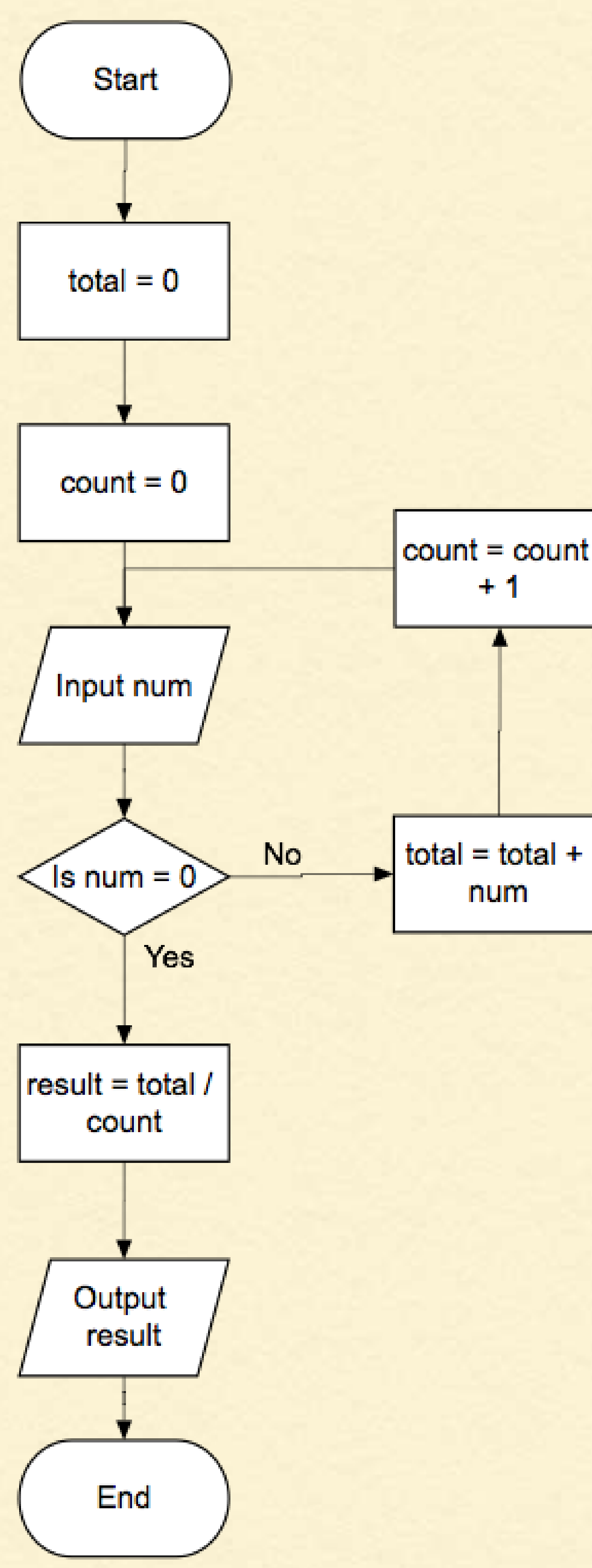
- Next we see if the user has entered a zero.
- We can use Branch Zero to do this.
- If the accumulator is zero we will jump to another section of the code.
- We need to give this section a label. I will call this section CALCULATE.
- I will need to do that code later.

```
INP
STA num
BRZ CALCULATE
```



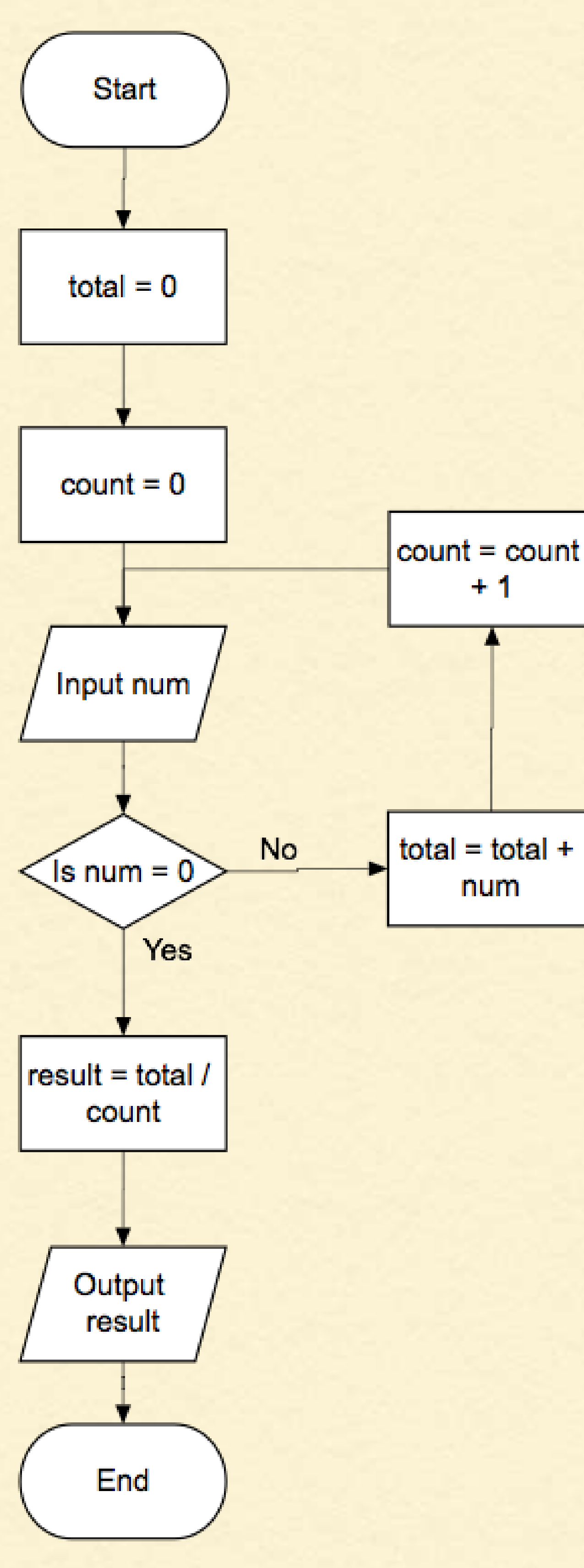
- The next section of code happens if num does NOT equal zero.
- Now I need to add the num to the total.
- I will load the total and then add the num.
    - LDA total
    - ADD num
- This then needs to be saved back as the total.
    - STA total

```
INP
STA num
BRZ CALCULATE
LDA total
ADD num
STA total
```



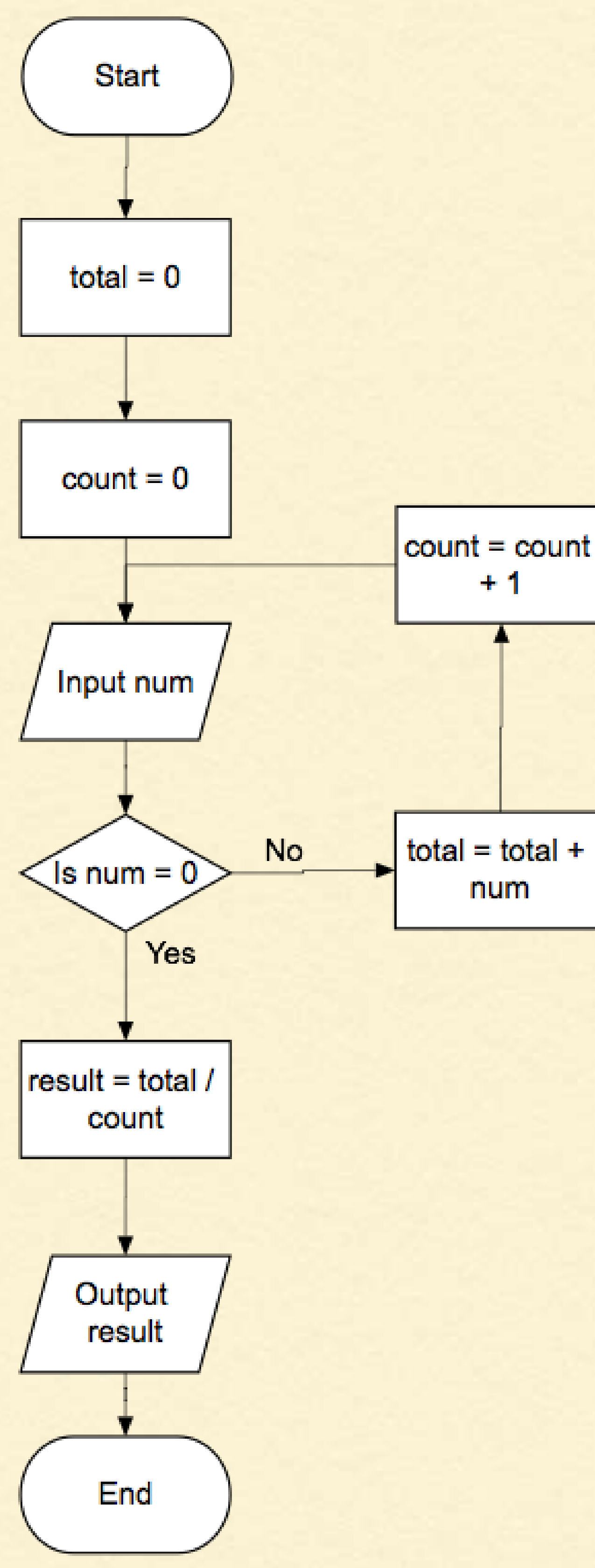
- Now I need to add one to the count.
- So i need to load count and then add one.
- The result needs to be saved as count

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



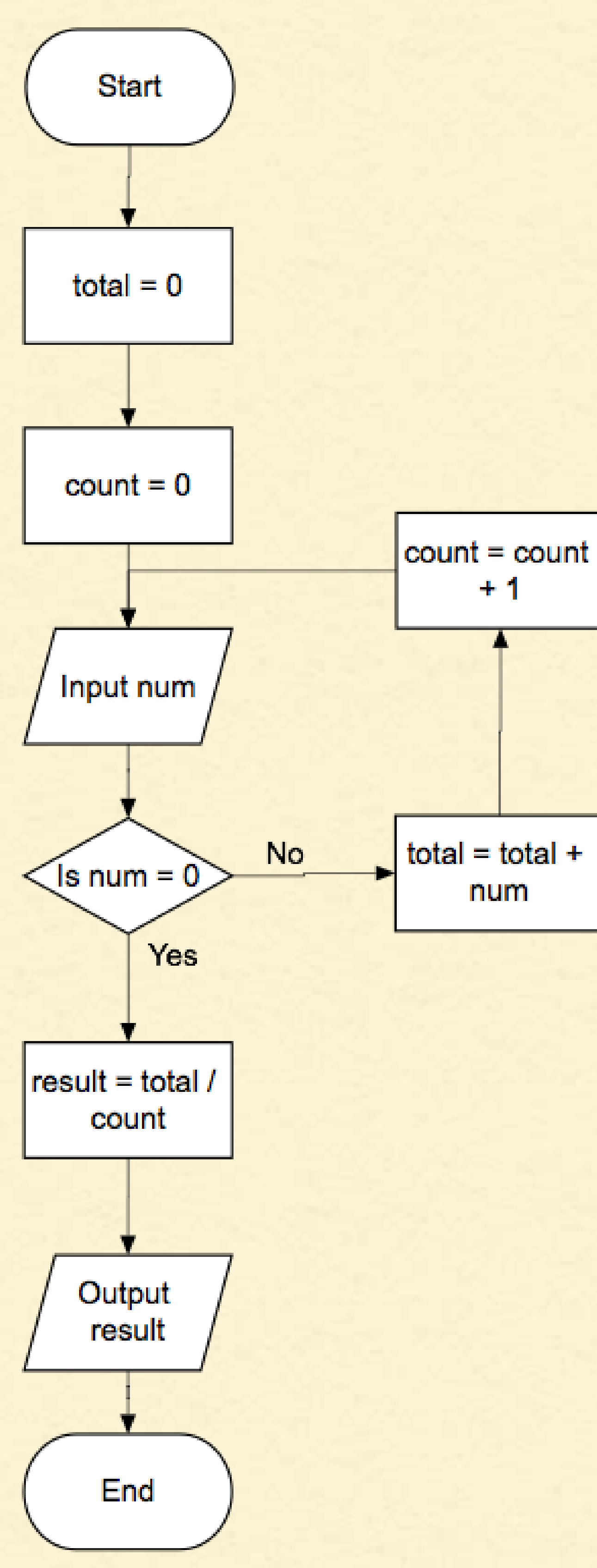
- The code now loops back to the Input command.
- We can use a Branch always command to do this.
- We need to label where we want the BRA command to jump to.
- I will call it LOOPTOP.
- I need to add this label to the INP command and use it in the BRA command.

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



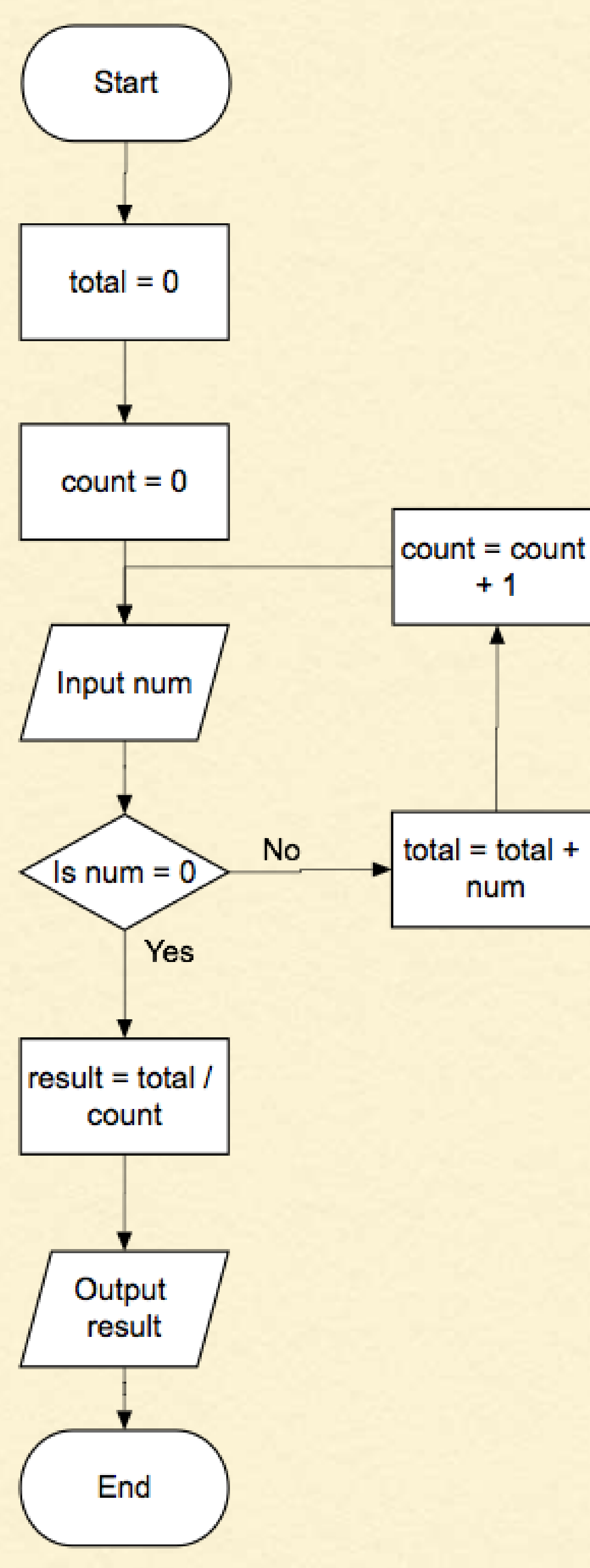
- Now we need to go back to out CALCULATE code.
- This code performs a division.
- LMC does not have a divide command.

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



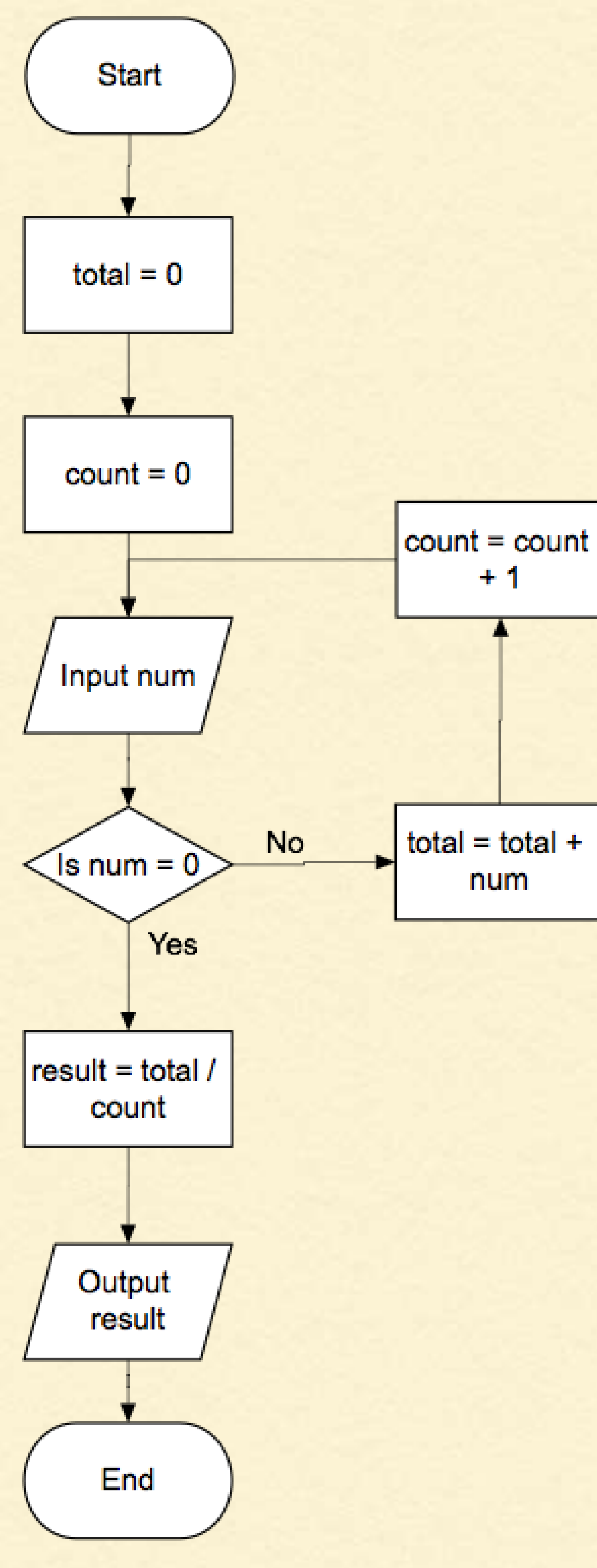
- We can perform a divide by repeatedly subtracting the count from the total until we get to zero.
- Keeping a count of how many times we successfully subtract the count will be the same as dividing.
- The result will store this count.

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



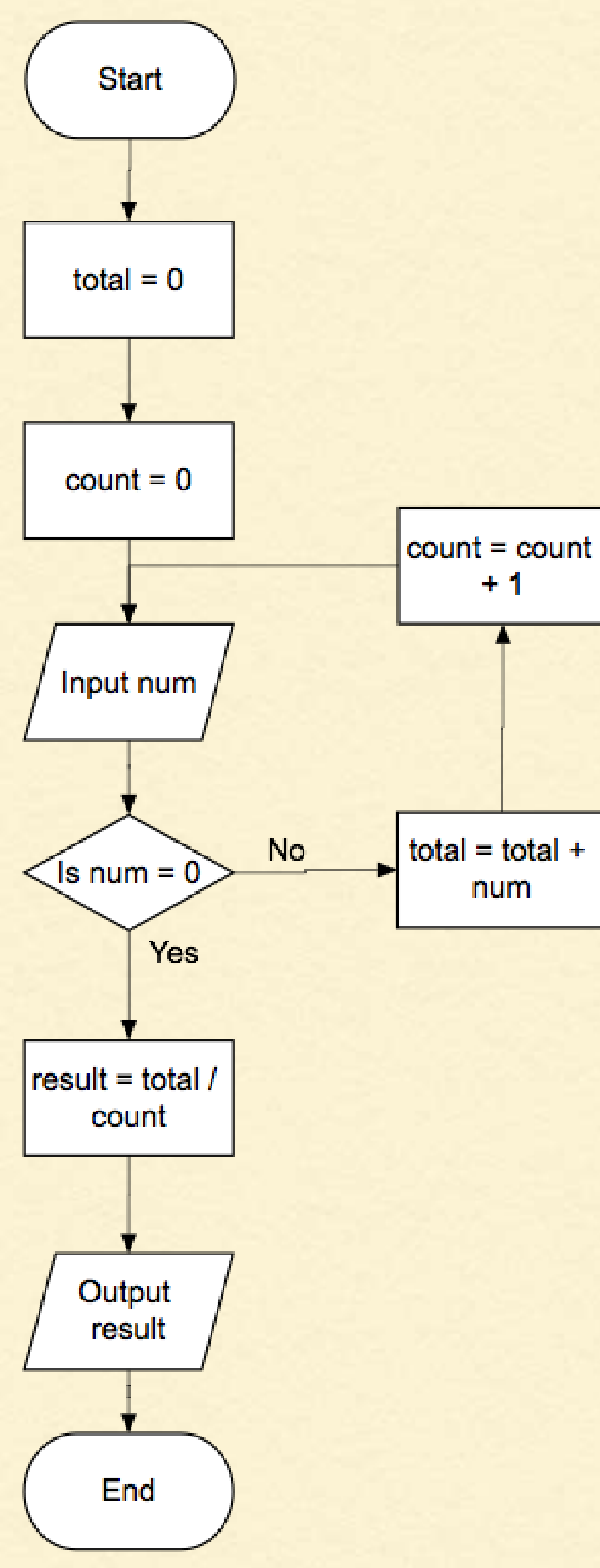
- This code will run in a loop.
- We need to load total and then subtract the count.
- We then need to see if the count is below zero.
- If it is not we will add one to the result and then loop around.

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



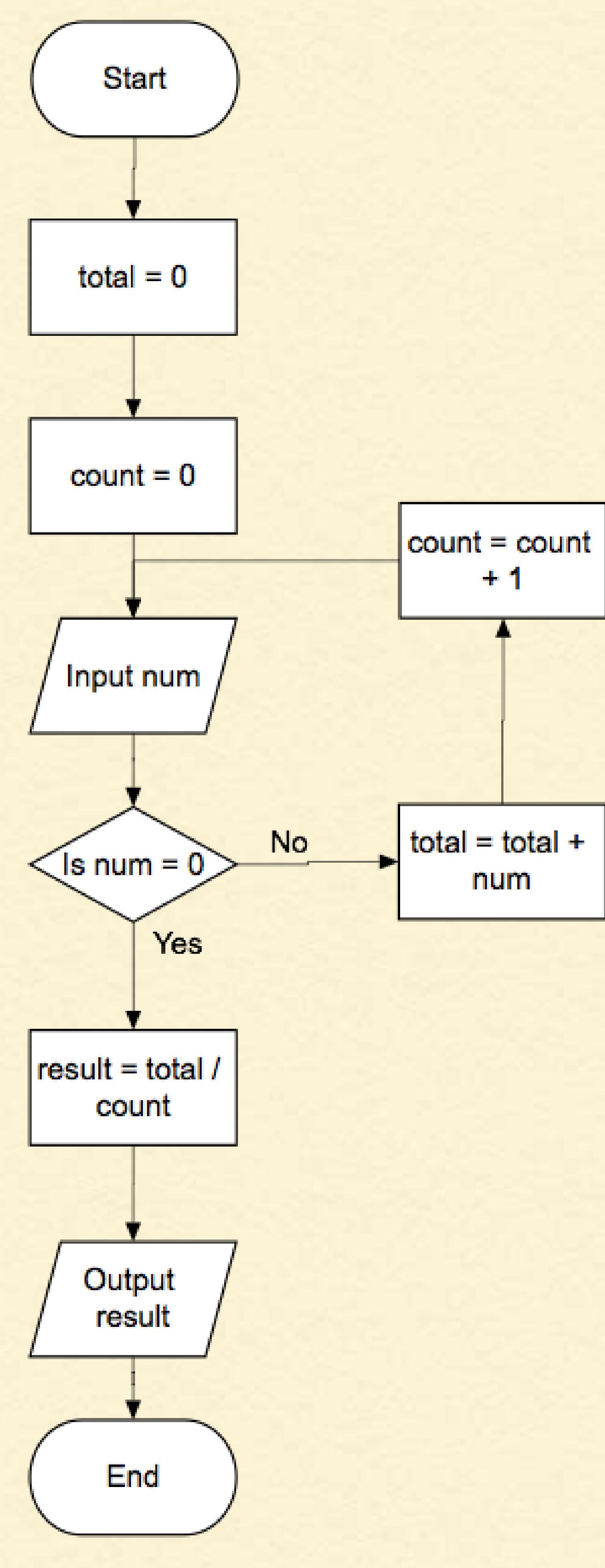
- So the first command is to load the total and subtract the count.
  - LDA Total
  - SUB count
- Then we Branch if the result is zero or higher, so we need BRP in order to add one to the result. I will give it the label DIVIDE and deal with that later.

```
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
```



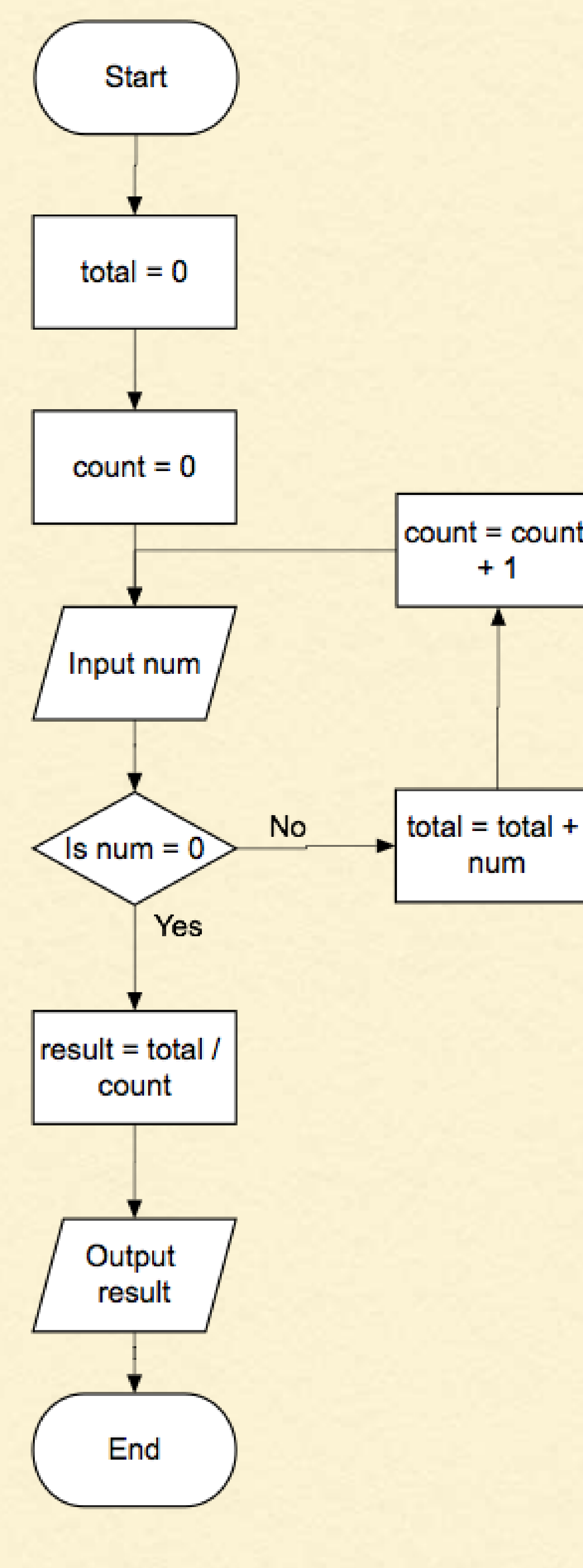
- If the value in the accumulator is negative then the BRP does not run.
- We now need to load the result and output it to the user.
- Once we do that the program is done.

```
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
```



- 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.

```
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
```



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

```
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
```