

Assembly Language

Isaac Student Booster

DO NOW:

What does this program do?

```

    INP
    STA 99
    INP
    ADD 99
    OUT

```


What does "99" mean?

How would we write this in Python or JavaScript?


Learning outcomes

By the end of this session you will be able to:

- Understand the purpose and characteristics of high- and low-level languages
- Identify and explain the features of assembly language
- Be able to interpret, complete and create simple programs in Little Man Computer Assembly language




Isaac Computer Science



During this course you may require access to the [Isaac Computer Science platform](#).

Accounts are free to create. You will be able to:

- use the platform to study
- test your knowledge with self-marking questions
- complete Gameboards set by your teacher



Programming in machine code

- The first programmers had to program in machine code.
- Machine code is a binary code representing the instructions a particular CPU can execute
- **Question:** what disadvantages are there to coding in binary?
 - Slow to code
 - Hard to learn and debug
 - Easy to make mistakes

MACHINE CODE
1110 0011 1010 0000
1110 0010 0100 0011
1110 0000 1000 0000
1110 0000 0000 0011



Assembly language

Uses mnemonics, i.e. abbreviations to represent the instructions, e.g.

MOV – move data

CMP – compare values

For example:

MOV R3, #5

which instructs the CPU to move 5 to register 3.

Register 3 is a physical hardware location inside the CPU, i.e. a real set of logic circuits.

How does "Register 3" differ from a variable, say "age" in a Python program?



High-level programming languages

- To speed up the writing of programs, high-level programming languages were developed. Including

- FORTRAN – scientific
- COBOL – commercial →

IDENTIFICATION DIVISION.
PROGRAM-ID. IDSAMPLE.
ENVIRONMENT DIVISION.
PROCEDURE DIVISION.
DISPLAY 'HELLO WORLD'.
STOP RUN.

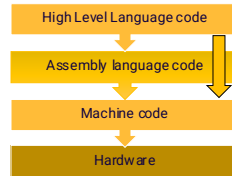
- C++, Python, Java, Ruby, Visual Basic and JavaScript are modern high-level programming languages.



Hierarchy of languages

Neither high-level code nor assembly code can run on the hardware directly and must be **translated**.

HLLs are **compiled** or **interpreted**. Assembly language is **assembled**.



Levels of abstraction

High level code is so-called because it sits at a **high level of abstraction** from the binary code.

High level of abstraction

```
var1 = input('num?')
var2 = input('num?')
print(var1 + var2)
```

Low level of abstraction

```
INP
STA 99
INP
ADD 99
OUT
```

No abstraction

```
000 00011100 00101
000 00001100 01111
000 00011100 00101
000 00000110 00111
000 00011100 00110
```

Activity – Evaluating Low-level languages



Spend **5 minutes** using the Isaac platform to research the **advantages and disadvantages of using low-level languages**. Record your findings on Handout 1.

isaac.computerscience.org/concepts/sys_proglang_low_level

Be ready to feedback using the text chat facility.

Advantages of low-level languages	Disadvantages of low-level languages

Solution – Evaluating Low-level languages



Advantages of low-level languages

- They allow a programmer to create optimised programs
- When a computer system has limited resources (processing power and memory) low-level languages allow a programmer to more directly control how the resources are used



Solution – Evaluating Low-level languages



Disadvantages of low-level languages

- It's more difficult to write programs in low-level: need to have good understanding of hardware
- Not portable, specific to a particular instruction set.
- No libraries of functions that can be imported
- No data structures such as arrays and records.



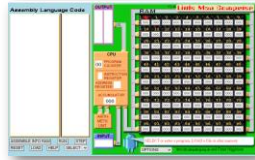
Instruction sets

- Assembly language is processor-specific.
- Each processor (CPU) has its own **instruction set** with a limited number of **opcodes** that are understood by the CPU's control unit.
- Instructions are made of an **opcode** and may be followed by an item of data, called the **operand**.

e.g. STA 3 =
STA (opcode)
3 (operand)

- **opcode** is the instruction that specifies which operation the processor should perform
- **operand** is a value that the opcode will operate on – it might be a number or an address in memory

Little Man Computer (LMC)



- The LMC is a simple model of a CPU.
- "LMC" because we imagine that inside the CPU is a little man running around, executing instructions stored in boxes which represent memory locations (RAM)
- The LMC models the architecture of a CPU, with its own simple instruction set.

peterhigginson.co.uk/lmc/



Opcode and operand

- Recall that an **instruction set** is the complete set of the instructions (opcodes) that can be executed by a processor.
- Instructions are made of an **opcode** and sometimes an item of data called the **operand**.

An example of a machine code instruction is **0100001** where **010** is the **opcode** and **0001** the **operand**

- The LMC uses opcodes and operands in **denary** rather than binary to make it easier to follow.
- In a "real" processor the opcodes and operands are stored and processed in **binary** and usually documented in **hexadecimal**.



LMC Instruction Set

Instruction	Mnemonic	Numeric Code
Load	LDA	5xx
Store	STA	3xx
Add	ADD	1xx
Subtract	SUB	2xx
Input	INP	901
Output	OUT	902
End	HLT	000
Branch if zero	BRZ	7xx
Branch if zero or positive	BRP	8xx
Branch always	BRA	6xx
Data storage	DAT	



LMC Instruction Set

Instruction	Mnemonic	Numeric Code
Load	LDA	5xx
Store	STA	3xx
Add	ADD	1xx
Subtract	SUB	2xx
Input	INP	901
Output	OUT	902
End	HLT	000
Branch if zero	BRZ	7xx
Branch if zero or positive	BRP	8xx
Branch always	BRA	6xx
Data storage	DAT	

Opcodes for writing programs using **sequence, input and output**



LMC Instruction Set

Instruction	Mnemonic	Numeric Code
Load	LDA	5xx
Store	STA	3xx
Add	ADD	1xx
Subtract	SUB	2xx
Input	INP	901
Output	OUT	902
End	HLT	000
Branch if zero	BRZ	7xx
Branch if zero or positive	BRP	8xx
Branch always	BRA	6xx
Data storage	DAT	

Opcodes for writing programs using **selection**



LMC Instruction Set

Instruction	Mnemonic	Numeric Code
Load	LDA	5xx
Store	STA	3xx
Add	ADD	1xx
Subtract	SUB	2xx
Input	INP	901
Output	OUT	902
End	HLT	000
Branch if zero	BRZ	7xx
Branch if zero or positive	BRP	8xx
Branch always	BRA	6xx
Data storage	DAT	

Opcodes for writing programs using **iteration**



LMC Instruction Set

Instruction	Mnemonic	Numeric Code
Load	LDA	5xx
Store	STA	3xx
Add	ADD	1xx
Subtract	SUB	2xx
Input	INP	901
Output	OUT	902
End	HLT	000
Branch if zero	BRZ	7xx
Branch if zero or positive	BRP	8xx
Branch always	BRA	6xx
Data storage	DAT	

Opcode used to reserve memory for data (variables)



Assembling LMC code

Let's assemble an example program:

```
LDA 98    598
SUB 99    299
OUT       902
HLT       000
```

Mnemonic	Numeric Code
LDA	5xx
STA	3xx
ADD	1xx
SUB	2xx
INP	901
OUT	902
HLT	000
BRZ	7xx
BRP	8xx
BRA	6xx
DAT	

Activity 2 – Assembling and Disassembling



Use **Handout 2**.

- a) Assemble this program into denary codes:

INP	
STA 99	
OUT	
HLT	

- a) Disassemble this program back to mnemonics:

901	
388	
284	
587	
902	
000	

Mnemonic	Numeric Code
LDA	5xx
STA	3xx
ADD	1xx
SUB	2xx
INP	901
OUT	902
HLT	000
BRZ	7xx
BRP	8xx
BRA	6xx
DAT	

Activity 2 ANSWERS – Assembling and Disassembling

Use Handout 2.

- a) Assemble this program into denary codes:

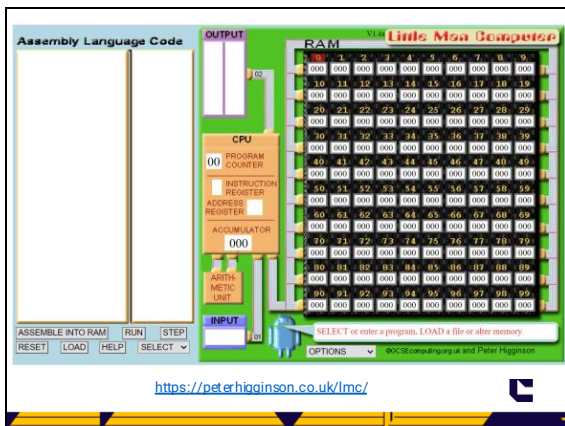
INP	901
STA 99	399
OUT	902
HLT	000

- a) Disassemble this program back to mnemonics:

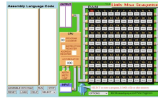
901	INP
388	STA 88
284	SUB 84
587	LDA 87
902	OUT
000	HLT

Mnemonic	Numeric Code
LDA	5xx
STA	3xx
ADD	1xx
SUB	2xx
INP	901
OUT	902
HLT	000
BRZ	7xx
BRP	8xx
BRA	6xx
DAT	

The Little Man Computer

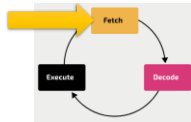


LMC Registers



- **Program Counter (PC)**: stores address of next instruction in RAM to fetch an instruction from (0 to 99)
- **Instruction Register (Current Instruction Register or CIR)**: stores the first digit of the instruction read from memory, the op-code, which contain the instruction to be performed
- **Address Register (Memory Address Register or MAR)**: stores the bottom two digits of the instruction, the operand, this contains an address which is associated with the instruction.
- **Accumulator**: stores the results of the last operation (used as a Flag status register for branch instructions) (-999 to 999) from the ALU
- **Memory Data Register (MDR/MBR)** – **not shown in LMC**: stores the contents *found* at the RAM address held by the MAR or data which *is to be transferred* to that RAM address.

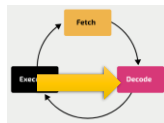
LMC FDE cycle



FETCH:

1. Check the **Program Counter (PC)** for the address of the next instruction in RAM
2. Fetch the instruction from that memory address.
3. Increment the **Program Counter** (add 1 to PC so that it contains the RAM address number of the next instruction.)

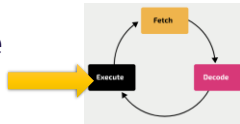
LMC FDE cycle



DECODE:

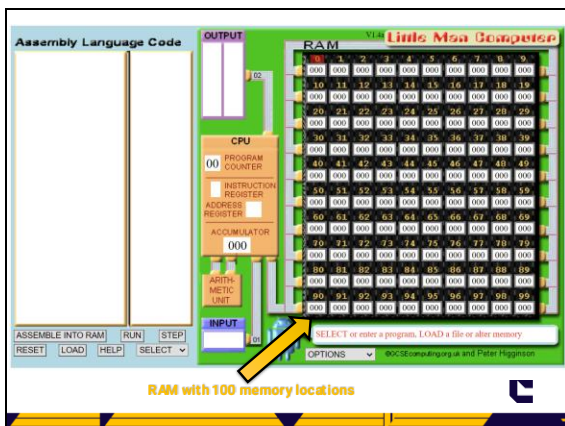
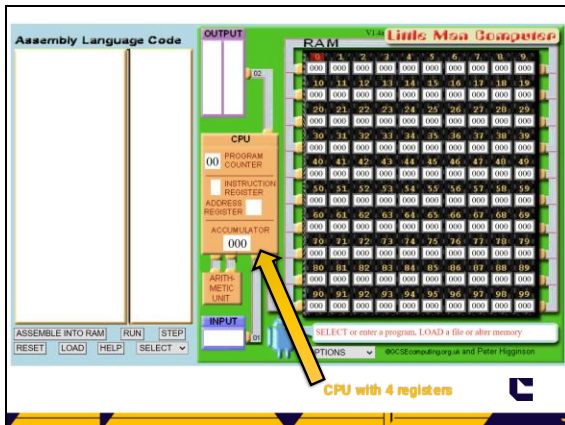
4. Split the instruction just fetched into **opcode** and **operand**
5. Store the opcode in the **instruction register**
6. Store the **operand** in the **address register**

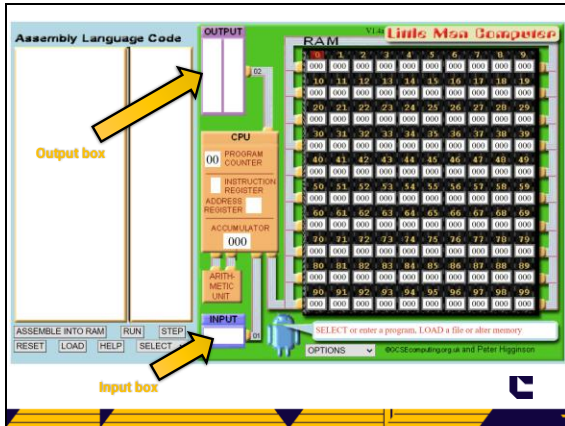
LMC FDE cycle

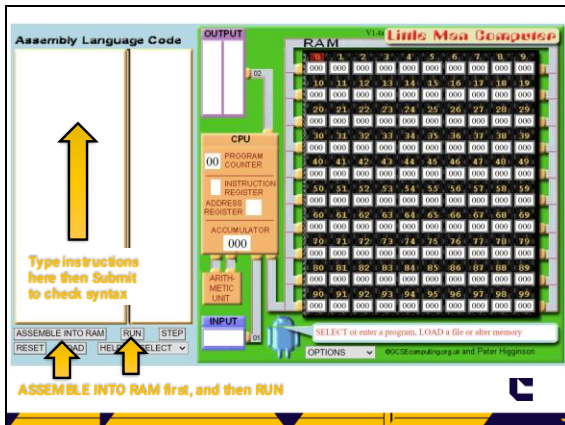


EXECUTE:

7. Execute the instruction based on the opcode given, using the **address register** (operand) as necessary.
 - a. Load data from **RAM** if executing an LDA instruction, store data in RAM if executing an STA instruction.
 - b. Store the result of arithmetic or LDA operations in the **accumulator**.
 - c. If the instruction is a BRA, or a BRP/ BRZ and the condition is met, update the **program counter** with the contents of the **address register** (operand).
8. If instruction is HLT (000) then **stop**, else return to the Fetch phase.







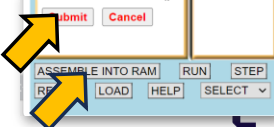
Worked Example 1

Let's assemble and run this program

Mnemonic	Machine code
INP	901
OUT	902
HLT	000

Assembly

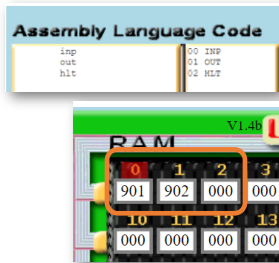
```
inp
out
hlt
```



Worked Example 1

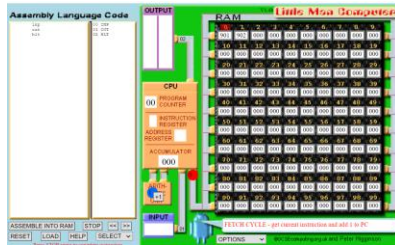
It's now assembled.
We can see it in RAM here.

Mnemonic	Machine code
INP	901
OUT	902
HLT	000



Demonstration

Mnemonic	Machine code
INP	901
OUT	902
HLT	000



Worked Example 2 – Add 2 numbers

We only have one accumulator and INP always overwrites the accumulator. So the pseudocode is:

```
input num1          INP
store num1 in RAM   STA 99
input num2          INP
add num1 from RAM    ADD 99
output              OUT
halt                HLT
```

In LMC assembly this would be...

Activity – Programming with the LMC



Now it's time for you to begin programming in assembly language using the LMC.

Visit zigzageducation.co.uk/lmc/

or peterhigginson.co.uk/lmc/

On Handout 3 complete Activities 1 and 2

You have 8 minutes!



ANSWERS: Activity 1



Modify:

INP
OUT
STA 9
HLT

Make:

INP
STA 97
INP
STA 98
INP
STA 99
OUT
HLT



ANSWERS: Activity 2



Modify:

INP
STA 99
INP
SUB 99
OUT

Make:

INP
STA 99
INP
ADD 99
STA 99
INP
SUB 99
OUT



"Variables" in the LMC (DAT)

- A label containing a DAT statement works as a variable, it labels a memory address with a text identifier.

one DAT 1
creates a label called **one** for the next free memory location and stores in it the value 1.

num DAT
creates label called **num** for the next free memory location and does **not** overwrite its contents



Branching in the LMC

- Branching uses **labels** and the branch commands to alter the order in which the instructions are executed.
- It achieves this by changing the **program counter**.

BRA	Branch to the address specified by the operand
BRZ	Branch to the address specified by the operand if the value in the accumulator is 0
BRP	Branch to the address specified by the operand if the value in the accumulator is 0 or positive

- BRZ and BRP branch **depending upon the value currently stored in the accumulator** and can be used as "if" conditions to allow selection between two possible paths through the program.



Branching and labels



- A label to the **left** of the instruction is converted to the memory address of the instruction or data.

loop INP

loop is the label and when assembled, resolves to the memory address where the instruction INP is stored.

- A label to the **right** of the instruction takes on the value of the memory address labelled previously.

BRA loop

loop will resolve to the location we labelled earlier, so the program will loop back to the **loop INP** instruction



Branching to implement a loop

Predict what this code does:

This block of code will loop forever, and the output will be:

10

9

8

etc... and it won't stop at 1 but continue through zero into the negative numbers!

```

LDA TEN
START OUT
      SUB ONE
      BRA START
ONE   DAT 1
TEN   DAT 10

```

Implementing a condition-controlled loop

If we want to stop this loop running forever, we can change BRA to **BRP** (branch if positive or zero).

Now what will it output?

10, 9, 8, ..., 0.

```

LDA TEN
START OUT
      SUB ONE
      BRP START
ONE   DAT 1
TEN   DAT 10

```

Conditions other than ACC = 0

What if we want to count **up** from zero to nine? i.e. implement this pseudocode →

We can't directly compare the accumulator with 10. We only have BRZ and BRP to play with.

What can we do?

```

count = 0
limit = 10
repeat
  output count
  count = count + 1
until count == limit

```

Subtract limit then branch if zero

Let's make a slight change to our pseudocode:

```
count = 0
limit = 10
repeat
  output count
  count = count + 1
  temp = count - limit
until temp = 0
```

Now we can code this:

```
START LDA TEN
      OUT
      SUB ONE
      STA COUNT
      SUB LIMIT
      BRZ END
      LDA COUNT
      BRA START
      END HLT
      ONE DAT 1
      TEN DAT 10
      COUNT DAT
      LIMIT DAT
```

Annotations:

- save count (points to STA COUNT)
- subtract limit (points to SUB LIMIT)
- break out if 0 (points to BRZ END)
- reload count (points to LDA COUNT)
- loop back (points to BRA START)

Activity continued – Programming with the LMC

Handout 3

Using zigzageducation.co.uk/lmc or peterhigginson.co.uk/lmc/

On Handout 3 complete Activities 3 and 4.
You have 10 minutes!

Predict
Run
Investigate
Modify
Make

ANSWERS: Activity 3 & 4

Handout 3

Predict:

```
INP
ADD FIVE
OUT
FIVE DAT 5
```

Inputs a number, adds five, outputs the result

Predict:

```
BEG INP
ADD TEN
OUT
BRA BEG
TEN DAT 10
```

Inputs a number, adds ten, outputs the result then starts again. Runs forever.

ANSWERS: Activity 4



Modify:

```

LOOP    LDA  START
        OUT
        SUB  ONE
        STA  START
        BRP  LOOP
        HLT

START    DAT  5
ONE      DAT  1

```

Make:

```

INP
STA  START
LOOP LDA  START
    OUT
    SUB  ONE
    STA  START
    BRP  LOOP
    HLT

START DAT
ONE   DAT 1

```



Branching to implement selection

Consider this pseudocode:

```

if a >= b then
    output 1
else

```

```

    output 2

```

How can we use SUB and BRP to implement this?

Let's rewrite as a "subtract and compare to zero"

```

a = a - b
if a >= 0 then
    output 1
else
    output 2

```



Branching to implement selection

Let's implement this code fragment in LMC assembly:

```

a = a - b
if a >= 0 then
    output 1
else
    output 2

```

```

SUB B
BRP ABIG
LDA B
OUT
BRA END
ABIG LDA A
    OUT
END  HLT

```

Notice that the *else* and *then* branches have swapped over!

This is the simplest way to code *if-then-else*.



Activity continued – Programming with the LMC



Using zigzageducation.co.uk/lmc or
peterhigginson.co.uk/lmc/

On Handout 3 complete **Activity 5**
and if time the **Challenge** activities 6 and 7.

You have 8 minutes!



ANSWER: Activity 5

Write an LMC
program that
outputs the larger
of two input values
(using selection)

```

INP
STA NUM1
INP
STA NUM2
SUB NUM1
BRP SEC
LDA NUM1
OUT
HLT
SEC LDA NUM2
OUT
HLT
NUM1 DAT
NUM2 DAT

```

ANSWERS: Challenges

Task 6 was
to multiply
two input
numbers:

```

INP          // input multiplicand
STA NUM1    // store as num1
INP          // input multiplier
STA NUM2    // store as num2
LOOP LDA TOTAL // load running total (0)
ADD NUM1    // add multiplicand
STA TOTAL   // store running total
LDA NUM2    // load multiplier
SUB ONE     // decrement by one
STA NUM2    // store multiplier
SUB ONE     // we want to stop at 1 not 0
BRP LOOP    // if multiplier > 0 loop
LDA TOTAL   // else load the total
OUT         // output it
HLT         // stop
NUM1 DAT
NUM2 DAT
ONE  DAT 1
TOTAL DAT 0

```

ANSWERS: Challenges

Task 7 was to check if a number is divisible by five...

```

      INP          // input a number
LOOP  BRZ TRUE    // number must be div by 5 so jump to "TRUE"
      SUB FIVE    // subtract 5
      BRP LOOP    // if still positive or zero, loop back
      LDA ZERO    // we've gone negative so not divisible by 5
      OUT
      BRA END     // jump to end
TRUE  LDA ONE     // this branch executes if we hit zero exactly
      OUT         // output a 1
      END HLT     // halt
ZERO  DAT 0
ONE   DAT 1
FIVE  DAT 5

```



Isaac Computer Science

- Each A level topic covered in depth, specific to each exam board.
- Multiple choice self-marking questions and videos for each topic.
- Free downloadable workbook containing over 450 questions, covering all the A level computer science topics, with space to write and work out answers.



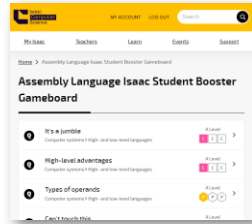
Practice Gameboard



Activity – Isaac Gameboard

Now complete the gameboard on Isaac Computer Science at bit.ly/ialowlevel

Use the hints to help you.

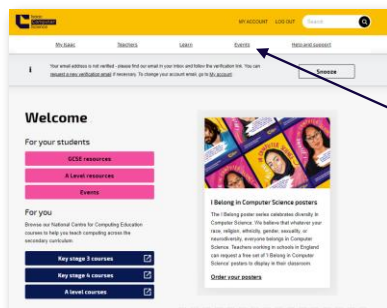


Learning outcomes

By the end of this session you will be able to:

- Understand the purpose and characteristics of high- and low-level languages
- Identify and explain the features of assembly language
- Be able to interpret, complete and create simple programs in Little Man Computer Assembly language

Check for more ISAAC boosters



Keep an eye out for more student booster events

Thank You