**Lab Manual**

**Computer Organization and**

**Assembly Language**

**(CS2523)**

**Contribution**

The manual was last updated on 22nd September, 2023 by Dr. Umair Rafique

**Outline**

|  |  |
| --- | --- |
| **Topic** | **Page No.** |
| Lab-1: Introduction to Computer Organization Using the Simulator Little Man Computer (LMC) | 7 |
| Lab-2: Instruction Cycle and Working with Some Basic Instructions in the LMC Simulator | 11 |
| Lab-3: Loops in Little Man Computer Simulator | 18 |
| Lab-4: Implementing Decimal to Binary Conversion in LMC (Part 1) | 23 |
| Lab-5: Implementing Decimal to Binary Conversion in LMC (Part 1) | 28 |
| Lab-6: Introduction to Intel 8086/8088 Assembly Language | 32 |
| Lab-7: Basic Instructions in Intel 8086/8088 |  |
| Lab-8: Addressing Modes, Arrays |  |
| Lab-9: Conditional Jumps |  |
| Lab-10: Unconditional Jumps, Sorting Algorithm |  |
| Lab-11: Implementing Multiplication Algorithm |  |
| Lab-12: Bit-wise logical operations |  |
| Lab-13: Subroutines, Stack |  |
| Lab 14: Parameter Passing through Stack |  |

**Rubric for the Evaluation of Lab Tasks**

# PSYCHOMOTOR

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Criteria**  **(Max Marks)** | **Level 0**  **(0%)** | **Level 1**  **0% ≤ S < 50%** | **Level 2**  **51% ≤ S < 65%** | **Level 3**  **66% ≤ S < 80%** | **Level 4**  **81% ≤ S < 100%** | **Total Score (S)** |
| **Understanding of problem**  **(20)** | Don’t know about problem understanding | Inadequate understanding of the provided problem in practice task | Brief understanding of the provided problem in practice task | Adequate understanding of the provided problem in practice task exists but missing some points | Complete understanding of the provided problem in practice task |  |
| **Program Logic**  **(20)** | Don’t know about the programming logic | Poor logic in program | Logic corrects to some extent but with major mistakes | Correct logic but with minor mistakes | Program Logic completely correct |  |
| **Program Implementation**  **(20)** | No task is  implemented | Makes several critical errors in syntax and in applying procedural knowledge | Makes few critical errors in syntax and in applying procedural knowledge | Makes some non-critical errors in syntax and in applying procedural knowledge | Applies the procedural knowledge and syntax in perfect ways |  |
| **Program Correctness**  **(10)** | Program totally incorrect | Program does not produce  correct answers or appropriate results for most inputs. | Program produces  correct answers or  appropriate results for most inputs, but contains miscalculations in some cases | Program produces correct answers or appropriate results for most inputs. | Program produces correct answers or appropriate results for all inputs tested |  |
| **Use of Software Tool**  **(10)** | Don’t know about the tool | Uses software tool, with limited competence | Uses software tool, with some competence | Uses software tool, with considerable competence | Uses software tool, with a high degree of competence |  |
| **Sub-Total Marks** | **80** | **Sub-Total Marks Obtained (PD)** | | | |  |

1. **AFFECTIVE**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Criteria**  **(Max Marks)** | **Level 0**  **(0%)** | **Level 1**  **0% ≤ S < 50%** | **Level 2**  **51% ≤ S < 65%** | **Level 3**  **66% ≤ S < 80%** | **Level 4**  **81% ≤ S < 100%** | **Total Score (S)** |
| **Viva**  **(20)** | All answers are incorrect | Answered for less than  40% of the questions  indicating a lack of  understanding of results | Answered for 60% of the  questions. But incomplete  understanding of results is still  evident | Answered for  80% of the  questions. Still  need some  improvements | Answered for more  than 90% of the  questions correctly,  good understanding  of results is  conveyed |  |
| **Sub-Total** | **20** | **Sub-Total Marks Obtained (AD)** | | | |  |

**Course Learning Outcomes**

1. Assembly language programming knowledge to write moderately complex assembly language subroutines. [2. Knowledge for Solving Computing Problems]
2. Assembly language subroutines interfacing to a high-level language i.e., C++. [4. Design/Development of Solutions]
3. Participate and answer the questions related to Assembly language. [7. Communication]

Lab-1

Introduction to Computer Organization

Using the Simulator Little Man Computer (LMC)

# Introduction

In this lab we will learn about the basic structural components of a computer and will work with a simulator to understand how a computer functions.

# Concept Map

The very first computer designed by von Neumann and his colleagues is called IAS computer. It is the prototype of all subsequent general- purpose computers. With rare exceptions, all of today’s computers have this same general structure and function and are thus referred to as von Neumann machines.

Generally, there are four main structural components of a computer:

* Central processing unit (CPU): Controls the operation of the computer and performs its data processing functions; often simply referred to as processor.
* Main memory: Stores data.
* I/O: Moves data between the computer and its external environment.
* System interconnection: Some mechanism that provides for communication among CPU, main memory, and I/O. A common example of system interconnection is by means of a system bus, consisting of a number of conducting wires to which all the other components attach.

A CPU has following structural components.

* Control unit: Controls the operation of the CPU and hence the computer.
* Arithmetic and logic unit (ALU): Performs the computer’s data processing functions.
* Registers: Provides storage internal to the CPU.
* CPU interconnection: Some mechanism that provides for communication among the control unit, ALU, and registers.

We will use a simulator to understand the inner workings of a computer. This simulator is called Little Man Computer.

**Little Man Computer - CPU simulator**

The Little Man Computer (LMC) is an instructional model of a computer, created by Dr. Stuart Madnick in 1965. It models the architecture of a simple computer and has all the basic features.

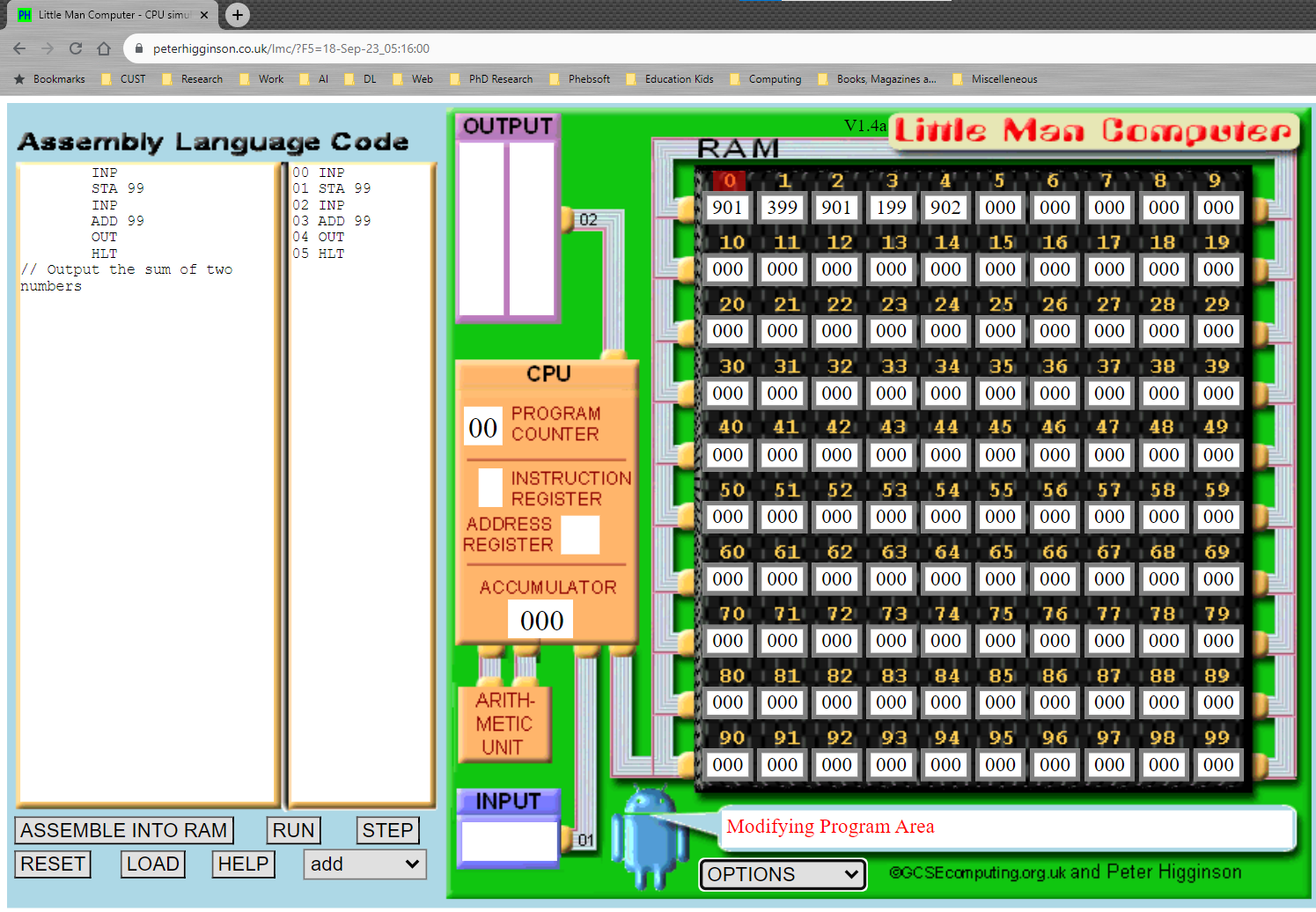
<https://www.peterhigginson.co.uk/lmc>

Inside the CPU there are:

* 100 Memory locations (numbered 0 to 99).
* An Arithmetic Unit to do calculations.
* An Accumulator to store the result of the last operation or calculation.
* A Program Counter to store the address (in memory) of the next instruction to do.
* An Instruction Register to hold the top digit of the instruction read from memory.
* An Address Register to hold the bottom two digits of the instruction read from memory.
* An Input tray into which a number can be typed when needed.
* An Output area where any numbers output are printed.

The Program Counter holds only 2 digits (0 to 99) and the Accumulator holds 3 digits and a sign (-999 to 999). The Output area holds a certain amount of information in two columns and then scrolls (and old information disappears).

When an instruction is read from memory the top digit is used to decide what to do. To help us write programs more easily each instruction type is given a name which the ASSEMBLE function converts to the corresponding code. The bottom two digits are used as an address associated with the instruction.



# Walkthrough Task

There is a SELECT button which will allow you to select from the supplied programs including the following two.

**Add:** Output the sum of two numbers. Requests input twice

**Add/subtr:** Input three numbers. Output the sum of the first two and the third minus the first.

To start using this LMC **SELECT**  the add program from the drop down list.

Click **RUN** to start execution. Execution starts at a medium/slow speed and you can use the >> button to speed it up or the << button to slow it down. The lowest five speeds show the flow of execution of an instruction as follows:

FETCH Get the contents of the next instruction to be executed. In parallel add 1 to the Program Counter.

EXECUTE Decode the instruction loaded and perform the operation requested.

The STEP button does one instruction only (at a slow speed). You can RESET to go back to the beginning (which is always with Program Counter = 0 on the LMC) or set the Program Counter to any memory address.

You can input into the INPUT box (when the running program requests it) and (when the program is not running) into the individual memory locations, the Program Counter or the Assembly Language area. You cannot alter the Code area or the CPU registers apart from the Program Counter. For memory, the INPUT box and the Program Counter only numbers are accepted.

# Practice Task

## Practice Task 1 [Expected time = 20mins]

Select the **add/su**btr program from the select menu and complete its execution using the step function of the simulator. Observe how the program is loaded in the memory and how the steps are performed.

## Practice Task 2 [Expected time = 30mins]

Modify the above program so that now it takes 4 inputs. It outputs the sum of the first two inputs and the difference of the last two inputs.

# Evaluation Task (Unseen) [Expected time = 30mins for tasks]

The lab instructor will give you unseen task depending upon the progress of the class.

# Evaluation criteria

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Marks** |
| 1 | Understanding of Problem | 20 |
| 2 | Program Logic | 20 |
| 3 | Program Implementation | 20 |
| 4 | Program Correctness | 10 |
| 5 | Use of Tool | 10 |
| 6 | Viva | 20 |
| **Total** | | **100** |

Lab-2

Instruction Cycle and Working with Some Basic Instructions in the LMC Simulator

# Introduction

In the previous lab we learned about the basic structural components of a computer and worked with a basic simulator, Little Man Computer (LMC), to understand how a computer functions. In this lab we will continue with the same simulator and will focus on its instruction set.

# Concept Map

In the LMC simulator, when an instruction is read from memory, the top digit is used to decide what to do. To help us write programs more easily each instruction type is given a name. The ASSEMBLE function converts this name to the corresponding code. The bottom two digits are used as an address associated with the instruction.

In other words, the first digit is the “opcode” and the other two digits represent the address.

**Instruction Cycle**

The simulator has two steps in its instruction cycle.

FETCH Get the contents of the next instruction to be executed. In parallel add 1 to the Program Counter.

EXECUTE Decode the instruction loaded and perform the operation requested.

**Instruction Set**

The simulator has the following instructions.

|  |  |  |
| --- | --- | --- |
| **Code** | **Name** | **Description** |
| 0 | HLT | Stop (Little Man has a rest). |
| 1 | ADD | Add the contents of the memory address to the Accumulator |
| 2 | SUB | Subtract the contents of the memory address from the Accumulator |
| 3 | STA or STO | Store the value in the Accumulator in the memory address given. |
| 4 |  | This code is unused and gives an error. |
| 5 | LDA | Load the Accumulator with the contents of the memory address given |
| 6 | BRA | Branch - use the address given as the address of the next instruction |
| 7 | BRZ | Branch to the address given if the Accumulator is zero |
| 8 | BRP | Branch to the address given if the Accumulator is zero or positive |
| 9 | INP or OUT | Input or Output. Take from Input if address is 1, copy to Output if address is 2. |
| 9 | OTC | Output accumulator as a character if address is 22. (Non-standard instruction) |

**Labels**

The assembler allows us to give names (called labels) to addresses. Any word that is not a recognized instruction is assumed to be a label. We can use these labels with branch instructions to go to a certain part of the program if the accumulator is 0 (BRZ) or if the accumulator is zero or positive (BRP).

**Comments**

You can also put comments into the program. Any line starting with // is ignored by the assembler.

**Input**

You can input into the INPUT box (when the running program requests it) and (when the program is not running) into the individual memory locations.

# Walkthrough Task

We want to write a program that can find the bigger of two numbers stored at given locations in the memory.

We could do the same thing in C++ using the following code. The two numbers are stored in variables x and y.

**int x=5;**

**int y=6;**

**if(x>=y)**

**cout<<x;**

**else**

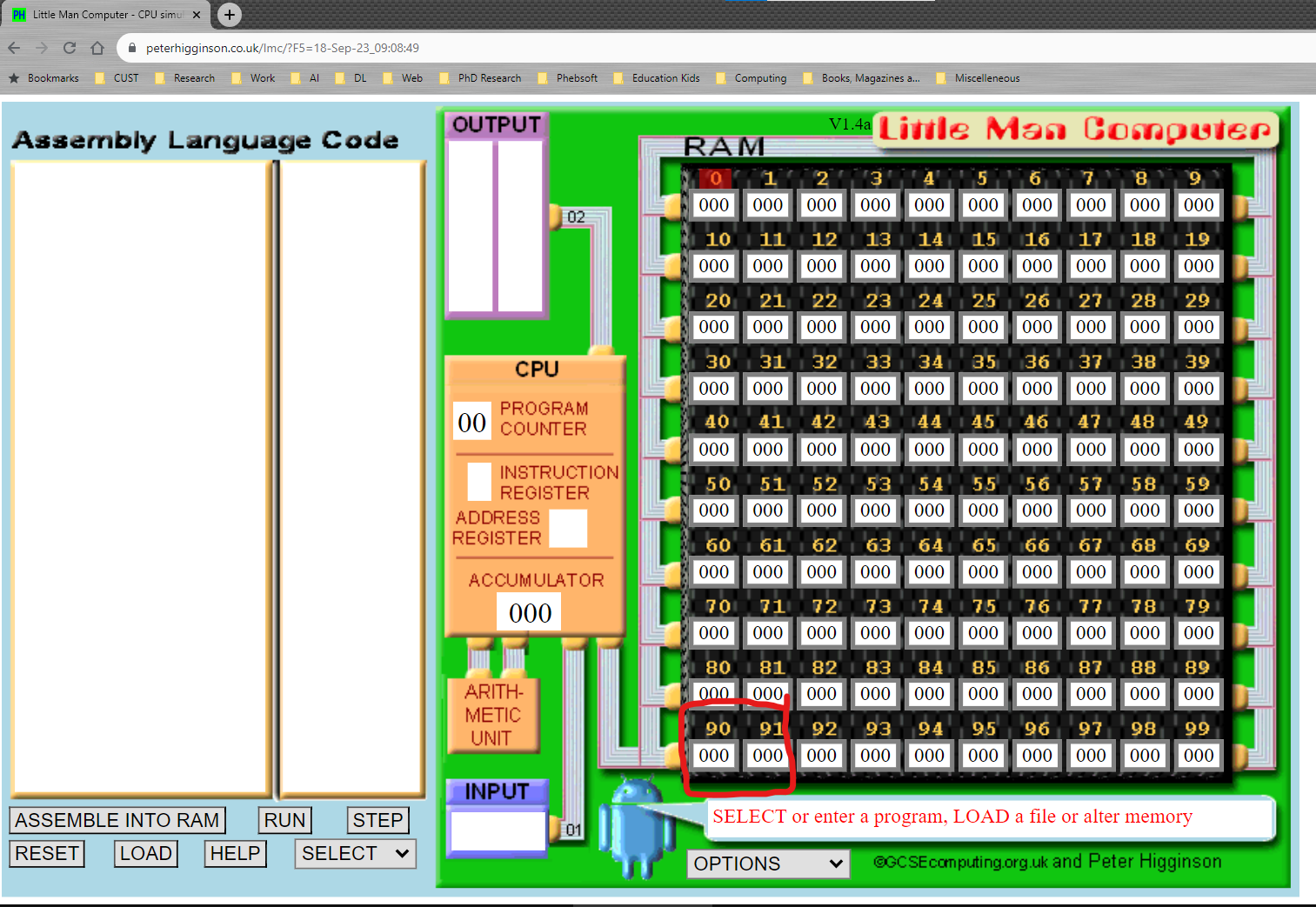
**cout<<y;**

However, in this case we are writing the assembly language program using the instructions available to us in the simulator. We have only a limited set of instruction so we need to build the logic accordingly.

Open the simulator using the following link.

<https://www.peterhigginson.co.uk/lmc>

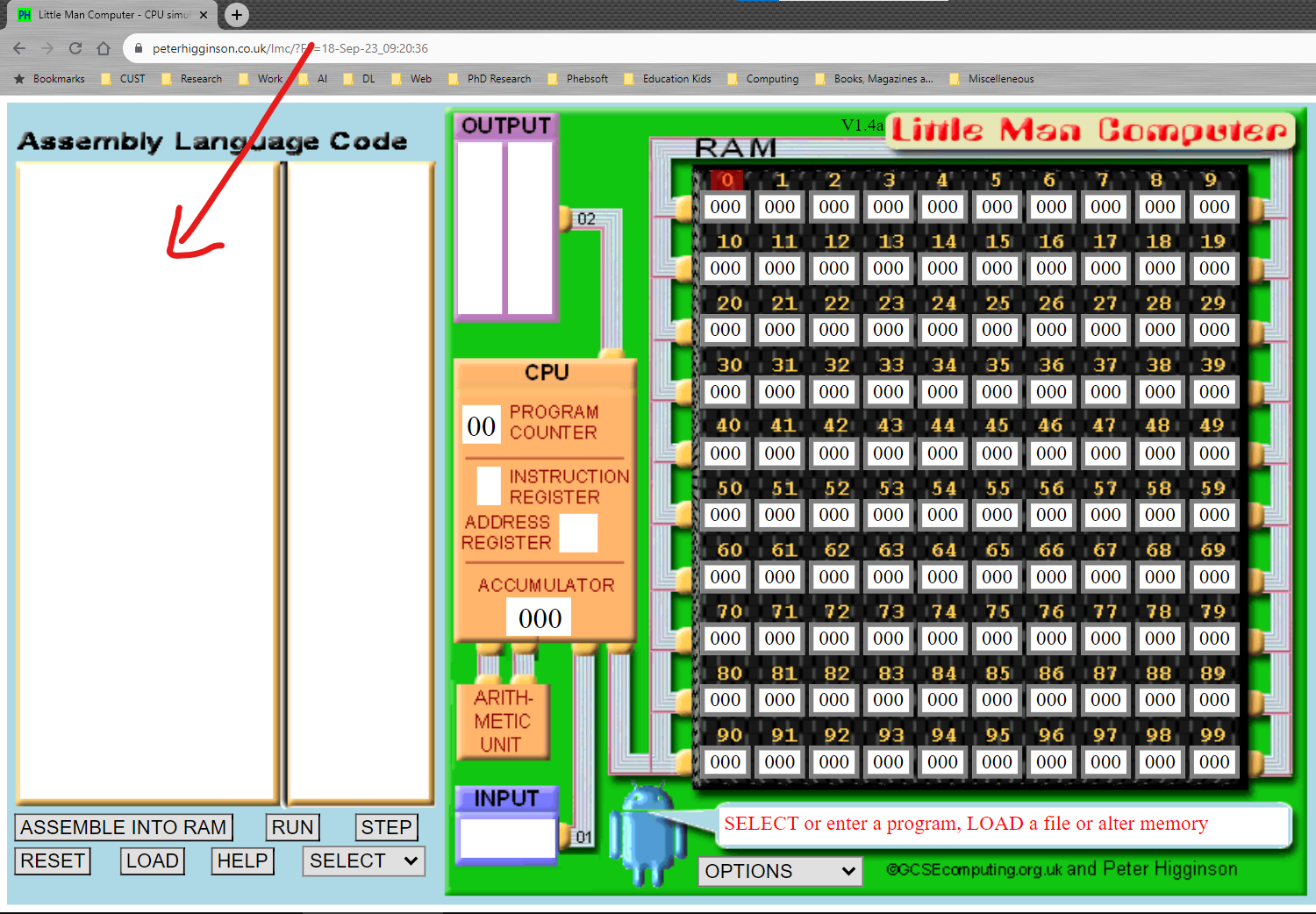
The first thing is that we want the two numbers to be stored at some locations. We choose location 90 and 91. We can simply type the two numbers there. We can type 5 at address 90 and 6 at address 91.



That means now we have the two numbers at known addresses (corresponding to the two variables x and y as given in the program above).

Now we need to compare the two numbers. We do not have any instruction in LMC’s instruction set that can compare two numbers. However, we have an instruction that can subtract from the accumulator (SUB) and we also have an instruction that can go to a certain address depending upon the value of the accumulator (BRZ if the accumulator is 0 or BRP if the accumulator is zero or positive). We will use these instructions to develop the logic of the program.

We write our program in the left most part of the interface.



We will first load the first number in the accumulator. We use the following instruction for this purpose. It will load 5 into accumulator.

**LDA 90**

To compare it with the other number stored at 91, we subtract that number from the accumulator using the following instruction

**SUB 91**

Depending on which number is bigger, the content of the accumulator will be negative (if number1 < number2) or 0 (if number1 and number2 are equal) or positive (if number1>number2).

We want to print the bigger number. That means if the accumulator is 0 or positive, we want to print the number at address 90. If that is not the case, then we want to print the number at address 91.

To be able to execute a set of instructions based on a decision, we use a label to identify that group of instructions. Then we go to that label if a certain condition holds.

Let’s write the set of instructions that would print number1 (stored at 90). Note that OUT instruction only prints what is in the accumulator, so we need to first load the number into the accumulator and then write OUT.

**LDA 90**

**OUT**

We assign a label **L1** to these two instructions and the resulting instructions will look like.

**L1 LDA 90**

**OUT**

We will go to these instructions (identified by L1) if the accumulator is zero or positive. We use BRP in the following way.

**BRP L1**

This means branch to L1 if the accumulator is zero or positive.

And if that is not the case, we want to print the other number (stored at 91). We can do so in the following way.

**LDA 91**

**OUT**

The complete program looks like the following. We add HLT to the point where we want to stop the program.

**LDA 90**

**SUB 91**

**BRP L1**

**LDA 91**

**OUT**

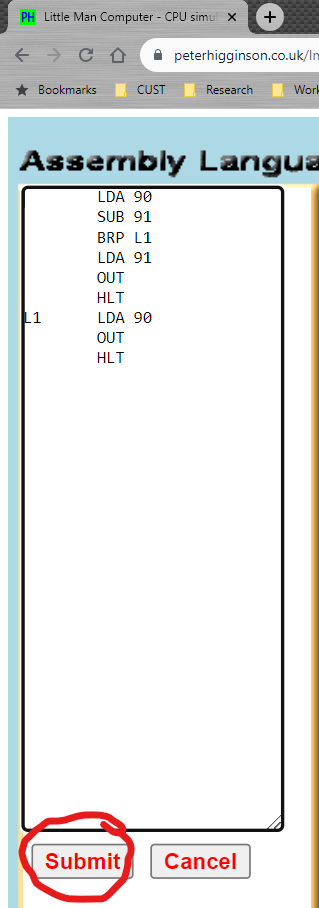
**HLT**

**L1 LDA 90**

**OUT**

**HLT**

Click on submit to assemble and load the program.



Click on step to run the program step by step.

Focus on the changes made to Program Counter, Instruction Register, Address Register and Accumulator.

# Practice Task

## Practice Task 1 [Expected time = 20mins]

Write a program to find the smaller of the two numbers stored at addresses 90 and 91.

## Practice Task 2 [Expected time = 30mins]

Write a program to find the bigger of three numbers stored at addresses 90, 91 and 92.

# Evaluation Task (Unseen) [Expected time = 30 mins]

The lab instructor will give you unseen task depending upon the progress of the class.

# Evaluation criteria

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Marks** |
| 1 | Understanding of Problem | 20 |
| 2 | Program Logic | 20 |
| 3 | Program Implementation | 20 |
| 4 | Program Correctness | 10 |
| 5 | Use of Tool | 10 |
| 6 | Viva | 20 |
| **Total** | | **100** |

Lab-3

Loops in Little Man Computer Simulator

# Introduction

In the previous lab we learnt the instruction set of Little Man Computer (LMC). We used it to write a simple program to find bigger of two numbers stored in the memory. We learnt how we can use basic instructions to implement decision making logic in our program using BRP (branch if accumulator is 0 or positive) instruction and labels.

In this lab (and next 2 labs) we will continue to work with LMC and will use its basic instruction set to implement increasingly complex logics. Our focus in this lab is to learn how to implement loops.

Although LMC has a very basic instruction set, the concepts we are learning are relevant for the more advanced 8086/8088 assembly language which we will start in Lab 6.

# Concept Map

LMC has the following instructions as we have seen in the previous lab.

|  |  |  |
| --- | --- | --- |
| **Code** | **Name** | **Description** |
| 0 | HLT | Stop (Little Man has a rest). |
| 1 | ADD | Add the contents of the memory address to the Accumulator |
| 2 | SUB | Subtract the contents of the memory address from the Accumulator |
| 3 | STA or STO | Store the value in the Accumulator in the memory address given. |
| 4 |  | This code is unused and gives an error. |
| 5 | LDA | Load the Accumulator with the contents of the memory address given |
| 6 | BRA | Branch - use the address given as the address of the next instruction |
| 7 | BRZ | Branch to the address given if the Accumulator is zero |
| 8 | BRP | Branch to the address given if the Accumulator is zero or positive |
| 9 | INP or OUT | Input or Output. Take from Input if address is 1, copy to Output if address is 2. |
| 9 | OTC | Output accumulator as a character if address is 22. (Non-standard instruction) |

We have two types of branches here. Instruction 6 is an “unconditional branch”. That means it does not depend on any condition. Whatever address we specify in the instruction, that address will be taken as the address of the next instruction. The jump to this new instruction is not dependent on any condition.

Instruction 7 and 8 are “conditional branches”. That means the branch to the specified address is made only if the corresponding condition is true. In case of BRZ, if the accumulator is 0, the branch is taken. In case of BRP if the Accumulator is 0 or positive, the branch is taken. We used BRP in the previous lab to implement a simple if structure.

We can use the same logic to implement a loop. This is demonstrated using the walkthrough task.

# Walkthrough Task

In this task we will write a basic loop to print numbers from 1 to 5.

Open the simulator using the following link. <https://www.peterhigginson.co.uk/lmc>

We can start with the starting number (1), increment it by 1 in each iteration and will continue till this number is equal to the ending number (5).

We only have an instruction which can add to the accumulator the value given at the specified address. That means we need to store this “1” in the memory too.

We will store the starting number at address 80.

Ending number at address 81. We will store 6 here as we want to print till 5.

The increment value (1) at address 82.

We will use address 83 as a temporary storage so that we do not change the original numbers.

**Steps of the program**

1. Load the starting number from address 80 **(LDA 80)**
2. Save it to address 83 which we will use as temporary storage. **(STA 83)**
3. Load the number from address 83 to the accumulator. **(LDA 83)**
4. Print it **(OUT)**
5. Add to the accumulator whatever is at address 82 (increment value) **(ADD 82)**
6. Store the accumulator at address 83 **(STA 83)**
7. Sub from the accumulator whatever is at address 81 (6 which is ending value + 1) **(SUB 81)**
8. End the program if the accumulator is 0.
9. Go to step 3.

Note that if step 9 will execute only if line 8 did not terminate the program.

The complete code for the above program is given below. We have used LOOP and END as labels.

LDA 80

STA 83

LOOP LDA 83

OUT

ADD 82

STA 83

SUB 81

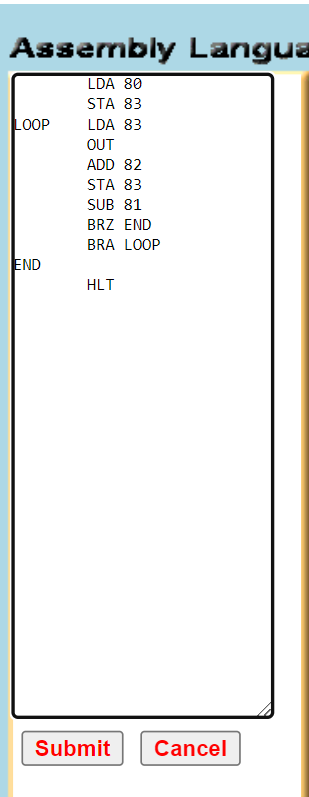
BRZ END

BRA LOOP

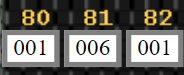
END

HLT

Write the program in the left most box and then press submit.



Update the values at addresses 80, 81 and 82 after clicking on submit. Our memory will look like the following.



Run the program step by step and trace its execution. The final output will be the following.



# Practice Task

## Practice Task 1 [Expected time = 30mins]

Change the above program so that it prints the number in the descending order like

5

4

3

2

1

# Evaluation Task (Unseen) [Expected time = 30mins for tasks]

The lab instructor will give you unseen task depending upon the progress of the class.

# Evaluation criteria

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Marks** |
| 1 | Understanding of Problem | 20 |
| 2 | Program Logic | 20 |
| 3 | Program Implementation | 20 |
| 4 | Program Correctness | 10 |
| 5 | Use of Tool | 10 |
| 6 | Viva | 20 |
| **Total** | | **100** |

Lab-4

Implementing Decimal to Binary Conversion in LMC

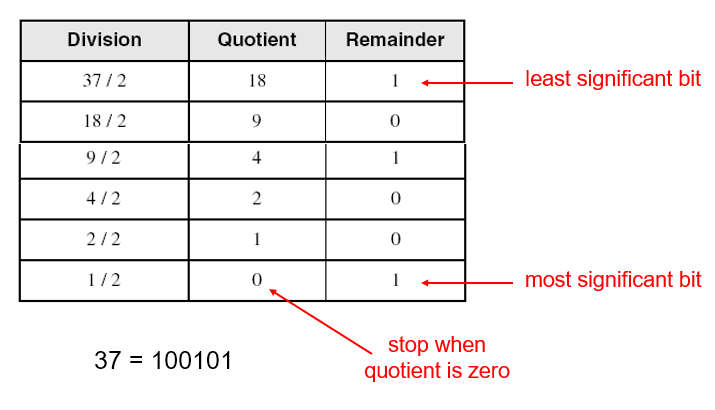
Part 1

# Introduction

In this lab (and the next) we will implement decimal to binary conversion in LMC.

# Concept Map

To convert a decimal number into binary, we repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value.



We want to implement this logic in Little Man Computer (LMC).

One main issue here is that there is no division instruction in the LMC. Hence, we need to implement division using repeated subtraction. In this lab we will focus on this task. The procedure will be explained in the walkthrough task. The conversion will then be discussed in the next lab (Lab 5).

# Walkthrough Task

In this lab we are going to focus on implementing division by 2 using repeated subtraction.

Open the simulator using the following link. <https://www.peterhigginson.co.uk/lmc>

We will use the following addresses for special purposes.

Will store dividend at address 80

Divisor at address 81

Quotient at address 82. Initially 0 will be stored here.

1 will be stored at address 83. We will use it when we want to increment the quotient.

Address 84 will be used as temporary storage.

The basic idea of the program is as following.

1. Load the dividend in the accumulator. Subtract from it the divisor.
2. If the dividend is 0 or positive, increment the quotient. In the following program the increment quotient part has been identified with the label INCQ.
3. If this jump to the INCQ does not happen, that means the dividend is now negative. Hence, we do not increment the quotient and go to the label END.
4. If the jump to INCQ happened, then we increment the quotient and then jump (branch) to the label loop. That means going back to step 1 above.
5. Once we reach the label END, we load the current value from address 84. Actually, it represents the remainder but the divisor has been subtracted from it one extra time in the previous loop (we come to know about it only when we get negative value at address 84). To correct it we add the divisor one time (**ADD 81**) to this value and then store it at address 84 (**STA 84**).

The complete program is as shown below. The first two lines of the program where we copy the dividend to address 84 is to keep a copy of the dividend which we work with and do not destroy the original. This program will print Remainder (address 84) and then Quotient (address 82) in the output.

LDA 80

STA 84

LOOP LDA 84

SUB 81

STA 84

BRP INCQ

BRA END

INCQ LDA 82

ADD 83

STA 82 //INCQ ends

BRA LOOP

END LDA 84 //Remainder

ADD 81 //subtr correction

STA 84

OUT

LDA 82 //Quotient

OUT

HLT

//80 stores dividend

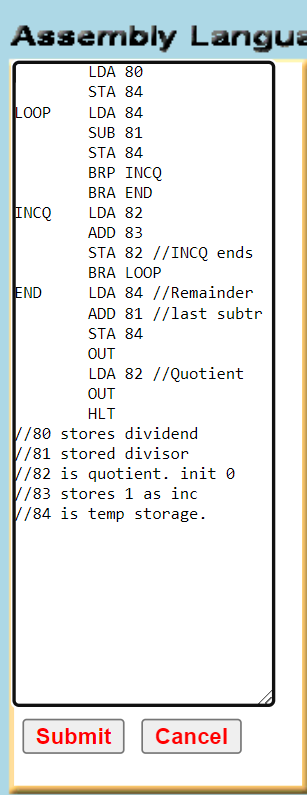
//81 stored divisor

//82 is quotient. init 0

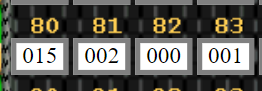
//83 stores 1 as inc

//84 temp storage. Stores Remainder.

Write the program in the left most box in the simulator and then click submit.



Update the values in the memory storing dividend at 80, divisor at 81. 0 at 82 and 1 at 83. For dividing 15 by 2, the memory will look like the following.



Run the program and see how it updates different memory locations.

# Practice Task

## Practice Task 1 [Expected time = 50mins]

Implement multiplication as repeated addition using a logic similar to the previous program.

# Evaluation Task (Unseen) [Expected time = 30mins for tasks]

The lab instructor will give you unseen task depending upon the progress of the class.

# Evaluation criteria

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Marks** |
| 1 | Understanding of Problem | 20 |
| 2 | Program Logic | 20 |
| 3 | Program Implementation | 20 |
| 4 | Program Correctness | 10 |
| 5 | Use of Tool | 10 |
| 6 | Viva | 20 |
| **Total** | | **100** |

Lab-5

Implementing Decimal to Binary Conversion in LMC

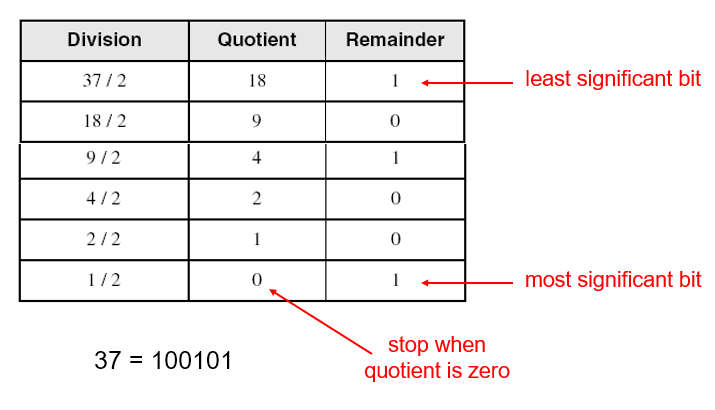
Part 2

# Introduction

In the previous lab, we implemented division as repeated subtraction and multiplication as repeated addition because we do not have any division or multiplication instruction in Little Man Computer (LMC). In this lab we will use this implementation of division to implement decimal to binary conversion.

# Concept Map

The following figure illustrates the method of decimal to binary conversion.



We have already implemented the division as repeated subtraction using the following code in the previous lab.

LDA 80

STA 84

LOOP LDA 84

SUB 81

STA 84

BRP INCQ

BRA END

INCQ LDA 82

ADD 83

STA 82 //INCQ ends

BRA LOOP

END LDA 84 //Remainder

ADD 81 //subtr correction

STA 84

OUT

LDA 82 //Quotient

OUT

HLT

//80 stores dividend

//81 stored divisor

//82 is quotient. init 0

//83 stores 1 as inc

//84 is temp storage. Stores Remainder

Using this implementation whenever we need division, we will implement decimal to binary conversion. The logic of the program is explained in the following walkthrough task.

# Walkthrough Task

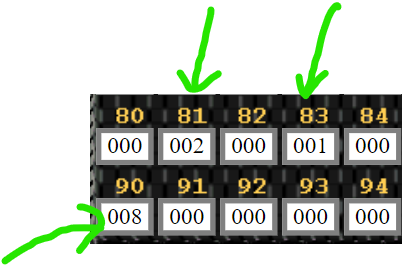
The basic idea of the program is as following.

1. Load the number to be converted in the accumulator from some fixed memory location. We will use address **90** for this purpose.
2. Store this number at address **80** as our division algorithm uses this address to store the dividend.
3. Store value 2 at address **81**. This is the address where the division algorithm stores divisor.
4. Store value 1 at address 83. This is where the division algorithm stores increment 1.
5. Mark the division algorithm with a label. We will use label DIV here. Branch to the DIV
   1. It is like calling a function for division which will divide the value stored at address 80 with value 2 stored at address 81.
   2. Once the division completes, it will store the remainder at address 84 and the quotient at address 82. We print this remainder and store the quotient at address 90 for the next iteration.
   3. **An important point here is that in the previous lab we did not set quotient back to its initial value 0 because we performed only one division. Now as this division algorithm will be called again and again, we need to set the quotient to 0 before next iteration.**

The complete program with description of each line is given below.

|  |  |  |
| --- | --- | --- |
| Line |  | Description |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | START LDA 90 //number  STA 80 //Divisor  BRA DIV  DIVRET BRZ ENDPROG  BRA START  DIV LDA 80  STA 84  LOOP LDA 84  SUB 81  STA 84  BRP INCQ  BRA END  INCQ LDA 82  ADD 83  STA 82 //INCQ ends  BRA LOOP  END LDA 84 //Remainder  ADD 81 //subtr corr  STA 84 //Remainder  OUT  LDA 82 //Quotient  STA 90 //next time  LDA 99 //0 value  STA 82 //init Q  LDA 90 //quotient  BRA DIVRET  ENDPROG HLT | Number to be converted  Store the number at address 80  Branch to Division part  Label to return from division. End if Acc 0  Otherwise branch to start for next iteration  Div algorithm starts here. Load dividend  Store it at 84 as temp value  Loop for div as repeated subtraction  Division is complete now. Load remainder  Add divisor for correction (see last lab)  Update remainder to 84  Print the remainder  Load quotient  Store it to address 90 for next time  Load value 0 from address 99  Store 0 to address 82 for next iteration.  Load quotient back to accumulator  Branch to DIVRET. return from function |

As an example let’s convert 8 to binary which will be 1000. The memory will look like the following.



Address 90 stores the number.

Address 81 stores 2 for division with 2 each time.

Address 83 stores 1 as an increment value of 1.

Run the program and observe its behavior.

# Practice Task

## Practice Task 1 [Expected time = 50mins]

Write a program to convert binary back to decimal.

# Evaluation Task (Unseen) [Expected time = 30mins for tasks]

The lab instructor will give you unseen task depending upon the progress of the class.

# Evaluation criteria

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Marks** |
| 1 | Understanding of Problem | 20 |
| 2 | Program Logic | 20 |
| 3 | Program Implementation | 20 |
| 4 | Program Correctness | 10 |
| 5 | Use of Tool | 10 |
| 6 | Viva | 20 |
| **Total** | | **100** |

Lab-6

Introduction to Intel 8086/8088 Assembly Language

# Introduction

By now we are familiar with computer organization and even have written simple programs in a basic simulator. Hence it is now time to move to the assembly language we will be using in this course which is Intel 8086/8088 Assembly Language. In this lab we will learn how to setup the software and will write a simple program in this language.

# Concept Map

We need the following software.

* Notepad (for writing programs)
* NASM (Assembler). Netwide Assembler
* AFD (Debugger). Advanced Full-screen Debugger
* DOSBox (DOS emulator)

Follow the following steps to download and install the software

* Download the DOS version of nasm (Netwide Assembler) from <https://www.nasm.us/pub/nasm/releasebuilds/2.15.05/dos/>

(Download nasm-2.15.05-dos.zip)

or

<https://www.dropbox.com/s/vooq8cvomi4l49c/nasm-2.15.05-dos.zip?dl=0>

* Unzip nasm folder to a drive (C: or D:). Rename it to "Assembly-DOS".
* AFD is the debugger (to view program execution). Download AFD.exe from

<https://www.dropbox.com/s/q7p2g78ya4rt05x/AFD.EXE?dl=0>

Copy AFD.exe to the folder "Assembly-DOS" we created above.

* AFD Tutorial can be downloaded from the following link

<https://www.dropbox.com/s/a6a1u2lj3x7yi6g/AFD_Tutorial.pdf?dl=0>

* Download and Install DOSBox from

<https://www.dropbox.com/s/dk3vokjvk8jh9l4/DOSBox0.74-win32-installer.exe?dl=0>

**How to run a program?**

1. Write your program in notepad. Save your program file to the “Assembly-DOS” folder using **.asm** extension (e.g. file name is "**ex01.asm**").
2. Run DOSBox and mount the folder "Assembly-DOS" as a drive using the following command. Here **D:\Assembly-DOS** is the path to the folder

**mount X D:\Assembly-DOS**

1. Navigate to Assembly-DOS folder (drive X) from DOSBox by writing the following command

**X:**

1. Use the following command to assemble the program

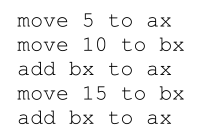
**nasm ex01.asm –o ex01.com –l ex01.lst**

1. Use the following command to run the program

**afd ex01.com**

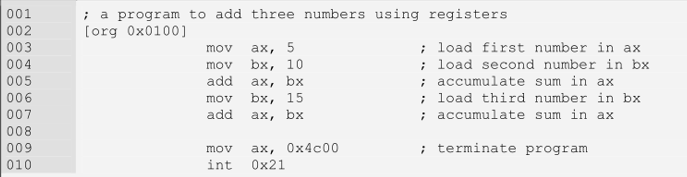
# Walkthrough Task

The first program that we will write will only add three numbers. It will clarify most of the basic concepts of assembly language. In plain English, the steps of our program are listed below.

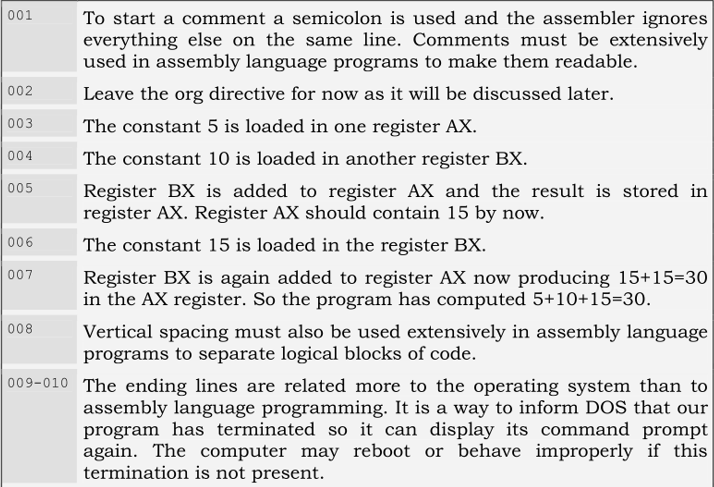


Here ax and bx are two general purpose registers which can hold values. 8086/8088 architecture has many registers and we will discuss these later.

Write the following code in a .asm file.



Description of the code is given below.



Use the following command to assemble our program.

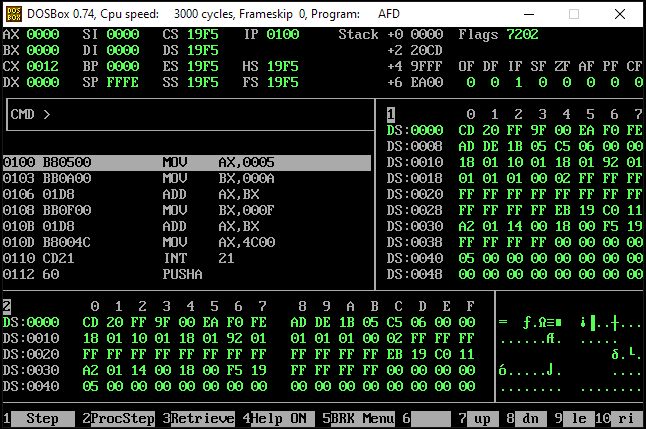
**nasm ex01.asm –o ex01.com**

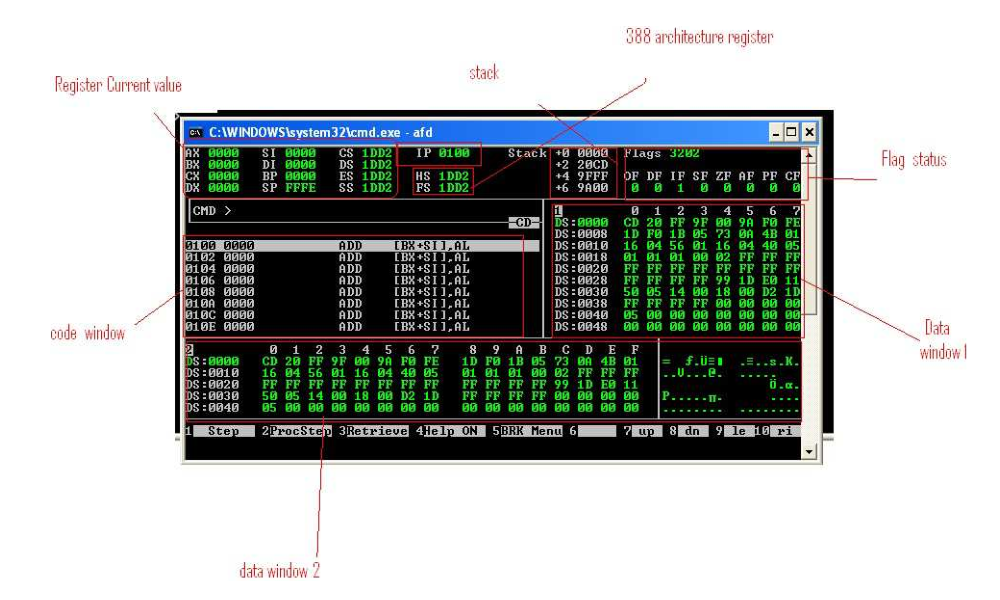
Use the following command to run the program.

**Afd ex01.com**

The debugger shows the values of registers, flags, stack, our code, and one or two areas of the system memory as data. Debugger allows us to step through our program one instruction at a time and observe its effect on the registers and program data.

Use F1 and F2 to step through the program





* Use quit command to exit from the shell.
* On DOS command prompt type cls and press Enter to clear the screen

# Practice Task

## Practice Task 1 [Expected time = 20mins]

Modify the above program to add 5 numbers.

## Practice Task 2 [Expected time = 30mins]

Using the add instruction, write a program that can multiply 6 with 10.

# Evaluation Task (Unseen) [Expected time = 30mins for tasks]

The lab instructor will give you unseen task depending upon the progress of the class.

# Evaluation criteria

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Description** | **Marks** |
| 1 | Understanding of Problem | 20 |
| 2 | Program Logic | 20 |
| 3 | Program Implementation | 20 |
| 4 | Program Correctness | 10 |
| 5 | Use of Tool | 10 |
| 6 | Viva | 20 |
| **Total** | | **100** |