Practical Sheet: Little Man Computer

All materials are at http://www.eecs.qmul.ac.uk/~william/CAS-London-2016.html

1 Introduction

This session introduces the Little Man Computer, a simple compute simulator used in both GCSE and A level syllabuses.

There are several versions available. We are using: http://peterhigginson.co.uk/LMC/

1.1 LMC Registers

1. **Program counter**: this register holds the address of the next instruction to be executed.
2. **Accumulator** (or calculator): this is the computer working memory. Arithmetic operations and load/store use it as one of the operands.
3. **Memory address register** (MAR): this register holds the address at which the memory is accessed. In the fetch part of the cycle, it has the address of the instruction; in the execute stage it has the address of the data value.
4. **Instruction register** (IR): this holds the opcode of h current instruction.
5. **Memory data register** (MDR): this register holds the data passing to or from the memory. *Not visible in the version we are using*.

Only the accumulator is used directly by the programmer. The other registers help to show the fetch-execute cycle in action.

The **memory** has 100 locations, numbered 00 to 99. It holds values in denary (not binary), with three digits. (Some versions allow negative numbers).

1.2 Instructions

The following table shows the available instructions:

Name	Op Code	Operand (or n/a)	Description
ADD	1xx	xx = address of data	Calculate Acc + data
SUB	2xx		Calculate Acc – data
STO	3xx		Store Acc at the address
LDA	5xx		Load data from the address to Acc
BR	6xx	xx = program address	Branch to new address
BRZ	7xx		Branch if Acc is zero
BRP	8xx		Branch if Acc is positive
IN	901	n/a	Input from user to Acc
OUT	902	n/a	Output from Acc to user
HLT	000	n/a	Halt or Stop
DAT	n/a	Initial value	Storage location

The mnemonics are used in the assembly code. The format is:

LABEL <tab> OPCODE <tab> OPERAND

The label is optional. The operand is either a label or the initial value of data. Note that 'DAT' is not an instruction but rather a directive to the assembler to reserve space for a variable. This is why it does not have an opcode.

2 Simple Programming Exercises

Exercise 2.1: Enter a Program in LMC

Consider the following Python program:

```
z = x + y
```

This can be compiled (*note: you are the compiler*) into the follow LMC program

```
        LDA   x
        ADD   y
        STA   z                 or STO      z
        HLT
x       DAT   11
y       DAT   17
z       DAT
```

Enter this program and assemble it. Answer the following questions:

1. What is the memory location for the variable 'x'?
2. The first memory location has value '504'. Explain how this value represents the first instruction in the program.
3. Describe the program in words.

Exercise 2.2: How Registers are Used in the LMC

The program of Exercise 2.1 has 4 instructions. Run the program, step by step and complete the following table to show the state of the system after each step:

Step	Program Counter	Memory Address Register	Instruction Register	Accumulator (Calculator)
0	00	---	---	000
1				
2				
3				

In addition, look at which memory location is change. *Note: the names of the registers vary a bit between different LMC versions*

Exercise 2.3: Simple Programming Challenge

We wish to input two numbers and output their sum. Here is some 'pseudo assembly code' (like Python) for this problem:

```
acc = input
A = acc
acc = input
acc = acc + A
Output acc
```

Write the program using the following steps

1. Chose to the operation name required for each step
2. Write the textual assembly code, using the LMC software
3. Work out the translations
4. Use the LMC software to assemble your code and check the answer
5. Test it!

Exercise 2.4: Further Simple Programming Challenges

Try any of the following:

1. Input three numbers and output the sum
2. Input a number, double it twice and output (e.g. 7 input gives 28 out).
3. Input two numbers and calculate the difference. What happens if the answer is negative?

Before writing the assembly code, write a 'Python version', as shown above. Consider what other techniques you could use (e.g. flowcharts) as a stepping-stone to the assembly code.

3 LMC Programs with Branches

We have not yet used the LMC with BR, BRZ and BRP opcodes. These allow us to write program with if-statements and loops, though it is much less convenient than in a high-level language.

3.1 Writing IF-Statements in Assembly Code

Consider the following Python-like program:

```
input x
input y
if x > y:
    output x
else :
    output y
```

The problem here is how to translate `x > y` into LMC instructions. The following Python-like program shows the principles, using an accumulator:

Equivalent Python	LMC Assembly Code
`acc = input`	`IN`
`x = acc`	`STA x`
`acc = input`	`IN`
`y = acc`	`STA y`
`acc = acc – x`	`SUB x`
`if acc <= 0:`	`BRP yGTx`
`    acc = x`	`LDA x`
`    output acc`	`OUT`
	`HLT`
`else :`	
`    acc = y`	`yGTx LDA y`
`    output acc`	`OUT`
	`HLT`

Check that you understand this step of the translation.

3.2 If Statements Exercises

Exercise 3.1: Largest of Two

Enter the 'largest of two' program described in the reference notes and test it with different values. Step through it and look at the value of the program counter.

Can you explain how the 'if statement' has been compiled using the `BRP` instruction? Note: if you know about flowcharts, you might find it useful to draw a flowchart of the program.

Exercise 3.2: LMC Challenge Problems Write a MC program to solve the following problems:

1. Input two numbers and output the smaller one
2. Input three numbers and output the largest.

You are recommended to write the program in simplified Python (as above) and/or draw a flowchart before attempting to write the assembly code.

3.3 Writing Loops in Assembly Code

Like If-statements, loops are created using branches. Consider the following program in Python-like language. If the input is 5, the program outputs 5, 4, 3, 2, 1

```
input x
while x > 0 :
      output x
      x = x – 1
```

The following Python-like program shows the principles of implementing this in LMC, using an accumulator. We have used a 'goto' statement, which of course Python does not have. The lines have line numbers on the left:

```
1:    acc = input
2:    if acc == 0 : goto line 6
3:    output acc
4:    acc = acc – 1
5:    goto line 2
6:    halt
```

The final code is:

```
      IN
loop  BRZ   end
      OUT
      SUB   one
      BR    loop
end   HLT
one   DAT   1
```

In general, loops correspond to **backward jump**. In the example above:

- An unconditional branch (BR opcode) jumps back to the start of the loop.
- A conditional branch (opcode BRZ) exits the loop when the condition given in the while statement is no longer true.

3.4 Exercises with Loops

Exercise 3.3: LMC Counting Down

Enter the 'down counter' loop program described above and test it with different values. Step through it and look at the value of the program counter. Again, explain how the loop is translated to LMC using a flowchart.

Exercise 3.4: LMC Looping Challenge Problems Try any of the following:

1. Enter two numbers C, N: C is a counter and N is a number. Output the first C multiples of N, starting with 0. So if C is 5 and N is 3, the output is 0, 3, 6, 9, 12.
2. LMC does not have a multiply instructions but multiplication can be done by repeatedly adding. For example 3 x 4 is equal to 4 + 4 + 4.