Session 5 Reference Notes

CPU Architecture and Assembly

1 Little Man Computer

1.1 Versions

Little Man Computer is a widely used simulator of a (very simple) computer. There are a number of implementations. So far, I have found:

1. A web applet, using Java, with instructions and examples (see home page) from York University in Canada.
    a. Home page: http://www.yorku.ca/sychen/research/LMC/
    b. Web applet: http://www.yorku.ca/sychen/research/LMC/LittleMan.html
    c. The applet is good if it will run in your browser; I find that only one digit of the memory is displayed. The word 'compile' is used instead of assemble.

2. Another Java version from the University of Minnesota; also available as a web applet:
    a. See from http://www.d.umn.edu/~gshute/cs3011/LMC.html
    b. A minor issue: this version does no show the MAR and MDR registers and does not simulate the separate stages of the fetch-execute cycle.

3. A version from Durham University for Mac or Windows
    a. See http://www.dur.ac.uk/m.j.r.bordewich/LMC.html
    b. This version calls the 'assembler' a 'compiler' which is unfortunate, but it is otherwise good.

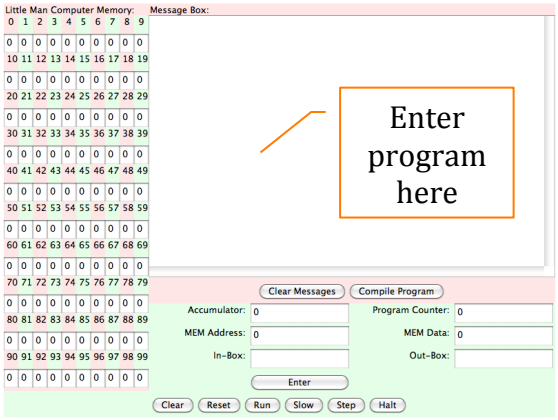
4. A MS Windows version, requiring .NET. Excellent if you can run it.
    a. Available from http://www.gcsecomputing.org.uk/lmc/lmc.html

5. A spread sheet version
    a. http://www.ictcool.com/2011/12/16/download-lmc-simulation-v-1-5-2-requires-microsoft-excel/
    b. This version does not include an assembler: you can enter the 3-digit codes instead.

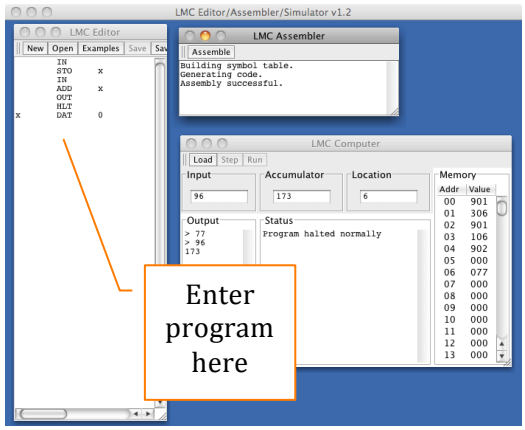
6. A flash version
    a. Available as a CAS resource from http://community.computingatschool.org.uk/resources/1383
    b. This version is mainly used for demonstration rather than programming; there is no assembler. The web page is useful.

There is also an informative Wikipedia page
http://en.wikipedia.org/wiki/Little_man_computer

Applet version



Java version



Important note: the applet uses the mnemonic 'STA' whereas the Java version uses 'STO' for store accumulator.

1.2 Registers

The LMC registers are:

1. **Program counter**: this register holds the address of the next instruction to be executed. This automatically increases by 1 after each instruction, except for branching instructions.

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 is not used directory by the programmer. It 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 (if data is moved to/from memory).

4. Memory data register (MDR): this register is not used directly by the programmer. It holds the data passing to or from the memory. This data can be either a program word or a data value.

The MAR and MDR registers help to show the fetch-execute cycle in action.

The **memory**

- Has 100 locations, numbered 00 to 99.
- Holds values in denary (not binary), with three digits. (Some versions allow negative numbers).

1.3 Instructions

The following table shows the available instructions:

Mnemonic	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.

1.4 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.

1.5 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 Pythin 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.

2 Understanding Compilers and Interpreters

2.1 What we Learn from Assembly Code

Learning about assembly code, remind us that:

1. Variables correspond to memory locations.
2. The memory of a computer contains both data and code.
3. If statements and loops are created by changing the Program Counter.

2.2 Compilers

A compiler is a translator from a high level language to the assembly code of a particular CPU. A compiled program works on

- the particular CPU and
- Operating System

that it was compiled for. Internally, the compiler has several stages:

1. A parser checks that the source code follows the syntax of the language. A tree is constructed representing the program code. At this stage, syntax errors are generated.

2. The type checker checks that the expressions in the program are correctly typed and how much space is need for each variable. At this stage, the errors generated concern variables (and other names) that are not declared and code that is incorrectly types.
3. The code generator then translates the program to assembly code. Compilers usually include an assembler so the output is usually in binary (call object code) rather than assembly code. The two main tasks are i) deciding which register to use (as, unlike LMC, modern CPUs have many registers) and ii) choose the CPU instructions.

The first two steps are determined by the language being compiled; the final step is determined by the processor being targeted. Modern compilers have a modular structure, which front-ends for different source languages and back-ends for different CPUs. In addition, modern compilers include optimisers that make the generated code faster without changing its meaning. These optimisers typically operate on an intermediate language, which is used for all source languages and all CPUs.

2.3 Interpreter

An interpreter is simpler than a compiler. It includes the parser but instead of the code generator, the interpreter goes through the internal representation of the source code (such as an abstract syntax tree) and 'executes' the code directly.

Although in principle any language can be compiled or interpreted, languages that are usually compiled tend to be dynamically typed and scoped, while compiled languages are statically typed and lexically scoped.

> **Dynamic v static typing**: in dynamic typing, the type of a variable depends on its use and may change at different points in the program. Since the type is not know in advance, the operation (e.g. integer versus floating point arithmetic) can not be determined either, which is inconvenient for a compiler.
>
> **Dynamics scoping**: scoping is about matching names to variables (or memory locations). In a lexically scoped language, such as C, the compiler matches names to variables. In a more dynamic language like Python, names are 'resolved' at run time and the process depends on the variables that exist when a reference to a name is executed. Many languages include aspects of both approaches.

One way to understand how an interpreter works is to write one for the LMC. (Note: an interpreter for a CPU is often called an emulator or a simulator). Here are some fragments of such a program to illustrate the idea:

Represent the state of the system. The LMC state is its registers and memory.

```
acc = 0
mdr = 0
mar = 0
pc = 0
memory = [504,105,306, 0, 11, 17,...]
```

Update the state: this means following the rules of the CPU. In the LMC, the way the data moves between registers depends on the opcode:

```
def execute(memory, opcode, arg):
    global acc, mar, mdr, pc
    if opcode == ADD:
        mar = arg
        readMem(memory)
        acc = acc + mdr
    elif opcode == SUB:
        mar = arg
        readMem(memory)
        acc = acc - mdr
    elif opcode == STO:
        mar = arg
        mdr = acc
        writeMem(memory)
    elif opcode == LDA:
        mar = arg
        readMem(memory)
        acc = mdr
    elif opcode == BR:
        pc = opcode
    elif ...
```

Some of the additional functions needed to complete the interpreter are shown below:

```
def readMem(memory):
    global mdr
    mdr = memory[mar]

def writeMem(memory):
    memory[mar] = mdr

def fetch(memory):
    global pc, mar
    mar = pc
    pc = pc + 1
    readMem(memory)
```

2.4 Java and Virtual Machine

Many systems combine aspects of both compilers and interpreters. A notable example is Java and the similar approach taken by the Microsoft .net language family.

Java is a compiled language but it is not compiled for real CPUs. Instead, the compiled code is for a Java Virtual Machine (JVM). As there are no real JVM CPUs, they are emulated. This approach has many advantages: for example, only one compiled version of a program is needed and it can be run on any machine with an emulator, but it is much faster than a pure interpreter.