Little Man Computer

Additional Notes on Compiler and Interpreters

1 Understanding Compilers and Interpreters

1.1 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.

1.2 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.

1.3 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.

2 Writing an LMC Interpreter

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). Below, we outline how an LMC interpreter can be written. Here is an example of a possible solution being used.

```
Load (L) Run(R) Stop(S) > R
MAR = 0 MDR = 0 ACC = 0 PC = 0
Program halted
Load (L) Run(R) Stop(S) > L
Location = 0 Enter value (or '.') 504
Location = 1 Enter value (or '.') 105
Location = 2 Enter value (or '.') 306
Location = 3 Enter value (or '.') 0
Location = 4 Enter value (or '.') 11
Location = 5 Enter value (or '.') 17
Location = 6 Enter value (or '.') .
Load (L) Run(R) Stop(S) > R
MAR = 0 MDR = 0 ACC = 0 PC = 0
Press enter to continue
MAR = 4 MDR = 11 ACC = 11 PC = 1
Press enter to continue
MAR = 5 MDR = 17 ACC = 28 PC = 2
Press enter to continue
MAR = 6 MDR = 28 ACC = 28 PC = 3
Program halted
Load (L) Run(R) Stop(S) >
```

2.1 Design Suggestions

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)
```

2.2 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.

3 Appendix 1: LMC Versions

Little Man Computer is a widely used simulator of a (very simple) computer. There are a number of implementations.

1. An excellent version in java script:
   a. Available from http://peterhigginson.co.uk/LMC/
   b. It has adequate documentation at http://peterhigginson.co.uk/LMC/help.html
   c. The MDR is not visible.

2. A MS Windows version, requiring .NET and functionally the same as 1. Excellent if you can run it.
   a. Available from http://www.gcsecomputing.org.uk/lmc/lmc.html

3. 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. Increasing incompatible with modern browsers.

4. 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.

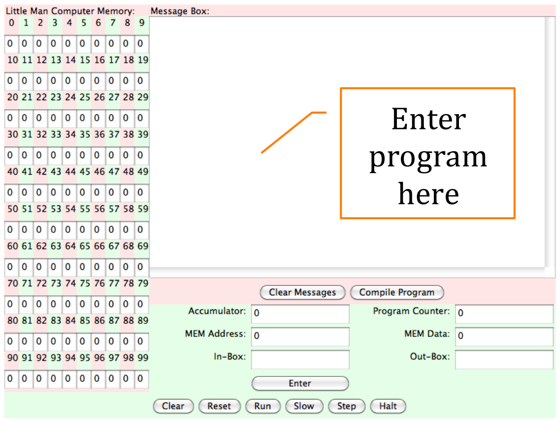
5. A version from Durham University for Mac or Windows
   a. See http://www.dur.ac.uk/m.j.r.bordewich/LMC.html
   b. This version calls the 'assembler' a 'compiler' which is unfortunate, but it is otherwise good.

6. 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.

7. 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

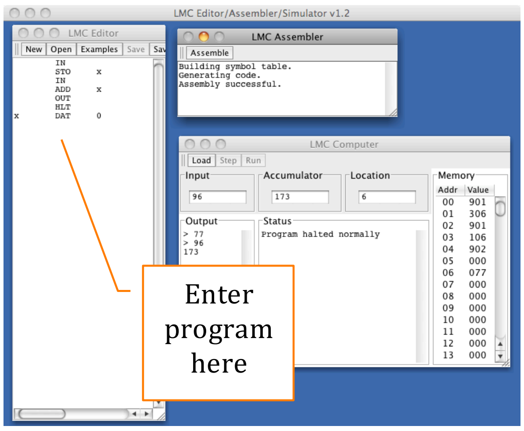


Little Man Computer Memory:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0
10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0
20	21	22	23	24	25	26	27	28	29
0	0	0	0	0	0	0	0	0	0
30	31	32	33	34	35	36	37	38	39
0	0	0	0	0	0	0	0	0	0
40	41	42	43	44	45	46	47	48	49
0	0	0	0	0	0	0	0	0	0
50	51	52	53	54	55	56	57	58	59
0	0	0	0	0	0	0	0	0	0
60	61	62	63	64	65	66	67	68	69
0	0	0	0	0	0	0	0	0	0
70	71	72	73	74	75	76	77	78	79
0	0	0	0	0	0	0	0	0	0
80	81	82	83	84	85	86	87	88	89
0	0	0	0	0	0	0	0	0	0
90	91	92	93	94	95	96	97	98	99
0	0	0	0	0	0	0	0	0	0

Accumulator: 0
Program Counter: 0
MEM Address: 0
MEM Data: 0
In-Box:
Out-Box:

Java version



```
        IN
        STO     x
        IN
        ADD     x
        OUT
        HLT
x       DAT     0
```

```
Building symbol table.
Generating code.
Assembly successful.
```

Input: 96
Accumulator: 173
Location: 6

Output

```
> 77
> 96
173
```

Status:

```
Program halted normally
```

Memory

Addr	Value
00	901
01	306
02	901
03	106
04	902
05	000
06	077
07	000
08	000
09	000
10	000
11	000
12	000
13	000

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