FML: a VM implemented in SML

Henrik Sommerland, Oskar Ahlberg, Aleksander Lunqvist

March 5, 2014

1 Introduction

For our project we have decided to build a virtual machine(VM) in SML. The name FML is just an arbitrary thre letter name and has no meaning or interpertation. The VM is a RISC machine using a Von-Neuman architecture. It has a very minimalistic instruction set. The design of FML resembles those of older 8-bit architectures such as the MOS 6510 and the Z80 microprocessors commonly in use during the late 70s and early 80s. The FML machine has no "bus width" and works exclusivley with signed integers[i]. The lack of a physical bus enables the VM to do things wich an ordinary CPU could not achive such as reading from two registers at the same time. Even though the cpu have very few operations (only 27) a very effective instruction set architecture makes these operations very flexibels and there are roughly 600 valid instruction codes. It is also noteworthy that FML is asycrounous and has now predefined clock frequenzy.[ii]

So even though FML is a very minimalistic machine it is quite powerfull. We have allso built a fully featured assembler for the FML machine.

2 The VM

Here a informal description of the workings of the machine. For a more detailed description se the VM specifications in the appendix.

2.1 General

The FML machine is built up as a very simple von-neuman architecture. The machine consists ony of a few major components. It's notewhorthy that there is no instruction decoder present. This is since all of the instruction decoding and handling takes place within the software implementation of the machine. The size of the memory the machine has avaliable is arbitrary and is defined

[i]The details of the integers used are dependent on which SML implementation is used

[ii]Allthogh for debugging purposues one can use both manual stepping and a fixed update speed.

at the initilazion of the machine. Below will follow a dataflow diagram of the machine, describing all of the components and how they can communicate.

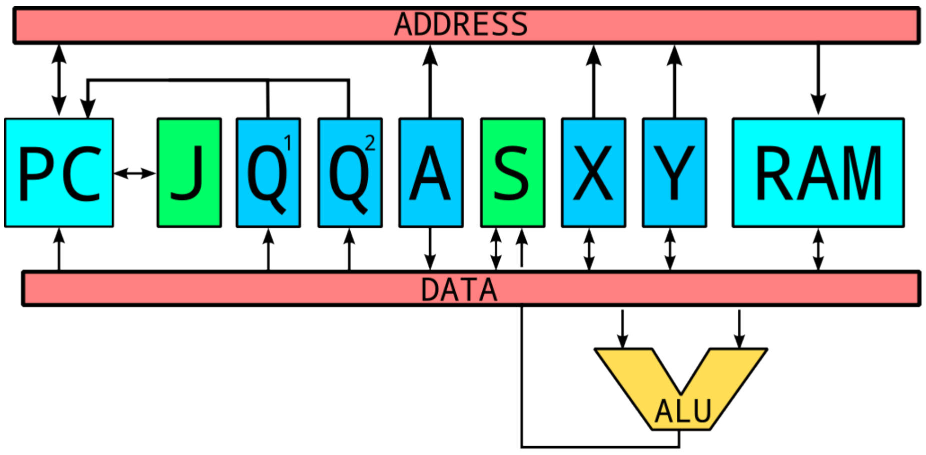

Figure 1: Dataflow diagram of the FML machine

Now this image might be a little bit confusing. One should consider the two read rectangles DATA and ADDRESS as "viritual buses". One can interpret the picture as: X can both read and write from other components and be used for addressing. Below will follow brief descriptions of the components. More indepth descriptions are given in the appendix.

X and Y

These are the two general purpose registers wich can be read, written to and used for addressing.

S

This is the general purpose stack. It can be both read and written to. Everytime some thing gets written to the stack it gets pushed onto the stack and everytime something is read from the stack the stack gets popped. The stack can not be used for addressing.

A

This is only a virtual register. It is read only and can be used for addressing. This is only used if an instructuion uses a non-register argument[iii].

Q^1 and Q^2

These are the two interupt registers. These are very special and can only be written to. They will hold the adresses to wich the machine should jump if an prepherial component makes a interupt request. More on this later.

[iii]A non-registry argument is a argument wich is not any of the registers, the stack, or something from the memmory. The value of A will (if used) be at the memmory cell directly folowing the one at wich the program counter is.

PC

This is the program counter. It keeps track on where in the memmory the instructions are being read from. It allso handles the jumping.

J

This is the jump stack. This stack is used to store the return addresses for subroutine jumps. This stack can only be manipulated by the program counter.

ALU

This is not really a ALU. The machine does not have a seppareta ALU component but this is just here to illustrate that the all of the components which can be read from can be used as arguments for arithmetic and logical operations. All of the results from the arithmetic and logical operations are always put on the stack.

RAM

This is the random access memmory of the machine.

2.2 Instruction Set Architecture

The ISA of the VM is built in a special but simple fashion. Each instruction corresponds to a six digit integer.. Where each digit corresponds to specific information regarding different types of opcodes. The digits counting from right to left is:

First Second argument

Second First argument

Third Arithmetic operations

Fourth Logic operations

Fifth Jump operations

Sixth Special

This system of encoding information into each digit of the instruction makes the implementyation of the instruction controler and the assembler much easier. It allows for all the operation types to be grouped into numerical ranges and it gives a lot of flexibility. Note that some of the instructions may be invalid and some might be nonsensical but the instruction controler crashes if a invalid instruction is encountered. The assembler is written in such a way that it can only generate valid instructions. So an example would be: 000401. Where the 4 tells us that it we should perform a modulo operation, he 0 says that the second argument is the **X**register and the last 1 says that the first argument is the **Y**register. Notice that the order of the last two digits is reversed in respect to the order of the arguments in the operation. This is due to a design choise made

early in the design phase. It makes the instructions code a bit more confusing to read but it makes the assembly code become far more intuitive.
For a more detailed description of the ISA se the VM specifications in the appendix.

3 Assembler

3.1 General

The assembler wich we have written for the FMl machine is a very basic yet powerfull assembler. The assembler doesnt do much more than catch invalid opcodes and arguments. It allso enables the use of both label pointers and value pointers. The main tasks of the assembler is the instruction code generation and address resolution. The syntax of the assembler is inspired by the syntax for the MOS 6510 assembly lanuage and primarily the syntax of the Turbo Assembler for the Commodore 64. Below a short example of a assembly program will follow:

```
% This a simple program wich fills a part
% of the memmory with 100 consecutive integers
% trough rellative addressing.
#start
MOV 0 x
@start_address
MOV start_addres y
#loop
MOV x $y
INC x
INC y
BLE x 100
JMP loop
HLT
```

A line starting with a `#` declares a label. The address of the label will correspond to where in the code the the lebel gets declared. A line starting with a `@` declares a value. The address of the label will be assigned independent of where in the code it apepars. For a more indepth description of how the assembly lanuage se the Assembler part of the description.

3.2 Implementation

3.2.1 General

The assembler works in a fairly straight forward way. The first step in the process of asemblying is the lexiographical analysis in wich the lines in the text file gets tokenized. In this tage an "intermediate structure" [iv] gets constructed. This is an object wich contains all of the labels, values and alist of the tokenized lines. The list of the tokens contains tupels of `(label,offsett,token)` wherer the lable is the last initialized label and the offset is how many addresses away from that line the current token is. Allof the labels and values will not be assigned an adress in this phase. It is in theis phase where the opcodes and their arguments gets converted in to there corresponding numerical instruction code. It is also during this phase in wich the syntax gets checked. If a syntax error is encountered the assembler will stop emideatly. When the lexiographical analysis has been completed a check for duplicate pointer declaration is performed.

The next phase in the assembly is the address resolution phase. This is done in two phases, in the first one the labels gets resolved and in the second the values gets resolved. It begins by first resolving the labels. This is done by first giving the assembler a base address wich is the adress of the first label. As of now the first non comment line in the input file has to be a label since every line has to have a label assigned to it. Then the address resolution function continues down the intermediate sturcture and remebering wich line it is at and what its last read label was. When it runns into a new label token it will sett the new label to its current address and then continue on untill it has gone trough the entire intermediate structure. After the labels have been resolved the assembler starts to resolve the values. This is done in a very straightforward way. The assembler just looks at the last address of the last entry in the output of the firsst pass and looks at the last address, adds one to it and the just places the all the values in after that address in the same order as they appeared in the file. After all the adress has been resolved the assembler runns trough the list of tokens and replaces every pointer token with its correct adress.

After this is completed the assembler finalizes the code by converting everything into a list of integers wich then gets outputed to a file. And that children is how assembly code gets turned in to machine code.

the assembler runns in linear time with respect to the number of lines in the code. This is under the assumption that the number of lines are far greater than the number of values and labels in the code. This is a safe assumption for any resonably written code.

3.2.2 Flow chart

Below a flow chart will folow for how the assembler assembler the assembles the assembly code.

[iv]The use of the word structure here is a bit ambigous since it actually is a structure in sml. But in this text it will reffere to an abstract structure of data.

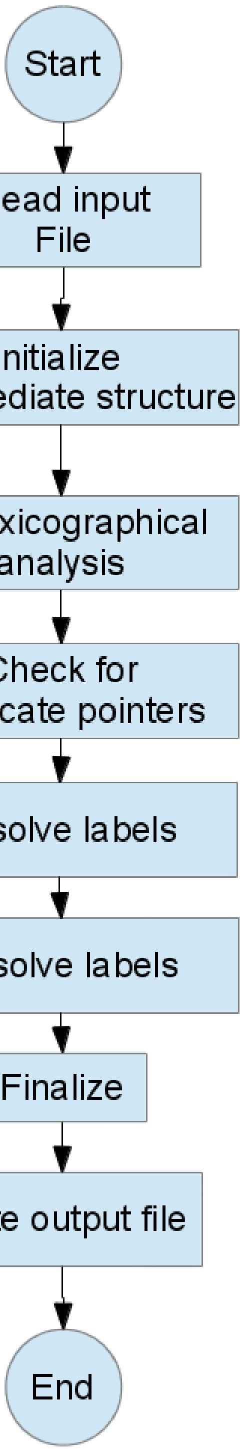
Start
Read input File
Initialize Intermediate structure
Do lexicographical analysis
Check for Duplicate pointers
Resolve labels
Resolve labels
Finalize
Write output file
End

Figure 2: Dataflow diagram of the assembler

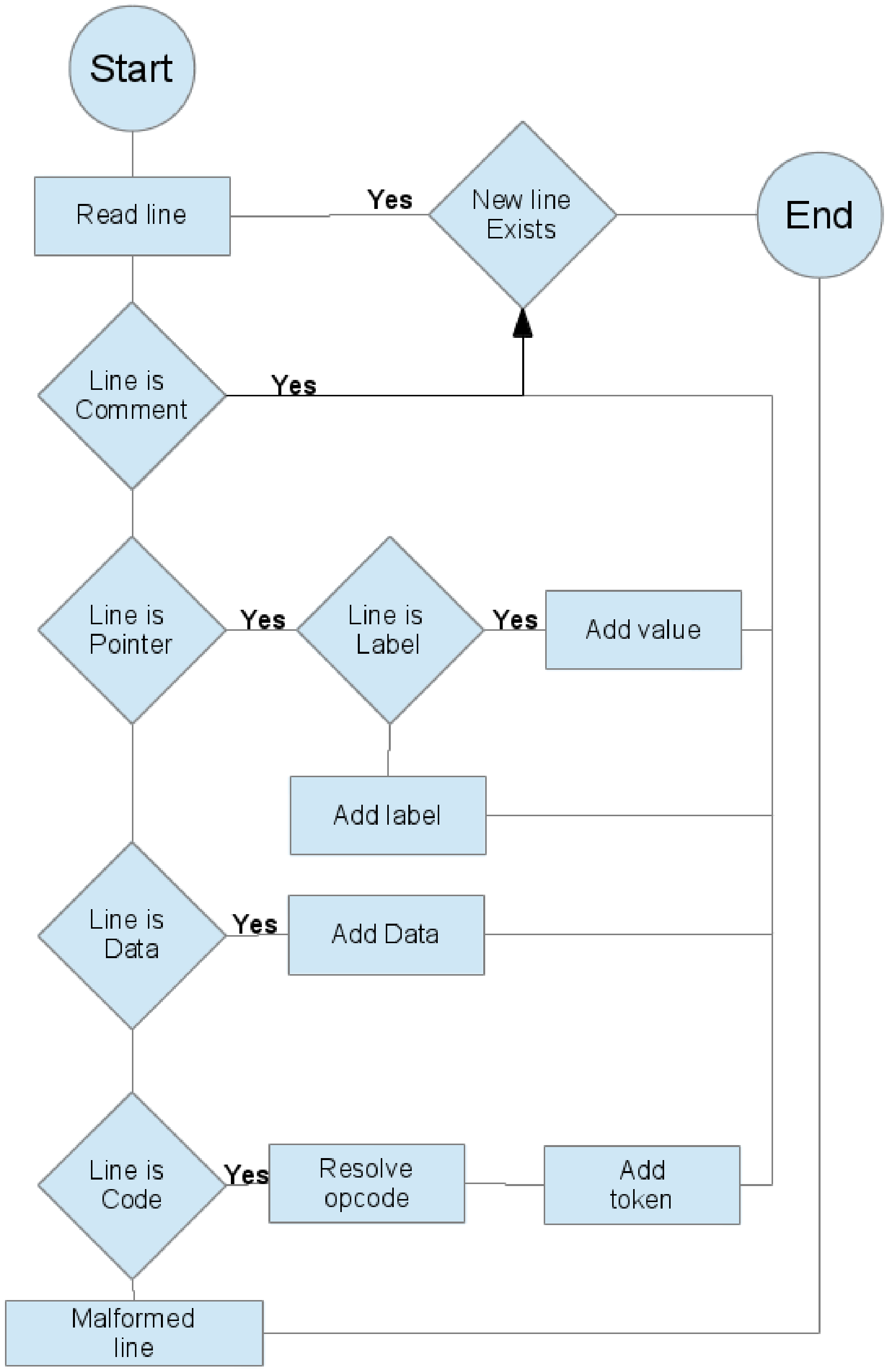

Figure 3: Dataflow diagram of the tokenization phase

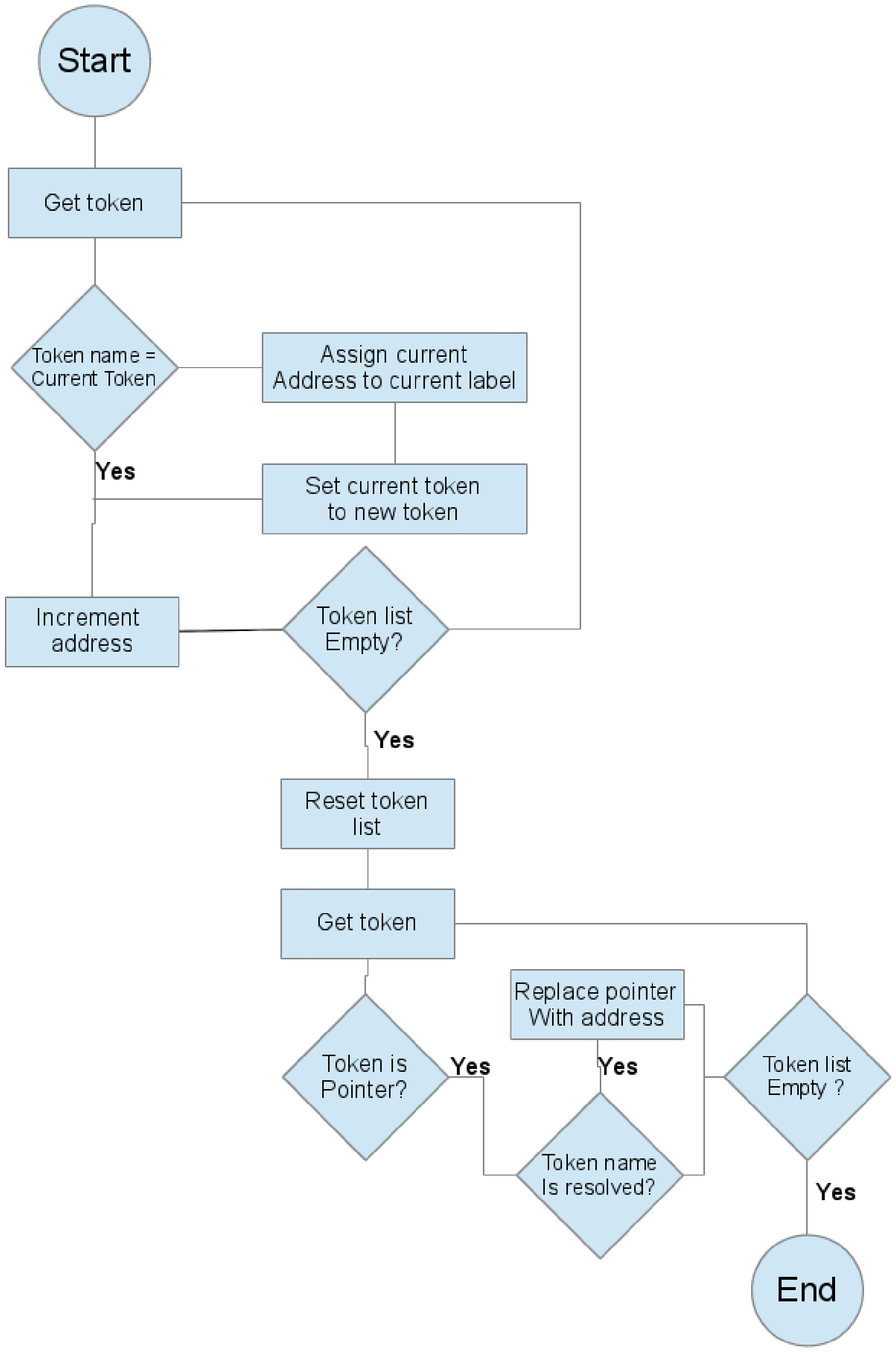

Figure 4: Dataflow diagram of the label address resolution

The i have not included a flowchart for the address resolution of the values or the finalization part since these are trivial.

3.2.3 Usage

To use the assembler propperly one has to know how to write assembly code and understand the detailed workings of the machine. We recomend studying both the VM spcifications and the assembler documentation in the appendix before you start to write programs for the machine.

The working of the assembler program is very straight forward. just frite your assembly code in a file called `in.asm` and run the `Assembler.sml` file in the sml interperter of your choosing and if there are no errors encountered during the assembling of the program the assembeled program will be outputed to a file named `out.fml`