FML: a VM implemented in SML

Henrik Sommerland, Oskar Ahlberg, Aleksander Lunqvist

March 5, 2014

Abstract

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.

[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.

Contents

1 Our work

We have tried to work as independent as possible. This has ofcourse led to some difference in how we have commented the code, some slight differnces in naming and indentation. The way we have chosen do describe how our programs work differes a bit here depending on who wrote the code for the given part of the VM.

We have not been using any form of unit testing. But instead we have done a series of more and more complicated online tests. This due to the scale and the complexity of the various funtions and algorithms of the project.

We have continously have meetings both with just us in the group and also some with our assigned TA[iii]. These meetings have primairly been about informing eathother about how the VM works and how the various components of it should be implemented. We have then had an ongoing discussion on facebook regarding details and problems wich we have encountered.

We have been using Git as a source code management system. We have been using BitBucket for to host the project and troughout all of the development process the repository has been hidden and only we in the group and our TA have had access to the code.

We are using an array in the current implementation of the memory for the VM. Now i know that it is stated in the project description that the project should be written in a functional and pure way. I discussed with both Dave Clarke and Tjark Weber about using a "monad like" structure to hide the sideffects of the array hadling and they said that it was okay. The Signature handling the memory is written in such a way that any other part of the program implementing the memmory structure will not be able to se that there are any side effects. I.e there are no semantically observable side effects of the memory structure. All of the code would look exactly the same from "the outside" regardles of how we implemented the memory. And thus the program is pure dissregarding I/O

1.1 Personal notes

In this section we will give som personal notes regarding our parts in the project.

1.1.1 Henrik Sommerland

I have been incharge of designing the VM and writing the assembler. I have also written some signatures for the others in the group to help them get started. I began work very early, as soon as we had gotten permission to start on the project. I began by writing the specifications for the VM.

Even though i have never had any formal education in how computer achitectures work I have learnt a lot about it on my own. When i was younger

[iii]Our TA has during this project been Tobias Neil

(around 17) i designed a 8-bit cpu from TTL logic chps (the 7400 series). It was during this time I learned how to build a cpu. So the design of the FML machine was for me very straight forward. I have done all of the design myself and i have not copied anything from books or any previous designs. Allthough my way of thinking and reasoning about cpu architectures comes primarily from my own work and the designs of older 8-bit cpus. The design of the cpu took roughly one or two days to finish and then there has been a continous process of ironing out bugs and inconsitencies.

Then I started to write the assembler. From the start I had a pretty good idea about what I wanted the assembler to do and I had a pretty good idea how to implement it. I wrote the assembler in about three days and I then had a fully working assembler.

After I had completed the assembler I sat out to start testing it and to write the documentation for it. As I wrote the documentation and did more extensive testing of the assembler I worked out the last few bugs and such in the code.

Then I started to write signature files and such for the others in the group to get to work on. I allso started to write on the major report for the entire program whilst guiding the others in the group.

In general I have found this project to be increadibly fun and interesting. This will sound very sad (wich it is) but doing things like this and climbing is what makes life worth living. In the beggining I was worried that the others in the group where not going to be able to understand how it all woked and even though it's really hard to explain something complex wich is crystal clear in my head to other people the others in the group have ben very enthusiastic and have really pulled trough this mammoth of a project.
I know i might have gotten a bit carried away here. I'm actually on medication not to do stuff like this.

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 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.

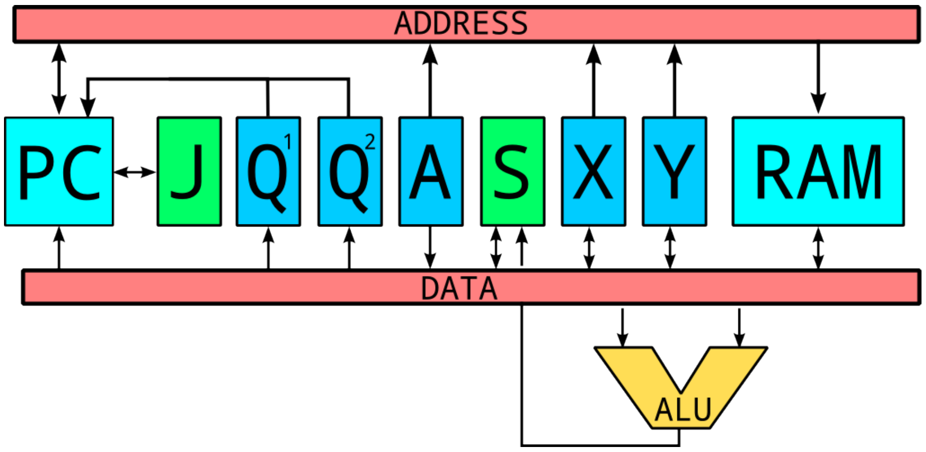

Figure 1: Dataflow diagram of the FML machine

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[iv].

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.

[iv] 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 Utillities

For this program we have written some utillity files. The IO.sml and the StringUtills.sml files just contains general helper funtions and thus play little importance in the larger scheme of things. These two files will not be described here.
We will though give a shorter descrptions of the OpcodeResolve.sml file. This is an important file since it works [v] as a interface for both the assembler and the VM. If both the assembler and the VM addheres to this structure the assembler will not generate any instructions not accepted by the VM. In general the `ResolveOpcode` structure just contains a lot of lookup tables in one can find important information regarding the different instructions and opcodes, such as which number corresponds to what type of operation, which arguments are allowed for each operations and so forth. Since this structure was not propperly used in the VM implementation there are still some things left to write for it, such as a series of reverse lookup functions and checking for invalid opcodes.

4 Assembler

4.1 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. The assembler is now fully functional and there we dont see any need to augment it or redisgning any spects of it. The assembler should only generate valid instructions but due to a major design error in the implementation of the VM this is not neccesarily true any more. 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
```

[v] It was supposed to work like this but due to an implemtation fault in the VM it does not.

```
@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.

4.1.1 Implementation

4.1.1.1 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" [vi] 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

[vi]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.

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.

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

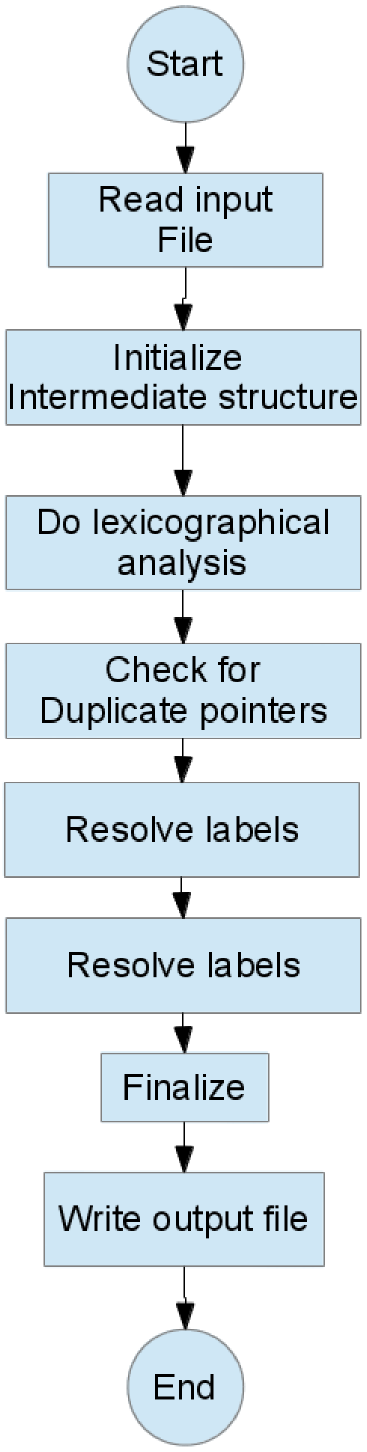
Start
Read input File
Initialize Intermediate structure
Do lexicographical analysis
Check for Duplicate pointers
Resolve labels
Resolve labels
Finalize
Write output file
End

Figure 2: Dataflow diagram of the assembler

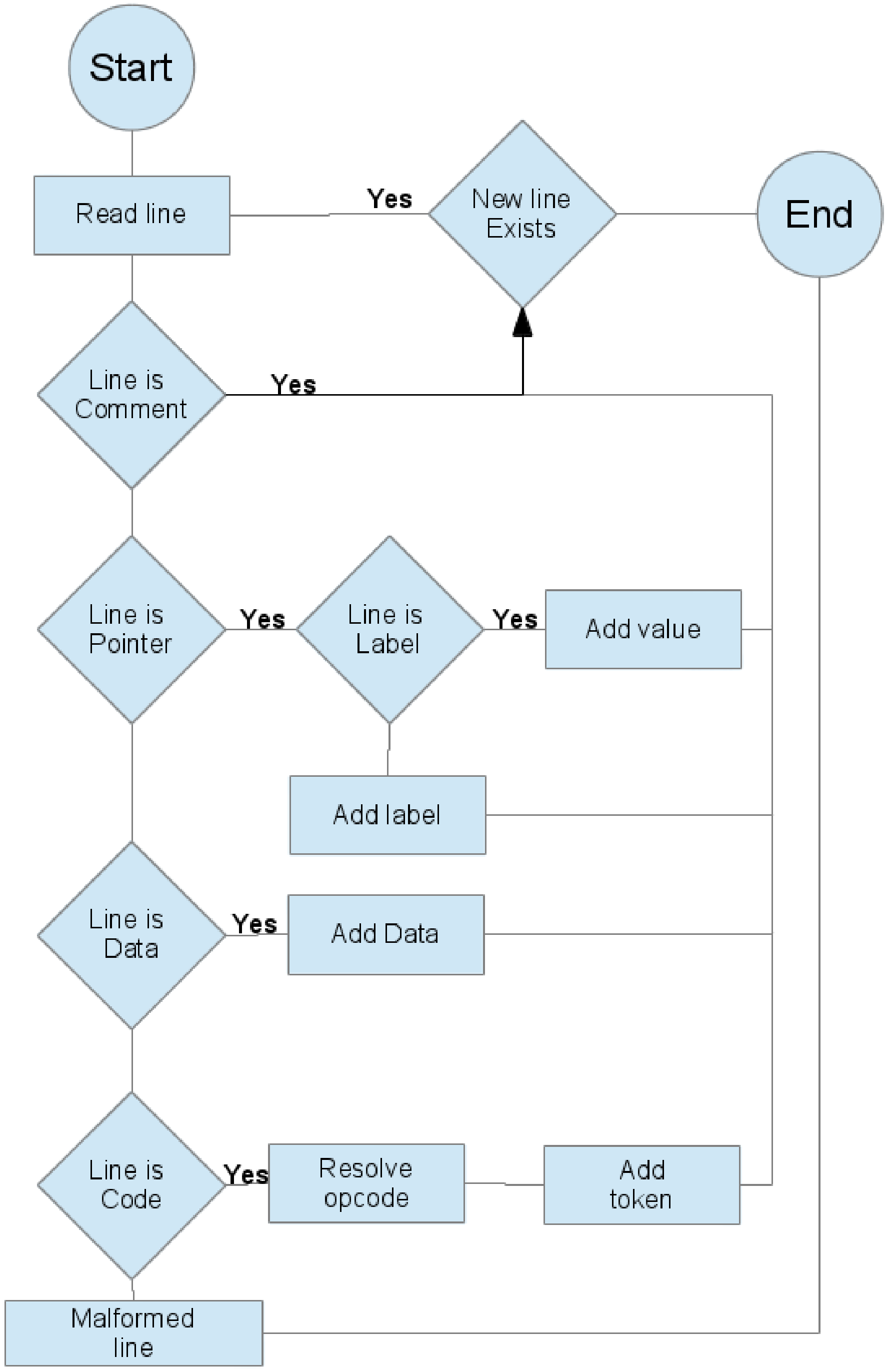

Figure 3: Dataflow diagram of the tokenization phase

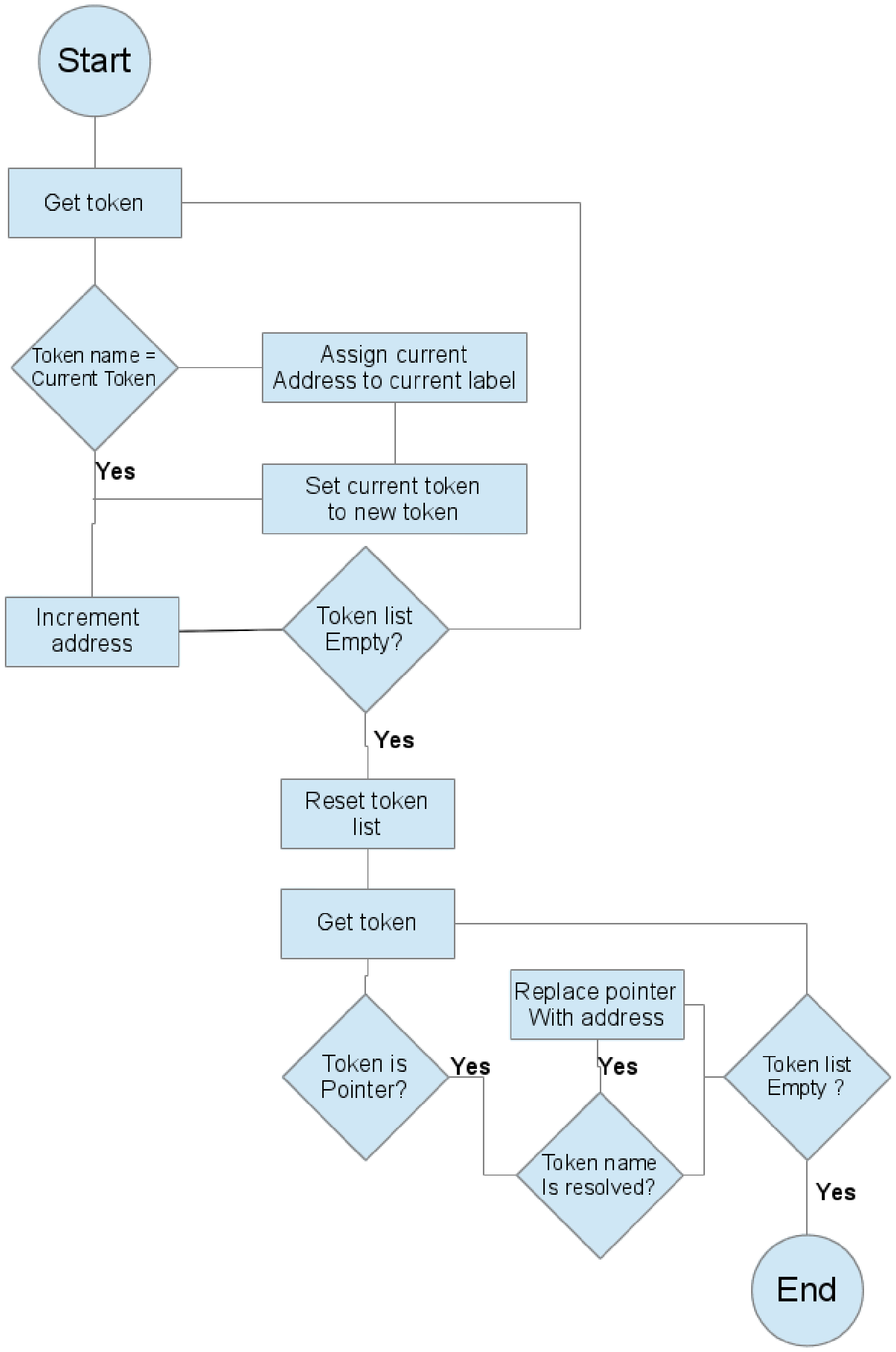

Figure 4: Dataflow diagram of the label address resolution

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

4.1.1.3 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`.

5 Appendix

5.1 VM specifications

5.1.1 Structure

The VM consists of 9 components. Two general purpouse registers(**X** , **Y**), one general purpose stack (**S**), One virtual read only register (**A**), One jump stack **J**, Two IRQ adress registers ($\mathbf{Q_1}$, $\mathbf{Q_2}$), One "ALU", One program counter (**PC**) and of course a random acces memory.

5.1.1.1 The general purpose registers

The two general purpose registers **X** and **Y** are both capable of being used for all arithmetic operations and their values can also be used as addresses. These two registers can be incremented and decremented.

5.1.1.2 The stack

The stack **S** is a standard LIFO stack of unlimited size. The stack can not be used for adressing. One can not read the top of the stack without popping it. If one tries to get a value from an empty stack an exception will be raised and the VM must halt.

5.1.1.3 The Argument Register

Now this is just a virtual read only register. The argument **A** is only accesible if the instruction being executed takes a predefined argument. The argument will be the value of the memory location after the location at which the **PC** is currently pointing. No well formed instruction should refere to **A** unless it is supposed to.

5.1.1.4 The Jump Stack

The jump stack **J** is not accesible by anything besides the **PC**. The program counter is of infinite size. The jump stack is responsible for keeping track of the return address when a subroutine is performed. Every time someone issues a subroutine jump the current address will be pushed onto the stack. When a return jump is issued **J** gets popped and its value gets assigned to the **PC**. The top entry on the stack can not be accsessed without popping the stack. If someone tries to execute a return jump if the jump stack is empty a exception shall be raised and the VM must crash.

5.1.1.5 The IRQ registers

The IRQ registers $\mathbf{Q_1}$ and $\mathbf{Q_2}$ are two pointers to the memory. These are two write only registers and can only be read by the **PC**. The IRQ registers can be assigned values like all the other registers. If a interrupt is issued the **PC** will be assigned to the value of the corresponding IRQ register and the current value of the **PC** will be pushed onto **J**.

5.1.1.6 RAM

The RAM in this machine works pretty much like any other random access memory. If any instruction tries to wrtie or read from addresses lying outisde of the size of the ram the VM should crash.

5.1.2 ISA

Every opcode is represented by a integer where each digit provides information about what the VM is to do in that step. The digits are from right to left as follows.

First Read location

Second Write location

Third Arithmetic operations

Fourth Logic operations

Fifth Jump operations

Sixth Special

Below is a table describing what each digit value corresponds to:

Value	0	1	2	3	4	5	6	7	8	9
Read	**X**	**Y**	**S**	$\mathbf{M_X}$	$\mathbf{M_Y}$	$\mathbf{M_A}$	**A**			
Write	**X**	**Y**	**S**	$\mathbf{M_X}$	$\mathbf{M_Y}$	$\mathbf{M_A}$	$\mathbf{Q_1}$	$\mathbf{Q_2}$	**A**	
Arit		`INC`	`DEC`	`ADD`	`SUB`	`MUL`	`DIV`	`MOD`		
Logic		`EQL`	`GRT`	`LES`	`BRL`	`BRR`	`AND`	`ORR`	`XOR`	`NOT`
Jump		`JMP`	`BEQ`	`BLE`	`BGR`	`JSR`	`RET`			
Special		H	Se	`POP`						

Here $\mathbf{M_X}$, $\mathbf{M_Y}$and $\mathbf{M_A}$is to be read as address of **X**, **Y** and **A**. All arithmetic and logic operations operations writes their output to the stack.
Here some examples follows:
`000042` $\rightarrow$ Move value at **S** to memory cell at the address stored in **Y**.
`000401` $\rightarrow$ Get **X** mod **Y** and write result to **S**
`020046` $\rightarrow$ Skip next instruction if $\mathbf{M_Y}$ is equal to **A**

First i would like to mention that `000000` will be the `NOP` operation since it would translate to just moving **X** to **X** . We can now group the instructions in to numerical ranges:

`000000`	`NOP`
`000001-000076`	Move operations
`000100-000776`	Arithmetic operations
`001000-009076`	Logic operations
`010000-070000`	Jump operations
`100000`	Special

As is apparent from this list many values would yield invalid or nonsense operations. The instruction decoder must take this into consideration.

Below will follow specifications for all the instruction types.

5.1.3 Instruction types

Every instruction will take exactly one cycle. Almost every instruction needs only one memory cell and should increment the **PC** by one. Any operation using a argument i.e **A** will occupy two memory cells and increment the **PC** by two.
Using jump operations may affect the **PC**in other ways. No operation except moves to the IRQ registers ($\mathbf{Q_1}$ and $\mathbf{Q_2}$) are allowed. If any other operation where to try to access the IRQ registers the opcode is invalid and the VM should crash.

5.1.3.1 Move operations

The only invalid move operations are those where the second digit is a 8 since one can not write to **A**. Although some are nonsensical such as `000011` since it would move **Y** to **Y** .

5.1.3.2 Arithmetic operations

The increment(++) and decrement(−−) operations only take one write argument and the read argument should be ignored. Incrementing or decrementing a register or memory cell updates the value stored in that registry directly and does not affect any thing else.
The other arithmetic operations takes the write digit as the first argument to the operation and the write operation will be the second argument. The result of the operation is always stored on the stack.
If one tries division by zero a exception should be thrown and the VM shall crash.

5.1.3.3 Logic operations

The logic operations work in the same way as the arithmetic operations. The comparison operations will return 0 if the result is false and 1 otherwise. Any logic operation where the 3:d digit is non zero is an illegal instruction and a exception should be thrown and the VM shall crash.

5.1.3.4 Jump operations

The standard address jump (J) will jump the **PC** to the address given by its read digit.
The conditional breaks takes the write digit as its first argument and the read digit as its second argument. If the test fails the **PC** will skipp the next instruction. This will require som tricks to implement. The VM must , at runtime, identify weather or not the following instruction takes up one or two memmory cells.

A `JSR` (subroutine jump) will take a argument in **A** and move the **PC** there and it will also put its current value on **J**.
A return jump will jump to the address at the top of **J** plus one or two depending on weather a non register argument is used. [vii] and then pop the stack.If the jump where to be empty the VM should crash and an exception should be raised.

5.1.3.5 Special

The Halt operation which just stops the VM and raises an exception.
And the `SEM` or Stack empty operation which returns 1 if the stack is empty and 0 else. The `POP` just pops the stack. I.e removing the top object.

[vii] If the return jump where to return to the value at the top of the stack it where to return to the address where the subroutine jump is and thus get stuck in a loop

5.1.4 Opcodes

Below a short summary of all the avliable opcodes will follow.

Mnemonic	**Description**	**X**	**Y**	**S**	**A**	$\mathbf{M_A}$	$\mathbf{M_X}$	$\mathbf{M_Y}$	Args
`NOP`	No Operation	x	x	x	x	x	x	x	0
`MOV`	Move operations	b	b	b	r	b	b	b	1
`INC`	Increment	b	b	x	x	x	x	x	1
`DEC`	Decrement	b	b	x	x	x	x	x	1
`ADD`	Add	r	r	b	r	r	r	r	2
`SUB`	Subtract	r	r	b	r	r	r	r	2
`MUL`	Multiply	r	r	b	r	r	r	r	2
`DIV`	Division	r	r	b	r	r	r	r	2
`MOD`	Modulus	r	r	b	r	r	r	r	2
`EQL`	Equal	r	r	b	r	r	r	r	2
`LES`	Less	r	r	b	r	r	r	r	2
`GRT`	Greater	r	r	b	r	r	r	r	2
`BRL`	Rotate L	r	r	b	r	r	r	r	1
`BRR`	Rotate R	r	r	b	r	r	r	r	1
`AND`	And	r	r	b	r	r	r	r	2
`ORR`	Or	r	r	b	r	r	r	r	2
`XOR`	Xor	r	r	b	r	r	r	r	2
`NOT`	Not	r	r	b	r	r	r	r	2
`JMP`	Jump	r	r	x	r	r	r	r	1
`BEQ`	Jump Equal	r	r	r	r	r	r	r	2
`BLE`	Jump Less	r	r	r	r	r	r	r	2
`BGR`	Jump Greater	r	r	r	r	r	r	r	2
`JSR`	Subroutine Jump	r	r	x	r	r	r	r	1
`RET`	Return Jump	x	x	x	x	x	x	x	0
`HLT`	HALT	x	x	x	x	x	x	x	0
`SEM`	Stack Empty	x	x	w	x	x	x	x	0
`POP`	Pop Stack	x	x	w	x	x	x	x	0

5.2 Assembler usage guide

In this part of the appendix a brief explanation of how the assembly lanuage works is given here.

5.2.1 Syntax

The syntax for the assembly code is pretty straight forward. Each declaration is written on a single line. There are a few reserved identifiers:

Indentifier	Name	Description
`%<text>`	Comment	Will be ignored by the assembler
`#<name>`	Label	Declares a label called `<name>`
`@<name>`	Value	Declares a value called `<name>`
`:<data>`	Raw input	Returns `<data>` as is
`$<a>`	Address	Derefferences `a`
`x`	x	The **X** register
`y`	y	The **Y** register
`s`	s	The stack **S**
`q1`	IRQ1	The $\mathbf{Q_1}$ register
`q2`	IRQ2	The $\mathbf{Q_2}$ register

The names given to *label*s and *value*s can contain any characters except for whitespace ones.
Operations are declared in a straightforward approach as:
`<opcode> <arg1> <arg2>`
Wich arguments are allowed are dependent upon the opcode.
Any non whitespace character can be used for names of labels and values.
Each file has to start with a label.

5.2.2 Usage

5.2.2.1 General

One noteworthy thing to point out is the limitations on the arguments. Due to limitations in the VM only one "none registry" argument can be used for any operation. A "non registry" argument is one wich is either a number or a a pointer. Derefferencing a pointer is a registry operations so they are valid. Below follows some examples:

```
#label
@value
MOV label value    This is not accepted
MOV $label $value  This is perfectly fine
MOV 10 value       This is invalid
MOV 10 $value      But this is
ADD 1 10           This is invalid
ADD $1 $10         This is valid
ADD 1 $1           So is this
```

5.2.2.2 Registers

The useage of the registers is pretty straight forward. One has to remeber that `q1` and `q2` are write only registers and that `s` cant be used for addressing so `$s` is not allowed and will generate a `syntax` error. It is also good to keep in mind that all operations reading from the stack will consume what is on top of the stack.

5.2.2.3 Pointers

Using pointers is fairly straight forward. Alltough one has to keep in mind how the addresses are resolved. All pointers will be resolved after the tokenazation fo the code. First the *labels* will be resolved and then the *values*. This means that the first *value* will lie after the last line of code. Since the address of a *label* depends on where in the code their addresses are easy to reason about. However for *values* things are bit different. Since vlues will be given addresses wich are "independent" of where in the code they appear it is hard to reason about the address of a *value*. Although the *value* pointers are resolved in order the first *value* declared whill lie intermediatley after the last line of code and the last *value* declared will lie "at the end" of the memmory used by the program. This can be exploited to use rellative addressing. Allthoug great care has to be taken.

Its important to remember that all pointers are reffered to troughout the entire program therefor it's not allowed to define two pointers with the same name. If this where to be allowed it would generate unpredictable behaviour so instead the assembler will return a `assembler` error.

*label*s and *value*s are interchangable. Since opcodes takes pointers as arguments and has no idea weather or not they are *label*s or *value*s. From this the need for caution arises. Since one can use *value* pointers as arguments to jump operation like this:

```
@bad_idea
ADD x y
MOV s x
MUL x y
JMP bad_idea
```

Since it is not known what where `bad_idea` points jumping to it is suicidal.

Since pointers are just numbers under the hood one needs to take into account weather or not one uses them for their adress orr for their *value*s. Here is some examples

```
@pointer
% This stores x in pointer
MOV x value
% This stroes x in the address wich is
% stored at pointer
MOV x $value
% This adds one to the value stored at
% pointer
ADD $pointer 1
% This adds one to the address of pointer
ADD pointer 1
```

Pointers are imutable and once they has been declared they can not be changed. One has to do some tricking to achive relative addressing using *labels* or *values*.

5.2.2.4 Labels

Labels are declared using the `#` identifier. *Labels* are resolved first and their addresses correspond to location in the code where they are written. For example:

```
MOV x y
#loop
INC x
MOV x s
JMP loop
```

In this code `loop` points to the address where `INC x` is stored. In the tokenization of the assembly code the lines where a pointer is defined will be ignored and the address where the next instruction or raw entry occurs. This can lead to that poorly written code becomes ambigous. For example:

```
MOV x y
#loop
#silly
INC y
```

Here `loop` and `silly` will both point to the same address wich is silly.

Because tokenaization of the code happens before the address resolving a *label* will be "in scope" troughout the enitre code. So this code is perfectly valid:

```
MOV x y
JMP ahead
INC x
ADD x y
#ahead
ADD s x
```

The `JMP ahead` will jump to `ADD s x` even though the `ahead` flag is defined after the jump. This was not a concious design choise but it is actually quite usefull since one can define subroutines anywhere in the code wich can be accesed form anywhere in the code.
One possible piffall arises due to the fact that the assembler does not know the difference between a *label* and a *value* after their adresses has been resolved. So this code is valid assembly code:

```
#loop
ADD s x
MOV s $loop
JMP loop
```

Allthough what this will do is that it will change what is at the address of `loop`. But there `ADD s x` lies! This is what is known as self-modifiying code and it's the spawn of satan and should be avoided like one avoids Miami beach during spring break. Allthough in some cases the interchangability of *value* and *label* can be verry usefull if one wants to have "arrays" in ones code. This is easily achived like this:

```
#array
:0
:1
:2
:3
```

Here `array` can be used as a pointer to the array. One can then manipulate the array trough using rellative arressing of `array` like this:

```
ADD 2 array
MOV s y
MOV 5 $y
#array
:0
:1
:2
:3
```

This code would change the 2 into a 5. But great care needs to be taken since one could easily end up outside of the "array" and corrupt the program.

5.2.2.5 Values

*Value*s are far more straight forward than *label*. One only has to take into account that what address a *value* is given is somewhat independet of where in the code it gets defined.

5.2.2.6 Jumping

Doing oridary jumps using the `JMP` operation is very straight forward. The machine will just jump to the address given to the `JMP` operator.

But for conditional branching things become a little bit less obvious. If the test given to a conditional test fails the machine will skip the next instruction. Letts illustrate this with a few examples:

```
MOV 10 x
BLE x 2
JMP this_does_not_happen
BGR x 2
JMP this_happens
```

Subroutine jumps work in a very straight forward fashion. You just make a subroutine call using `JSR <address>` and then you use the `RET` operations to return to the address imediatley after the one from which the jump was issued. One has to be carefull not to execute a `RET` jump unless one has actually made a subroutine jump. The VM will crash if a return jump is issued and the jump stack is empty.

5.2.2.7 Arithmetic and logic operations
The arithmetic and logic operations are quite straight forward. The arguments given to the operations appear as they would in the normal case. So `ADD x y` is x+y and `MOD x y` is x mod y. All of these operations (except for `INC` and `DEC`) store their result on the stack.

5.3 Componets.sml description

Due to misscomunication a radicaly diffrent description of how the Components.sml file works was written and have been included here as an appendix.

5.3.1 Introduction

The following structures and signatures are present in Components.sml are the Ram, Stack, Register and ProgramCounter.

5.3.2 The Ram structure

5.3.2.1 Synopsis
signature RAM
structure Ram :>RAM

The Ram structure provides a base of the functions of a ram memory. This structure acts as something akin to a “monad”. It hides all of the sidefects sued for the array handling.

5.3.2.2 INTERFACE
type memory = int array
val initialize : int $\rightarrow$ memory
val getSize : (memory) $\rightarrow$ int
val write :(memory * int * int) $\rightarrow$ memory
val read : (memory * int) $\rightarrow$ int
val load : (memory* int list) $\rightarrow$ memory
val writeChunk : (memory * int * (int array)) $\rightarrow$ memory
val readChunk : (memory * int * int) $\rightarrow$ int array
val dump : memory $\rightarrow$ string

5.3.2.3 Description
val initialize : int $\rightarrow$ memory
Initialize the ram to a memory with the size of int, when int ¿ 0
val getSize : (memory) $\rightarrow$ int
Gets the size of the memory
val write :(memory * int * int) $\rightarrow$ memory
write takes a memory and writes a new value of int at the pointer of the first int and returns the memory
val read: (memory * int) $\rightarrow$ memory

read takes a memory and reads the value of the place of int
val load: (memory * int list) → memory
load takes a list of values and loads them to the memory
val writeChunk: (memory* int *(int array)) → memory
writeChuck takes a memory and a start pointer and adds a chunk to the memory
val readChunk: (memory * int *int) → int array
readChumk takes a memory and reads a chunk form first int to the last int and gives the values as an int array
val dump: memory → string
dump takes a memory and returns the value as strings

5.3.3 The Stack structure

5.3.3.1 Synopsis

signature STACK
structure Stack :>STACK
The Stack structure provides a base for the stack part of the Pc structure.

5.3.3.2 INTERFACE

datatype stack = Stack of (int list)
val empty : stack
val push : stack * int → stack
val pop : stack → stack
val top : stack → int
val isEmpty : stack → bool
val dumpStack : stack → string

5.3.3.3 Description

val empty : stack
is a definition of a empty Stack
val push : stack * int → stack
takes a stack and adds the value of int to the stack.
val pop : stack → stack
takes a Stack and pops the first element of the stack.
val top : stack → int
takes the stack and returns the first element of the stack
val isEmpty : stack → bool
takes a stack and checks if it is empty if it is then true else false.
val dumpStack : stack → string
takes a stack, then pops the stack until its empty and returns all values as string

5.3.4 The Register structure

5.3.4.1 Synopsis

signature REGISTER

structure Register :>REGISTER

The Register structure provides a base structure of the different register that is contained in the Pc as well the Virtual machine. The vm has two different registers.

5.3.4.2 INTERFACE

datatype reg = Reg of int
val setData : (reg * int) → reg
val getData : reg → int
val increment : reg → reg
val decrement : reg → reg
val dumpRegister : reg → string

5.3.4.3 Description

val setData : (reg * int) → reg
Setups a new Register
val getData : reg → int
Gets the value of the reg as an int
val increment : reg → reg
Takes a reg and increment it with one.
val decrement : reg → reg
Takes a reg and decrements it with one.
val dumpRegister : reg → string
Takes the register and adds all elements to a string.

5.3.5 The Program Counter structure

5.3.5.1 Synopsis

signature PROGRAM_COUNTER
structure ProgramCounter :>PROGRAM_COUNTER

The ProgramCounter structure controls the execution flow of the VM

5.3.5.2 INTERFACE

datatype pc = Pc of (int * Stack.stack * Register.reg * Register.reg)
val incrementPointer : (pc * int) → pc
val jump : (pc * int) → pc
val subroutineJump : (pc * int) → pc
val return : pc → pc
val interrupt : (pc * int) → pc

val dumpPc : pc → string

5.3.5.3 Description

val incrementPointer : (pc * int) → pc
Takes a Pc and adds a int ¿ 0
val jump : (pc * int) → pc
Takes a Pc and jumps the pc counter to the value of int ¿ 0
val subroutineJump : (pc * int) → pc
Takes a Pc and preforms SubrutineJump with the value of int ¿ 0 and adds the value of the pointer + 1 to the stack
val return : pc → pc
Takes a pc and gets the value from the pointer and pops the stack with the value
val interrupt : (pc * int) → pc
if the value of a is 1 or 2, then the value of i is added to s
val dumpPc : pc → string
Takes a pc and dumps the content of the pc as a string (the Pc contained a pointer, Stack, and tow registers)