1 General

The VM features a very basic assembler capable of little more than adress resolution. But yet it gives us an ability to create some neat little programs.

2 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.

3 Usage

3.1 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
```

3.2 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.

3.3 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*.

3.3.1 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.

3.3.2 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.

3.4 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.

3.5 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.