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
`$`	Address	Treats `a` as a address
`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 labels and values 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.

3 Usage

3.1 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.

3.2 Pointers

Using pointers is fairly straight forward. Alltough one has to keep in mind how the addresses are resolved. All pointers will be resolved afterr 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.

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

```
... LOTS OF CODE ...

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

... LOTS OF CODE ...
```

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 values. 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.

3.2.1 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.