Specs for a lame VM implemented in SML

February 19, 2014

This VM is a very simple and minimalistic machine. It has an memory of arbitrary size which is specified at run-time. The memory works with signed integers of arbitrary size as words. The VM is asynchronous and each operation takes one step.

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.

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.

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.

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.

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.

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**.

1.6 RAM

The RAM in this machine works pretty much like any other random access memory.
The RAM is devided into three part. There are some special cells with negative addresses. These addresses are read only and contain information regarding the layout of the memory. Any write to these special locations will result in the VM crashing. The main part of the RAM is then sectioned in two two parts. One "ordinary" part and one external part. Prepherial components such as I/O handlers and such can only accsess the external memory.
The two special addresses are -1 where the size of the entire memory is stored and -2 stores the address at which the external memory start.

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		$++$	$--$	$+$	$-$	$\star$	$\div$	$\%$		
Logic		$=$	$<$	$>$	$<<$	$>>$	and	or	xor	not
Jump		J	$J_=$	$J_<$	$J_>$	J_{SR}	Ret			
Special		H	Se							

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.

2.1 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.

2.1.1 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** .

2.1.2 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.

2.1.3 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.

2.1.4 Jump operations

The standard address jump (J) will jump the **PC** to the address given by its read digit.
The conditional jumps takes the write digit as its first argument and the read digit as its second argument. If the condition is met the **PC** will be incremented by 1 (or 2 if **A** is used) otherwise the **PC** will be incremented by 2 (or 3 if **A** is used).
A J_{SR} (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** and then pop the stack.If the jump where to be empty the VM should crash and an exception should be raised.
For all jump operations the third and fourth digit must be 0 or the instruction is invalid and the VM should crash. The first and second digit will be ignored if they are not used.

2.1.5 Special

The two special operations exist. The Halt operation which just stops the VM and raises an exception. An the Se or Stack empty operation which returns 1 if the stack is empty and 0 else.

2.2 Opcodes

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	w	r	r	r	r	2
`SUB`	Subtract	r	r	w	r	r	r	r	2
`MUL`	Multiply	r	r	w	r	r	r	r	2
`DIV`	Division	r	r	w	r	r	r	r	2
`MOD`	Modulus	r	r	w	r	r	r	r	2
`EQL`	Equal	r	r	w	r	r	r	r	2
`LES`	Less	r	r	w	r	r	r	r	2
`GRT`	Greater	r	r	w	r	r	r	r	2
`BRL`	Rotate L	r	r	w	r	r	r	r	1
`BRR`	Rotate R	r	r	w	r	r	r	r	1
`AND`	And	r	r	w	r	r	r	r	2
`ORR`	Or	r	r	w	r	r	r	r	2
`XOR`	Xor	r	r	w	r	r	r	r	2
`NOT`	Not	r	r	w	r	r	r	r	2
`JMP`	Jump	r	r	x	r	r	r	r	1
`JEQ`	Jump Equal	r	r	r	r	r	r	r	2
`JLE`	Jump Less	r	r	r	r	r	r	r	2
`JGR`	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