

# The SAM Virtual Machine

Reuben Thomas

21st September 2025

## Typographical notes

VM instructions and registers are shown in `Typewriter` font; interface calls are shown in **Bold** type, and followed by empty parentheses.

SAM can be built as a library and embedded in other programs. A small interface is provided for other programs to control SAM.

Addresses are given in bytes and refer to SAM's address space except where stated. Addresses are written in hexadecimal; hex numbers are prefixed with "0x".

## 1 Introduction

SAM is a simple virtual machine designed playfully to teach some elements of low-level programming. It is programmed with an assembly language written as YAML.

SAM is self-contained, and performs I/O via the **TRAP** instruction, which provides access to I/O and possibly implementation-dependent facilities.

## 2 Architecture

SAM is a stack-based VM. Its memory is a single stack of items, each of which is a VM instruction or a stack. It has no other memory. It has a handful of registers.

### 2.1 Registers

The registers are set out in table 1.

All of the registers are word-sized.

### 2.2 The stack

The stack is represented as a series of VM instructions encoded as 4-byte words.

The stack is addressed with signed integers.

A positive address  $n$  refers to the  $n$ th word, starting from zero at the bottom of the stack.

Register	Function
SSIZE	The Stack SIZE. The size of the stack in words.
SP	The Stack Pointer. The number of words in the current stack.
IR	The Instruction Register holds the currently-executing instruction word.
OP	The OPerand is the operand encoded in the current instruction.
I	The opcode of the currently executing Instruction.
PC	The Program Counter points to the next instruction.
PC0	Points to the start of the currently executing stack.

Table 1: SAM's registers

A negative address  $-n$  refers to the  $n$ th stack item, counting from 1 at the top of the stack.

A valid address is one that points to a valid instruction word whose opcode's name does not start with an underscore.

## 2.3 Operation

Before SAM is started, the stack should be given suitable contents, SSIZE and SP set appropriately, and PC0 set to point to some code in the stack.

```

perform a KET instruction
begin
    load the stack item at PC into IR
    set I to the least significant byte of IR
    set OP to IR arithmetically shifted one byte to the right
    increment PC
    execute the instruction whose opcode is in I
repeat

```

Note that **run()** does not perform the initialisation specified above; that must be performed before calling it.

## 2.4 Termination and errors

When SAM encounters a HALT instruction (see section 3.11), it pops an integer from the top of the stack and returns it as its reason code. If an error occurs while popping the integer, that error code is returned instead.

Error codes are signed numbers. Negative codes are reserved for SAM; positive error codes may be used by user code.

Table 2 lists the errors and the conditions under which they are raised. Some further specific error conditions are documented with the instructions that raise them.

Code	Value	Meaning
OK	0	No error.
INVALID_OPCODE	−1	An attempt was made to execute an invalid opcode (see section 3.13).
INVALID_ADDRESS	−2	Invalid address.
STACK_UNDERFLOW	−3	The stack has underflowed, that is, an attempt was made to pop when it was empty.
STACK_OVERFLOW	−4	The stack has overflowed, that is, an attempt was made to push to it when it already contained SSIZE words, or an attempt was made to access beyond the current top of the stack.
NOT_NUMBER	−5	A stack item expected to be a number was not.
NOT_INT	−6	A stack item expected to be an integer was not.
NOT_FLOAT	−7	A stack item expected to be a float was not.
NOT_CODE	−8	An item expected to be code was not.
BAD_BRACKET	−9	No matching KET found for a BRA, or vice versa.
UNPAIRED_FLOAT	−10	A FLOAT instruction was not followed by _FLOAT.
UNPAIRED_PUSH	−11	A PUSH instruction was not followed by _PUSH.
INVALID_SWAP	−12	A SWAP instruction was given invalid arguments.
INVALID_FUNCTION	−13	An invalid function number was given to TRAP.

Table 2: Errors raised by SAM

### 3 Instruction set

The instruction set is listed in sections 3.2 to 3.12, with the instructions grouped according to function. The instructions are given in the following format:

NAME ( *before* -- *after* )  
Description.

The first line has the name of the instruction on the left and the stack comment, which shows the effect of the instruction on the stack, on the right. Underneath is the description.

**Stack comments** are written ( *before* -- *after* ) where *before* and *after* are stack pictures showing the items on top of a stack before and after the instruction is executed (the change is called the **stack effect**). An instruction only affects the items shown in its stack comments. The brackets and dashes serve merely to delimit the stack comment and to separate *before* from *after*. **Stack pictures** are a representation of the top-most items on the stack, and are written  $i_1\ i_2 \dots i_{n-1}\ i_n$  where the  $i_k$  are stack items, each of which occupies a whole number of words, with  $i_n$  being on top of the stack. The symbols denoting different types of stack item are shown in table 3.

### 3.1 Types and their representations

Symbol	Data type
$i$	a signed integer
$f$	a floating-point number
$n$	a number (integer or floating point)
$l$	a link (pointer to code)
$s$	a scalar (anything other than a stack)
$c$	a code item (stack or link)
$x$	an unspecified item

Table 3: Types used in stack comments

Each type may be suffixed by a number in stack pictures; if the same combination of type and suffix appears more than once in a stack comment, it refers to identical stack items.

Integers are stored as the top three bytes of an INT instruction, in twos-complement form.

Floats are 32-bit IEEE floats, whose top three bytes are stored in the top three bytes of a FLOAT instruction and bottom byte is in the second byte of the paired \_FLOAT instruction.

A stack is encoded as a BRA instruction followed by the nested stack items and ending with a KET instruction. A link to a stack is encoded in a LINK instruction as the address of its BRA instruction.

### 3.2 Do nothing

NOP ( -- )  
Do nothing.

### 3.3 Literals

These instructions encode literal values.

INT ( -- *i* )

Push 0P on to the stack.

FLOAT ( -- *f* )

Push the float encoded in the FLOAT and following \_FLOAT instruction on to the stack, or raise the error UNPAIRED\_FLOAT if the following instruction is not a \_FLOAT instruction. Increment PC.

\_FLOAT ( )

Raise the error UNPAIRED\_FLOAT. This instruction should never be executed.

### 3.4 Numeric type conversion

Numeric conversions:

I2F ( *i* -- *f* )

Raise the error NOT\_INT if the top-most stack item is not an integer. Otherwise, pop it, convert the integer to a float, and push the float.

F2I ( *f* -- *i* )

Raise the error NOT\_FLOAT if the top-most stack item is not a float. Otherwise, pop it, convert the float to an integer, and push the integer.

### 3.5 Stack manipulation

These instructions manage the stack.

PUSH ( -- *x* )

Push the word encoded in the PUSH and following \_PUSH instruction on to the stack, or raise the error UNPAIRED\_PUSH if the following instruction is not a \_PUSH instruction. Increment PC.

\_PUSH ( )

Raise the error UNPAIRED\_PUSH. This instruction should never be executed.

POP ( *x* -- )

If the stack is empty, raise the error STACK\_UNDERFLOW. Pop *x* from the stack.

DUP ( *i* -- *x* )

Pop *i* from the top of the stack. Push the *i*th stack item to the stack. If that item is a stack, push a LINK instruction pointing to it.

SWAP ( *x*<sub>1</sub> *i* -- *x*<sub>2</sub> )

Pop *i* from the stack. Swap the stack item *x*<sub>1</sub> with the *i*th item, *x*<sub>2</sub>.

IDUP (  $i_1$   $i_2$  --  $x$  )  
 Push the  $i_1$ th item of the  $i_2$ th stack element, which must be a stack or link, to the top of the stack.

ISSET (  $x$   $i_1$   $i_2$  -- )  
 Set the  $i_1$ th item of the  $i_2$ th stack element, which must be a stack or link, to  $x$ .

### 3.6 Control structures

These instructions implement loops, conditions and subroutine calls.

BRA ( --  $l$  )  
 Push  $PC - 1$  on to the stack, and add  $0P + 1$  to  $PC$ .

KET (  $l_1$   $l_2$  -- )  
 Pop  $l_2$  into  $PC$  and  $l_1$  into  $PC0$ .

LINK ( --  $x$  )  
 Push  $IR$  on to the stack.

DO (  $i$  --  $x_1$   $x_2$  )  
 Pop  $i$ . If the stack item at that index is not a stack, raise `NOT_CODE`. Push  $PC0$  then  $PC$  to the stack, and set both  $PC0$  and  $PC$  to  $i$ .

IF (  $i$   $c_1$   $c_2$  -- )  
 Pop  $c_1$  and  $c_2$ . If either stack item is not code, raise `NOT_CODE`. Pop  $i$ . If it is non-zero, perform the action of `DO` on  $c_1$ , otherwise on  $c_2$ .

WHILE (  $i$  -- )  
 Pop  $i$ . If it is zero, perform the action of `KET`.

LOOP ( )  
 Set  $PC$  to  $PC0$ .

### 3.7 Logic and shifts

These instructions consist of bitwise logical operators and bitwise shifts. The result of performing the specified operation on the argument or arguments is left on the stack.

Logic functions:

NOT (  $x_1$  --  $x_2$  )  
 Invert all bits of  $x_1$ , giving its logical inverse  $x_2$ .

AND (  $x_1$   $x_2$  --  $x_3$  )  
 $x_3$  is the bit-by-bit logical “and” of  $x_1$  with  $x_2$ .

OR (  $x_1$   $x_2$  --  $x_3$  )  
 $x_3$  is the bit-by-bit inclusive-or of  $x_1$  with  $x_2$ .

XOR (  $x_1$   $x_2$  --  $x_3$  )  
 $x_3$  is the bit-by-bit exclusive-or of  $x_1$  with  $x_2$ .

Shifts:

LSH (  $x_1$   $u$  --  $x_2$  )  
 Perform a logical left shift of  $u$  bit-places on  $x_1$ , giving  $x_2$ . Put zero into the least significant bits vacated by the shift.

RSH (  $x_1$   $u$  --  $x_2$  )  
 Perform a logical right shift of  $u$  bit-places on  $x_1$ , giving  $x_2$ . Put zero into the most significant bits vacated by the shift.

ARSH (  $x_1$   $u$  --  $x_2$  )  
 Perform an arithmetic right shift of  $u$  bit-places on  $x_1$ , giving  $x_2$ . Copy the most significant bit into the most significant bits vacated by the shift.

### 3.8 Comparison

EQ (  $s_1$   $s_2$  --  $i$  )  
 $i$  is 1 if  $n_1$  and  $n_2$  are equal, and 0 otherwise. Integers, floats and links are compared.

LT (  $n_1$   $n_2$  --  $i$  )  
 $i$  is 1 if  $n_1$  is less than  $n_2$  and 0 otherwise.

### 3.9 Arithmetic

These instructions consist of monadic and dyadic operators. All calculations are made without bounds or overflow checking, except as detailed for certain instructions.

The result of dividing by zero is zero. Integer division rounds the quotient towards zero; signed division of  $-2^{31}$  by  $-1$  gives a quotient of  $-2^{31}$  and a remainder of 0.

NEG (  $n_1$  --  $n_2$  )  
 Negate  $n_1$ , giving its arithmetic inverse  $n_2$ .

ADD (  $n_1$   $n_2$  --  $n_3$  )  
 Add  $n_2$  to  $n_1$ , giving the sum  $n_3$ .

MUL (  $n_1$   $n_2$  --  $n_3$  )  
 Multiply  $n_1$  by  $n_2$ , giving the product  $n_3$ .

DIV (  $n_1$   $n_2$  --  $n_3$  )  
 Divide  $n_1$  by  $n_2$ , giving the quotient  $n_3$ .

REM (  $n_1$   $n_2$  --  $n_3$  )  
 Divide  $n_1$  by  $n_2$ , giving the remainder  $n_3$ .

POW (  $n_1$   $n_2$  --  $n_3$  )  
 Raise  $n_1$  to the power  $n_2$ , giving the result  $n_3$ .

### 3.10 Trigonometry

Trigonometric functions:

SIN (  $f_1$  --  $f_2$  )

Calculate  $\sin f_1$ , giving the result  $f_2$ .

COS (  $f_1$  --  $f_2$  )

Calculate  $\cos f_1$ , giving the result  $f_2$ .

DEG (  $f_1$  --  $f_2$  )

Convert  $f_1$  radians to degrees, giving the result  $f_2$ .

RAD (  $f_1$  --  $f_2$  )

Convert  $f_1$  degrees to radians, giving the result  $f_2$ .

### 3.11 Errors

These instructions give access to SAM's error mechanisms.

HALT (  $x$  -- )

Stop SAM, returning reason code  $x$  to the calling program (see section 4.2).

### 3.12 External access

These instructions allow access to I/O and other system facilities.

TRAP (  $n$  -- )

Execute trap  $n$ . Further stack items may also be consumed and returned, depending on  $n$ . If the trap is invalid, raise `INVALID_FUNCTION`.

### 3.13 Opcodes

Table 4 lists the opcodes in numerical order. All undefined opcodes raise error `INVALID_OPCODE`.

## 4 External interface

SAM's external interface comes in three parts. The calling interface allows SAM to be controlled by other programs. The TRAP instruction allows implementations to provide access to system facilities, code written in other languages, and the speed of machine code in time-critical situations. The assembly format allows compiled code to be saved, reloaded and shared between systems.

### 4.1 Assembly format

TODO.



Opcode	Instruction	Opcode	Instruction
0x00	NOP	0x15	AND
0x01	INT	0x16	OR
0x02	FLOAT	0x17	XOR
0x03	_FLOAT	0x18	LSH
0x04	I2F	0x19	RSH
0x05	F2I	0x1a	ARSH
0x06	PUSH	0x1b	EQ
0x07	_PUSH	0x1c	LT
0x08	POP	0x1d	NEG
0x09	DUP	0x1e	ADD
0x0a	SWAP	0x1f	MUL
0x0b	IDUP	0x20	DIV
0x0c	ISSET	0x21	REM
0x0d	BRA	0x22	POW
0x0e	KET	0x23	SIN
0x0f	LINK	0x24	COS
0x10	DO	0x25	DEG
0x11	IF	0x26	RAD
0x12	WHILE		
0x13	LOOP	0xfe	HALT
0x14	NOT	0xff	TRAP

Table 4: SAM's opcodes

## 4.2 Calling interface

See `sam.h`.