SaM v2.2 Design Documentation

Ivan Gyurdiev
David Levitan

11/14/2003

1 Introduction

1.1 What is SAM?

SAM is the codename for a Java-based computer emulator. It is an acronym for StAck Machine. SAM is a virtual machine which executes programs composed of SAM assembly instructions. It is a tool, which allows students to learn how computers operate, and to write a compiler/translator to the simplified SAM assembly, testing their code using the virtual machine.

1.2 What is SAM 2?

SAM 2 is a complete rewrite of the original stack machine. Its main objective is to restructure the SAM code, and divide it into components that resemble real computer hardware and software more closely. The new SAM code also enhances the instruction set with numerous additions, such as bitwise logic, floating point instructions, and string instructions. It provides a typed stack, which supports Integer, Floating Point Number, Character, Program Address, and Memory Address types. It implements a stop-and-copy garbage collector for the heap. SAM 2 provides better error handling using exceptions. Finally, it provides new more powerful front-ends.

1.3 What are the major SAM Components?

SAM is divided into three packages: user interface (ui), core, and input-output (io). The user interface package contains the SAM front ends, which are used to execute SAM

assembly programs. The io package contains a tokenizer, used to properly parse such programs. The core package contains components that emulate real-world hardware and software.

1.4 Program Execution

SAM Programs are executed according to the following order:

1. A front-end is invoked.
2. The front-end invokes the SamAssembler.
3. The assembler uses the SamTokenizer to examine the program's code, and generate a Program java object, which consists of a sequence of Instruction objects.
4. The assembler returns this Program object to the front-end.
5. The front-end passes the Program object to the SamProcessor and begins execution.

2 Hardware Components

2.1 The Processor

(CORE/PROCESSOR.JAVA, CORE/SAMPROCESSOR.JAVA)

The SAM processor is responsible for the loading of a Program object, and the execution of Instruction objects, enclosed within that Program object. Like a real world processor, it also provides registers, which assist program execution.

Program Execution To execute a program, it must first be loaded by the processor. The following methods allow this to happen:

- void load(Program prog)
- Program getProgram()

Following the successful loading of a program, it may be executed one instruction at a time or all instructions with one call:

- void step() - Step executes one instruction.
- void run() - Run executes all instructions, until the HALT register becomes nonzero

Registers The SAM Processor implements several registers that allow the Processor to execute programs. These are not used for data storage, but rather are internal processor registers.

- HALT - Execution Status
  Allowed Value: 0 (running) or 1 (stopped)
  Start Value: 0
  It is used to stop the execution of a program.
- PC - Program Counter
  Allowed Value: 0 <= PC < Program Length
  Start Value: 0
  Used to track the instruction that will be executed next.
- SP - Stack Pointer
  Allowed Value: 0 <= SP < Stack Limit
  Start Value: 0
  Used to track the first free memory location on the stack.
- HP - Heap Pointer
  Allowed Value: Stack Limit <= HP < Memory Limit
  Start Value: Memory.STACKLIMIT
  Used to track the first free memory location on the heap.
- FBR - Frame Based Register
  Allowed Value: 0 <= FBR < Memory Limit
  Start Value: 0
  Used for relative addressing when constructing frames on the stack

The registers may be manipulated using the following methods. However, changing the register values may affect program execution.

- int get(byte reg)
- int[] getRegisters()
- void set(byte reg, int value)
- int inc(byte reg)
- int dec(byte reg)
- void reset()

2.2 Memory

(CORE/MEMORY.JAVA, CORE/SAMMEMORY.JAVA)

Memory is responsible for data storage. It is capable of storing integer-compatible data types (<=32 bits). Its size is defined as Memory.MEMORYLIMIT, and is presently 10000. It is implemented in SamMemory as an array of integers. Internally, type and data information alternate locations, but this is transparent to the end user, who sees 10000 locations and type and data information associated with each location.

Types The following data types are supported:

- Integer (INT)
  When an integer value is requested as an integer, a standard Java integer containing the value should be returned.
- Floating Point (FLOAT)
  When a floating point value is requested as an integer, an IEEE 754 representation of the floating point number should be returned.
- Character (CH)
  When a character is requested as an integer, the ASCII value of the character should be returned.
- Memory Address (MA)
  When a memory address is requested as an integer, its location in memory should be returned as an integer.

- Program Address (PA)
  When a program address is requested as an integer, the location should be returned as an integer.

Low-Level Access It is possible to access the memory directly by using one of the following methods that are part of the Memory interface:

- int getMem(int pos)
- byte getType(int pos)
- void setMem(int pos, int value)
- void setMem(int pos, byte type, int value)
- int[] getMemory()
- byte[] getTypes()
- void reset()

Stack Zone Memory is divided into stack space, and heap space. Currently the stack boundary is defined as Memory.STACKLIMIT=1000. Memory can be manipulated using the higher-level functions for stack management:

- void push(int value)
- void push(byte type, int value)
- int pop()

push() without a type uses INT by default. The SAM stack grows upwards from 0.

Heap Zone The SAM heap grows upwards. The malloc(int) method allocates the requested amount of memory on the heap, pushes the current HP on the stack, and updates the HP to the first free location on the heap.

Note that malloc() allocates size+1 locations in memory, and stores the requested size in the first location. We define this to be the format of an object - one location used for the object signature at offset 0 from the HP (used to store size of the object), followed by size locations for data. Students should use offsets 1 through size from the stored HP for data.

Garbage Collection The SamMemory implementation of the Memory interface also includes a stop-and-copy garbage collector, as described in the CS312 Garbage Collection Notes `http://www.cs.cornell.edu/courses/cs312/2003sp/lectures/rec20.html`.

The garbage collector divides the heap into two zones of equal size, only one of which is active at any time. When the active zone's free space is exhausted past Memory.GTHRESHOLD (defined as 10% of the heap zone space left), the garbage collector is triggered. The collector is also triggered if there is no sufficient space to allocate a large object. When triggered, the garbage collector traces all live objects, and copies them to the inactive zone, using breadth-first traversal. It updates any pointers to and within those objects. Anything left in the active zone after this transfer is considered garbage. The active zone is switched, and allocation continues in the new zone. If there is insufficient space following a garbage collection, malloc() fails with a SysException.

2.3 Video Card

(CORE/VIDEO.JAVA)

The video card is represented by an interface, which front ends can implement and provide extra functionality for the SAM instructions. This hardware component is strictly optional, and instructions should have an alternate solution for systems without a video card (front ends that do not implement the interface).

The Video interface consists of several functions that allow both reading and writing strings, ints, and floats.

2.4 System Chipset

(CORE/SYS.JAVA)

This component provides a unified way to access the system devices in SAM. It would be the equivalent of a computer chipset. The three components currently accessible from this class are the Processor, the Memory, and the Video card.

The Sys class was originally designed to be static, but it was later redesigned as a non-static class in order to allow components to work in parallel (multiple systems). This works by sending a Sys object as a parameter to other components, which need it. The Processor, Memory, and GUI constructors take a Sys argument. The Processor and

Memory are actually constructed when a Sys is constructed, since every system will have memory and processor. The SAM instructions use a method(void setSystem(Sys sys)) to get access to the system at execution time - they don't need it prior to that time.

To use this class, simply create a new Sys object. This will initialize the processor and memory. If you want to set the video component, use setVideo(). Then pass the Sys instance as an argument to any classes that will be working with the same system. The respective components can be accessed using the cpu(), mem(), and video() methods, which return objects of the respective types.

3 Internal Simulator Classes

SAM also contains several classes that handle program execution and other necessary functions. While these are not found in hardware, these are critical to SAM being an extensible and flexible language.

3.1 SAM Instructions

(CORE/INSTRUCTION.JAVA, CORE/SAMINSTRUCTION.JAVA)

A SAM Instruction is a class, implementing the Instruction interface. The exec() method manipulates the hardware components of the SAM system and must be overwritten by all instructions. All instructions are expected to manually change the PC register, either incrementing it in the case of most instructions, or changing its value to a new value for a jump instruction. See section 5 on page 11 for the SAM Instruction Set Manual.

Every instruction is represented as its own class, all of which extend SamInstruction or one of its subclasses. An instruction that extends SamInstruction is one that does not have any operands. Instructions that need an operand extend one of the subclasses of SamInstruction - SamIntInstruction, SamFloatInstruction, SamCharInstruction, etc... These subclasses provide the *op* variable in the appropriate type which is set to the operand provided to the instruction.

Instructions do not check the types of any input values they use. In the default SamProcessor implementation, floats, for example, are stored as integers using standard Java float->integer conversion routines. An integer operation that is given a float as input will produce unexpected results.

Apart from the *op* variable that is provided to operand classes, instructions also have access to the *cpu* (the processor), *mem* (the memory), *video* (the video interface), and *symTable* (the symbol table for the current program). All these variables are of the more generic types (Processor, Memory, Video, and SymbolTable, respectively), but are not guaranteed to be of the types that are included with this package.

Creating a new Instruction To create a new instruction:

1. Decide whether your instruction needs an operand, and select the approriate class to extend.
2. Create a new class definition that inherits the class you have selected.
3. Override the exec() method

Things to remember:

- Remember that the PC must be set manually - typically it must just be incremented by 1.
- Make sure you use the correct superclass.

3.2 SAM Symbol Table

(CORE/SYMBOLTABLE.JAVA, CORE/SAMSYMBOLTABLE.JAVA)

A SAM program supports the use of labels, defined as a string ending with ':', such as "ThisIsALabel:" Jump instructions can then take the name of such a label, and jump to that address in the program. The SAM Symbol Table is responsible for mapping labels (symbols) to addresses, and vice versa. It is implemented by using two Hash tables, enabling a search using a symbol, or a search using an address.

To support multiple labels per address, one of the hashtables was implemented with vectors. Unfortunately, as of SAM 2.2, the GUI does not handle multiple stacked labels quite properly, and simplifies them down to one. This is nontrivial to fix, but will be corrected in a future release.

3.3 SAM Program

(CORE/PROGRAM.JAVA, CORE/SAMPROGRAM.JAVA)

A SAM Program is an enclosing container for Instructions, and a SymbolTable. The assembler generates a Program object. The processor executes each instruction of the program object, using the PC register as a numeric index.

4 SAM Front Ends

There are several classes that are not used directly by the SAM System components, yet are required to load and execute programs.

4.1 SAM Assembler

(CORE/SAMASSEMBLER.JAVA)

The SAM assembler is the equivalent of a real-world assembler, which translates assembly code into binary. It can be invoked with a filename argument, a Reader argument, or a Tokenizer argument. The assembler uses the tokenizer to read the program tokens one-by-one and create Instruction objects using Java reflection for obtaining class names. It reads the operands for each instruction, based on the abstract class it extends (String for SamStringInstruction, character for SamCharInstruction, etc..). The assembler resolves all labels to integers, and constructs a SymbolTable. It combines all this information inside a Program object.

4.2 SAM User Interfaces

There are two SAM Interfaces that are included in SAM 2.2. The primary one is a Swing-based GUI that allows program execution and debugging. A text-only interface is also provided that allows fast execution of SAM programs from the command line.

4.2.1 SamGUI - Graphical UI

SamGUI is a feature-rich graphical front-end, which displays the stack contents, the program, and the registers at every step. It allows a user to start and stop execution at

will, or step through the program. It supports a capture feature, which saves the memory and register contents at every step, creating a series of snapshots of the execution process. The capture feature has its own GUI component, which allows the saving and loading of capture output.

Usage java ui.SamGUI [<filename> | -capture [<filename>]]

SamGUI can be called either with the filename of a SAM program (which will then be loaded and ready for exectuion, or invoked in capture mode (also with an optional filename of a SAM capture).

4.2.2 SamText - Text UI

SamText is a small console front-end designed to execute a program, and report its return value. This front-end is great for quick testing, or grading of student programs. This front-end also allows a user to type a program at the console and execute it without saving. It also allows piping a program in as the input.

Usage java ui.SamText <filename> <options>

If the options are omitted, the program runs without limits. If the filename is omitted, System.in is used for input.

Options +tl <integer>: Time limit in miliseconds

+il <integer>: Instruction limit

4.3 Execution Convenience Classes

(UI/RUNTHREAD.JAVA, UI/THREADEDFRONTEND.JAVA)

To simplify the creation of SAM Simulator user interfaces, the RunThread class exists to easily allow a front end to run a program without losing control of the executing program. It allows the front end to stop the program at any point, and also calls a function to update the status after every instruction is executed. This class uses the same access methods that are available to any other program, but exit for convenience reasons. All front ends that use the RunThread class must implement ThreadedFrontend.

5 SAM Instruction Set Architecture Manual

Instruction Set Architecture Manual Format Each instruction contains the input values to the instruction, the output values for the instruction, any operand for this instruction, and a description of what the instruction does. The input values are ordered from top to bottom. The leftmost (first) input value is the one at the top of the stack, while each succeeding input value is one below the previous. The rightmost (last) output value is the value place at the top of the stack, and the leftmost output value is the bottom value placed on the stack by the instruction. The version of SaM the instruction first appeared in is also included.

Types Thoughout the manual, many input/output values are specified with types. Please note that no instruction requires a specific input type. However, instructions do not automatically convert values and treat all values as the input type required. Depending on the implementation, different types may be stored in incompatible types.

Instruction Order Execution All instructions change the PC value. Most instructions will simply increase the PC value by 1. However, jumps may change this to a different value.

5.1 Type Converters

Type conversion instructions convert a value from one type to another. Since there is no guarantee regarding how a particular type is stored in the stack, these instructions must be used to conver a type before executing a type specific instruction on a particular stack value.

5.1.1 FTOI

Input Values Float

Output Values Integer

Operand None

Since 2.0

Description Converts a float to an integer by truncating any decimal portion.

5.1.2 FTOIR

Input Values Float

Output Values Integer

Operand None

Since 2.0

Description Converts a float to an integer by rounding based on the decimal portion.

5.1.3 ITOF

Input Values Integer

Output Values Float

Operand None

Since 2.0

Description Converts an integer to a float.

5.2 Stack Insertion

These instructions allow new values to be pushed onto the stack.

5.2.1 PUSHIMM

Input Values None

Output Values Integer

Operand Integer

Since 1.0

Description Places the integer operand onto the stack.

5.2.2 PUSHIMMF

Input Values None

Output Values Float

Operand Float

Since 2.0

Description Places the float operand onto the stack.

5.2.3 PUSHIMMCH

Input Values None

Output Values Character

Operand Character

Since 2.0

Description Places the character operand onto the stack.

5.2.4 PUSHIMMSTR

Input Values None

Output Values Memory Address

Operand String

Since 2.0

Description Allocates space for the string on the heap, stores the sequence of characters on the heap (with type CH) starting with the first letter as the lowest heap location. Note that the string is stored as an object - offset 0 is the string size, offsets 1-size hold the string's characters. The instruction finally pushes the address of the string's beginning on the stack. The HP is updated as necessary by mem.malloc().

5.3 SP/FBR Manipulation

These instructions manipulate the SP and FBR registers.

5.3.1 PUSHSP

Input Values None

Output Values Memory Address

Operand None

Since 1.0

Description Pushes the value of the SP register onto the stack.

5.3.2 PUSHFBR

Input Values None

Output Values Memory Address

Operand None

Since 1.0

Description Pushes the value of the FBR register onto the stack.

5.3.3 POPSP

Input Values Memory Address

Output Values None

Operand None

Since 1.0

Description Sets the SP register to the value at the top of the stack.

5.3.4 POPFBR

Input Values Memory Address

Output Values None

Operand None

Since 1.0

Description Sets the FBR register to the value at the top of the stack. This is often used to undo LINK.

5.4 Stack Manipulation

5.4.1 DUP

Input Values Any type

Output Values Two of the input type

Operand None

Since 1.0

Description Duplicates the value at the top of the stack preserving the type.

5.4.2 SWAP

Input Values Two values

Output Values The reverse of the two values

Operand None

Since 1.0

Description Switches the places of the first two values on the stack. Type information is preserved for each value.

5.5 Stack/Heap Allocation

These instructions allow space to be allocated for data on the heap/stack.

5.5.1 ADDSP

Input Values Memory Address

Output Values None

Operand None

Since 1.0

Description Increments the SP register by the provided value. This essentially creates new values at the top of the stack of type integer with value 0.

5.5.2 MALLOC

Input Values Integer

Output Values Memory Address

Operand None

Since 1.0 (Modified in 2.0)

Description This instruction allocates space in the heap of size provided by the input value. It writes the address of the allocated space to the stack. Note that the space allocated is allocated as a new object and the object size is written in offset 0. Offsets 1 - size become available for storage. The HP is also updated as appropriate.

5.6 Absolute Store/Retrieve

These instructions provide access to absolute addresses. They should generally be used for heap access, or for access to known, fixed, locations (for example, static variables).

5.6.1 PUSHIND

Input Values Memory Address

Output Values Value

Operand None

Since 1.0

Description Pushes the data at that memory address onto the stack, preserving its type.

5.6.2 STOREIND

Input Values Value, Memory Address

Output Values None

Operand None

Since 1.0

Description Sets the address provided by the second input value to the value/type of the first input value.

5.7 Relative Store/Retrieve

These instructions provide access to memory addresses relative to the current frame. They should generally be used for stack access.

5.7.1 PUSHOFF

Input Values None

Output Values Value

Operand Integer

Since 1.0

Description Pushes the data at the memory address of the FBR+operand onto the stack.

5.7.2 STOREOFF

Input Values Value

Output Values None

Operand Integer

Since 1.0

Description Sets the address provided by the FBR+operand to the value/type of the input value.

5.8 Simple Integer Algebra

These instructions perform simple algebra on the stack. Please remember that all instructions operate on any data type, and mixing data types, such as floats and integers in simple algebra instructions is strongly discouraged due to the internal data handling.

5.8.1 ADD

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Adds the first input value to the second input value and places the result on the stack.

5.8.2 SUB

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Subtracts the first input value from the second input value and places the result on the stack.

5.8.3 TIMES

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Multiplies the first input value by the second input value and places the result on the stack.

5.8.4 DIV

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Divides the first input value into the second input value and places the result on the stack. If the result is not an integer, it is truncated and then placed on the stack as an integer.

5.8.5 MOD

Input Values Integer, Integer

Output Values Integer

Operand None

Since 2.0

Description Divides the first input value into the second input value and places the remainder on the stack.

5.9 Simple Floating Point Algebra

These instructions perform simple algebra on the stack. Please remember that all instructions operate on any data type, and mixing data types, such as Floats and Integers in simple algebra instructions is strongly discouraged due to the internal data handling.

5.9.1 ADDF

Input Values Float, Float

Output Values Float

Operand None

Since 2.0

Description Adds the first input value to the second input value and places the result on the stack.

5.9.2 SUBF

Input Values Float, Float

Output Values Float

Operand None

Since 2.0

Description Subtracts the first input value from the second input value and places the result on the stack.

5.9.3 TIMESF

Input Values Float, Float

Output Values Float

Operand None

Since 2.0

Description Multiplies the first input value by the second input value and places the result on the stack.

5.9.4 DIVF

Input Values Float, Float

Output Values Float

Operand None

Since 2.0

Description Divides the first input value into the second input value and places the result on the stack.

5.10 Shifts

These instructions perform signed bitwise shifting. Shifting essentially moves all the bits in the shifted value over by the specified amount in a certain direction. With signed bitwise shifting, the sign of the value is preserved when shifting to the right. Shifting left also has the effect of effectively multiplying the value by 2, while shifting right effectively divides the value by 2.

5.10.1 LSHIFT

Input Values Integer

Output Values Integer

Operand Integer

Since 2.0

Description Shifts the input values to the left by the number of places specified by the operand.

5.10.2 RSHIFT

Input Values Integer

Output Values Integer

Operand Integer

Since 2.0

Description Shifts the input values to the right by the number of places specified by the operand. The sign of the value is preserved (so a negative number will have ones added to the left, while a positive number will have zeroes add to the left).

5.11 Logic

These instructions perform logical operations on input values. They essentially treat all non-negative numbers as 1. See section 5.12 on page 27.

5.11.1 AND

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description If both values are non-zero, pushes 1 onto the stack. Otherwise, pushes 0.

5.11.2 OR

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Performs an inclusive or. If either value is non-zero, pushes 1 onto the stack. Otherwise, pushes 0.

5.11.3 NOR

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description If either value is non-zero, pushes 0 onto the stack. Otherwise, pushes 1. Essentially equivalent to an OR followed by a NOT.

5.11.4 NAND

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description If both values are non-zero, pushes 0 onto the stack. Otherwise, pushes 1. Essentially equivalent to an AND followed by a NOT.

5.11.5 XOR

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Performs an exclusive or. If only one of the two values is non-zero, pushes 1 onto the stack. Otherwise, pushes 0.

5.11.6 NOT

Input Values Integer

Output Values Integer

Operand None

Since 1.0

Description If the value is non-zero, pushes 0 onto the stack. Otherwise, pushes 1.

5.12 Bitwise Logic

These are logic operations that are performed on a bitwise level. Each individual bit is compared and the operation is performed. For example, the binary value 110 (decimal value of 6) BITAND'd with 101 (decimal value of 5) produces an output of 100 (decimal value of 4).

5.12.1 BITAND

Input Values Integer, Integer

Output Values Integer

Operand None

Since 2.0

Description Performs a bitwise AND operation on the two integers. For each bit, if both bits are 1, the resulting bit is a 1. Otherwise, it is a zero.

5.12.2 BITOR

Input Values Integer, Integer

Output Values Integer

Operand None

Since 2.0

Description Performs a bitwise inclusive OR operation on the two integers. For each output bit, if either input bit is 1, the resulting bit is a 1. Otherwise, it is a 0.

5.12.3 BITNOR

Input Values Integer, Integer

Output Values Integer

Operand None

Since 2.0

Description Performs a bitwise NOR operation. Essentially equivalent to a BITOR followed by a BITNOT.

5.12.4 BITNAND

Input Values Integer, Integer

Output Values Integer

Operand None

Since 2.0

Description If both values are non-zero, pushes 0 onto the stack. Otherwise, pushes 1. Essentially equivalent to a BITAND followed by a BITNOT.

5.12.5 BITXOR

Input Values Integer, Integer

Output Values Integer

Operand None

Since 2.0

Description Performs an bitwise exclusive or. For each bit in the output value, if only of the input bits is 1, the resulting bit is a 1. Otherwise, it is a 0.

5.12.6 BITNOT

Input Values Integer

Output Values Integer

Operand None

Since 2.0

Description For each bit in the output value, if the input bit is a 0, the output bit is set to a 1. Otherwise it is set to a 0.

5.13 Comparison

These instructions allow two values to be compared.

5.13.1 CMP

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Compares the two input values. If the first input value is bigger than the second input value, a 1 is placed on the stack. If they are equal, a 0 is placed on the stack. If the first input value is smaller than the second input value, a -1 is placed on the stack.

5.13.2 CMPF

Input Values Float, Float

Output Values Integer

Operand None

Since 2.2.4

Description Compares the two input values. If the first input value is bigger than the second input value, a 1 is placed on the stack. If they are equal, a 0 is placed on the stack. If the first input value is smaller than the second input value, a -1 is placed on the stack.

5.13.3 GREATER

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Compares the two input values. If the second input value is bigger than the first input value, a 1 is placed on the stack. Otherwise, a 0 is placed on the stack.

5.13.4 LESS

Input Values Integer, Integer

Output Values Integer

Operand None

Since 1.0

Description Compares the two input values. If the second input value is smaller than the first input value, a 1 is placed on the stack. Otherwise, a 0 is placed on the stack.

5.13.5 EQUAL

Input Values Value, Value

Output Values Integer

Operand None

Since 1.0

Description Compares the two input values. If the second input value is equal than the first input value, a 1 is placed on the stack. Otherwise, a 0 is placed on the stack.

5.13.6 ISNIL

Input Values Value

Output Values Integer

Operand None

Since 1.0

Description If the input value is 0, a 1 is placed on the stack. Otherwise a 0 is place on the stack. This is equivalent to the NOT instruction.

5.13.7 ISPOS

Input Values Integer

Output Values Integer

Operand None

Since 1.0

Description If the input value is greater than 0, a 1 is placed on the stack. Otherwise a 0 is place on the stack.

5.13.8 ISNEG

Input Values Integer

Output Values Integer

Operand None

Since 1.0

Description If the input value is less than 0, a 1 is placed on the stack. Otherwise a 0 is placed on the stack.

5.14 Jumps

Jumps are special instructions used for moving to other pieces of code. They are useful for such things as loops and, especially, functions. Jumps mostly use labels for their operands, which are then translated to the correct instruction at assemble-time.

5.14.1 JUMP

Input Values None

Output Values None

Operand Label

Since 1.0

Description Sets the PC to the instruction specified by the label and continues execution starting with that instruction.

5.14.2 JUMPC

Input Values Integer

Output Values None

Operand Label

Since 1.0

Description If the input value is non-zero, the PC is set to the instruction specified by the label and execution is continued starting with that instruction. Otherwise, the execution is continued as normal with the instruction following the JUMPC.

5.14.3 JUMPIND

Input Values Program Address

Output Values None

Operand None

Since 1.0

Description Sets the PC to the input value and continues execution with that instruction. This if often used to undo a JSR.

5.14.4 JSR

Input Values None

Output Values Program Address

Operand Label

Since 1.0

Description Sets the PC to the instruction found at the label, pushes the current PC + 1 onto the stack, and continues execution at the next instruction. This is usually used with LINK for subroutines.

5.14.5 JSRIND

Input Values Program Address

Output Values Program Address

Operand None

Since 1.0

Description Sets the PC to the input value, pushes the current PC + 1 onto the stack, and continues execution at the next instruction. This is usually used with LINK for subroutines.

5.14.6 BRANCH

Input Values Program Address

Output Values None

Operand None

Since 2.0

Description Sets the PC to the input value + current PC + 1. The effect is that if the popped value is 0, execution continues as normal. If the popped value is -1 the PC stays the same, and -2 and below will move the PC back by that value minus one.

5.15 Stack Frames

Stack frames are used for relative addressing and are usually defined for every subroutine. These are two different stack formats.

5.15.1 LINK

Input Values None

Output Values None

Operand None

Since 1.0

Description Pushes the FBR register on the stack with type MA and sets the FBR register to the SP register - 1. This is often undone with POPFBR.

5.15.2 FSET

Input Values None

Output Values None

Operand Integer

Since 2.2

Description Sets the FBR register to the SP register minus the operand plus one.

5.16 Input/Output

The I/O instructions are a special set of instructions that allow the SAM Program to interact with the outside world. These are not guaranteed to be implemented in all implementations, as they require a Video interface to be specified for the particular system being used. The implementations of these instructions will differ depending on the simulator used. If no Video interface is available, zero or an empty string are placed onto the stack.

5.16.1 READ

Input Values None

Output Values Integer

Operand None

Since 1.0

Description Asks the Video interface for an integer (using the readInt() method) and push it onto the stack. If there is no Video defined, pushes 0 onto the stack.

5.16.2 READF

Input Values None

Output Values Float

Operand None

Since 2.0

Description Asks the Video interface for an float (using the readFloat() method) and push it onto the stack. If there is no Video defined, pushes 0 onto the stack.

5.16.3 READCH

Input Values None

Output Values Character

Operand None

Since 2.2

Description Asks the Video interface for a character (using the readChar() method) and push it onto the stack. If there is no Video defined, pushes null (ASCII code 0) onto the stack.

5.16.4 READSTR

Input Values None

Output Values Memory Address

Operand None

Since 2.0

Description Asks the Video interface for a string (using the readString() method) and allocates for space for the string on the heap. The string is stored as a sequence of characters starting with the first character at the lowest available heap location. The memory address of the string is pushed onto the stack. Note that offset of 0 from that address is reserved for an object signature. The HP is updated as appropriate by mem.malloc(). If there is no video card, the instruction performs the operation above on an empty string. That results in an object signature written (with size 0), and the address being pushed on the stack.

5.16.5 WRITE

Input Values Integer

Output Values None

Operand None

Since 2.0

Description Writes the integer to the Video interface using the writeInt() method. If there is no video interface, there is no change in the result except that the integer will not be sent to any Video interface.

5.16.6 WRITEF

Input Values Float

Output Values None

Operand None

Since 2.0

Description Writes the float to the Video interface using the writeFloat() method. If there is no Video interface, there is no change in the result except that the float will not be sent to any Video interface.

5.16.7 WRITECH

Input Values Character

Output Values None

Operand None

Since 2.2

Description Writes the character to the Video interface using the writeChar() method. If there is no Video interface, there is no change in the result except that the character will not be sent to any Video interface.

5.16.8 WRITESTR

Input Values Memory Address

Output Values None

Operand None

Since 2.0

Description Writes the string at the memory address provided (the string must be stored in the standard format). If there is no Video interface, there is no change in the result except that the string will not be sent to any Video interface. Note that the string is not automatically removed by this instruction.

5.17 Program Control

5.17.1 STOP

Input Values None

Output Values None

Operand None

Since 1.0

Description Sets the HALT register to 1, effectively stopping program execution.