SaM v2.6 Design Documentation

Ivan Gyurdiev
David Levitan

9/5/2005

Contents

Chapter 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. 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 four packages: user interface (*ui*), core, utilities (*utils*), 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. The utililities package contains common pieces of code used that should be reused. There are also some subpackages.

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.

Chapter 2

Components

2.1 Hardware Components

2.1.1 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 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 init() - initialize the state of the processor
- void load(Program prog) - load a Program object into the processor
- Program getProgram() - obtain the current Program object

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.
- FBR - Frame Based Register
  Allowed Value: 0 <= FBR < Memory Limit
  Start Value: 0
  Used for relative addressing when constructing frames on the stack

Registers can be manipulated using the following methods:

- void set(int register, int value) - set a register to a given value
- int inc (int register) - increment a register by one, and return the result
- int dec (int register) - decrement a register by one, and return the result
- int[] getRegisters() - obtain the register array

2.1.2 Memory

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

Memory is responsible for data storage. It is capable of storing data of size Memory.UNIT_SIZE bits in each of Memory.MEMORYLIMIT locations. It is implemented in SamMemory as an array of integers. SaM memory is typed - it supports associating a data type with each memory location. This functionality can be used for error checking, presentation clarity, or other purposes. Internally, type and data information alternate locations, but this is not visible to the end user. The API provides methods for setting and getting data and type separately, or simultaneously, using the Memory.Data wrapper object. SaM memory is divided into two zones - stack and heap. The boundary between them is fixed at Memory.STACKLIMIT (the last stack 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.

General Access

All memory locations can be accessed using the following methods:

- Data getMem (int pos) - retrieve the Data object at this location
- int getValue (int pos) - retrieve the value at this location
- Type getType (int pos) - retrieve the type at this location
- void setMem (int pos, Memory.Data data) - store a Data object at this location
- void setMem (int pos, int data, Memory.Type type) - store a (type, value) pair at this location
- void setValue (int pos, int data) - set the value at this location
- void setType (int pos, Memory.Type type) - set the type at this location

Stack Zone

The stack can be manipulated specifically using the following functions:

- void push (Memory.Data data) - pushes a Data object on the stack
- void push (int value, Memory.Type type) - pushes the value and type separately
- void pushINT (int i) - pushes value with type integer
- void pushCH (char ch) - pushes value with type character
- void pushMA (int ma) - pushes value with type memory address
- void pushPA (int pa) - pushes value with type program address
- void pushFLOAT (float fl) - pushes value with type floating point
- Memory.Data pop () - pops a Data object off the stack
- int popValue () - pops a value as an integer off the stack
- int popINT () - pops an integer value off the stack
- char popCH () - pops a character value off the stack
- int popMA () - pops a memory address value off the stack
- int popPA () - pops a program address value off the stack
- float popFLOAT () - pops a floating point value off the stack

- public List<Memory.Data> getStack () - retrieves the entire stack as a list of Data objects

Heap Zone

The heap zone is manipulated using a HeapAllocator. It is used for dynamic allocation of memory space. Memory provides the following methods for working with the heap zone:

- HeapAllocator getHeapAllocator () - obtain the heap allocator
- void setHeapAllocator (HeapAllocator heap) - set the heap allocator
- public List<Memory.Data> getAllocation (HeapAllocator.Allocation alloc) - retrieves a heap memory allocation as a list of Data objects

2.1.3 The Heap Allocator

(CORE/HEAPALLOCATOR.JAVA, CORE/EXPLICITFREEALLOCATOR.JAVA)

The SaM memory allocator is responsible for managing the heap. It must support reserving chunks of memory of a particular size, and reclaiming the space later, without fragmentation.

The current allocator is based on the Doug Lea's malloc allocator: Doug Lea's malloc allocator `http://gee.cs.oswego.edu/dl/html/malloc.html`, or rather the high-level description of it. The implementation was written independently, and was simplified for our purposes.

Access Interface

The heap allocator supports the following methods:

- void init () - initialize the state of memory
- void malloc (int size) - allocate a memory chunk of the requested size, and place its address on the stack
- void free (int pos) - free a previously allocated memory chunk
- Iterator<HeapAllocator.Allocation> getAllocations () - obtain an iterator for all allocations

- Memory getMemory () - obtain the Memory object associated with this allocator
- void setMemory(Memory mem) - set the Memory object for this allocator

Design

The allocator's main feature is the division of memory space into chunks (slices) with power-of-two sizes. Those chunks are grouped together into linked lists with other chunks of the same size. The linked list is attached to an "anchor," whose memory offset from the base of the heap corresponds to $\log_2$(size) of chunks contained within - this makes it trivial to locate chunks of a required size, using simple bitwise algebra. Offset 0 is special, and contains chunks of various sizes that have already been allocated. Each memory chunk includes accounting metadata before and after the actual space visible to the user. This is used internally to maintain the linked lists, and for error detection.

malloc:

When a chunk of size x is requested, x is rounded to a power of two, and the anchor is traversed looking of free memory of that size, or larger (by a factor of 2 for each anchor index). If no such slice is found, exception is thrown (OMEM). If a slice is found, it is disconnected from the linked list. The slice is cut back to the smallest power-of-two slice sufficiently large to contain the requested allocation. The remainder is distributed throughout the anchor as free space. The allocated chunk's metadata is updated to reflect how large it is, and what the neighbor chunks are, and the user-visible address of it (past the metadata) is returned by malloc.

free:

When a chunk is freed, it is removed from the list of allocations and it is merged together with preceding free chunks and following free chunks. This process prevents fragmentation of memory - it reassembles small chunks into larger ones, which allow the allocator to handle larger requests. There may be more than one consecutive free chunk, because, after the chunks are merged, the result is cut and redistributed back into bins of various sizes. Unless the result was an exact power of two, this may cause multiple consecutive free chunks to exist in memory. However, the distribution algorithm makes sure that this number is bound by the size of the anchor, which ensures that free is kept O(1).

getAllocations:

An interesting consequence of keeping the list of allocations in anchor 0 is that we can iterate this list. This function returns a Java iterator (in O(1)) which allows a debugging program to iterate the contents of memory, if necessary. This can be useful, for example, to check for memory leaks (if any allocations remain).

Notes:

Enable the constant DEBUG_ALLOCATOR to print the allocator bins after every malloc and free call.

This allocator was successfully translated to SaM assembly from an enhanced version of the Bali language in Spring 2005. It executed correctly, at about 100% overhead. The compiler and the code may be found in the SVN repository for 2005sp/part3c.

Further Work:

While power-of-two blocks simplify computations, and are easy to manipulate, the allocator is extremely wasteful on requests of size 2k + c for large k and small c. To address this, a better distribution of bin sizes should be designed - this is an opportunity for future improvement of SaM.

2.1.4 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 following methods are avaliable for reading and writing video data:

- void writeInt(int i) - write an integer
- void writeString(String str) - write a string
- void writeFloat(float f) - write a floating point number
- void writeChar(char c) - write a character
- int readInt() - read an integer
- float readFloat() - read a floating point number

- char readChar() - read a character
- String readString() - read a string

2.1.5 System Chipset

(CORE/SYS.JAVA**)**

This component provides a unified way to access the system devices in SaM. It is 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 the Sys object 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 pass an integer to the constructor you can instantiate multiple processors - this feature is currently of limited functionality, but could be used to implement multiprocessing in the future. The following methods are available to work with the Sys object:

- Processor cpu() - return the first processor of this system
- Collection<Processor> cpus() - return the entire collection of processors
- Memory mem() - return the system memory
- Video video() - return the system video object (null if not available)
- void setVideo(Video v) - set the system video object

2.2 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.

2.2.1 SaM Instructions

(CORE/INSTRUCTIONS/***)**

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 3 on page 23 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. All included instructions are prefixed by *SAM_*, which is removed automatically when the instruction is actually used.

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. Some instructions, such as ADD, SUB, DUP, SWAP, ISNIL, NOT and EQUALS are defined on multiple data types. Refer to the ISA manual for details.

Apart from the *op* variable, which is provided to operand classes, instructions also have access to the *cpu* (the processor), *mem* (the memory), *video* (the video interface), *sys* (the system object), and *prog* (the program that the instruction is a part of).

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.

2.2.2 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.

The SaM core package will support multiple labels per address. Unfortunately, as of SaM 2.3, 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.

The symbol table provides the following methods:

- void add(String symbol, int address) - add a symbol with the specified address
- String resolveSymbol(int address) - return the symbol for the given address
- Collection<String> resolveSymbols(int address) -return all symbols for the given address
- int resolveAddress(String label) - return the address for the given symbol
- Collection<String> getSymbols() - return a collection of all symbols
- String toString() - return a string representation of this table

2.2.3 SaM Reference Table

(CORE/REFERENCETABLE.JAVA, CORE/SAMREFERENCETABLE.JAVA)

Introduced in SaM 2.4, the reference table maps symbols to places in the program where they are used. This sets the stage for future work on dynamic linking, where

reference resolution is performed at linking time. Currently the reference table is used only during assembly.

The reference table provides the following methods:

- void add(String symbol, int ref_address) - add a reference to Symbol at the specified address
- void deleteSymbol(String symbol) - delete all references to the given symbol
- Collection<Integer> getReferences(String symbol) - return a collection of references for this symbol
- int size() - return the number of symbols in the reference table
- String toString() - return a string representation of this table

2.2.4 SaM Program

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

A SaM Program is an enclosing container for Instructions, a ReferenceTable, and a SymbolTable. The assembler generates a Program object, and resolves any references using the constructed symbol table. The processor loads the program, and executes each instruction of the program object, using the PC register as a numeric index.

The SaM program provides the following methods:

- void addInst (Instruction ins) - append an instruction object to the program
- void addInst (Instruction[] ins) - append an array of instruction objects to the program
- Instruction getInst(int pos) - return the instruction at the specified index
- List<Instruction> getInstList() - return all instructions of this program as a list
- int getLength() - return the number of instructions
- SymbolTable getSymbolTable() - return the symbol table for this program
- ReferenceTable getReferenceTable() - return the reference table for this program

- void setSymbolTable(SymbolTable table) - set the symbol table for this program
- void setReferenceTable(ReferenceTable table) - set the reference table for this program
- boolean isExecutable() - return if the program is executable (all references are resolved)
- void resolveReferences() - resolve the program's references from its symbol table
- void resolveReferencesFrom(Program prog) - resolve another program's references

2.2.5 SaM I/O

(CORE/TOKENIZER.JAVA, CORE/SAMTOKENIZER.JAVA)

The SaM tokenizer was designed to break text into tokens, and make it easier to parse both high level and assembly programs. The tokenizer's functionality is better described at the SaM Javadoc API documentation. However, the following are the various types of tokens produced by these classes:

- Integers: defined as a sequence of digits starting with either a dash or a digit and containing only digits after the first digit.
- Floating Point Numbers: defined as a sequence of digits starting with either a dash, a period, or a digit and containing only digits and one period after the first digit.
- Words: defined as a letter followed by a sequence of alphanumeric characters or underscores (_) without any whitespace. Also, any numbers that have two or more periods.
- Strings: anything between two quotation signs. Valid escapes (\n, \r. \\, \", \', \xxx, etc...) are evaluated in this case. This is only enabled if the correct option is passed to SamTokenizer.
- Characters: a single character betweeen two apostrophes. Valid escapes (\n, \r. \\, \", \', \xxx, etc...) are evaluated in this case. This is only enabled if the correct option is passed to SamTokenizer.

- Comments: any text on the same line after //. This is only enabled if the correct options is passed to SamTokenizer.
- Operators: defined as any non alphanumeric character and represented by the java type *char*.
- EOF: Such a token represents that the end of the stream has been reached and that there are no more tokens.

2.3 SaM Front Ends

SaM frontends make use of the core package to assemble and execute SaM programs.

2.3.1 SaM Assembler

(CORE/SAMASSEMBLER.JAVA**)**

The SaM assembler is the equivalent of a real 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 tracks symbol references into a ReferenceTable, and resolves the symbols currently available into a SymbolTable. It combines all this information inside a Program object.

2.3.2 SaM Graphical User Interfaces

2.3.2.1 SamGUI - Graphical UI

SamGUI is a feature-rich graphical front-end, which displays the stack and heap 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. Finally, breakpoints are supported to ease debugging.

2.3.2.2 SamCapture - Capture Viewer

SamCapture allows the user to view captures created by the SamGUI. This feature allows the user to view the state of the processor and memory side-by-side form one step to the next.

2.3.2.3 SamTester - Test Script Execution

SamTester is a front-end to the ui.utils.TestScript class. It allows the user to create a TestScript and execute multiple SaM programs quickly and easily. It reports the output, and also allows the XML-based test scripts to be saved.

2.3.2.4 Usage

The three graphical user interfaces described all have a similar execution syntax. Each can execute up to two arguments. The first argument can be either a filename (in which case the an instance of the class that was launched is started with the provided file) or one of *-gui*/*-capture*/*-tester*. In the latter case, the appropriate GUI is launched and any provided second argument is loaded as a file into this GUI.

2.3.3 SaM Text User Interface

When working in a console based enviroment it is essential to have a text only solution to executing SaM programs. Currently one such interface to the SaM simulator is provided. It is also important to note that this interface is much faster than any of the GUIs.

2.3.3.1 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

-load <class>: Loads a new instruction from the provided class

Chapter 3

SaM Instruction Set Architecture Manual

Instruction Set Architecture Manual Format

Each instruction specifies input types expected, output types, operand type, version, and description. The input and output values are ordered from top to bottom. The leftmost value corresponds to the top of the stack (for both input and output).

Types

Input/output types correspond to the memory types supported by SaM (see Memory). The allowed operand types are Integer, Float, Character (single quotes), String (double quotes), and/or Label (unquoted string). Please note that no instruction requires a specific input type on the stack - all memory types are treated as integers, if they are not converted with the appropriate instructions.

Instruction Order Execution

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

3.1 Type Converters

Type conversion instructions convert a value from one type to another.

3.1.1 FTOI

Stack Input: Float

Stack Output: Integer

Operand: None

Since: 2.0

Description:

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

3.1.2 FTOIR

Stack Input: Float

Stack Output: Integer

Operand: None

Since: 2.0

Description:

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

3.1.3 ITOF

Stack Input: Integer

Stack Output: Float

Operand: None

Since: 2.0

Description:

Converts an integer to a float.

3.2 Stack Insertions

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

3.2.1 PUSHIMM

Stack Input: None

Stack Output: Integer

Operand: Integer

Since: 1.0

Description:

Places the integer operand onto the stack.

3.2.2 PUSHIMMF

Stack Input: None

Stack Output: Float

Operand: Float

Since: 2.0

Description:

Places the float operand onto the stack.

3.2.3 PUSHIMMCH

Stack Input: None

Stack Output: Character

Operand: Character

Since: 2.0

Description:

Places the character operand onto the stack.

3.2.4 PUSHIMMMA

Stack Input: None

Stack Output: Memory Address

Operand: Integer

Since: 2.4

Description:

Places the integer operand onto the stack as a memory address.

3.2.5 PUSHIMMPA

Stack Input: None

Stack Output: Program Address

Operand: Label or Integer

Since: 2.3.2

Description: This instruction pushes the address of the label onto the stack as a program address.

3.2.6 PUSHIMMSTR

Stack Input: None

Stack Output: Memory Address

Operand: String

Since: 2.0

Description:

Allocates space for the string on the heap, stores the sequence of characters on the heap, starting with the first letter as the lowest heap location. The string is null-terminated automatically. The object's address is pushed onto the stack.

3.3 Register Manipulation

These instructions manipulate the processor registers.

3.3.1 PUSHSP

Stack Input: None

Stack Output: Memory Address

Operand: None

Since: 2.4

Description

Pushes the value of the SP register onto the stack.

3.3.2 PUSHFBR

Stack Input: None

Stack Output: Memory Address

Operand: None

Since: 1.0

Description:

Pushes the value of the FBR register onto the stack.

3.3.3 POPSP

Stack Input: Memory Address

Stack Output: None

Operand: None

Since: 2.4

Description:

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

3.3.4 POPFBR

Stack Input: Memory Address

Stack Output: 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.

3.4 Stack Manipulation

3.4.1 DUP

Stack Input: Any type

Stack Output: Two of the input type

Operand: None

Since: 1.0

Description:

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

3.4.2 SWAP

Stack Input: Any two types

Stack Output: The reverse of the two types

Operand: None

Since: 1.0

Description:

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

3.5 Stack/Heap Allocation

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

3.5.1 ADDSP

Stack Input: None

Stack Output: None

Operand: Integer

Since: 1.0

Description

Increments the SP register by the provided value.

3.5.2 MALLOC

Stack Input: Integer

Stack Output: Memory Address

Operand: None

Since: 1.0 (Modified in 2.0, 2.6)

Description:

This instruction allocates space on the heap of size provided by the input value. It writes the address of the allocated space to the stack.

3.5.3 FREE

Stack Input: Memory Address

Stack Output: None

Operand: None

Since: 2.6

Description:

This instruction reclaims the space used by a previous allocation. It pops the address of the allocation off the stack, and marks the memory space as free.

3.6 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).

3.6.1 PUSHIND

Stack Input: Memory Address

Stack Output: Value

Operand: None

Since: 1.0

Description:

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

3.6.2 STOREIND

Stack Input: Value, Memory Address

Stack Output: 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.

3.6.3 PUSHABS

Stack Input: None

Stack Output: Value

Operand: Integer

Since: 2.4

Description:

Pushes the data at the memory address specified by the operand, onto the stack, preserving its type.

3.6.4 STOREABS

Stack Input: Value

Stack Output: None

Operand: Integer

Since: 2.4

Description:

Sets the address provided by the operand to the value/type of the stack input.

3.7 Relative Store/Retrieve

These instructions provide access to memory addresses relative to the FBR register. They should generally be used for local variables, parameters, and temporary stack operations.

3.7.1 PUSHOFF

Stack Input: None

Stack Output: Value

Operand: Integer

Since: 1.0

Description:

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

3.7.2 STOREOFF

Stack Input: Value

Stack Output: None

Operand: Integer

Since: 1.0

Description:

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

3.8 Integer Algebra

These instructions perform integer algebra on the stack.

3.8.1 ADD

Stack Input: Value (Float disallowed), Value (Float disallowed)

Stack Output: Integer, Memory Address (if exactly one of the input values is a Memory Address), or Program Address (if exactly one of the input values is a Program Address)

Operand: None

Since: 1.0 (modified in 2.0, 2.6)

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

3.8.2 SUB

Stack Input: Value (Float disallowed), Value (Float disallowed)

Stack Output: Integer, Memory Address (if exactly one of the input values is a Memory Address), or Program Address (if exactly one of the input values is a Program Address)

Operand: None

Since: 1.0 (modified in 2.0, 2.6)

Description:

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

3.8.3 TIMES

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 1.0

Description:

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

3.8.4 DIV

Stack Input: Integer, Integer

Stack Output: 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.

3.8.5 MOD

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 2.0

Description:

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

3.9 Floating Point Algebra

These instructions perform floating point algebra on the stack.

3.9.1 ADDF

Stack Input: Float, Float

Stack Output: Float

Operand: None

Since: 2.0

Description:

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

3.9.2 SUBF

Stack Input: Float, Float

Stack Output: Float

Operand: None

Since: 2.0

Description:

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

3.9.3 TIMESF

Stack Input: Float, Float

Stack Output: Float

Operand: None

Since: 2.0

Description:

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

3.9.4 DIVF

Stack Input: Float, Float

Stack Output: Float

Operand: None

Since: 2.0

Description:

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

3.10 Shifts

These instructions perform signed bitwise shifting. Shifting moves all the bits in the shifted value over by the specified amount left or right. With signed bitwise shifting, the sign of the value is preserved when shifting to the right. Bitwise shift left is eqivalent of multiplying an integer by two. Bitwise shift right is equivalent to dividing an integer by two.

3.10.1 LSHIFT

Stack Input: Integer

Stack Output: Integer

Operand: Integer

Since: 2.0

Description:

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

3.10.2 LSHIFTIND

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 2.6

Description:

Shifts the second input value to the left by the number of places specified by the first input value.

3.10.3 RSHIFT

Stack Input: Integer

Stack Output: Integer

Operand: Integer

Since: 2.0

Description:

Shifts the input value 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 added to the left).

3.10.4 RSHIFTIND

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 2.6

Description:

Shifts the second input value to the right by the number of places specified by the first input value. 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 added to the left).

3.11 Logic

These instructions perform logical operations on input values. They treat all non-negative numbers as 1.

3.11.1 AND

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 1.0

Description:

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

3.11.2 OR

Stack Input: Integer, Integer

Stack Output: 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.

3.11.3 NOR

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 1.0

Description:

If either value is non-zero, pushes 0 onto the stack. Otherwise, pushes 1. Equivalent to OR followed by NOT.

3.11.4 NAND

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 1.0

Description:

If both values are non-zero, pushes 0 onto the stack. Otherwise, pushes 1. Equivalent to AND followed by NOT.

3.11.5 XOR

Stack Input: Integer, Integer

Stack Output: 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.

3.11.6 NOT

Stack Input: Integer

Stack Output: Integer

Operand: None

Since: 1.0

Description:

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

3.12 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).

3.12.1 BITAND

Stack Input: Integer, Integer

Stack Output: 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.

3.12.2 BITOR

Stack Input: Integer, Integer

Stack Output: 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.

3.12.3 BITNOR

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 2.0

Description:

Performs a bitwise NOR operation. For each output bit, if either input bit is 1, the resulting bit is a 0. Otherwise, it is a 1. Equivalent to BITOR followed by BITNOT.

3.12.4 BITNAND

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 2.0

Description:

Performs a bitwise NAND operation. For each output bit, if both input bits are 1, the resulting bit is a 0. Otherwise, it is a 1. Equivalent to BITAND followed by BITNOT.

3.12.5 BITXOR

Stack Input: Integer, Integer

Stack Output: Integer

Operand: None

Since: 2.0

Description:

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

3.12.6 BITNOT

Stack Input: Integer

Stack Output: Integer

Operand: None

Since: 2.0

Description:

Performs a bitwise NOT operation. 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.

3.13 Comparison

These instructions allow two values to be compared.

3.13.1 CMP

Stack Input: Integer, Integer

Stack Output: 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.

3.13.2 CMPF

Stack Input: Float, Float

Stack Output: 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.

3.13.3 GREATER

Stack Input: Integer, Integer

Stack Output: 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.

3.13.4 LESS

Stack Input: Integer, Integer

Stack Output: 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.

3.13.5 EQUAL

Stack Input: Value, Value

Stack Output: 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.

3.13.6 ISNIL

Stack Input: Value

Stack Output: 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.

3.13.7 ISPOS

Stack Input: Integer

Stack Output: 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.

3.13.8 ISNEG

Stack Input: Integer

Stack Output: 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.

3.14 Jumps

Jumps are special instructions used for transferring control to other pieces of code. They are useful for such things as loops and, especially, functions. Jumps can use labels or integer addresses for their operands. Labels are translated to the correct address at assemby and/or linking time.

3.14.1 JUMP

Stack Input: None

Stack Output: None

Operand: Label or Integer

Since: 1.0

Description:

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

3.14.2 JUMPC

Stack Input: Integer

Stack Output: None

Operand: Label or Integer

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.

3.14.3 JUMPIND

Stack Input: Program Address

Stack Output: 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.

3.14.4 RST

Stack Input: Program Address

Stack Output: None

Operand: None

Since: 2.4

Description:

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

3.14.5 JSR

Stack Input: None

Stack Output: Program Address

Operand: Label or Integer

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.

3.14.6 JSRIND

Stack Input: Program Address

Stack Output: 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.

3.14.7 SKIP

Stack Input: Integer

Stack Output: None

Operand: None

Since: 2.3

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.

3.15 Stack Frames

Stack frames are used for relative addressing and are usually defined for every subroutine.

3.15.1 LINK

Stack Input: None

Stack Output: Memory Address

Operand: None

Since: 1.0

Description:

Pushes the FBR register on the stack and sets the FBR register to the SP register - 1. This should be undone with UNLINK.

3.15.2 UNLINK

Stack Input: Memory Address

Stack Output: None

Operand: None

Since: 2.4

Description:

Sets the FBR register to the value at the top of the stack. This is often used to undo LINK and is currently the same as POPFBR.

3.16 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, the appropriate zero value for the data type is placed onto the stack.

3.16.1 READ

Stack Input: None

Stack Output: Integer

Operand: None

Since: 2.0

Description:

Asks the Video interface for an integer and pushes it onto the stack. If there is no Video defined, pushes 0 onto the stack.

3.16.2 READF

Stack Input: None

Stack Output: Float

Operand: None

Since: 2.0

Description:

Asks the Video interface for n float and pushes it onto the stack. If there is no Video defined, pushes 0.0 onto the stack.

3.16.3 READCH

Stack Input: None

Stack Output: Character

Operand: None

Since: 2.2

Description:

Asks the Video interface for a character and pushes it onto the stack. If there is no Video defined, pushes '\0' onto the stack.

3.16.4 READSTR

Stack Input: None

Stack Output: Memory Address

Operand: None

Since: 2.0

Description:

Asks the Video interface for a string and allocates 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. The string is null-terminated. If there is no video card, the instruction performs the operation above on an empty string.

3.16.5 WRITE

Stack Input: Integer

Stack Output: None

Operand: None

Since: 2.0

Description:

Writes the integer to the Video interface. 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.

3.16.6 WRITEF

Stack Input: Float

Stack Output: None

Operand: None

Since: 2.0

Description:

Writes the float to the Video interface. 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.

3.16.7 WRITECH

Stack Input: Character

Stack Output: None

Operand: None

Since: 2.2

Description:

Writes the character to the Video interface. 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.

3.16.8 WRITESTR

Stack Input: Memory Address

Stack Output: None

Operand: None

Since: 2.0

Description

Writes the string at the memory address provided to the Video interface. 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 freed by this instruction, but its address is removed from the stack.

3.17 Program Control

3.17.1 STOP

Stack Input: None

Stack Output: None

Operand: None

Since: 1.0

Description

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