Civilized C Virtual Machine (CiviC-VM) and Civilized C Assembler (CiviC-AS)

Reference Manual

Version 0.9

Mike Lankamp **Clemens Grelck**

University of Amsterdam

Computer Systems Architecture

February 2013

Contents

Chapter 1

Instruction Set Architecture

1.1 Introduction

This part of the documents describes the Instruction Set Architecture that CiviC programs will be run in. It comprises the conceptual execution model and environment and a specification of all instructions.

1.2 Execution Environment

This section describes the execution environment of the virtual machine as seen by assembly-language programmers. It describes how the CiviC-VM executes instructions and how it stores and manipulates data. The parts of the execution environment described here include the stack and registers.

1.2.1 Overview

Any program running on the CiviC-VM is given a set of resources for executing programs and for storing data. These resources make up the execution environment of the CiviC-VM. More precisely, it consists of three disjoint memories and three registers:

Instruction memory. The instruction memory is a read-only memory that contains the code of a program after being loaded into the CiviC-VM.

Stack memory. The stack is a contiguous array of values. Conceptually, the stack can be infinitely long. Most instructions operate on the top elements of the stack, removing them from the top of the stack and pushing some result back onto the stack. The top of the stack is identified by the stack pointer register.

Heap memory. The heap memory is a random access memory; it is mainly used to store arrays.

Instruction Pointer register. This register contains the address of the currently executing instruction. The IP register is comprised of two parts, a module identifier and an offset into that module's instruction stream. Branches, jumps and subroutine calls modify the offset part of the IP register; jumps to external subroutines modify the entire register.

Stack pointer register. The stack pointer register points to the top of the stack, more precisely it points to the next available entry in stack memory.

Frame Pointer register. The frame pointer register points to the beginning of the currently executing function's current frame and this permits access to the function's parameters and local variables.

Note that all three registers cannot be directly read or written by programs but are manipulated by certain instructions. For example, instructions that push or pop values onto or from the stack,

respectively, implicitly manipulate the stack pointer register; all instructions manipulate the instruction pointer register by either advancing it to the next instruction or by adding a certain offset to it in the case of jump and branch instructions. Last not least, instructions implementing the procedure call convention implicitly manipulate the frame pointer register.

1.2.2 Modules

A program consists of one or more modules. Each module consists of a

sequence of instructions. The instructions in a module are a byte stream of variable-length instructions; they are packed one after the other to form a single array of instruction bytes. Each instruction has either zero, one or two arguments of varying numbers of bytes.

global variable table. The global variable table is an array of values that is created, but left uninitialized, when the module is loaded. Certain instructions can move values between this table and the stack.

constant table. The constant table is an array of values that is created and initialized when the module is loaded. Certain instructions can move values from this table onto the stack.

function import table. The function import table is a list of functions that must be located in different modules. These imported functions will be connected with exported functions in other modules. Certain instructions can perform a subroutine jump to an import entry. Executing such an instruction will result in a subroutine call into a different module. If an import entry could not be found among other modules when the module is loaded, execution of the program is aborted.

function export table. The function export table is a list of functions in the current module that should be made available to other modules for importing.

variable import table. The variable import table is a list of variables that must be located in different modules. Like imported functions, imported variables will be connected with exported variables in other modules. Imported variables can be loaded and manipulated using specialised instructions which operate on the global variable table of another module. If an import entry could not be found among other modules when the module is loaded, execution of the program is aborted.

variable export table. The function export table is a list of variables in the current module that should be made available to other modules for importing.

1.2.3 Operands

Instructions have zero, one or two operands. We distinguish a number of differently typed operands for a number of different purposes:

- a 1-byte unsigned number;
- a 1-byte unsigned index into the current frame;
- a 2-byte unsigned index into the constant table;
- a 2-byte unsigned index into the global table;
- a 2-byte unsigned index into an import table;
- a 2-byte signed offset on the instruction pointer.

The number and type of operands is unambiguously determined by the instruction itself. Note that in contrast to register transfer instruction set architectures (both hardware and virtual), the runtime stack implicitly provides the data that certain instructions operate on.

1.3 Data Types

The CiviC-VM supports four different types:

int. Integer numbers are signed and guaranteed to be at least 32 bits;

float. Floating-point numbers conform to the IEEE 754 standard and are guaranteed to be at least single precision;

bool. Booleans can only have two values: 'true' or 'false';

array references. Arrays are inherently one-dimensional; the element type is either int, float or bool; There is no support for nested arrays in the CiviC-VM.

For safety, all items in tables, on the stack and in arrays are typed during initialization. This type can never be changed. Instructions that deal in reading or writing these values come in different shapes, depending on the type of the data that is operated on. For instance, the `iadd` instruction adds two integer values, while the `fadd` instruction adds two floating-point values, etc.
In case of a mismatch between the operand types expected by some instructions and the types of the values found on the stack, in the heap, in the constant pool or in the global frame, the CiviC-VM aborts with an appropriate error message.

1.4 Instruction Set Reference

This section describes the complete CiviC-VM instruction set. The instruction descriptions are grouped according to their function. Instructions consist of one primary opcode byte and, depending on the instruction, one or more 8-bit or 16-bit operands. For each instruction, the opcode is given, as well as the required operands and a description of each.

1.4.1 Interpreting the reference pages

This section describes the information contained in the various sections of the following instruction reference pages. It also explains the notational conventions and abbreviations used in them.

Opcode column

The "Opcode" column gives the 8-bit opcode for the instruction. The opcode is given in hexadecimal representation.

Instruction column

The "Instruction" column gives the syntax of the instruction statement as it would appear in an assembly program. The following is a list of symbols used to represent operands in the instruction statements:

- *A*—An unsigned 8-bit count of the number of arguments that a subroutine takes.
- *L*—An unsigned 8-bit offset into the current frame for access to a parameter or local variable.
- *N*—An unsigned 8-bit static link counter. Instructions take an operand of this type can refer to frames other than their own. This operand will identify how many (lexical) frames to skip past.
- *C*—An unsigned 16-bit index into the module's constant table.
- *G*—An unsigned 16-bit index into the module's global table.
- *I*—An unsigned 16-bit index into the module's import table.
- *O*—A signed 16-bit offset relative to the instruction's PC.

Description column

The "Description" column briefly explains the various forms of the instruction. The following "Description" section contains more information.

Description

The "Description" section describes the purpose of the instructions and the required operands.

1.4.2 Arithmetic instructions

Addition

Opcode	Instruction	Description
00	iadd	Add two integer numbers
20	fadd	Add two floating-point numbers

Description

Pops two values from the top of the stack, adds them and pushes the result onto the stack. `iadd` pops and pushes integer values. `fadd` pops and pushes floating point values.

Subtraction

Opcode	Instruction	Description
01	isub	Subtract two integer numbers
21	fsub	Subtract two floating-point numbers

Description

Pops two values from the top of the stack, subtracts the first popped value from the second popped value and pushes the result onto the stack. `isub` pops and pushes integer values. `fsub` pops and pushes floating point values.

Multiplication

Opcode	Instruction	Description
02	imul	Multiply two integer numbers
22	fmul	Multiply two floating-point numbers

Description

Pops two values from the top of the stack, multiplies them and pushes the result onto the stack. `imul` pops and pushes integer values. `fmul` pops and pushes floating point values.

Division

Opcode	Instruction	Description
03	idiv	Divide two integer numbers
23	fdiv	Divide two floating-point numbers

Description

Pops two values from the top of the stack, divides the second popped value by the first popped value and pushes the result onto the stack. `idiv` pops and pushes integer values. `fdiv` pops and pushes floating point values.

Remainder

Opcode	Instruction	Description
04	irem	Calculate remainder of integer division of two integer numbers

Description

Pops two values from the top of the stack, divides the second popped value by the first popped value (i.e., the values are popped right-to-left) and pushes the remainder onto the stack. `irem` pops and pushes integer values.

Negation

Opcode	Instruction	Description
05	ineg	Negate integer
25	fneg	Negate floating-point
45	bnot	Negate boolean

Description

Pops a value from the top of the stack, negates it and pushes the result onto the stack. `ineg` pops and pushes integer values. `fneg` pops and pushes floating point values. `bnot` pops and pushes boolean values.

Increment / Decrement

Opcode	Instruction	Description
60	iinc *L C*	Increment local by constant
61	iinc_1 *L*	Increment local by one
62	idec *L C*	Decrement local by constant
63	idec_1 *L*	Decrement local by one

Description

Increments or decrements the local variable identified by *L* with the value loaded from the constant table at index *C*. `iinc` and `idec` increment and decrement, respectively. `iinc_1` and `idec_1` increment and decrement by 1, and do not take the constant table index *C*. All instructions operate on integer local variables. The stack pointer is not modified.

Locical conjunction and disjunction

Opcode	Instruction	Description
44	badd	Get the logical disjunction of two boolean values
4e	bmul	Get the logical conjunction of two boolean values

Description

Pops two values from the top of the stack, calculates disjunction or conjunction and pushes the result onto the stack. `badd` calculates disjunction, `bmul` calculates conjunction.

1.4.3 Comparison instructions

Relational Operations

Opcode	Instruction	Description
08	ine	Test if not equal
09	ieq	Test if equal
0A	ilt	Test if less
0B	ile	Test if less or equal
0C	igt	Test if greater
0D	ige	Test if greater or equal
28	fne	Test if not equal
29	feq	Test if equal
2A	flt	Test if less
2B	fle	Test if less or equal
2C	fgt	Test if greater
2D	fge	Test if greater or equal
48	bne	Test if not equal
49	beq	Test if equal

Description

These instructions pop two values off the top of the stack, compare them, and push the boolean result onto the stack. The `i??` instructions pop two integer values, the `f??` instructions pop two floating-point values and the `b??` instructions pop two boolean values. The values are popped right-to-left (i.e., the first popped value is on the right side of the comparison).

1.4.4 Control flow instructions

Initiate Subroutine call

Opcode	Instruction	Description
68	isr	Initiate call to subroutine in current scope
69	isrn *N*	Initiate call to subroutine in outer scope
6A	isrl	Initiate call to subroutine local to current scope
6B	isrg	Initiate call to global subroutine

Description

These instructions push an activation record onto the stack which will be finalized by the `jsr` or `jsre` instructions. `isr` pushes an activation record for a call to a subroutine which is at the same nesting level as the currently executing subroutine. `isrn` pushes an activation record for a call to a subroutine which is at a higher nesting level than the currently executing subroutine. *N* specifies the number of nesting levels that the target subroutine is higher at. *N* must be greater than 0 and less than the nesting depth of the currently executing subroutine. `isrl` pushes an activation record for a call to a subroutine which is nested level directly below the currently executing subroutine (i.e., the nesting depth difference is exactly 1). `isrg` pushes an activation record for a call to a subroutine which is at the global level (i.e., a call to a subroutine which is not nested).
Each of these instructions must be paired with a `jsr` or `jsre` instruction. The number of values pushed onto the stack in between this instruction and the `jsr` or `jsre` must be exactly equal to the amount of arguments that the target subroutine expects.

Jump to Subroutine

Opcode	Instruction	Description
6D	jsr *A O*	Jump to subroutine
6E	jsre *I*	Jump to external subroutine

Description

Writes the program counter of the next instruction into the activation record pushed onto the stack by the preceding `isr`, `isrn`, `isrg` or `isrl` instruction. The activation record must be directly preceding the arguments on the stack. The number of arguments on the stack is taken from *A* in the `jsr` instruction, or from the import entry in case of a `jsre` instruction.
`jsr` calls a subroutine in the same module; the location of this subroutine is $pc + O$, where *pc* is the program counter before executing the instruction and *O* is the signed integer offset taken from the instruction.
`jsre` calls a subroutine in another (or possibly the same) module. The location of the subroutine is an entry in the import table, at index *I*. Upon loading the various modules, the virtual machine will resolve all imports such that `jsre` will work.

Enter Subroutine

Opcode	Instruction	Description
6C	esr *L*	Enter subroutine

Description

Advances the top of the stack with *L* elements, thus reserving space for *L* local variables.
`esr` should be issued at the beginning of a subroutine, thus ensuring that the locals and arguments are contiguous in the stack, i.e., the first argument is addressable as local 0, and so on.

Return from Subroutine

Opcode	Instruction	Description
0F	ireturn	Return integer number from subroutine
2F	freturn	Return floating-point number from subroutine
4F	breturn	Return boolean value from subroutine
6F	return	Return from subroutine without function value

Description

Restores the stack top to the current subroutine's activation record, set the current program counter to the return address in the activation record and pops the activation record. `ireturn`, `freturn` and `breturn` additionally pop an integer, floating-point or boolean value off the top of the stack at the beginning, and push that value onto the stack at the end.

Jumps and Branches

Opcode	Instruction	Description
64	jump *O*	Jump by offset
65	branch_t *O*	Branch by offset if true
66	branch_f *O*	Branch by offset if false

Description

These instructions either (unconditionally) jump or conditionally branch to $pc + O$, where pc is the current value of the instruction pointer register (i.e., the old program counter, before executing the instruction), and O is the signed integer offset in the instruction.
`jump` jumps to $pc + O$, i.e. it resets the instruction pointer register to $pc + O$ `branch_t` and `branch_f` pop a boolean value from the top of the stack and branch if and only if that value is `true` or `false`, respectively. Otherwise, they continue with the next instruction in sequence.

1.4.5 Variables and Constants

Load Local Variable

Opcode	Instruction	Description
18	iload *L*	Load local integer
10	iload_0	Load local integer #0
11	iload_1	Load local integer #1
12	iload_2	Load local integer #2
13	iload_3	Load local integer #3
38	fload *L*	Load local floating-point
30	fload_0	Load local floating-point #0
31	fload_1	Load local floating-point #1
32	fload_2	Load local floating-point #2
33	fload_3	Load local floating-point #3
58	bload *L*	Load local boolean
50	bload_0	Load local boolean #0
51	bload_1	Load local boolean #1
52	bload_2	Load local boolean #2
53	bload_3	Load local boolean #3
78	aload *L*	Load local array reference
70	aload_0	Load local array reference #0
71	aload_1	Load local array reference #1
72	aload_2	Load local array reference #2
73	aload_3	Load local array reference #3

Description

Reads a local variable and pushes its value onto the stack. `iload`, `fload`, `bload` and `aload` load a local integer, floating-point, boolean or array reference variable at index *L*, respectively. *L* indexes the current frame in such a manner that the first argument to the current subroutine is 0, and so on. Additional local space can be allocated behind the arguments with the `esr` instruction at the beginning of the subroutine.
The other instructions are short-hand notation. The number in their name indicates the value of *L* that they load.

Load Relatively Free Variable

Opcode	Instruction	Description
19	iloadn *N L*	Load local integer from higher frame
39	floadn *N L*	Load local floating-point from higher frame
59	bloadn *N L*	Load local boolean from higher frame
79	aloadn *N L*	Load local array reference from outer frame

Description

Reads a local variable from a higher frame and pushes its value onto the stack. `iloadn`, `floadn`, `bloadn` and `aloadn` load an local integer, floating-point, boolean or array reference variable at index *L* in the frame that is *N* levels higher, respectively. *N* must be greater than 0 and less than the nesting depth of the currently executing subroutine.

Load Global Variable

Opcode	Instruction	Description
1A	iloadg *G*	Load integer from global table
3A	floadg *G*	Load floating-point from global table
5A	bloadg *G*	Load boolean from global table
7A	aloadg *G*	Load array reference from global table

Description

Loads a value from the module's global table at index *G* and pushes it onto the stack. `iloadg`, `floadg`, `bloadg` and `aloadg` load and push an integer, floating-point, boolean and array reference, respectively. *G* must be a valid index into the module's global table.

Load Imported Variable

Opcode	Instruction	Description
40	iloade *I*	Load integer from import table
41	floade *I*	Load floating-point from import table
42	bloade *I*	Load boolean from import table
43	aloade *I*	Load array reference from import table

Description

Loads a value from the module's variable import table at index *I* and pushes it onto the stack. `iloade`, `floade`, `bloade` and `aloade` load and push an integer, floating-point, boolean and array reference, respectively. *I* must be a valid index into the module's variable import table.

Load Constant

Opcode	Instruction	Description
17	iloadc *C*	Load integer constant from constant table
37	floadc *C*	Load floating-point constant from constant table
57	bloadc *C*	Load boolean constant from constant table
14	iloadc_0	Load 0
34	floadc_0	Load 0.0
54	bloadc_t	Load `true`
15	iloadc_1	Load 1
35	floadc_1	Load 1.0
55	bloadc_f	Load `false`
16	iloadc_m1	Load -1

Description

Push a constant value onto the stack. `iloadc`, `floadc` and `bloadc` load the constant value from the constant table, at index *C*. The other instructions push the following values as indicated in their description.

Store Local Variable

Opcode	Instruction	Description
1C	istore *L*	Store local integer number
3C	fstore *L*	Store local floating-point number
5C	bstore *L*	Store local boolean value
7C	astore *L*	Store local array reference

Description

Pops a value from the stack and stores it into a local variable. `istore`, `fstore`, `bstore` and `astore` store an integer, floating-point, boolean or array reference into the local variable at index *L*, respectively. *L* indexes the current frame in such a manner that the first argument to the current subroutine is 0, and so on. Additional local space can be allocated behind the arguments with the `esr` instruction at the beginning of the subroutine.

Store Relatively Free Variable

Opcode	Instruction	Description
1D	istoren *N L*	Store local integer in higher frame
3D	fstoren *N L*	Store local floating-point in higher frame
5D	bstoren *N L*	Store local boolean in higher frame

Description

Pops a value from the stack and stores it into a local variable. `istoren`, `fstoren` and `bstoren` store an integer, floating-point, boolean or array reference into the local variable at index *L* in the frame that is *N* levels higher, respectively. *N* must be greater than 0 and less than the nesting depth of the currently executing subroutine.

Store Global Variable

Opcode	Instruction	Description
1E	istoreg *G*	Store integer in global table
3E	fstoreg *G*	Store floating-point in global table
5E	bstoreg *G*	Store boolean in global table
7E	astoreg *G*	Store array reference in global table

Description

Pops a value from the stack and stores it into the module's global table at index *G*. `istoreg`, `fstoreg`, `bstoreg`, `astoreg` pop and store an integer, floating-point, boolean and array reference, respectively. *G* must be a valid index into the module's global table.

Store Imported Variable

Opcode	Instruction	Description
4A	istoree *I*	Store integer in import table
4B	fstoree *I*	Store floating-point in import table
4C	bstoree *I*	Store boolean in import table
4D	astoree *I*	Store array reference in import table

Description

Pops a value from the stack and stores it into the variable at index *I* of the module's variable import table (in the global table of another module). `istoree`, `fstoree`, `bstoree`, `astoree` pop and store an integer, floating-point, boolean and array reference, respectively. *I* must be a valid index into the module's variable import table.

1.4.6 Arrays

Array Creation

Opcode	Instruction	Description
06	inewa	Create new integer array
26	fnewa	Create new floating-point array
46	bnewa	Create new boolean array

Description

`inewa`, `fnewa` and `bnewa` create a new (one-dimensional) array of integer, floating-point or boolean values, respectively. These instructions pop an integer value off the top of the stack (the array size to allocate), and push a reference to the created array onto the stack.

Read Element from Array

Opcode	Instruction	Description
1B	iloada	Read integer from array
3B	floada	Read floating point from array
5B	bloada	Read boolean from array

Description

Pops an array reference off the top of the stack. Then pops an integer off the top of the stack, which is used to index the array. Then, `iloada`, `floada`, `bloada` take the indexed array integer, floating-point or boolean element, respectively, and push it on top of the stack.

Write Element to Array

Opcode	Instruction	Description
1F	istorea	Write integer into array
3F	fstorea	Write floating-point into array
5F	bstorea	Write boolean into array

Description

Pops an array reference off the top of the stack. Then pops an integer off the top of the stack, which is used to index the array in order of popping them off the stack. Then, `istorea`, `fstorea`, `bstorea` pop an integer, floating-point or boolean value off the top of the stack and store that in the indexed array element.
The type of the array must match the type of the value popped off the stack.

1.4.7 Type conversion instructions

Type Conversion

Opcode	Instruction	Description
0E	i2f	Convert integer to floating-point
2E	f2i	Convert floating-point to integer

Description

`i2f` pops an integer value from the top of the stack, converts it to a floating-point value (possibly losing accuracy in the lesser-significant digits) and pushes the resulting floating-point value onto the stack. `f2i` pops a floating point value from the top of the stack, converts it to an integer value (possibly truncating or rounding the value) and pushes the resulting integer value onto the stack.

1.4.8 Stack maintenance instructions

Pop

Opcode	Instruction	Description
07	ipop	Pop integer number from top of stack
27	fpop	Pop floating-point number from top of stack
47	bpop	Pop boolean value from top of stack

Description

`ipop`, `fpop` and `bpop` pop an integer number, floating-point number and boolean value off the top of the stack, respectively. The popped value is discarded. These instructions are useful whenever the returned value of some function is not bound to a variable in the calling context, i.e. despite the fact that a function does return a value (and the callee will leave that value on the runtime stack upon completion) the calling context is not interested into the function's result value, which in this case needs to be popped from the top of the runtime stack after control has returned to the caller.

Chapter 2

Tool Chain

2.1 CiviC Assembler (CiviC-AS)

2.1.1 Introduction

The Civilized Assembler CiviC-AS constructs an executable module from an assembly input file consisting of pseudo instructions and the instructions described in chapter 1. This chapter of this document describes the format of the assembly file and how its contents affects the generated module file. Appendix B lists an example assembly file.

2.1.2 Assembly format

An assembly file consists of zero or more statements, separated by newlines. A statement is either an instruction, label definition or pseudo-instruction. Comments exist only as single-line comments; everything after a semicolon (`;`) on a line is ignored. Empty lines (after comments removal) are ignored. The assembler parses the file line-by-line.

Types

In certain pseudo-instructions, it is necessary to specify types of variables or parameters. The following types are supported:

Type	⇒	*BasicType*
	|	*ArrayType*
BasicType	⇒	`int`
	|	`float`
	|	`bool`
ArrayType	⇒	*BasicType* `[]`
RetType	⇒	*BasicType*
	|	`void`

In other words, there are the three basic types of integer, floating point and boolean, and a multi-dimensional array type. The latter is specified by adding `[]` after the array's basic type.
Finally, a *return type* exists which can be any of the basic types, and *void*.

Function signatures

In some pseudo-instructions, a *function signature* must be specified. A function signature consists of a quote-delimited string identifying the name, a return type, and a list of parameter types:

FunctionSignature ⇒ *Name RetType Type**
Name ⇒ " [*any non-quote, non-newline character*]* "

Labels

Labels identify offset in the code by name, absolving the programmer or compiler from having to calculate offset manually. Labels are defined on their own line and exist of a token, followed by a colon (`:`):

Label ⇒ *token* :

A label is used by referring to its name in any (pseudo-) instruction that expects a label:

LabelRef ⇒ *token*
| *integer*

Note that, as the grammar indicates, direct offsets in the form of (signed) integer values can be used instead of label references as well. Note that, since labels can be numbers, a label reference is first tried to be located as an actual. Only if that fails, it is interpreted as an offset.

2.1.3 Pseudo-instructions

Pseudo instruction: .exportfun

Format:
`.exportfun` *FunctionSignature LabelRef*

Description

The `.exportfun` pseudo-instruction defines an entry in the module's *function export table* which consists of a function signature and an offset in the module to the first instruction of the exported function.

For instance, the following export definition:
`.exportfun "foo" void int int[] foo`
defines an export entry named "foo" that exports a function that takes an array of integers (the first `int` is the array dimension parameter, giving the size of the first dimension) and returns nothing (void). The offset in the code of the function is defined by label `foo`.

Pseudo instruction: .importfun

Format:
`.importfun` *FunctionSignature*

Description

The `.importfun` pseudo-instruction defines an entry in the module's *function import table* which consists of a function signature.

For instance, the following import definition:
`.importfun "foo" void int int[]`
defines an import entry that should, at load-time, be linked up with an external definition of a function named "foo" that takes an array of integers (the first `int` is the array dimension parameter, giving the size of the first dimension) and returns nothing (void). Certain instructions can refer to this import entry by index to refer to externally defined functions.

Pseudo instruction: .exportvar

Format:
`.exportvar` *Name integer*

Description

The `.exportvar` pseudo-instruction defines an entry in the module's *variable export table* which consists of an export name and an index in the global table.

For instance, the following export definition:
`.exportvar "foo" 0`
defines an export entry named "foo" that exports the variable at index 0 in the global table. Note that the type is not mentioned here, since that is already defined by the `.global` pseudo-instruction.

Pseudo instruction: .importvar

Format:
`.importvar` *Name Type*

Description

The `.importvar` pseudo-instruction defines an entry in the module's *variable import table* which consists of an export name and a variable type.

For instance, the following import definition:
`.importvar "foo" int`
defines an import entry that should, at load-time, be linked up with an external definition of an integer variable named "foo". Certain instructions can refer to this import entry by index to refer to externally defined variables.

Pseudo instruction: .const

Format:
`.const` *BasicType value*

Description

The `.const` pseudo-instruction defines an entry in the module's *constants table* which consists of a type and value. The type can be an integer, floating-point or boolean type and the value is parsed accordingly. Integers and floating-point values are parsed similar to C. Boolean values must be `true` or `false`.
Certain instructions can refer to entries in the constants table by index.

Pseudo instruction: .global

Format:
`.global` *Type*

Description

The `.global` pseudo-instruction defines an entry in the module's global table, which is defined by a type. Both basic types (integer, floating point and boolean) as well as array types can be used here. The global table is a table of typed, but uninitialized entries that can be read and written by instructions by index.

2.1.4 Instructions

If a line does not contain a label definition or pseudo-instruction, it is interpreted as in instruction. The first token should be the name of an instruction from chapter 1. The rest of the tokens on the line are interpreted as the arguments to the instruction.

All instructions in an assembly file are assembled into a sequence of bytes and appended (without padding) in the order in which they occur in the file into a stream of bytes which defines the module's executable code. Offsets in the global table are offsets from the beginning of this byte stream.

2.2 CiviC Virtual Machine (CiviC-VM)

The Civilized Virtual Machine CiviC-VM loads one or more module files as produced by the Civilized Assembler CiviC-AS and executes them.

2.2.1 Order of operation

Specifically, the CiviC-VM performs the following steps:

1. Load and verify all specified module files. This step involves parsing the module binary files, ensuring the input is valid and loading all elements into memory.

2. Initialize the global table in each module. The global table from the binary is used to setup the type information for the module's global value table. Note that the values themselves remain uninitialized.

3. Patch imports in each module. For every import entry, the CiviC-VM will look through all other modules, and built-in functions for a match on the function signature and connect the import entry to that found function. If a match cannot be found, the CiviC-VM will return an error.

4. Execute every module's `__init` function, if any. The `__init` function can be used to initialize globals and do other setup before `main`. Note that this function is not required in a module.

5. Find and execute `main`.

6. Return the return value from `main` as exit code.

2.2.2 Type Safety

Since the ISA contains typed instructions, the CiviC-VM verifies the type safety of instructions by ensuring that the data they operate on is of the correct type, as indicated by the instruction. If this is not the case, the CiviC-VM will abort execution with an error message indicating which instruction caused the invalid access.

2.2.3 Built-in Functions

The CiviC-VM defines several built-in functions that can be used as existing exported function by adding the appropriate import entry in a module. The functions are:

Built-in function: printInt

Signature:
```
void printInt(int i);
```

Description

The `printInt` function prints the signed integer argument to standard out in base 10, followed by a newline. The number is prefixed with a minus sign if the value is negative.

Built-in function: printFloat

Signature:
```
void printFloat(float i);
```

Description

The `printFloat` function prints the floating point value to standard out in base 10, decimal notation, with default precision, followed by a newline. The number is prefixed with a minus sign if the value is negative.

Built-in function: scanInt

Signature:
```
int scanInt();
```

Description

The `scanInt` prompts the user to enter an integer value on standard error, then reads the integer value from the standard input and returns it. If an invalid value was entered, `scanInt` returns 0.

Built-in function: scanFloat

Signature:
```
float scanFloat();
```

Description

The `scanFloat` prompts the user to enter an floating point value on standard error, then reads the floating point value from the standard input and returns it. If an invalid value was entered, `scanFloat` returns 0.0.

Appendix A

Opcode Table

	0	1	2	3	4	5	6	7
0	iadd	iload_0	fadd	fload_0	iloade	bload_0	iinc	aload_0
1	isub	iload_1	fsub	fload_1	floade	bload_1	iinc_1	aload_1
2	imul	iload_2	fmul	fload_2	bloade	bload_2	idec	aload_2
3	idiv	iload_3	fdiv	fload_3	aloade	bload_3	idec_1	aload_3
4	irem	iloadc_0		floadc_0	badd	bloadc_t	jump	
5	ineg	iloadc_1	fneg	floadc_1	bnot	bloadc_f	branch_t	
6	inewa	iloadc_m1	fnewa		bnewa		branch_f	
7	ipop	iloadc	fpop	floadc	bpop	bloadc		
8	ine	iload	fne	fload	bne	bload	isr	aload
9	ieq	iloadn	feq	floadn	beq	bloadn	isrn	aloadn
A	ilt	iloadg	flt	floadg	istoree	bloadg	isrl	aloadg
B	ile	iloada	fle	floada	fstoree	bloada	isrg	
C	igt	istore	fgt	fstore	bstoree	bstore	esr	astore
D	ige	istoren	fge	fstoren	astoree	bstoren	jsr	
E	i2f	istoreg	f2i	fstoreg	bmul	bstoreg	jsre	astoreg
F	ireturn	istorea	freturn	fstorea	breturn	bstorea	return	

This table shows all instructions ordered by opcode. The table should be read as follows: the columns represent the high 4 bits of the opcode and the rows represent the low 4 bits of the opcode. For instance, the instruction `fneg` has an opcode of `0x25`.

Appendix B

Assembly Example

Listed below is an example assembly file for the following CiviC program:

```
extern void printInt(int i);
extern int scanInt();

export int main() {
    int n = scanInt();
    int[n] values;
    readValues(values);
    printValues(values);
    return 0;
}

void readValues(int[n] v) {
    for (int i = 0, n) {
        v[i] = scanInt();
    }
}

void printValues(int[n] v) {
    for (int i = 0, n) {
        printInt(v[i]);
    }
}
```

This could be compiled into the following assembly code:

```
main:
    esr 3
    isrg
    jsre 1
    istore 0
    iload_0
    istore 1
    iload_1
    inewa
    astore 2
    isrg
    iload_1
    aload_2
    jsr 2 readValues
    isrg
    iload_1
    aload_2
    jsr 2 printValues
    iloadc_0
    ireturn

readValues:
    esr 2
    iloadc_0
    istore 2
    iload_0
    istore 3
3_while:
    iload_2
    iload_3
    ilt
    branch_f 4_end
    isrg
    jsre 1
    iload_2
    aload_1
    istorea
    iinc_1 2
    jump 3_while
4_end:
    return

printValues:
    esr 2
    iloadc_0
    istore 2
    iload_0
    istore 3
1_while:
    iload_2
    iload_3
    ilt
```

```
    branch_f 2_end
    isrg
    iload_2
    aload_1
    iloada
    jsre 0
    iinc_1 2
    jump 1_while
2_end:
    return

.exportfun "main" int main
.importfun "printInt" void int
.importfun "scanInt" int
```