The

CKPMvm

Virtual Machine

Compiler Construction Course

Summer 2006

Team Members:

Clemens Krainer 9020112

Salzburg, 13 July 2006

Content

1. Introduction

This paper describes the CKPMvm virtual machine which is the target of the CKPMcc C-compiler.

The rest of this work is structured as follows:

- Chapter 2 contains a short introduction to the architecture of this virtual machine, and briefly describes its byte code instruction set, registers, stack and heap.
- Chapter 3 discusses the byte code instruction set in detail.
- Chapter 4 explains the executable file format.

2. Architecture

The CKPMvm virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. This virtual machine knows nothing of the C programming language, only of a particular binary format, the `cklf` file format. A `cklf` file contains CKPMvm virtual machine instructions and a symbol table, as well as other supplementary information.

The CKPMvm virtual machine can be divided into four fundamental parts:

- A bytecode instruction set
- A set of registers
- A stack
- A heap

The memory area used by the CKPMvm virtual machine is not required to be at any particular place in the memory. The design of this virtual machine was essentially inspired by the design of the Java™ virtual machine, described in [1] and [2].

2.1 Byte Codes

The CKPMvm virtual machine instruction set is optimized to be small and compact. C source code is compiled into byte codes and stored in a `.cklf` file. This is performed by the `CKPMcc` tool. A byte code instruction consists of a one-byte opcode that serves to identify the instruction involved and zero or more operands, each of which may be more than one byte long, that encode the parameters the opcode requires. When operands are more than one byte long, they are stored in big endian order, high-order byte first.

2.2 Registers

The registers in the CKPMvm virtual machine are like the registers in a real computer. The following are the CKPMvm registers:

- PC, the program counter, which indicates what byte code is being executed
- FP, the frame pointer, that references the execution environment of the current subroutine
- SP, the stack pointer. It references to the top of the operand stack, which is used to evaluate all arithmetic expressions.

The virtual machine defines these registers to be 32 bit wide.

2.3 Stack

The CKPMvm virtual machine is stack based. A CKPMvm stack frame is similar to the stack frame of other programming languages – it holds the state for a single subroutine invocation. Frames for nested method invocations are stacked on top of this frame. Each stack frame contains four (possibly empty) sets of data: the supplied parameters, the execution environment, the local variables for the subroutine invocation, and the operand stack. The execution environment helps to maintain the stack itself. It contains the return address and a pointer to the previous stack frame. The operand stack is a one-word wide last-in-first-out (LIFO) stack that is used to store the parameters and return values of most byte code instructions. Each primitive data type has unique instructions that know how to extract, operate, and push back operands of that type.

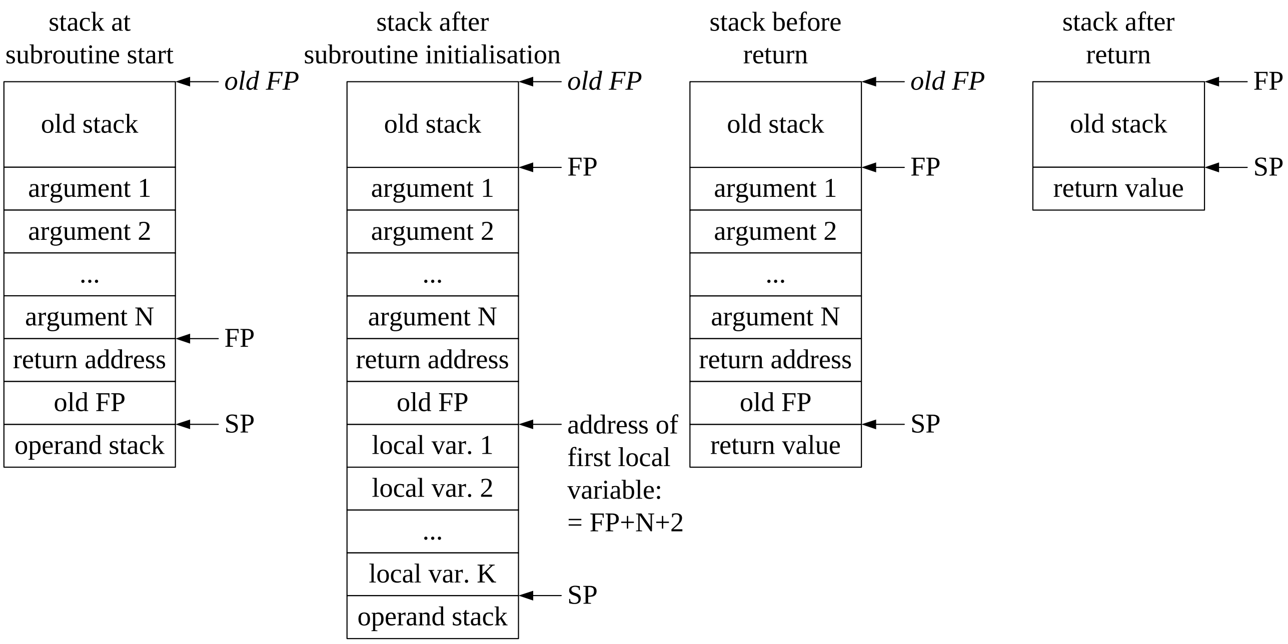

Figure 2.1. Calling conventions on the CKPMvm virtual machine stack

When calling a subroutine, the calling code pushes the arguments on the stack and invokes the subroutine via the `jsr` instruction. This instruction pushes the return address and the current value of the FP register on the stack, as shown in Figure 2.1. At subroutine start the FP register points to the return address and the operand stack is empty. The first instructions of the subroutine code move the FP down to argument 1 and reserve space for the local variables on the stack. There is no register that contains the address of the first local variable, because the local variables have to be addressed relatively to FP. The invoked subroutine is responsible for resetting SP before return. The `retx` instructions reset FP to its previous value and leave only the return value on the stack. If the subroutine returns no value, the `ret` instruction is called.

At start time, the virtual machine pushes the number of passed arguments and a pointer to the arguments onto the stack before calling the `main()`subroutine of the loaded program. The virtual machine uses the return value of the `main()`subroutine as its exit code.

2.4 Heap

The heap is that part of memory from which chunks of memory are allocated. The heap is not garbage collected, so allocated blocks have to be freed explicitly. The virtual machine hides the implementation of the functions for memory allocation and de-allocation for two reasons. First, the performance of the code provided by the virtual machine is much higher and second, for easier debugging. Currently the virtual machine manages the heap memory with a used and a free list applying the first fit algorithm for memory allocation.

3. Byte Codes in Detail

A byte code instruction has a size of at least one byte. Depending on the instruction one or more subsequent bytes contain its parameters. Table 3.1 summarises the byte code instruction set of the CKPMvm virtual machine.

	0 / 8	1 / 9	2 / A	3 / B	4 / C	5 / D	6 / E	7 / F
0x00	push_b	push_s	push_i	push_sp	push_fp	-	-	-
0x08	ld_b $x	ld_s $x	ld_i $x	ld_l $x	-	ld_f $x	ld_d $x	-
0x10	ld_b $x,fp	ld_s $x,fp	ld_i $x,fp	ld_l $x,fp	-	ld_f $x,fp	ld_d $x,fp	-
0x18	ld_b ($x)	ld_s ($x)	ld_i ($x)	ld_l ($x)	-	ld_f ($x)	ld_d ($x)	-
0x20	ld_b_sp	ld_s_sp	ld_i_sp	ld_l_sp	-	ld_f_sp	ld_d_sp	-
0x28	st_b	st_s	st_i	st_l	-	st_f	st_d	-
0x30	st_b $x,fp	st_s $x,fp	st_i $x,fp	st_l $x,fp	-	st_f $x,fp	st_d $x,fp	-
0x38	st_b ($x)	st_s ($x)	st_i ($x)	st_l ($x)	-	st_f ($x)	st_d ($x)	-
0x40	st_b_sp	st_s_sp	st_i_sp	st_l_sp	-	st_f_sp	st_d_sp	-
0x48	ldc_i_0	-	add_i	add_l	-	add_f	add_d	-
0x50	ldc_i_1	-	sub_i	sub_l	-	sub_f	sub_d	-
0x58	ldc_i_-1	-	mul_i	mul_l	-	mul_f	mul_d	-
0x60	ldc_l_0	-	div_i	div_l	-	div_f	div_d	-
0x68	ldc_l_1	-	rem_i	rem_l	-	rem_f	rem_d	-
0x70	ldc_l_-1	-	neg_i	neg_l	-	neg_f	neg_d	-
0x78	ldc_f_0	-	shl_i	shl_l	-	-	-	-
0x80	ldc_f_1	-	shr_i	shr_l	-	-	-	-
0x88	ldc_f_-1	-	and_i	and_l	-	-	-	-
0x90	ldc_d_0	-	or_i	or_l	-	-	-	-
0x98	ldc_d_1	-	xor_i	xor_l	-	-	-	-
0xA0	ldc_d_-1	-	cmp_i	cmp_l	-	cmp_f	cmp_d	-
0xA8	i2b	i2s	-	i2l	-	i2f	i2d	-
0xB0	-	-	l2i	-	-	l2f	l2d	-
0xB8	-	-	f2i	f2l	-	-	f2d	-
0xC0	-	-	d2i	d2l	-	d2f	-	-
0xC8	-	-	ret_i	ret_l	-	ret_f	ret_d	-
0xD0	pop	pop2	dup	dup2	-	-	-	-
0xD8	-	-	swap	swap2	-	-	-	-
0xE0	-	dec_fp $x	inc_sp,$x	dec_sp,$x	-	read	write	-
0xE8	-	-	inc_i $x	inc_l $x	-	-	-	-
0xF0	ret	jmp	jsr	fopen	fclose	getc	putc	halt
0xF8	beq	bne	bge	bgt	ble	blt	new	del

Table 3.1. Numeric Codes of the Instruction Set

In the following subchapters a specific instruction, with type information, is built by replacing the *T* in the instruction template by the first letter of the corresponding type.

3.1 Load and Store Instructions

The load and store instructions transfer values between the operand stack and both, local and global variables respectively. Push instructions load constant values onto the operand stack.

- Load a variable onto the operand stack: *ld_b*, *ld_s*, *ld_i*, *ld_l*, *ld_f*, *ld_d*.
- Store a value into a variable: *st_b*, *st_s*, *st_i*, *st_l*, *st_f*, *st_d*.
- Load a constant onto the operand stack: *push_b*, *push_s*, *push_i*, *push_sp*, *push_fp*, *ldc_i_<i>*, *ldc_l_<i>*, *ldc_f_<i>*, *ldc_d_<i>*.

3.1.1 push_b – push one byte integer constant onto the operand stack

The immediate *byte* is sign-extended to an `int` value. That value is pushed onto the operand stack.

Format:

push_b	*byte*

Operand Stack: … ⇒ …, value

3.1.2 push_s – push two-byte integer constant

The immediate unsigned *byte1* and *byte2* are combined to a short value that is sign-extended to an `int` value. That value is pushed onto the operand stack.

Format:

push_s	*byte1*	*byte2*

Operand Stack: … ⇒ …, value

3.1.3 push_s – push four-byte integer constant

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value. That value is pushed onto the operand stack.

Format:

push_i	*byte1*	*byte2*	*byte3*	*byte4*

Operand Stack: … ⇒ …, value

3.1.4 push_sp – push the current stack pointer onto the stack

The current stack pointer is pushed onto the operand stack.

Format:

push_sp

Operand Stack: … ⇒ …, value

3.1.5 push_fp – push the current frame pointer onto the stack

The current frame pointer is pushed onto the operand stack.

Format:

push_fp

Operand Stack: … ⇒ …, value

3.1.6 ld_*T* $x – load from absolute address $x

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. The value at this address is read from the memory and pushed onto the operand stack.

Format:

ld_T $x	*byte1*	*byte2*	*byte3*	*byte4*

Operand Stack: … ⇒ …, value

3.1.7 st_*T* $x – store at address $x

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. The value is popped from the operand stack and stored at this address.

Format:

st_T $x	*byte1*	*byte2*	*byte3*	*byte4*

Operand Stack: …, value ⇒ …

3.1.8 ld_*T* $x,fp – load from absolute address $x + FP

The immediate unsigned *byte1* and *byte2* are combined to a `short` value and the content of register FP is added to that value. The result is interpreted as an address in the virtual machine memory. The value at this address is read from the memory and pushed onto the operand stack.

Format:

ld_T $x, fp	*byte1*	*byte2*

Operand Stack: … ⇒ …, value

3.1.9 st_*T* $x,fp – store at address $x + FP

The immediate unsigned *byte1* and *byte2* are combined to a `short` value and the content of register FP is added to that value. The result is interpreted as an address in the virtual machine memory. The value is popped from the operand stack and stored at this address.

Format:

st_T $x, fp	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.1.10 ld_*T* ($x) – load indirect

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. The value at this address is interpreted as a pointer to the memory, which value is read and pushed onto the operand stack.

Format:

ld_T_($x)

Operand Stack: … ⇒ …, value

3.1.11 st_*T* ($x) – store indirect

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. The value at this address is interpreted as a pointer to the memory to which the value from the operand stack is stored.

Format:

st_T_ind

Operand Stack: …, value ⇒ …

3.1.12 ld_*T* $x,sp – load from absolute address $x + SP

The immediate unsigned *byte1* and *byte2* are combined to a `short` value and the content of register SP is added to that value. The result is interpreted as an address in the virtual machine memory. The value at this address is read from the memory and pushed onto the operand stack.

Format:

ld_T $x, sp	*byte1*	*byte2*

Operand Stack: … ⇒ …, value

3.1.13 st_*T* $x,sp – store at address $x + SP

The immediate unsigned *byte1* and *byte2* are combined to a `short` value and the content of register SP is added to that value. The result is interpreted as an address in the virtual machine memory. The value is popped from the operand stack and stored at this address.

Format:

st_T $x, sp	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.1.14 ldc_*T*_<i> – load constant

This instruction pushes a constant <i> onto the operand stack. For each data type *T* exists such an instruction to load the values “-1”, “0”, “1”.

Format: $\boxed{\textit{ldc\_T\_<i>}}$

Operand Stack: … ⇒ …, value

3.2 Arithmetic Instructions

Arithmetic instructions compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack.

- Add: *add_i*, *add_l*, *add_f*, *add_d*.
- Substract: *sub_i*, *sub_l*, *sub_f*, *sub_d*.
- Multiply: *mul_i*, *mul_l*, *mul_f*, *mul_d*.
- Divide: *div_i*, *div_l*, *div_f*, *div_d*.
- Remainder: *rem_i*, *rem_*l, *rem_f*, *rem_d*.
- Negate: *neg_i*, *neg_l*, *neg_f*, *neg_d*.
- Shift: *shl_i*, *shl_l*, *shr_i*, *shr_l*.
- Bitwise OR: *or_i*, *or_l*.
- Bitwise AND: *and_i*, *and_l*.
- Bitwise exclusive OR: *xor_i*, *xor_l*.
- Comparison: *cmp_i*, *cmp_l*, *cmp_f*, *cmp_d*.
- Increment: *inc_i*, *inc_l*
- Decrement: *dec_fp*, *inc_sp*, *dec_sp*

3.2.1 add_*T* – add

This instruction takes two variables from the operand stack adds them and pushes the result onto the stack.

Format: $\boxed{\textit{add\_T}}$

Operand Stack: …, v1, v2 ⇒ …, result

3.2.2 sub_*T* – subtract

This instruction takes two variables from the operand stack subtracts them and pushes the result onto the stack.

$\boxed{\textit{sub\_T}}$

Format:

Operand Stack: …, v1, v2 ⇒ …, result

3.2.3 mul_*T* – multiply

This instruction takes two variables from the operand stack multiplies them and pushes the result onto the stack.

Format: | *mul_T* |

Operand Stack: …, v1, v2 ⇒ …, result

3.2.4 div_*T* – divide

This instruction takes two variables from the operand stack divides them and pushes the result onto the stack.

Format: | *div_T* |

Operand Stack: …, v1, v2 ⇒ …, result

3.2.5 rem_*T* – remainder

This instruction takes two variables from the operand stack divides them and pushes the remainder onto the stack.

Format: | *rem_T* |

Operand Stack: …, v1, v2 ⇒ …, result

3.2.6 neg_*T* – negate

This instruction negates the value on top of the stack.

Format: | *neg_T* |

Operand Stack: …, value ⇒ …, (-value)

3.2.7 shl_*T* – shift left

This instruction shifts an integer or a long value v1 to the left by v2 bits and pushes the result on top of the stack.

Format: *shl_T*

Operand Stack for *shl_i*: …, v1, v2 ⇒ …, result

Operand Stack for *shl_l*: …, v1.word1, v1.word2, v2 ⇒ …, result.word1, result.word2

3.2.8 shr_*T* – shift right

This instruction shifts an integer or a long value v1 to the right by v2 bits and pushes the result on top of the stack.

Format: *shr_T*

Operand Stack for *shr_i*: …, v1, v2 ⇒ …, result

Operand Stack for *shr_l*: …, v1.word1, v1.word2, v2 ⇒ …, result.word1, result.word2

3.2.9 and_*T* – bitwise AND

This instruction performs a bitwise AND for types integer and long.

Format: *and_T*

Operand Stack for *and_i*: …, v1, v2 ⇒ …, result

Operand Stack for *and_l*: …, v1.word1, v1.word2, v2.word1, v2.word2 ⇒ …, r.word1, r.word2

3.2.10 or_*T* – bitwise OR

This instruction performs a bitwise OR for types integer and long.

Format: *or_T*

Operand Stack for *or_i*: …, v1, v2 ⇒ …, result

Operand Stack for *or_l*: …, v1.word1, v1.word2, v2.word1, v2.word2 ⇒ …, r.word1, r.word2

3.2.11 xor_*T* – bitwise XOR

This instruction performs a bitwise XOR for types integer and long.

Format: *xor_T*

Operand Stack for x*or_i*: …, v1, v2 ⇒ …, result

Operand Stack for x*or_l*: …, v1.word1, v1.word2, v2.word1, v2.word2 ⇒ …, r.word1, r.word2

3.2.12 cmp_*T* – comparison

This instruction compares the two variables on top of the operand stack and pushes -*1* for *v1 < v2*, *0* for *v1 == v2* or *1* for *v1 > v2* as result onto the stack.

Format:

cmp_T

Operand Stack: …, v1, v2 ⇒ …, result

3.2.13 inc_*T* $x – increment at absolute address $x with type *T*

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. The value at this address is incremented by one.

Format:

inc_T $x	*byte1*	*byte2*	*byte3*	*byte4*

Operand Stack: … ⇒ …

3.2.14 dec_fp $x – decrement frame pointer register

The immediate unsigned *byte1* and *byte2* are combined to a `short` value that is subtracted from the FP register.

Format:

dec_fp $x	*byte1*	*byte2*

Operand Stack: … ⇒ …

3.2.15 inc_sp $x – increment stack pointer register

The immediate unsigned *byte1* and *byte2* are combined to a `short` value that is added to the SP register.

Format:

inc_sp $x	*byte1*	*byte2*

Operand Stack: … ⇒ …

3.2.16 dec_sp $x – decrement stack pointer register

The immediate unsigned *byte1* and *byte2* are combined to a `short` value that is subtracted from the SP register.

Format:

dec_sp $x	*byte1*	*byte2*

Operand Stack: … ⇒ …

3.3 Type Conversion Instructions

The type conversion instructions allow conversion between the virtual machine numeric types. The virtual machine directly supports the following narrowing and widening conversions:

- `int` to `long`, `float` or `double`: *i2b*, *i2s*, *i2l*, *i2f*, *i2d*.
- `long` to `int`, `float` or `double`: *l2i*, *l2f*, *l2d*.
- `float` to `int`, `long` or `double`: *f2i*, *f2l*, *f2d*.
- `double` to `int`, `long` or `float`: *d2i*, *d2l*, *d2f*.

3.3.1 i2*T* – integer to type *T*

This instruction converts an integer value to a value of type *T*.

Format:

i2T

Operand Stack for *i2b*: …, value ⇒ …, result

Operand Stack for *i2s*: …, value ⇒ …, result

Operand Stack for *i2l*: …, value ⇒ …, result.word1, result.word2

Operand Stack for *i2f*: …, value ⇒ …, result

Operand Stack for *i2d*: …, value ⇒ …, result.word1, result.word2

3.3.2 l2*T* – long to type *T*

This instruction converts a long value to a value of type *T*.

Format:

l2T

Operand Stack for *l2i*: …, v1.word1, v1.word2 ⇒ …, result

Operand Stack for *l2f*: …, v1.word1, v1.word2 ⇒ …, result

Operand Stack for *l2d*: …, v1.word1, v1.word2 ⇒ …, result.word1, result.word2

3.3.3 f2*T* – float to type *T*

This instruction converts a float value to a value of type *T*.

Format: *f2T*

Operand Stack for *f2i*: …, value ⇒ …, result

Operand Stack for *f2l*: …, value ⇒ …, result.word1, result.word2

Operand Stack for *f2d*: …, value ⇒ …, result.word1, result.word2

3.3.4 d2*T* – double to type *T*

This instruction converts a float value to a value of type *T*.

Format: *d2T*

Operand Stack for *d2i*: …, v1.word1, v1.word2 ⇒ …, result

Operand Stack for *d2l*: …, v1.word1, v1.word2 ⇒ …, result.word1, result.word2

Operand Stack for *d2f*: …, v1.word1, v1.word2 ⇒ …, result

3.4 Memory Management

Two instructions are responsible for allocating and freeing memory from the heap: *new*, *del*.

3.4.1 new – allocate memory

This instruction allocates memory of size *value* bytes on the heap of the virtual machine and pushes the first address of the allocated memory as result onto the stack.

Format: *new*

Operand Stack: …, value ⇒ …, address

3.4.2 del – free memory

This instruction frees a previously allocated chunk of heap memory.

Format: *del*

Operand Stack: …, address ⇒ …

3.5 Operand Stack Manipulation

A quantity of instructions is provided for direct manipulation of the operand stack: *pop*, *pop2*, *dup*, *dup2*, *swap*, *swap2*.

3.5.1 pop – pop word

This instruction pops the top word from the stack.

Format: *pop*

Operand Stack: …, word ⇒ …

3.5.2 pop2 – pop double word

This instruction pops the top two words from the stack.

Format: *pop2*

Operand Stack: …, word1, word2 ⇒ …

3.5.3 dup – duplicate word

This instruction duplicates the top word on the stack.

Format: *dup*

Operand Stack: …, word ⇒ …, word, word

3.5.4 dup2 – duplicate double word

This instruction duplicates the top two words on the stack.

Format: *dup2*

Operand Stack: …, word1, word2 ⇒ …, word1, word2, word1, word2

3.5.5 swap – swap word

This instruction swaps the two top words on the stack.

Format: | *dup* |
|---|

Operand Stack: …, word1, word2 ⇒ …, word2, word1

3.5.6 swap2 – swap double word

This instruction swaps the top two double words on the stack.

Format: | *dup2* |
|---|

Operand Stack: …, word1, word2, word3, word4 ⇒ …, word3, word4, word1, word2

3.6 Control Transfer Instructions

The control transfer instructions conditionally or unconditionally cause the virtual machine to continue the execution with an instruction other than the one following the control transfer instruction. They are:

- Conditional branch: *beq*, *bne*, *bgt*, *bge*, *blt*, *ble*.
- Unconditional branch: *jmp*, *jsr*, *ret*, *ret_i*, *ret_l*, *ret_f*, *ret_d*.
- Machine stop: *halt*

3.6.1 beq – branch on equal

The immediate unsigned *byte1* and *byte2* are combined to a sign-extended short value that is added to the program counter if the value on the stack equals to zero.

Format:

beq	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.6.2 bne – branch on not equal

The immediate unsigned *byte1* and *byte2* are combined to a sign-extended short value that is added to the program counter if the value on the stack equals not to zero.

Format:

bne	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.6.3 bgt – branch on greater

The immediate unsigned *byte1* and *byte2* are combined to a sign-extended short value that is added to the program counter if the value on the stack is greater than zero.

Format:

bgt	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.6.4 bge – branch on greater or equal

The immediate unsigned *byte1* and *byte2* are combined to a sign-extended short value that is added to the program counter if the value on the stack is positive or equal zero.

Format:

bge	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.6.5 blt – branch on lower

The immediate unsigned *byte1* and *byte2* are combined to a sign-extended short value that is added to the program counter if the value on the stack is lower than zero.

Format:

bgt	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.6.6 ble – branch on lower or equal

The immediate unsigned *byte1* and *byte2* are combined to a sign-extended short value that is added to the program counter if the value on the stack is negative or equal zero.

Format:

ble	*byte1*	*byte2*

Operand Stack: …, value ⇒ …

3.6.7 jmp – jump unconditionally

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. This instruction sets the program counter to this value.

Format:

jmp	*byte1*	*byte2*	*byte3*	*byte4*

Operand Stack: *unchanged*

3.6.8 jsr – jump to subroutine

The immediate unsigned *byte1*, *byte2*, *byte3* and *byte4* are combined to an `int` value that is interpreted as an address in the virtual machine memory. This instruction pushes the address of the succeeding instruction and the content of the register FP on the stack. Subsequently it sets the FP register to SP-2 and jumps to the subroutine by setting the program counter to the given address.

Format:

jmp	*byte1*	*byte2*	*byte3*	*byte4*

Operand Stack: … ⇒ …, return-address, FP

3.6.9 ret – return from subroutine

This instruction returns from a subroutine without returning a value. It sets register FP to the old value on the stack, pops the return address and sets the program counter to this address.

Format:

ret

Operand Stack: …, return-address, FP ⇒ …

3.6.10 ret_*T* – return from subroutine with value

This instruction returns from a subroutine returning a value of type *T*. It sets register FP to the old value on the stack, pops the return address, pushes the return value and sets the program counter to the return address.

Format:

ret_T

Operand Stack: …, return-address, FP ⇒ …, value

3.6.11 halt – stop the virtual machine

This instruction halts the virtual machine. The value on the stack is used as exit code for the virtual machine.

Format: *halt*

Operand Stack: …, value ⇒ …

3.7 Input and Output Instructions

The input and output instructions allow interaction to files outside the virtual machine:

- Open a file: *fopen*
- Close a file: *fclose*
- Get one character from the file: *getc*
- Put one character to the file: *putc*
- Read from file: *read*
- Write to file: *write*

3.7.1 fopen – file open

This instruction pops the address to a zero terminated string containing file name, the access mode flags, as well as the permission modes for file creation from the stack, opens the file and pushes a file handle onto the stack. If the given file can not be opened, the value *-1* is pushed.

Format: *fopen*

Operand Stack: …, address, flags, mode ⇒ …, file-handle

3.7.2 fclose – file close

This instruction pops a file handle from the stack and closes the associated file. It pushes the value *0* for success and *-1* if an error occurs, respectively.

Format: *fclose*

Operand Stack: …, file-handle ⇒ …, value

3.7.3 getc – get character

This instruction reads one character from a file and pushes it onto the stack. On end of file, the value *-1* is pushed.

Format: | *getc* |

Operand Stack: …, file-handle ⇒ …, value

3.7.4 putc – put character

This instruction writes one character to a file and pushes *0* for success or *-1* for failure onto the stack.

Format: | *putc* |

Operand Stack: …, file-handle, character ⇒ …, result

3.7.5 read – read from file

This instruction reads a byte chunk from a file stores it at a given address and pushes the number of read bytes onto the operand stack.

Format: | *read* |

Operand Stack: …, file-handle, buffer address, buffer length ⇒ …, value

3.7.6 write – write to file

This instruction writes a byte chunk from to a file and pushes the number of written bytes onto the operand stack.

Format: | *write* |

Operand Stack: …, file-handle, buffer address, buffer length ⇒ …, value

4. Object File Format

This chapter describes the object file format, called CKLF (CK Linking Format). A CKLF file holds code and data suitable for linking with other object files and execution, respectively. Figure 2 shows an object file's organisation.

CKLF Header
Code Segment
Data Segment
Symbol Table
String Table

Figure 2 The CKLF Object File Format

The CKLF header resides at the beginning and holds a description of the file's organisation. The code segment holds virtual machine instructions and the data segment contains initialised variables. The symbol table holds a list of exported, imported and local symbols as well as references to the string table for the symbol's names.

4.1 CKLF Header

The object file sections can be of variable size, because the CKLF header contains their actual position.

```
#define     LD_MAX_IDENT      8

struct _cklf_hdr {
   char     ident[LD_MAX_IDENT];
   int      version;
   int      prog;
   int      data;
   int      symtab;
   int      strtab;
   int      total_sz;
};
```

Figure 3. The CKLF Header

ident The initial bytes mark the file as an CKLF object file. It always contains the eight byte long identification string `“\177CKLF   “` (‘\0’ included).

version This member identifies the object file version. The value 1 identifies the original file format. Extensions will create new version s with higher numbers.

prog This member holds the code segment's file offset in bytes.

data This member holds the data segment's file offset in bytes.

symtab This member holds the symbol table's file offset in bytes.

strtab This member holds the string table's file offset in bytes.

total_sz This member holds the total size of the CKLF object file.

4.2 Code Segment and Data Segment

The code segment contains the virtual machine executable code. The virtual machine fixes the addresses to subroutines and the global data are at loading time.

4.3 Symbol Table

The symbol table section can be of variable size, suitable to the number of local and global symbols. The symbol table is an array of table entries shown in Figure 4.

```
struct _cklf_symtab {
   int          name;
   int          addr;
   int          offs;
   st_type_t    type;
   st_bind_t    bind;
};
```

Figure 4. The symbol table entry format

`name` This member specifies the name of the symbol. Its value is an index into the string table section, giving the location of a null-terminated string.

`addr` This members value gives the byte offset of the symbol in the section it refers to. For external symbols it contains zero.

`offs` This member refers to the first member of the fix-up chain in the code segment. The value at this address in the code segment refers to the next address to be fixed up. The last location in the fix-up chain contains the value zero. This member contains zero if the symbol is not referred to in the code segment.

`type` This member categorises the symbol’s type as follows:

`STT_NOTYPE` This value marks the symbol as an imported symbol.

`STT_OBJECT` The symbol refers to a chunk in the data segment.

`STT_FUNC` The symbol refers to a subroutine in the code segment.

`bind` This member specifies the symbol’s binding scope as follows:

`STB_LOCAL` The specified symbol is valid only in the local scope and can not be accessed externally.

`STB_GLOBAL` The specified symbol is an exported symbol that can referred to externally.

4.4 String Table

String table sections hold null-terminated character sequences, commonly called strings. The CKLF file uses these strings to represent symbol names. One references a string as an index into the string table section. The string table's last byte is defined to hold a null character, ensuring null termination of all strings. The following Table 4.1 and Table 4.2 show a string table with 13 bytes and the strings associated with various indexes.

Index	**+0**	**+1**	**+2**	**+3**	**+4**	**+5**	**+6**	**+7**	**+8**	**+9**
0	m	a	i	n	\0	f	o	o	\0	b
10	a	r	\0	\0						

Table 4.1 An example string table

Index	**String**
0	main
5	foo
9	bar
13	*null string*

Table 4.2 String Table Indexes

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once, references to substrings may exist, and a single string may be referenced multiple times. Unreferenced strings are also allowed.

5. References

[1] Lindholm, T. and Yellin, F. (1999). *The Java™ Virtual Machine Specification, Second Edition.* Addison Wesely Professional.

[2] Lamay, L. and Perkins, C. (1997). *Teach yourself Java™ 1.1 in 21 days, Second Edition.* Sams.net Publishing, Indianapolis, Indiana.