A No-Frills Introduction to Lua 5 VM Instructions

by Kein-Hong Man, esq. <khman AT users.sf.net>

Version 0.3, 20060221

Contents

1 Introduction

This is a no-frills introduction to the instruction set of the Lua 5 virtual machine. Compared to Perl or Python, the compactness of Lua makes it relatively easier for someone to peek under the hood and understand its internals. I think that one cannot completely grok a scripting language, or any complex system for that matter, without slitting the animal open and examining the entrails, organs and other yucky stuff that isn't normally seen. So this document is supposed to help with the "peek under the hood" bit.

Output from ChunkSpy (URL: `http://luaforge.net/projects/chunkspy/`), a Lua 5 binary chunk disassembler which I wrote while studying Lua internals, was used to generate the examples shown in this document. The brief disassembly mode of ChunkSpy is very similar to the output of the listing mode of `luac`, so you do not need to learn a new listing syntax. ChunkSpy can be downloaded from LuaForge (URL: `http://luaforge.net/`); it is licensed under the same type of MIT-style license as Lua 5 itself.

ChunkSpy has an interactive mode: you can enter a source chunk and get an immediate disassembly. This allows you to use this document as a tutorial by entering the examples into ChunkSpy and seeing the results yourself. The interactive mode is also very useful when you are exploring the behaviour of the Lua code generator on many short code snippets.

This is a quick introduction, so it isn't intended to be a comprehensive or expert treatment of the Lua virtual machine (from this point on, "Lua" refers to "Lua 5" unless otherwise stated) or its instructions. It is intended to be a simple, easy-to-digest beginner's guide to the Lua virtual machine instruction set – it won't do cartwheels or blow smoke rings.

The objective of this introduction is to cover all the Lua virtual machine instructions and the structure of Lua 5 binary chunks with a minimum of fuss. Then, if you want more detail, you can use `luac` or ChunkSpy to study non-trivial chunks of code, or you can dive into the Lua source code itself for the real thing.

This is currently a draft, and I am not a Lua internals expert. So feedback is welcome. If you find any errors, or if you have anything to contribute please send me an e-mail (to khman AT users.sf.net *or* mkh AT pl.jaring.my*) so that I can correct it. Thanks.*

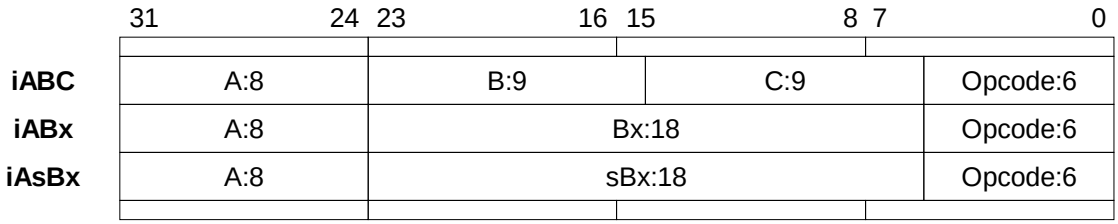
2 Lua Instruction Basics

The Lua virtual machine instruction set we will look at is a particular *implementation* of the Lua language. It is by no means the only way to skin the chicken. The instruction set just happens to be the way the authors of Lua chose to implement version 5 of Lua. The following sections are based on the instruction set used in Lua 5.0.2. The instruction set might change in the future – do not expect it to be set in stone. This is because the implementation details of virtual machines are not a concern to most users of scripting languages. For most applications, there is no need to specify how bytecode is generated or how the virtual machine runs, as long as the language works as advertised. So remember that there is no official specification of the Lua virtual machine instruction set, there is no need for one; the only official specification is of the Lua language.

In the course of studying disassemblies of Lua binary chunks, you will notice that many generated instruction sequences aren't as perfect as you would like them to be. This is perfectly normal from an engineering standpoint. The canonical Lua implementation is not meant to be an optimizing bytecode compiler or a JIT compiler. Instead it is supposed to load, parse and run Lua source code efficiently. It is the totality of the implementation that counts. If you really need the performance, you are supposed to drop down into native C functions anyway.

Lua instructions have a fixed size, 32 bits by default. Instructions are manipulated using the platform's native integer data type, which is usually a 32-bit signed integer on 32-bit machines. In binary chunks, endianness is significant, but while in memory, an instruction can be portably decoded or encoded in C using the usual integer shift and mask operations. The details can be found in `lopcodes.h`.

There are three instruction types and 35 opcodes (numbered 0 through 34) are currently in use as of Lua 5.0.2. The instruction types are enumerated as iABC, iABx, iAsBx, and may be visually represented as follows:



Lua 5 Instruction Formats

Instructions are encoded with unsigned integer fields, except for sBx. Field sBx can represent negative numbers, but it doesn't use 2s complement. Instead, it has a bias equal to half the maximum integer that can be represented by its unsigned counterpart, Bx. For a field size of 18 bits, Bx can hold a maximum integer value of 262143, and so the bias is 131071. A value of -1 will be encoded as (-1 + 131071) or 131070 or 1FFFE in hexadecimal.

Fields A, B and C usually refers to register numbers (I'll use the term "register" because of its similarity to processor registers). Although field A is the target operand in arithmetic operations, this rule isn't always true for other instructions. A register is really an index into the current stack frame, register 0 being the bottom-of-stack position.

Unlike the Lua C API, negative indices (counting from the top of stack) are not supported. For some instructions, where the top of stack may be required, it is encoded as a special operand value. Local variables are equivalent to certain registers in the current stack frame, while dedicated opcodes allow read/write of globals and upvalues.

Beyond a certain threshold, a value in fields B or C may become an encoding of the number of a constant in the constant pool of a function. By default, Lua has a maximum stack frame size of 250. This is encoded as `MAXSTACK` in `llimits.h`. So, a value of 251 in field B means that the operand is constant number 1 from the constant pool.

The maximum stack frame size in turn limits the maximum number of locals, and the limit is set at 200, encoded as `MAXVARS` in `llimits.h`. It is a useful bit of information to know, especially if you are doing something in Lua that pushes its capabilities to the limit. Other limitations found in `llimits.h` include the maximum number of upvalues (32), encoded as `MAXUPVALUES`, and the maximum number of parameters in a function (100), encoded as `MAXPARAMS`.

A summary of the Lua 5 virtual machine instruction set is as follows:

Opcode	Name	Description
0	MOVE	Copy a value between registers
1	LOADK	Load a constant into a register
2	LOADBOOL	Load a boolean into a register
3	LOADNIL	Load **nil** values into a range of registers
4	GETUPVAL	Read an upvalue into a register
5	GETGLOBAL	Read a global variable into a register
6	GETTABLE	Read a table element into a register
7	SETGLOBAL	Write a register value into a global variable
8	SETUPVAL	Write a register value into an upvalue
9	SETTABLE	Write a register value into a table element
10	NEWTABLE	Create a new table
11	SELF	Prepare an object method for calling
12	ADD	Addition
13	SUB	Subtraction
14	MUL	Multiplication
15	DIV	Division
16	POW	Exponentiation
17	UNM	Unary minus
18	NOT	Logical NOT
19	CONCAT	Concatenate a range of registers
20	JMP	Unconditional jump
21	EQ	Equality test
22	LT	Less than test
23	LE	Less than or equal to test
24	TEST	Test for short-circuit logical **and** and **or** evaluation
25	CALL	Call a closure
26	TAILCALL	Perform a tail call
27	RETURN	Return from function call
28	FORLOOP	Iterate a numeric **for** loop
29	TFORLOOP	Iterate a generic **for** loop
30	TFORPREP	Initialization for a generic **for** loop
31	SETLIST	Set a range of array elements for a table
32	SETLISTO	Set a variable number of table elements
33	CLOSE	Close a range of locals being used as upvalues
34	CLOSURE	Create a closure of a function prototype

3 Really Simple Chunks

Before heading into binary chunk and virtual machine instruction details, this section will demonstrate briefly how ChunkSpy can be used to explore Lua 5 code generation. All the examples in this document were produced using ChunkSpy 0.9.4.

First, start ChunkSpy in interactive mode (user input is set in bold):

```
$ lua ChunkSpy.lua --interact
ChunkSpy: A Lua 5 binary chunk disassembler with no dependencies
Version 0.9.4 (20041121)  Copyright (c) 2004 Kein-Hong Man
The COPYRIGHT file describes the conditions under which this
software may be distributed (basically a Lua 5-style license.)

Type 'exit' or 'quit' to end the interactive session. 'help' displays
this message. ChunkSpy will attempt to turn anything else into a
binary chunk and process it into an assembly-style listing.
A '\' can be used as a line continuation symbol; this allows multiple
lines to be strung together.

>
```

We'll start with the shortest possible binary chunk that can be generated:

```
>do end
; source chunk: (interactive mode)
; x86 standard (32-bit, little endian, doubles)

; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
[1] return     0   1
; end of function
```

ChunkSpy will treat your keyboard input as a small chunk of Lua source code. The library function `string.dump()` is first used to generate a binary chunk string, then ChunkSpy will disassemble that string and give you a brief assembly language-style output listing.

Some features of the listing: Comment lines are prefixed by a semicolon. The header portion of the binary chunk is not displayed with the brief style. Data or header information that isn't an instruction is shown as an assembler directive with a dot prefix. `luac`-style comments are generated for some instructions, and the instruction location is in square brackets.

A "`do end`" generates a single RETURN instruction and does nothing else. There are no parameters, locals, upvalues or globals. For the rest of the disassembly listings shown in this document, we will omit some common header comments and show only the function disassembly part. Instructions will be referenced by its marked position, e.g. line [1]. Here is another very short chunk:

```
>return
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
[1] return     0   1
[2] return     0   1
; end of function
```

A RETURN instruction is generated for every **return** in the source. The first RETURN (line [1]) is generated by the **return** keyword, while the second RETURN (line [2]) is always added by the code generator. This isn't a problem, because the second RETURN never gets executed anyway, and only 4 bytes is wasted. Perfect generation of RETURN instructions requires basic block analysis, and it is not done because there is no performance penalty for an extra RETURN, only a negligible memory penalty.

Notice in these examples, the minimum stack size is 2, even when the stack isn't used. The next snippet assigns a constant value of 6 to the global variable **a**:

```
>a=6
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.const  "a"  ; 0
.const  6  ; 1
[1] loadk      0   1        ; 6
[2] setglobal  0   0        ; a
[3] return     0   1
; end of function
```

All string and number constants are pooled on a per-function basis, and instructions refer to them using an index value which starts from 0. Global variable names need a constant string as well, because globals are maintained as a table. Line [1] loads the value 6 (with an index to the constant pool of 1) into register 0, then line [2] sets the global table with the constant "a" (constant index 0) as the key and register 0 (holding the number 6) as the value.

If we write the variable as a local, we get:

```
>local a="hello"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.const  "hello"  ; 0
[1] loadk      0   0        ; "hello"
[2] return     0   1
; end of function
```

Local variables reside in the stack, and they occupy a stack (or register) location for the duration of their existence. The scope of a local variable is specified by a starting program counter location and an ending program counter location; this is not shown in a brief disassembly listing.

The local table in the function tells the user that register 0 is variable **a**. This information doesn't matter to the VM, because it needs to know register numbers only – register allocation was supposed to have been properly done by the code generator. So LOADK in line [1] loads constant 0 (the string "hello") into register 0, which is the local variable **a**. A stripped binary chunk will not have local variable names for debugging.

Next we will take a look at the structure of Lua 5 binary chunks.

4 Lua Binary Chunks

Lua can dump functions as binary chunks, which can then be written to a file, loaded and run. Binary chunks behave exactly like the source code from which they were compiled.

A binary chunk consist of two parts: a header block and a top-level function. The header portion contains 12 elements:

Header block of a Lua 5 binary chunk
Default values are for a 32-bit little-endian platform with IEEE 754 doubles as the number format. The header size on this platform is 22 bytes.

4 bytes	Header signature: ESC, “Lua” • Binary chunk recognized by checking for ESC only
1 byte	Version number, 0x50 (80 decimal) for Lua 5.0.2 • High hex digit is major version number • Low hex digit is minor version number
1 byte	Endianness • 0=big endian, 1=little endian
1 byte	Size of int (in bytes) (default 4)
1 byte	Size of size_t (in bytes) (default 4)
1 byte	Size of Instruction (in bytes) (default 4)
1 byte	Size of OP field (in bits) (default 6)
1 byte	Size of A field (in bits) (default 8)
1 byte	Size of B field (in bits) (default 9)
1 byte	Size of C field (in bits) (default 9)
1 byte	Size of size of a Lua number (in bytes) (default 8)
8 bytes*	Test number (encoding of 3.14159265358979323846E7) • The only field in the header that is endian-dependent.

* Changes depending on the field size given in the global header.

Thus the first 14 bytes of a Lua 5 binary chunk have fixed locations. Since the characteristics of a Lua virtual machine is hard-coded, the Lua undump code has to check the header bytes and determine whether the binary chunk is fit for consumption or not. If you have a binary chunk that does not match the characteristics of the Lua platform you want to run it on, then Lua will usually refuse to load the chunk.

In theory, a Lua binary chunk is portable; in real life, there is no need for the undump code to support such a feature. If you need undump to load all kinds of binary chunks, you are probably doing something wrong. If however you somehow need this feature, you can try ChunkSpy’s rewrite option, which allows you to convert a binary chunk from one profile to another.

Anyway, most of the time there is little need to seriously scrutinize the header, because since Lua source code is usually available, a chunk can be readily compiled into the native binary chunk format.

The header block is followed immediately by the top-level function or chunk:

Function block of a Lua 5 binary chunk
Holds all the relevant data for a function. There is one top-level function.

String	source name
Integer	line defined
1 byte	number of upvalues
1 byte	number of parameters
1 byte	vararg function flag, true if non-zero
1 byte	maximum stack size (number of registers used)
List	source line positions for each instruction
List	list of locals
List	list of upvalues
List	list of constants
List	list of function prototypes
List	list of instructions (code)

A function block in a binary chunk defines the prototype of a function. To actually execute the function, Lua creates an instance (or *closure*) of the function first. A function in a binary chunk consist of a few header elements and a bunch of lists.

A **String** is defined in this way:

All strings are defined in the following format:

Size_t	String data size
Bytes	String data, includes a NUL (ASCII 0) at the end

The string data size takes into consideration a NUL character at the end, so an empty string (“”) has 1 as the size_t value. A size_t of 0 means zero string data bytes; the string does not exist. This is often used by the source name field of a function.

The **source name** is usually the name of the source file from which the binary chunk is compiled. It may also refer to a string. This source name is specified only in the top-level function; in other functions, this field consists only of a **Size_t** with the value 0.

The **line defined** is the line number where the function prototype was defined. Next comes the number of upvalues, the number of parameters, a boolean flag to show whether the function is a vararg function (a true is encoded as a 1,) and a maximum stack size, all single-byte values.

After the function header elements comes a number of lists that store the information that makes up the body of the function. Each list starts with an **Integer** as a list size count, followed by a number of list elements. Each list has its own element format. A list size of 0 has no list elements at all.

In the following boxes, a data type in square brackets, e.g. **[Integer]** means that there are multiple numbers of the element, in this case an integer. The count is given by the list size. Names in parentheses are the ones given in the Lua sources; they are data structure fields.

Source line position list
Holds the source line number for each corresponding instruction in a function. This information is used by error handlers or debuggers. In a stripped binary, the size of this list is zero. The execution of a function does not depend on this list.

Integer	size of source line position list (sizelineinfo)
[Integer]	list index corresponds to instruction position; the integer value is the line number of the Lua source where the instruction was generated

Next up is the locals list. Each local variable entry has 3 fields, a string and two integers:

Locals list
Holds list of local variable names and the program counter range in which the local variable is active.

Integer	size of locals list (sizelocvars)
[	
String	name of local variable (varname)
Integer	start of local variable scope (startpc)
Integer	end of local variable scope (endpc)
]	

The upvalues list and the constants list follows locals:

Upvalues list
Holds list of upvalue names.

Integer	size of upvalues list (sizeupvalues)
[String]	name of upvalue

Constants list
Holds list of constants (it's a constant pool.)

Integer	size of constants list (sizek)
[	
1 byte	type of constant (value in parentheses): LUA_TNIL (0), LUA_TNUMBER (3) or LUA_TSTRING (4)
Const	the constant itself: this field does not exist if the constant type is 0; it is a **Number** for type 3, and a **String** for type 4.
]	

Number is the Lua number data type, normally an IEEE 754 64-bit double. **Integer**, **Size_t** and **Number** are all endian-sensitive, and they are converted into the native endian during the binary chunk loading, or undump, process. Their sizes and formats are of course specified in the binary chunk header.

The function prototypes list comes after the constants list:

Function prototypes list Holds function prototypes defined within the function.	
Integer	size of function prototypes (sizep)
[Functions]	function prototype data, or function blocks

Function prototypes or function blocks have the exact same format as the top-level function or chunk. However, function prototypes that isn't the top-level function do not have the source name field defined. In this way, function prototypes at different lexical scoping levels are defined and nested. In a complex binary chunk, the nesting may be several levels deep. A closure will refer to a function by its number in the list.

The final list is the instruction list, or the actual code to the function. This is the list of instructions that will actually be executed:

Instructions list Holds list of instructions that will be executed.	
Integer	size of code (sizecode)
[Instruction]	virtual machine instructions

The format of the virtual machine instructions were given in the last chapter. All of these lists are not shared or re-used between functions: Locals, upvalues and constants referenced in the code are specified in the respective lists in the same function. A RETURN instruction is always generated by the code generator, so the size of the instructions list should be at least 1.

In addition, locals, upvalues, constants and the function prototypes are indexed using numbers starting from 0. In disassembly listings, both the source line position list and the instructions list are indexed starting from 1. Although all jump-related instructions use only signed displacements, the scope of local variables is encoded using absolute program counter positions, and these positions are based on a starting index of 1. This is also consistent with the output from `luac`.

How does it all fit in? You can easily generate a detailed binary chunk disassembly using ChunkSpy. Enter the following short bit of code and name the file `simple.lua`:

```
local a = 8
function b(c) d = a + c end
```

Next, run ChunkSpy from the command line to generate the listing:

```
$ lua ChunkSpy.lua --source simple.lua > simple.lst
```

The following is a description of the generated listing, split into segments.

```
Pos   Hex Data           Description or Code
------------------------------------------------------------------------
0000                     ** source chunk: simple.lua
                         ** global header start **
0000  1B4C7561           header signature: "\27Lua"
0004  50                 version (major:minor hex digits)
0005  01                 endianness (1=little endian)
0006  04                 size of int (bytes)
0007  04                 size of size_t (bytes)
0008  04                 size of Instruction (bytes)
0009  06                 size of OP (bits)
000A  08                 size of A (bits)
000B  09                 size of B (bits)
000C  09                 size of C (bits)
000D  08                 size of number (bytes)
000E  B6099368E7F57D41   sample number (double)
                         * x86 standard (32-bit, little endian, doubles)
                         ** global header end **
```

This is an example of a binary chunk header. ChunkSpy calls this the global header to differentiate it from a function header. For binary chunks specific to a certain platform, it is easy to match the entire header at one go instead of testing each field.

The global header is followed by the function header of the top-level function:

```
0016                     ** function [0] definition (level 1)
                         ** start of function **
0016  0B000000           string size (11)
001A  73696D706C652E6C+  "simple.l"
0022  756100             "ua\0"
                         source name: simple.lua
0025  00000000           line defined (0)
0029  00                 nups (0)
002A  00                 numparams (0)
002B  00                 is_vararg (0)
002C  02                 maxstacksize (2)
```

The source name is only present in the top-level function. A top-level chunk does not have a line number on which it is defined, so the field is 0. There are no upvalues or parameters either, and it does not accept a variable number of parameters. The stack size is set at the minimum of 2 for this very simple chunk.

Next we come to the various lists, starting with the source line position list:

```
                         * lines:
002D  05000000           sizelineinfo (5)
                         [pc] (line)
0031  01000000           [1] (1)
0035  02000000           [2] (2)
0039  02000000           [3] (2)
003D  02000000           [4] (2)
0041  02000000           [5] (2)
```

There are 5 instructions in the top-level function. The source for the first instruction was defined on line 1, while the other 4 instructions were defined on line 2.

```
                        * locals:
0045  01000000          sizelocvars (1)
0049  02000000          string size (2)
004D  6100              "a"
                        local [0]: a
004F  01000000            startpc (1)
0053  04000000            endpc   (4)
                        * upvalues:
0057  00000000          sizeupvalues (0)
```

The top-level function has one local variable, named “a”, active from location 1 to location 4, and it refers to register 0. There are no upvalues.

```
                        * constants:
005B  02000000          sizek (2)
005F  03                const type 3
0060  0000000000002040  const [0]: (8)
0068  04                const type 4
0069  02000000          string size (2)
006D  6200              "b"
                        const [1]: "b"
```

The top-level function requires two constants, the number 8 (which is used in the assignment on line 1) and the string “b” (which is used to refer to the global variable **b** on line 2.)

On line 2 of the source, a function prototype was declared. The function prototype list holds all the relevant information, a function block within a function block. ChunkSpy reports it as function prototype number 0, at level 2. Level 1 is the top-level function; there is only one level 1 function, but there may be more than one function prototype at other levels.

```
                        * functions:
006F  01000000          sizep (1)

0073                    ** function [0] definition (level 2)
                        ** start of function **
0073  00000000          string size (0)
                        source name: (none)
0077  02000000          line defined (2)
007B  01                nups (1)
007C  01                numparams (1)
007D  00                is_vararg (0)
007E  02                maxstacksize (2)
                        * lines:
007F  04000000          sizelineinfo (4)
                        [pc] (line)
0083  02000000          [1] (2)
0087  02000000          [2] (2)
008B  02000000          [3] (2)
008F  02000000          [4] (2)
                        * locals:
0093  01000000          sizelocvars (1)
0097  02000000          string size (2)
009B  6300              "c"
                        local [0]: c
009D  00000000            startpc (0)
00A1  03000000            endpc   (3)
```

Parameters are located from the bottom of the stack, so the single parameter **c** is at register 0. It is also listed as a local, with a `startpc` value of 0.

```
                        * upvalues:
00A5  01000000          sizeupvalues (1)
00A9  02000000          string size (2)
00AD  6100              "a"
                        upvalue [0]: a
```

There is also an upvalue, **a**, which refers to the local **a** in the parent (top) function.

```
                        * constants:
00AF  01000000          sizek (1)
00B3  04                const type 4
00B4  02000000          string size (2)
00B8  6400              "d"
                        const [0]: "d"
                        * functions:
00BA  00000000          sizep (0)
                        * code:
00BE  04000000          sizecode (4)
00C2  04000001          [1] getupval   1   0        ; a
00C6  0C800001          [2] add        1   1   0
00CA  07000001          [3] setglobal  1   0        ; d
00CE  1B800000          [4] return     0   1
                        ** end of function **
```

Function **b** has 4 instructions. Most Lua virtual machine instructions are easy to decipher, but some of them have details that are not immediately evident. This example however should be quite easy to understand. In line [1], 0 is the upvalue **a** and 1 is the target register, which is a temporary register. Line [2] is the addition operation, with register 1 holding the temporary result while register 0 is the function parameter **c**. In line [3], the global **d** (so named by constant 0) is set, and in the next line, control is returned to the caller.

After the specification of function blocks in the function prototypes list, the parent function block resumes with its own code listing:

```
                        * code:
00D2  05000000          sizecode (5)
00D6  01000000          [1] loadk      0   0        ; 8
00DA  22000001          [2] closure    1   0        ; 1 upvalues
00DE  00000000          [3] move       0   0
00E2  47000001          [4] setglobal  1   1        ; b
00E6  1B800000          [5] return     0   1
                        ** end of function **

00EA                    ** end of chunk **
```

The first line of the source code compiles to a single instruction, line [1]. Local **a** is register 0 and the number 8 is constant 0. In line [2], an instance of function prototype 0 is created, and the closure is temporarily placed in register 1. The MOVE instruction in line [3] is actually used by the CLOSURE instruction to manage the upvalue **a**; it is not really executed. This will be explained in detail in Chapter 14. The closure is then placed into the global **b** in line [4]; “b” is constant 1 while the closure is in register 1. Line [5] returns control to the calling function. In this case, it exits the chunk.

Now that we’ve seen a binary chunk in detail, we will proceed to look at each Lua 5 virtual machine instruction.

5 Instruction Notation

Before looking at some Lua virtual machine instructions, here is a little something about the notation used for describing instructions. Instruction descriptions are given as comments in the Lua source file `lopcodes.h`. The instruction descriptions are reproduced in the following chapters, with additional explanatory notes. Here are some basic symbols:

R(A)	Register A (specified in instruction field A)
R(B)	Register B (specified in instruction field B)
R(C)	Register C (specified in instruction field C)
PC	Program Counter
Kst(n)	Element **n** in the constants list
Upvalue[n]	Name of upvalue with index **n**
Gbl[sym]	Global variable indexed by symbol **sym**
RK(B)	Register B or a constant index
RK(C)	Register C or a constant index
sBx	Signed displacement (in field sBx) for all kinds of jumps

The notation used to describe instructions is a little like pseudo-C. The operators used in the notation are largely C operators, while conditional statements use C-style evaluation. Booleans are evaluated C-style. Thus, the notation is a loose translation of the actual C code that implements an instruction.

The operation of some instructions cannot be clearly described by one or two lines of notation. Hence, this guide will supplement symbolic notation with detailed descriptions of the operation of each instruction. Having described an instruction, examples will be given to show the instruction working in a short snippet of Lua code. Using ChunkSpy's interactive mode, you can choose to try out the examples yourself and get instant feedback in the form of disassembled code. If you want a disassembled listing plus the byte values of data and instructions, you can use ChunkSpy to generate a normal, verbose, disassembly listing.

The program counter of the virtual machine (PC) always points to the next instruction. This behaviour is standard for most microprocessors. The rule is that once an instruction is read in to be executed, the program counter is immediately updated. So, to skip a single instruction following the current instruction, add 1 (the displacement) to the PC. A displacement of -1 will theoretically cause a JMP instruction to jump back onto itself, causing an infinite loop. Luckily, the code generator is not supposed to be able to make up stuff like that.

As previously explained, registers and local variables are roughly equivalent. Temporary results are always held in registers. Instruction fields B and C can point to a constant instead of a register for some instructions, this is when the field value is MAXSTACK or bigger. For most instructions, field A is the target register. Disassembly listings preserve the A, B, C operand field order for consistency.

6 Loading Constants

Loads and moves are the starting point of pretty much all processor or virtual machine instruction sets, so we'll start with primitive loads and moves:

MOVE **A B** R(A) := R(B)

Copies the value of register R(B) into register R(A). If R(B) holds a table, function or userdata, then the *reference* to that object is copied. MOVE is often used for moving values into place for the next operation.

The opcode for MOVE has a second purpose – it is also used in creating closures, always appearing *after* the CLOSURE instruction; see CLOSURE for more information.

The most straightforward use of MOVE is for assigning a local to another local:

```
>local a,b = 10; b = a
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.local  "b"  ; 1
.const  10  ; 0
[1] loadk      0   0        ; 10
[2] loadnil    1   1
[3] move       1   0
[4] return     0   1
; end of function
```

Line [3] assigns (*copies*) the value in local **a** (register 0) to local **b** (register 1).

You won't see MOVE instructions used in arithmetic expressions because they are not needed by arithmetic operators. All arithmetic operators are in 2- or 3-operand style: the entire local stack frame is already visible to operands R(A), R(B) and R(C) so there is no need for any extra MOVE instructions.

Other places where you will see MOVE are:

- When moving parameters into place for a function call.
- When moving values into place for certain instructions where stack order is important, e.g. GETTABLE, SETTABLE and CONCAT.
- When copying return values into locals after a function call.
- After CLOSURE instructions (discussed in Chapter 14.)

There are 3 fundamental instructions for loading constants into local variables. Other instructions, for reading and writing globals, upvalues and tables are discussed in the following chapters. The first constant loading instruction is LOADNIL:

LOADNIL **A B** R(A) := ... := R(B) := nil

Sets a range of registers from R(A) to R(B) to **nil**. If a single register is to be assigned to, then R(A) = R(B). When two or more consecutive locals need to be assigned **nil** values, only a single LOADNIL is needed.

LOADNIL uses the operands A and B to mean a *range* of register locations. The example for MOVE in the last page shows LOADNIL used to set a single register to **nil**.

```
>local a,b,c,d,e = nil,nil,0
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 0 5
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
.local  "d"  ; 3
.local  "e"  ; 4
.const  0  ; 0
[1] loadnil    0   1
[2] loadk      2   0        ; 0
[3] loadnil    3   4
[4] return     0   1
; end of function
```

In this example, line [1] **nil**s locals **a** and **b**. Line [3] **nil**s locals **d** and **e**. If all the locals are to be initialized to **nil**, then only a single LOADNIL will be needed.

LOADK	**A Bx**	R(A) := Kst(Bx)
Loads constant number Bx into register R(A). Constants are usually numbers or strings. Each function has its own constant list, or pool.		

LOADK loads a constant from the constants list into a register or local. Constants are indexed starting from 0. Some instructions, such as arithmetic instructions, can use the constants list without needing a LOADK. Constants are pooled in the list, duplicates are eliminated. The list can hold **nil**s, numbers or strings.

```
>local a,b,c,d = 3,"foo",3,"foo"
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 0 4
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
.local  "d"  ; 3
.const  3  ; 0
.const  "foo"  ; 1
[1] loadk      0   0        ; 3
[2] loadk      1   1        ; "foo"
[3] loadk      2   0        ; 3
[4] loadk      3   1        ; "foo"
[5] return     0   1
; end of function
```

The constant 3 and the constant “foo” are both written twice in the source snippet, but in the constants list, each constant has a single location. The constants list contains the names of global variables as well, since GETGLOBAL and SETGLOBAL makes an implied LOADK operation in order to get the name string of a global variable first before looking it up in the global table.

The final constant-loading instruction is LOADBOOL, for setting a boolean value, and it has a little additional functionality.

LOADBOOL A B C R(A) := (Bool)B; if (C) PC++

Loads a boolean value (**true** or **false**) into register R(A). **true** is usually encoded as an integer 1, **false** is always 0. If C is non-zero, then the next instruction is skipped (this is used when you have an assignment statement where the expression uses relational operators, e.g. M = K>5.)

You can use any non-zero value for the boolean **true** in field B, but since you cannot use booleans as numbers in Lua, it's best to stick to 1 for **true**.

LOADBOOL is the only instruction for loading a boolean value. It's also used where a boolean result is supposed to be generated, because relational test instructions, for example, do not generate boolean results – they perform conditional jumps instead. The operand C is used to optionally skip the next instruction (by incrementing PC by 1) in order to support such code. For simple assignments of boolean values, C is always 0.

```
>local a,b = true,false
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.local  "b"  ; 1
[1] loadbool   0   1   0    ; true
[2] loadbool   1   0   0    ; false
[3] return     0   1
; end of function
```

This example is straightforward: Line [1] assigns **true** to local **a** (register 0) while line [2] assigns **false** to local **b** (register 1).

```
>local a = 5 > 2
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.const  5  ; 0
.const  2  ; 1
[1] lt         1   251 250  ; 2 5, to [3] if false
[2] jmp        1           ; to [4]
[3] loadbool   0   0   1    ; false, to [5]
[4] loadbool   0   1   0    ; true
[5] return     0   1
; end of function
```

This is an example of an expression that gives a boolean result. Notice that Lua does not optimize the expression into a **true** value; it is not intended to do such optimizations.

Since the relational operator LT (which will be covered in greater detail later) does not give a boolean result but performs a conditional jump, LOADBOOL uses its C operand to perform an unconditional jump in line [3] – this saves one instruction and makes things a little tidier.

In the disassembly, when LT tests 2 < 5, it evaluates to **true** and doesn't perform a conditional jump. Line [2] jumps over the "false" path, and in line [4], the local **a** (register 0) is assigned the boolean **true** by the instruction LOADBOOL. If 2 and 5 were reversed, line [3] will be followed instead, setting a **false**, and the "true" path (line [4]) will be skipped.

7 Upvalues and Globals

When the Lua virtual machine needs an upvalue or a global, there are dedicated instructions to load the value into a register. Similarly, when an upvalue or a global needs to be written to, dedicated instructions are used.

GETGLOBAL A Bx R(A) := Gbl[Kst(Bx)]

Copies the value of the global variable whose name is given in constant number Bx into register R(A).

SETGLOBAL A Bx Gbl[Kst(Bx)] := R(A)

Copies the value from register R(A) into the global variable whose name is given in constant number Bx.

The GETGLOBAL and SETGLOBAL instructions are very straightforward and easy to use. The instructions require that the global variable name be a constant, indexed by instruction field Bx. R(A) is either the source or target register. The names of the global variables used by a function will be part of the constants list of the function.

```
>a = 40; local b = a
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "b"  ; 0
.const  "a"  ; 0
.const  40  ; 1
[1] loadk      0   1        ; 40
[2] setglobal  0   0        ; a
[3] getglobal  0   0        ; a
[4] return     0   1
; end of function
```

From the example, you can see that “b” is the name of the local variable while “a” is the name of the global variable. Line [1] loads the number 40 into register 0 (functioning as a temporary register, since local **b** hasn’t been defined.) Line [2] assigns the value in register 0 to the global variable with name “a” (constant 0). By line [3], local **b** is defined and is assigned the value of global **a**.

GETUPVAL A B R(A) := UpValue[B]

Copies the value in upvalue number B into register R(A). Each function may have its own upvalue list.

The opcode for GETUPVAL has a second purpose – it is also used in creating closures, always appearing *after* the CLOSURE instruction; see CLOSURE for more information.

SETUPVAL A B UpValue[B] := R(A)

Copies the value from register R(A) into the upvalue number B in the upvalue list for that function.

GETUPVAL and SETUPVAL uses the upvalues list. Only the names of upvalues are stored in the list. During execution, upvalues are set up by a CLOSURE, and maintained by the Lua virtual machine. In the following example, function **b** is declared inside the main chunk, and is shown in the disassembly as a function prototype within a function prototype. The indentation helps to separate the two functions.

```
>local a; function b() a = 1 return a end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.const  "b"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .upvalue  "a"  ; 0
  .const  1  ; 0
  [1] loadk      0   0        ; 1
  [2] setupval   0   0        ; a
  [3] getupval   0   0        ; a
  [4] return     0   2
  [5] return     0   1
  ; end of function

[1] loadnil    0   0
[2] closure    1   0        ; 1 upvalues
[3] move       0   0
[4] setglobal  1   0        ; b
[5] return     0   1
; end of function
```

In the main chunk (function 0, level 1), local **a** is first initialized to **nil**. The CLOSURE in line [2] then instantiates function prototype 0 (function 0, level 2) with a single upvalue, **a**. Line [3] is part of the closure, it links local **a** in the current scope to upvalue **a** in the closure. Finally the closure is assigned to global **b**.

In function **b**, there is a single upvalue, **a**. In Pascal, a variable in an outer scope is found by traversing stack frames. However, instantiations of Lua functions are first-class values, and they may be assigned to a variable and referenced elsewhere. Managing upvalues thus becomes a little more tricky than traversing stack frames in Pascal. The Lua virtual machine solution is to provide a clean interface via GETUPVAL and SETUPVAL, while the management of upvalues part is handled by the virtual machine itself.

Line [2] in function **b** sets upvalue **a** (upvalue number 0 in the upvalue table) to a number value of 1 (held in temporary register 0.) In line [3], the value in upvalue **a** is retrieved and placed into register 0, where the following RETURN instruction will use it as a return value.

8 Table Instructions

Accessing table elements is a little more complex than accessing upvalues and globals:

GETTABLE A B C R(A) := R(B)[RK(C)]

Copies the value from a table element into register R(A). The table is referenced by register R(B), while the index to the table is given by RK(C), which may be the value of register R(C) or a constant number.

SETTABLE A B C R(A)[RK(B)] := RK(C)

Copies the value from register R(C) or a constant into a table element. The table is referenced by register R(A), while the index to the table is given by RK(B), which may be the value of register R(B) or a constant number.

All 3 operand fields are used, and some of the operands can be constants. A constant is specified by biasing the constant number by MAXSTACK (250). If RK(C) need to refer to constant 1, then it will have the value of (250+1) or 251. Allowing constants to be used directly reduces considerably the need for temporary registers.

```
>local p = {}; p[1] = "foo"; return p["bar"]
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "p"  ; 0
.const  1  ; 0
.const  "foo"  ; 1
.const  "bar"  ; 2
[1] newtable   0   0   0    ; array=0, hash=0
[2] settable   0   250 251  ; 1 "foo"
[3] gettable   1   0   252  ; "bar"
[4] return     1   2
[5] return     0   1
; end of function
```

In line [1], a new empty table is created and the reference placed in local **p** (register 0). Creating and populating new tables is a little involved so it will only be discussed later.

Table index 1 is set to “foo” in line [2] by the SETTABLE instruction. Both the index and the value for the table element are encoded constant numbers; 250 is constant 0 (the number 1) while 251 is constant 1 (the string “foo”.) The R(A) value of 0 points to the new table that was defined in line [1].

In line [3], the value of the table element indexed by the string “bar” is copied into temporary register 1, which is then used by RETURN as a return value. 252 is constant 2 (the string “bar”) while 0 in field B is the reference to the table.

9 Arithmetic and String Instructions

The Lua virtual machine's set of arithmetic instructions looks like 3-operand arithmetic instructions on an RISC processor. 3-operand instructions allow arithmetic expressions to be translated into machine instructions pretty efficiently.

ADD	**A B C**	R(A) := RK(B) + RK(C)
SUB	**A B C**	R(A) := RK(B) – RK(C)
MUL	**A B C**	R(A) := RK(B) * RK(C)
DIV	**A B C**	R(A) := RK(B) / RK(C)
POW	**A B C**	R(A) := RK(B) ^ RK(C)

Binary operators (arithmetic operators with two inputs.) The result of the operation between RK(B) and RK(C) is placed into R(A). These instructions are in the classic 3-register style. RK(B) and RK(C) may either be registers or constants in the constant pool.

ADD is addition. SUB is subtraction. MUL is multiplication. DIV is division. POW is exponentiation.

The source operands, RK(B) and RK(C), may be constants. If a constant is out of range of field B or field C, then the constant will be loaded into a temporary register in advance.

```
>local a,b = 2,4; a = a + 4 * b - a / 2 ^ b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 0 4
.local  "a"  ; 0
.local  "b"  ; 1
.const  2  ; 0
.const  4  ; 1
[1] loadk      0   0        ; 2
[2] loadk      1   1        ; 4
[3] mul        2   251 1    ; 4
[4] add        2   0   2
[5] pow        3   250 1    ; 2
[6] div        3   0   3
[7] sub        0   2   3
[8] return     0   1
; end of function
```

Each arithmetic operator translates into a single instruction. This also means that while the statement "`count = count + 1`" is verbose, it translates into a single instruction if **count** is a local. If **count** is a global, then two extra instructions are required to read and write to the global (GETGLOBAL and SETGLOBAL), since arithmetic operations can only be done on registers (locals) only.

Next up are instructions for performing unary minus and logical NOT:

UNM	**A B**	R(A) := -R(B)

Unary minus (arithmetic operator with one input.) R(B) is negated and the value placed in R(A). R(A) and R(B) are always registers.

NOT	**A B**	R(A) := not R(B)
Applies a boolean NOT to the value in R(B) and places the result in R(A). R(A) and R(B) are always registers.		

Here is an example:

```
>local p,q = 10,false; q,p = -p,not q
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "p"  ; 0
.local  "q"  ; 1
.const  10  ; 0
[1] loadk      0   0        ; 10
[2] loadbool   1   0   0    ; false
[3] unm        2   0
[4] not        0   1
[5] move       1   2
[6] return     0   1
; end of function
```

Both UNM and NOT do not accept a constant as a source operand. When an unary minus is applied to a constant number, the unary minus is optimized away. Similarly, when a **not** is applied to **true** or **false**, the logical operation is optimized away.

CONCAT	**A B C**	R(A) := R(B).. ... ..R(C)
Performs concatenation of two or more strings. In a Lua source, this is equivalent to one or more concatenation operators ('**..**') between two or more expressions. The source registers must be consecutive, and C must always be greater than B. The result is placed in R(A).		

Like LOADNIL, CONCAT accepts a range of registers. Doing more than one string concatenation at a time is faster and more efficient than doing them separately.

```
>local x,y = "foo","bar"; return x..y..x..y
; function [0] definition (level 1)
; 0 upvalues, 0 params, 6 stacks
.function  0 0 0 6
.local  "x"  ; 0
.local  "y"  ; 1
.const  "foo"  ; 0
.const  "bar"  ; 1
[1] loadk      0   0        ; "foo"
[2] loadk      1   1        ; "bar"
[3] move       2   0
[4] move       3   1
[5] move       4   0
[6] move       5   1
[7] concat     2   2   5
[8] return     2   2
[9] return     0   1
; end of function
```

In this example, strings are moved into place first (lines [3] to [6]) in the concatenation order before a single CONCAT instruction is executed in line [7].

10 Jumps and Calls

Lua does not have any unconditional jump feature in the language itself, but in the virtual machine, the unconditional jump is used in control structures and logical expressions.

JMP	**sBx**	PC += sBx

Performs an unconditional jump, with sBx as a signed displacement. sBx is added to the program counter (PC), which points to the next instruction to be executed. E.g., if sBx is 0, the VM will proceed to the next instruction.

JMP is used in loops, conditional statements, and in expressions when a boolean **true**/**false** need to be generated.

For example, since a relational test instruction makes conditional jumps rather than generate a boolean result, a JMP is used in the code sequence for loading either a **true** or a **false**:

```
>local m, n; return m >= n
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "m"  ; 0
.local  "n"  ; 1
[1] loadnil    0   1
[2] le         1   1   0    ; to [4] if false
[3] jmp        1           ; to [5]
[4] loadbool   2   0   1    ; false, to [6]
[5] loadbool   2   1   0    ; true
[6] return     2   2
[7] return     0   1
; end of function
```

In line [3], the JMP skips over the false path (line [4]) to the true path (line [5]). More examples where JMP is used will be covered in later chapters.

Next we will look at the CALL instruction, for calling instantiated functions:

CALL	**A B C**	R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))

Performs a function call, with register R(A) holding the reference to the function object to be called. Parameters to the function are placed in the registers following R(A). If B is 1, the function has no parameters. If B is 2 or more, there are (B-1) parameters.

If B is 0, the function parameters range from R(A+1) to the top of the stack. This form is used when the last expression in the parameter list is a function call, so the number of actual parameters is indeterminate.

Results returned by the function call is placed in a range of registers starting from R(A). If C is 1, no return results are saved. If C is 2 or more, (C-1) return values are saved. If C is 0, then multiple return results are saved, depending on the called function.

CALL always updates the top of stack value. The use of the top of stack is implied in CALL, RETURN and SETLISTO.

Generally speaking, for fields B and C, a zero means that multiple results or parameters (up to the top of stack) are expected. If the number of results or parameters are fixed, then the actual number is one less than the encoded field value. Here is the simplest possible call:

```
>z()
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.const  "z"  ; 0
[1] getglobal  0   0        ; z
[2] call       0   1   1
[3] return     0   1
; end of function
```

In line [2], the call has zero parameters (field B is 1), zero results are retained (field C is 1), while register 0 temporarily holds the reference to the function object from global **z**. Next we see a function call with multiple parameters or arguments:

```
>z(1,2,3)
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 0 4
.const  "z"  ; 0
.const  1  ; 1
.const  2  ; 2
.const  3  ; 3
[1] getglobal  0   0        ; z
[2] loadk      1   1        ; 1
[3] loadk      2   2        ; 2
[4] loadk      3   3        ; 3
[5] call       0   4   1
[6] return     0   1
; end of function
```

Lines [1] to [4] loads the function reference and the arguments in order, then line [5] makes the call with field B value of 4, which means there are 3 parameters. Since the call statement is not assigned to anything, no return results need to be retained, hence field C is 1. Here is an example that uses multiple parameters and multiple return values:

```
>local p,q,r,s = z(y())
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 0 4
.local  "p"  ; 0
.local  "q"  ; 1
.local  "r"  ; 2
.local  "s"  ; 3
.const  "z"  ; 0
.const  "y"  ; 1
[1] getglobal  0   0        ; z
[2] getglobal  1   1        ; y
[3] call       1   1   0
[4] call       0   0   5
[5] return     0   1
; end of function
```

First, the function references are retrieved (lines [1] and [2]), then function **y** is called first (temporary register 1). The CALL has a field C of 0, meaning multiple return values are accepted. These return values become the parameters to function **z**, and so in line [4], field B

of the CALL instruction is 0, signifying multiple parameters. After the call to function **z**, 4 results are retained, so field C in line [4] is 5. Finally, here is an example with calls to standard library functions:

```
>print(string.char(64))
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.const  "print"  ; 0
.const  "string"  ; 1
.const  "char"  ; 2
.const  64  ; 3
[1] getglobal  0   0        ; print
[2] getglobal  1   1        ; string
[3] gettable   1   1   252  ; "char"
[4] loadk      2   3        ; 64
[5] call       1   2   0
[6] call       0   0   1
[7] return     0   1
; end of function
```

When a function call is the last parameter to another function call, the former can pass multiple return values, while the latter can accept multiple parameters.

Complementing CALL is RETURN:

RETURN **A B** return R(A), ... ,R(A+B-2)
Returns to the calling function, with optional return values. If B is 1, there are no return values. If B is 2 or more, there are (B-1) return values, located in consecutive registers from R(A) onwards.
If B is 0, the set of values from R(A) to the top of the stack is returned. This form is used when the last expression in the return list is a function call, so the number of actual values returned is indeterminate.
RETURN also closes any open upvalues, equivalent to a CLOSE instruction. See the CLOSE instruction for more information.

Like CALL, a field B value of 0 signifies multiple return values (up to top of stack.)

```
>local e,f,g; return f,g
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 0 5
.local  "e"  ; 0
.local  "f"  ; 1
.local  "g"  ; 2
[1] loadnil    0   2
[2] move       3   1
[3] move       4   2
[4] return     3   3
[5] return     0   1
; end of function
```

In line [4], 2 return values are specified (field B value of 3) and those values are placed in consecutive registers starting from register 3. The RETURN in line [5] is redundant; it is always generated by the Lua code generator.

TAILCALL **A B C** return R(A)(R(A+1), ... ,R(A+B-1))

Performs a tail call, which happens when a **return** statement has a single function call as the expression, e.g. `return foo(bar)`. A tail call is effectively a *goto*, and avoids nesting calls another level deeper. Only Lua functions can be tailcalled.

Like CALL, register R(A) holds the reference to the function object to be called. B encodes the number of parameters in the same manner as a CALL instruction.

C isn't used by TAILCALL, since all return results are significant. In any case, Lua always generates a 0 for C, to denote multiple return results.

A TAILCALL is used only for one specific **return** style, described above. Multiple return results are always produced by a tail call. Here is an example:

```
>return x("foo", "bar")
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.const  "x"  ; 0
.const  "foo"  ; 1
.const  "bar"  ; 2
[1] getglobal  0   0        ; x
[2] loadk      1   1        ; "foo"
[3] loadk      2   2        ; "bar"
[4] tailcall   0   3   0
[5] return     0   0
[6] return     0   1
; end of function
```

Arguments for a tail call are handled in exactly the same way as arguments for a normal call, so in line [3], the tail call has a field B value of 3, signifying 2 parameters. Field C is 0, for multiple returns. In practice, field C is not used by the virtual machine since the syntax guarantees multiple return results.

Line [5] is a RETURN instruction specifying multiple return results. This is required when the function called by TAILCALL is a C function. In the case of a C function, execution continues to line [5] upon return, thus the RETURN is necessary. Line [6] is redundant. When Lua functions are tailcalled, the virtual machine does not return to line [5] at all.

Finally, we have a special form of a call instruction, SELF, which is used for object-oriented programming:

SELF **A B C** R(A+1) := R(B); R(A) := R(B)[RK(C)]

For object-oriented programming using tables. Retrieves a function reference from a table element and places it in register R(A), then a reference to the table itself is placed in the next register, R(A+1). This instruction saves some messy manipulation when setting up a method call.

R(B) is the register holding the reference to the table with the method. The method function itself is found using the table index RK(C), which may be the value of register R(C) or a constant number.

A SELF instruction saves an extra instruction and speeds up the calling of methods in object-oriented programming. In the following example:

```
>foo:bar("baz")
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.const  "foo"  ; 0
.const  "bar"  ; 1
.const  "baz"  ; 2
[1] getglobal  0   0        ; foo
[2] self       0   0   251  ; "bar"
[3] loadk      2   2        ; "baz"
[4] call       0   3   1
[5] return     0   1
; end of function
```

The SELF in line [2] is equivalent to a GETTABLE lookup (the table is in register 0 and the index is constant 1) and *at the same time*, a MOVE (copying the table from register 0 to register 1.) Without SELF, a GETTABLE cannot write to register 0 because the table reference will be overwritten before a MOVE can be done. Hence, SELF saves one instruction and one temporary register slot.

After setting up the method call using SELF, the call is made with the usual CALL instruction in line [4], which is equivalent to the following: **foo.bar(foo, "baz")**

Next we will look at more complicated instructions.

11 Relational and Logic Instructions

Relational and logic instructions are used in conjunction with other instructions to implement control structures or expressions. Instead of generating boolean results, these instructions performs a conditional jump over the next instruction. Hence, there is always a "true path" and a "false path".

EQ	**A B C**	if ((RK(B) == RK(C)) ~= A) then pc++
LT	**A B C**	if ((RK(B) < RK(C)) ~= A) then pc++
LE	**A B C**	if ((RK(B) <= RK(C)) ~= A) then pc++

Compares RK(B) and RK(C), which may be registers or constants. If the boolean result is not A, then skip the next instruction. Conversely, if the boolean result equals A, continue with the next instruction.

EQ is for equality. LT is for "less than" comparison. LE is for "less than or equal to" comparison. The boolean A field allows the full set of relational comparison operations to be synthesized from these three instructions. The Lua code generator produces either 0 or 1 for the boolean A.

By comparing the result of the relational operation with A, the sense of the comparison can be reversed. Obviously the alternative is to reverse the paths taken by the instruction, but that will probably complicate code generation some more.

```
>local x,y; return x ~= y
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "x"  ; 0
.local  "y"  ; 1
[1] loadnil    0   1
[2] eq         0   0   1    ; to [4] if true
[3] jmp        1           ; to [5]
[4] loadbool   2   0   1    ; false, to [6]
[5] loadbool   2   1   0    ; true
[6] return     2   2
[7] return     0   1
; end of function
```

In the above example, the inequality comparison is compiled into an EQ in line [2]. Relational expressions always perform the conditional jump for the false path, while for the true path, the next instruction is executed. Hence the true path is from [2] to [3] to [5]; the result is true if the EQ comparison evaluates to false, since we are using the **~=** operator. This is because A selects the comparison result to use the true path – in this case we want **x ~= y** to return **true** if the EQ comparison fails, and selecting "fail" means A is 0. The false path follows the conditional jump, from [2] to [4] to [6]. The C field in the LOADBOOL in line [4] is set so that line [5], which is part of the true path, can be skipped. In line [6], the boolean result which is now in temporary register 2 is returned to the caller.

ChunkSpy comments the EQ in line [2] by letting the user know when the conditional jump is taken. In this case, the jump to the false path is taken when "the value in register 0 equals to the value in register 1" is *true*. This is always the opposite of the A field value, which selects the true path to be taken. Anyway, note that these are Lua code generator conventions, and there are other ways to code **x ~= y** in terms of Lua virtual machine instructions.

For conditional statements, there is no need to set boolean results. So Lua is optimized for coding the more common conditional statements rather than conditional expressions.

```
>local x,y; if x ~= y then return "foo" else return "bar" end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "x"  ; 0
.local  "y"  ; 1
.const  "foo"  ; 0
.const  "bar"  ; 1
[1] loadnil    0   1
[2] eq         1   0   1    ; to [4] if false
[3] jmp        3           ; to [7]
[4] loadk      2   0       ; "foo"
[5] return     2   2
[6] jmp        2           ; to [9]
[7] loadk      2   1       ; "bar"
[8] return     2   2
[9] return     0   1
; end of function
```

In the above conditional statement, the same inequality operator is used in the source, but the sense of the EQ instruction in line [2] is now reversed. Since the EQ conditional jump can only skip the next instruction, additional JMP instructions are needed to allow large blocks of code to be placed in both true and false paths. In contrast, in the previous example, only a single instruction is needed to set a boolean value.

The true path (when **x ~= y** is true) goes from [2] to [4]–[6] and on to [9]. Since there is a RETURN in line [5], the JMP in line [6] and the RETURN in [9] are never executed at all; they are redundant but does not adversely affect performance in any way. The false path is from [2] to [3] to [7] onwards. So in a disassembly listing, you should see the true and false code blocks in the same order as in the Lua source.

```
>if 8 > 9 then return 8 elseif 5 >= 4 then return 5 else return 9 end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.const  8  ; 0
.const  9  ; 1
.const  5  ; 2
.const  4  ; 3
[01] lt         0   251 250  ; 9 8, to [3] if true
[02] jmp        3           ; to [6]
[03] loadk      0   0       ; 8
[04] return     0   2
[05] jmp        7           ; to [13]
[06] le         0   253 252  ; 4 5, to [8] if true
[07] jmp        3           ; to [11]
[08] loadk      0   2       ; 5
[09] return     0   2
[10] jmp        2           ; to [13]
[11] loadk      0   1       ; 9
[12] return     0   2
[13] return     0   1
; end of function
```

This example is a little more complex, with an **elseif**, but it is structured in the same order as the Lua source, so interpreting the disassembled code should not be too hard.

TEST	**A B C**	if (R(B) <=> C) then R(A) := R(B) else pc++

Used to implement **and** and **or** logical operators, or for testing a single register in a conditional statement.

Register R(B) is coerced into a boolean and compared to the boolean field C. If R(B) matches C, the next instruction is skipped, otherwise R(B) is assigned to R(A) and the VM continues with the next instruction. The **and** operator uses a C of 0 (false) while **or** uses a C value of 1 (true).

TEST is a little more complex than a boolean test and conditional jump combination because Lua has short-circuit LISP-style logical operators that retains and propagates operand values instead of booleans. First, we'll look at how **and** and **or** behaves:

```
>local a,b,c; c = a and b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] loadnil    0   2
[2] test       2   0   0    ; to [4] if true
[3] jmp        1           ; to [5]
[4] move       2   1
[5] return     0   1
; end of function
```

The **and** operator *propagates false operands* (which can be **false** or **nil**) because any **false** operands in a string of **and** operations will make the whole boolean expression **false**. When a string of **and** operations evaluates to true, the result is the *last* operand value.

In line [2], the first operand (the local **a**) is retained when the test is **false** (with a field C of 0), while the jump to [4] is made when the test is **true**, and then in line [4], the expression result is set to the second operand (the local **b**).

```
>local a,b,c; c = a or b
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] loadnil    0   2
[2] test       2   0   1    ; to [4] if false
[3] jmp        1           ; to [5]
[4] move       2   1
[5] return     0   1
; end of function
```

The **or** operator *propagates the first true operand*, because any **true** operands in a string of **or** operations will make the whole boolean expression **true**. When a string of **or** operations evaluates to **false**, all operands must have evaluated to **false**.

In line [2], the local **a** value is retained if it is **true**, while the jump is made if it is **false**. Thus in line [4], the local **b** value is the result of the expression if local **a** evaluates to **false**.

Short-circuit logical operators also means that the following Lua code does not actually use a boolean operation:

```
>local a,b,c; if a > b and a > c then return a end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] loadnil    0   2
[2] lt         0   1   0    ; to [4] if true
[3] jmp        3           ; to [7]
[4] lt         0   2   0    ; to [6] if true
[5] jmp        1           ; to [7]
[6] return     0   2
[7] return     0   1
; end of function
```

With short-circuit evaluation, **a** > **c** is never executed if **a** > **b** is **false**, so the logic of the Lua statement can be readily implemented using the normal conditional structure. If both **a** > **b** and **a** > **c** are **true**, the path followed is [2] (the **a** > **b** test) to [4] (the **a** > **c** test) and finally to [6], returning the value of **a**. The TEST instruction is not required.

For a single variable used in the expression part of a conditional statement, TEST is used to boolean-test the variable:

```
>if Done then return end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.const  "Done"  ; 0
[1] getglobal  0   0        ; Done
[2] test       0   0   0    ; to [4] if true
[3] jmp        1           ; to [5]
[4] return     0   1
[5] return     0   1
; end of function
```

In line [2], the TEST instruction jumps to the true path if the value in temporary register 0 (from the global **Done**) is **true**. If the test expression of a conditional statement consist of purely boolean operators, then a number of TEST instructions will be used in the usual short-circuit evaluation style:

```
>if Found and Match then return end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.const  "Found"  ; 0
.const  "Match"  ; 1
[1] getglobal  0   0        ; Found
[2] test       0   0   0    ; to [4] if true
[3] jmp        4           ; to [8]
[4] getglobal  0   1        ; Match
[5] test       0   0   0    ; to [7] if true
[6] jmp        1           ; to [8]
[7] return     0   1
[8] return     0   1
; end of function
```

In the last example, the true code block of the conditional statement is executed only if both **Found** and **Match** evaluates to **true**. The path is from [2] (test for **Found**) to [4] to [5] (test for **Match**) to [7] (the true path code block, which is an explicit **return** statement.)

Finally, here is how Lua's ternary operator (**:?**) equivalent works:

```
>local a,b,c; a = a and b or c
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "a"  ; 0
.local  "b"  ; 1
.local  "c"  ; 2
[1] loadnil    0   2
[2] test       0   0   0    ; to [4] if true
[3] jmp        2           ; to [6]
[4] test       0   1   1    ; to [6] if false
[5] jmp        1           ; to [7]
[6] move       0   2
[7] return     0   1
; end of function
```

The TEST in line [2] is for the **and** operator. First, local **a** is tested in line [2]. If it is **false**, then execution continues in [3], jumping to line [6]. Line [6] assigns local **c** to the end result because since if **a** is **false**, then **a and b** is **false**, and **false or c** is **c**.

If local **a** is **true** in line [2], the TEST instruction makes a jump to line [4], where there is a second TEST, for the **or** operator. If **b** evaluates to **true**, then the end result is assigned the value of **b**, because **b or c** is **b** if **b** is not **false**. If **b** is also **false**, the end result will be **c**.

For the instructions in line [2], [4] and [6], the target (in field A) is register 0, or the local **a**, which is the location where the result of the boolean expression is assigned.

12 Loop Instructions

Lua has dedicated instructions to implement the two types of **for** loops, while the other two types of loops uses traditional test-and-jump.

FORLOOP **A sBx** R(A)+=R(A+2)
if R(A) <?= R(A+1) then PC+= sBx

Performs an iteration of a numeric **for** loop. A numeric for loop requires 3 registers on the stack, and each register must be a number. R(A) holds the initial value and doubles as the loop variable; R(A+1) is the limit; R(A+2) is the stepping value.

A jump is made back to the start of the loop body if the limit has not been reached or exceeded. The sense of the comparison depends on whether the stepping is negative or positive, hence the "<?=" operator. The jump is encoded as a signed displacement in the sBx field. An empty loop has a sBx value of -1.

Since a **for** loop need to perform an initial test *prior* to the start of the first iteration, the initial value is given a negative step, i.e. the initial value is subtracted by the step and the loop starts at a FORLOOP instruction. The first time FORLOOP is reached, a step is made, thus restoring the initial value before the first comparison. See the examples for an illustration.

The loop variable ends with the last value before the limit is reached (unlike C) because it is not updated unless the jump is made. However, since loop variables are local to the loop itself, you should not be able to use it unless you cook up an implementation-specific hack.

For the sake of efficiency, FORLOOP contains a lot of functionality, so when a loop iterates, only *one* instruction, FORLOOP, is needed. Here is a simple example:

```
>local a = 0; for i = 1,100,5 do a = a + i end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 4 stacks
.function  0 0 0 4
.local  "a"  ; 0
.local  "i"  ; 1
.local  "(for limit)"  ; 2
.local  "(for step)"  ; 3
.const  0  ; 0
.const  1  ; 1
.const  100  ; 2
.const  5  ; 3
[1] loadk      0   0        ; 0
[2] loadk      1   1        ; 1
[3] loadk      2   2        ; 100
[4] loadk      3   3        ; 5
[5] sub        1   1   3
[6] jmp        1            ; to [8]
[7] add        0   0   1
[8] forloop    1   -2       ; to [7] if loop
[9] return     0   1
; end of function
```

In the last example, notice that the **for** loop causes two additional local pseudo-variables to be defined, apart from the loop index, **i**. The two pseudo-variables, named **(for limit)** and **(for step)** are required to completely specify the state of the loop, along with the loop index, and are not visible to Lua source code. The three, **i**, **(for limit)** and **(for step)**, are arranged in consecutive registers, with the loop index given by R(A).

The loop body is in line [7] while line [8] is the FORLOOP instruction that steps through the loop state. The sBx field of FORLOOP is negative, as it always jumps back to the beginning of the loop body.

Lines [2]–[4] initializes the three register locations where the loop state will be stored. If the loop step is not specified in the Lua source, a constant 1 is added to the constant pool and a LOADK instruction is used to initialize the pseudo-variable **(for step)** with the loop step.

Lines [5]–[6] makes a negative loop step and jumps to line [8] for the initial test to be done. In the example, at line [6], the loop index **i** (at register 1) will be (1-5) or -4. When the virtual machine arrives at the FORLOOP in line [8] for the first time, one loop step is made prior to the first test, so the value that is actually tested against the limit is (-4+5) or 1. Since 1 < 100, the conditional jump is made to line [7], starting the first iteration of the loop.

The loop at line [7]–[8] repeats until the loop index **i** exceeds the loop limit of 100. The conditional jump is not taken when that occurs and the loop ends. Beyond the scope of the loop body, the loop state (**i**, **(for limit)** and **(for step)**) is not valid. This is determined by the parser and code generator. The range of PC values for which the loop state variables are valid is located in the locals list. The brief assembly listings generated by ChunkSpy that you are seeing does not give the startpc and endpc values contained in the locals list. In theory, these rules can be broken if you write Lua assembly directly.

```
>for i = 10,1,-1 do if i == 5 then break end end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "i"  ; 0
.local  "(for limit)"  ; 1
.local  "(for step)"  ; 2
.const  10  ; 0
.const  1  ; 1
.const  -1  ; 2
.const  5  ; 3
[01] loadk      0   0        ; 10
[02] loadk      1   1        ; 1
[03] loadk      2   2        ; -1
[04] sub        0   0   2
[05] jmp        3            ; to [9]
[06] eq         0   0   253  ; 5, to [8] if true
[07] jmp        1            ; to [9]
[08] jmp        1            ; to [10]
[09] forloop    0   -4       ; to [6] if loop
[10] return     0   1
; end of function
```

In the second loop example above, except for a negative loop step size, the structure of the loop is identical. The body of the loop is from line [6] to line [9]. Since no additional stacks or states are used, a **break** translates simply to a JMP instruction (line [8]). There is nothing to clean up after a FORLOOP ends or after a JMP to exit a loop.

The next instruction, TFORPREP, is largely for compatibility with Lua 4 source code:

TFORPREP **A sBx** if type(R(A)) == table then
R(A+1):=R(A), R(A):=next;
PC += sBx

Optionally initializes the Lua 4 table form of the generic **for** loop, for compatibility. In Lua 5, the elements for the iterated form of the generic **for** loop should already be in place (see TFORLOOP below), and the only thing TFORPREP does is an unconditional jump in order to execute TFORLOOP for the first time. The sBx field contains the signed displacement for the jump.

For Lua 4 compatibility, R(A) should be a table. The table is moved to register R(A+1) as the *state*, while R(A) is set to the global function **next**, which will serve as the iterator function for the generic **for** loop. After changing to the Lua 5 form, the unconditional jump is then made.

Apart from a numeric **for** loop (implemented by FORLOOP), Lua has a generic **for** loop, implemented by TFORLOOP:

TFORLOOP **A C** R(A+2), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
if R(A+2) ~= nil then pc++

Performs an iteration of a generic **for** loop. A Lua 5-style generic **for** loop keeps 3 items in consecutive register locations to keep track of things. R(A)is the *iterator function*, which is called once per loop. R(A+1) is the *state*, and R(A+2) is the enumeration index. At the start, R(A+2) has an initial value.

Each time TFORLOOP executes, the iterator function referenced by R(A) is called with two arguments: the state and the enumeration index (R(A+1) and R(A+2).) The first return value must be the new value of the enumeration index, and it is assigned to R(A+2). Additional values may be returned in consecutive registers after R(A+2), and the field C specifies the number of additional results. If C is 0, the enumeration index, R(A+2), is the only returned result.

If the enumeration index becomes **nil**, then the iterator loop is at an end, and TFORLOOP skips the next instruction (which is usually a jump to the beginning of the loop body.)

A generic **for** loop's state is also kept in 3 consecutive registers, but the registers contain very different things. The iterator function is located in R(A), and is named **(for generator)** for debugging purposes. The state is in R(A+1), and has the name **(for state)**. The enumeration index is contained in register R(A+2), while additional results from the iterator function is placed into R(A+3), R(A+4) and so on.

The number of additional results is given in the C field. The enumeration index is always present, and along with additional results, have normal local variable names that are visible to the programmer. A generic **for** loop ends when the enumeration index becomes **nil**.

This example has a loop with one additional result (**v**) in addition the loop enumerator (**i**):

```
>for i,v in pairs(t) do print(i,v) end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 7 stacks
.function  0 0 0 7
.local  "(for generator)"  ; 0
.local  "(for state)"  ; 1
.local  "i"  ; 2
.local  "v"  ; 3
.const  "pairs"  ; 0
.const  "t"  ; 1
.const  "print"  ; 2
[01] getglobal  0   0        ; pairs
[02] getglobal  1   1        ; t
[03] call       0   2   5
[04] tforprep   0   4        ; to [9]
[05] getglobal  4   2        ; print
[06] move       5   2
[07] move       6   3
[08] call       4   3   1
[09] tforloop   0       1    ; to [11] if exit
[10] jmp        -6           ; to [5]
[11] return     0   1
; end of function
```

Line [1]–[3] prepares register 0 to 3. Note that the call to the **pairs** standard library function has 1 parameter and 4 results. After the call in line [3], register 0 is the iterator function, register 1 is the loop state, register 2 is the initial value of the enumeration index **i**, and register 3 is the initial value of the additional result **v**.

The TFORPREP in line [4] does not do anything for Lua 5-style generic loops; for Lua 5 it is essentially an unconditional JMP to line [9], where TFORLOOP is encountered for the first time. Since **pairs** generate the *zeroth* enumeration state, the first time TFORLOOP executes, the *first* enumeration state of the generic loop is produced. Additional results are generated as needed. If a loop is to be made, execution continues in the next line, which is a JMP to the body of the generic loop (lines [5]–[8]). To drop out of the loop, TFORLOOP skips the next line, continuing to line [11].

repeat and **while** loops use a standard test-and-jump structure:

```
>local a = 0; repeat a = a + 1 until a == 10
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.const  0  ; 0
.const  1  ; 1
.const  10  ; 2
[1] loadk      0   0        ; 0
[2] add        0   0   251  ; 1
[3] eq         0   0   252  ; 10, to [5] if true
[4] jmp        -3          ; to [2]
[5] return     0   1
; end of function
```

The body of the **repeat** loop is line [2], while the test-and-jump scheme is implemented in lines [3] and [4]. Two instructions are needed to loop the loop.

```
>local a = 1; while a < 10 do a = a + 1 end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.const  1  ; 0
.const  10  ; 1
[1] loadk      0   0        ; 1
[2] jmp        1            ; to [4]
[3] add        0   0   250  ; 1
[4] lt         1   0   251  ; 10, to [6] if false
[5] jmp        -3           ; to [3]
[6] return     0   1
; end of function
```

A **while** loop is quite similar to a **repeat** loop; the body of the loop comes first (line [3]) while the test-and-jump structure (lines [4] and [5]) is at the end of the loop body. Since **while** does its test at the start of the loop, a JMP (line [2]) is added.

In `lparser.c`, it is explained that the reason for coding the condition after the loop body is optimization, because one jump in the loop is avoided. It is not clear whether having condition testing at the start is slower than having it at the end. The way the **while** code generator function is implemented, with the condition at the end, leads to a limit to the complexity of **while** conditions, about 100 instructions, defined as MAXEXPWHILE. This is due to a fixed buffer used to temporarily hold the instructions making up the condition.

Here is one way (untested) of implementing condition testing before the body of the loop:

```
>local a = 1; while a < 10 do a = a + 1 end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "a"  ; 0
.const  1  ; 0
.const  10  ; 1
[1] loadk      0   0        ; 1
[2] lt         0   0   251  ; 10, to [4] if true
[3] jmp        1            ; to [5]
[4] add        0   0   250  ; 1
[5] jmp        -4           ; to [2]
[6] return     0   1
; end of function
```

The sense of the condition test is reversed, while the loop body is at line [4]. Line [3] jumps out of the **while** loop, and line [5] jumps back to the condition test to repeat the loop.

13 Table Creation

There are three instructions for table creation and initialization. A single instruction creates a table while the other two instructions sets table elements.

NEWTABLE A B C R(A) := {} (size = B,C)

Creates a new empty table at register R(A). B and C are the encoded size information for the array part and the hash part of the table, respectively. Appropriate values for B and C are set in order to avoid rehashing when initially populating the table with values or key-value pairs.

B is a "floating point byte" (so named in `lobject.c`), encoded as `mmmmmxxx` in binary, where the actual value is: `xxx*2^mmmmm`. The actual size of the array is rounded up and then encoded in field B. The parser increments the array size for every *exp* field in the table constructor.

C is the $\log_2$ value of the size of the hash portion, plus 1 and truncating the fractional part. E.g. A size of 5 gives (int)($\log_2 5 + 1$), or 3. The value of 0 is reserved for a hash size of 0. The parser increments the hash size for every *name=exp* field in the table constructor.

Creating an empty table forces both array and hash sizes to be zero:

```
>local q = {}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "q"  ; 0
[1] newtable   0   0   0    ; array=0, hash=0
[2] return     0   1
; end of function
```

In later examples, we will see how the size values are encoded.

SETLIST A Bx R(A)[Bx-Bx%FPF+i] := R(A+i),
where 1 <= i <= Bx%FPF+1

Sets the values for a range of elements in a table referenced by R(A). Field Bx contains an encoding of the range of elements to set, while the values are located in the registers after R(A).

Bx is encoded using a block size, FPF. FPF is "fields per flush", coded as LFIELDS_PER_FLUSH in the source file `lopcodes.h`, with a value of 32. The remainder Bx%FPF gives the range, where 1 <= i <= Bx%FPF+1, while the starting index for the table is Bx-Bx%FPF+1. SETLIST always sets between 1 to FPF elements in a table.

Example: To set indices 33 to 64, Bx will be 63, then Bx%FPF is (63%32)=31 and the range will be 1 to (31+1)=32, while the starting index, Bx-Bx%FPF+1 is (63-31+1)=33. Thus indices 33 to 64 will be set, using values from R(A+1) to R(A+32).

We'll start with a simple example:

```
>local q = {1,2,3,4,5,}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 6 stacks
.function  0 0 0 6
.local  "q"  ; 0
.const  1  ; 0
.const  2  ; 1
.const  3  ; 2
.const  4  ; 3
.const  5  ; 4
[1] newtable   0   5   0    ; array=5, hash=0
[2] loadk      1   0        ; 1
[3] loadk      2   1        ; 2
[4] loadk      3   2        ; 3
[5] loadk      4   3        ; 4
[6] loadk      5   4        ; 5
[7] setlist    0   4        ; index 1 to 5
[8] return     0   1
; end of function
```

A table with the reference in register 0 is created in line [1] by NEWTABLE. The array part of the table has a size of 5, while the hash part has a size of 0. Constants are then loaded into temporary registers (lines [2] to [6]) before the SETLIST instruction in line [7] assigns each value to consecutive table elements. SETLIST's Bx value decodes to a block position of 0 (4 - 4%32) and an index range of 1 to 5. Table values are retrieved from temporary registers 1 to 5, since field A is 0.

Next up is a larger table. Some lines have been removed and ellipsis (...) added to save space.

```
>local q = {1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0, \
>> 1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 33 stacks
.function  0 0 0 33
.local  "q"  ; 0
.const  1  ; 0
.const  2  ; 1
...
.const  0  ; 9
[01] newtable   0   29  0    ; array=40, hash=0
[02] loadk      1   0        ; 1
[03] loadk      2   1        ; 2
[04] loadk      3   2        ; 3
...
[30] loadk      29  8        ; 9
[31] loadk      30  9        ; 0
[32] loadk      31  0        ; 1
[33] loadk      32  1        ; 2
[34] setlist    0   31       ; index 1 to 32
[35] loadk      1   2        ; 3
[36] loadk      2   3        ; 4
[37] loadk      3   4        ; 5
[38] setlist    0   34       ; index 33 to 35
[39] return     0   1
; end of function
```

Since FPF is 32, SETLIST works in blocks of 32, and the 35 elements of table **q** is split into a block with a range of 1 to 32, and a second block with a range of 33 to 35.

In line [1], NEWTABLE has a field B value of 29, or 11101 in binary. From the description of NEWTABLE, xxx is 101_2, while mmmmm is 11_2. Thus, the size of the array portion of the table is 5 x 2^3 or 40. With a maximum value of 7 x 2^31, the floating point byte is an elegant way of encoding a table size.

In line [34], SETLIST has a Bx value of 31. The starting index is (31 - 31%32 + 1) or 1, and the ending index is (1 + 31%32) or 32. Source register locations are 1 to 32 (field A is 0).

In line [38], SETLIST has a Bx value of 34. The starting index is (34 - 34%32 + 1) or 33, and the ending index is (33 + 34%32) or 35. Source register locations are 1 to 3 (field A is 0).

Here is a table with hashed elements:

```
>local q = {a=1,b=2,c=3,d=4,e=5,}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "q"  ; 0
.const  "a"  ; 0
.const  1  ; 1
.const  "b"  ; 2
.const  2  ; 3
.const  "c"  ; 4
.const  3  ; 5
.const  "d"  ; 6
.const  4  ; 7
.const  "e"  ; 8
.const  5  ; 9
[1] newtable   0   0   3    ; array=0, hash=8
[2] settable   0   250 251  ; "a" 1
[3] settable   0   252 253  ; "b" 2
[4] settable   0   254 255  ; "c" 3
[5] settable   0   256 257  ; "d" 4
[6] settable   0   258 259  ; "e" 5
[7] return     0   1
; end of function
```

In line [1], NEWTABLE is executed with an array part size of 0 and a hash part size of 8. The hash size is encoded as an exponent, so from field C, it is calculated as 2^3 = 8. Key-value pairs are set using SETTABLE; SETLIST is only for initializing array elements. Using SETTABLE to initialize the key-value pairs of a table is quite efficient as it can reference the constant pool directly.

The other table-creation instruction is SETLISTO:

SETLISTO A Bx R(A)[Bx-Bx%FPF+i] := R(A+i),
where A+1 <= A+i <= top of stack

SETLISTO is almost similar to SETLIST except that it is only used for the *last* batch of values to be set, when the *final element is a function call*. Since the function call can return a variable number of values, SETLISTO sets the table with all values from R(A+1) up to the top of the stack. The starting index is still calculated in the same way, as Bx-Bx%FPF+1. Only the range of elements to be set is now variable.

The SETLISTO instruction is generated when the final field of a table constructor is a function. The Lua language specification states that all return results of the function will be entered into the table as array elements. SETLISTO is very similar to SETLIST except that the number of values to be set is up to the top of stack. The Bx field is still used, to specify the starting table index to be set. Here is an example:

```
>return {1,2,3,a=1,b=2,c=3,foo()}
; function [0] definition (level 1)
; 0 upvalues, 0 params, 5 stacks
.function  0 0 0 5
.const  1  ; 0
.const  2  ; 1
.const  3  ; 2
.const  "a"  ; 3
.const  "b"  ; 4
.const  "c"  ; 5
.const  "foo"  ; 6
[01] newtable   0   4   2    ; array=4, hash=4
[02] loadk      1   0        ; 1
[03] loadk      2   1        ; 2
[04] loadk      3   2        ; 3
[05] settable   0   253 250  ; "a" 1
[06] settable   0   254 251  ; "b" 2
[07] settable   0   255 252  ; "c" 3
[08] getglobal  4   6        ; foo
[09] call       4   1   0
[10] setlisto   0   3        ; index 1 to top
[11] return     0   2
[12] return     0   1
; end of function
```

The table is created in line [1] with its reference in register 0, and it has both array and hash elements to be set. The size of the array part is 4 while the size of the hash part is also 4. Their sizes are encoded as 4 and 2, respectively, in fields B and C of the NEWTABLE instruction.

Lines [2]–[4] loads the values for the first 3 array elements. Lines [5]–[7] sets the 3 key-value pairs for the hashed part of the table. In line [8] and [9], the call to function **foo** is made, and then in line [10], the SETLISTO instruction sets the first 3 array elements (in registers 1 to 3,) plus whatever results returned by the **foo** function call (from register 4 onwards.) If no results are returned by the function, the top of stack is at register 3 and only the 3 constant array elements in the table are set.

14 Closures and Closing

The final two instructions of the Lua virtual machine are a little involved because of the handling of upvalues. The first is CLOSURE, for instantiating function prototypes:

CLOSURE **A Bx** R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))

Creates an instance (or closure) of a function. Bx is the function number of the function to be instantiated in the table of function prototypes. This table is located after the constant table for each function in a binary chunk. The first function prototype is numbered 0. Register R(A) is assigned the reference to the instantiated function object.

For each upvalue used by the instance of the function KPROTO[Bx], there is a pseudo-instruction that follows CLOSURE. Each upvalue corresponds to either a MOVE or a GETUPVAL pseudo-instruction. Only the B field on either of these pseudo-instructions are significant.

A MOVE corresponds to local variable R(B) in the current lexical block, which will be used as an upvalue in the instantiated function. A GETUPVAL corresponds upvalue number B in the current lexical block. The VM uses these pseudo-instructions to manage upvalues.

If the function prototype has no upvalues, then CLOSURE is pretty straightforward: Bx has the function number and R(A) is assigned the reference to the instantiated function object. However, when an upvalue comes into the picture, we have to look a little more carefully:

```
>local u; \
>>function p() return u end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "u"  ; 0
.const  "p"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .upvalue  "u"  ; 0
  [1] getupval   0   0        ; u
  [2] return     0   2
  [3] return     0   1
  ; end of function

[1] loadnil    0   0
[2] closure    1   0        ; 1 upvalues
[3] move       0   0
[4] setglobal  1   0        ; p
[5] return     0   1
; end of function
```

In the example, the upvalue is **u**, and within the main chunk there is a single function prototype (indented in the listing above for clarity.) In the top-level function, line [2], the closure is made. In line [4] the function reference is saved into global **p**. Line [3] is a part of the CLOSURE instruction (it not really an actual MOVE,) and its B field specifies that upvalue number 0 in the closed function is really local **u** in the enclosing function.

Here is another example, with 3 levels of function prototypes:

```
>local m \
>>function p() \
>>  local n \
>>  function q() return m,n end \
>>end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "m"  ; 0
.const  "p"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .local  "n"  ; 0
  .upvalue  "m"  ; 0
  .const  "q"  ; 0

    ; function [0] definition (level 3)
    ; 2 upvalues, 0 params, 2 stacks
    .function  2 0 0 2
    .upvalue  "m"  ; 0
    .upvalue  "n"  ; 1
    [1] getupval   0   0        ; m
    [2] getupval   1   1        ; n
    [3] return     0   3
    [4] return     0   1
    ; end of function

  [1] loadnil    0   0
  [2] closure    1   0        ; 2 upvalues
  [3] getupval   0   0        ; m
  [4] move       0   0
  [5] setglobal  1   0        ; q
  [6] return     0   1
  ; end of function

[1] loadnil    0   0
[2] closure    1   0        ; 1 upvalues
[3] move       0   0
[4] setglobal  1   0        ; p
[5] return     0   1
; end of function
```

First, look at the top-level function and the level 2 function – there is one upvalue, **m**. In the top-level function, the closure in line [2] has one more instruction following it, for the upvalue **m**. This is similar to the previous example.

Next, compare the level 2 function and the level 3 function – now there are two upvalues, **m** and **n**. The **m** upvalue is found 2 levels up. In the level 2 function, the closure in line [2] has two instructions following it. The first is for upvalue number 0 (**m**) – it uses GETUPVAL to indicate that the upvalue is one or more level lower down. The second is for upvalue number 1 (**n**) – it uses MOVE which indicate that the upvalue is in the same level as the CLOSURE instruction. For both of these pseudo-instructions, the B field is used to point either to the upvalue or local in question. The Lua virtual machine uses this information (CLOSURE information and upvalue lists) to manage upvalues; for the programmer, upvalues just works.

Our last instruction also deals with upvalues:

CLOSE	**A**	close all variables in the stack up to (>=) R(A)

Closes all local variables in the stack from register R(A) onwards. This instruction is only generated if there is an upvalue present within those local variables. It has no effect if a local isn't used as an upvalue.

If a local is used as an upvalue, then the local variable need to be placed somewhere, otherwise it will go out of scope and disappear when a lexical block enclosing the local variable ends. CLOSE performs this operation for all affected local variables for **do end** blocks or loop blocks. RETURN also does an implicit CLOSE when a function returns.

It is easier to understand with an example:

```
>do \
>>  local p,q \
>>  r = function() return p,q end \
>>end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 3 stacks
.function  0 0 0 3
.local  "p"  ; 0
.local  "q"  ; 1
.const  "r"  ; 0

  ; function [0] definition (level 2)
  ; 2 upvalues, 0 params, 2 stacks
  .function  2 0 0 2
  .upvalue  "p"  ; 0
  .upvalue  "q"  ; 1
  [1] getupval   0   0        ; p
  [2] getupval   1   1        ; q
  [3] return     0   3
  [4] return     0   1
  ; end of function

[1] loadnil    0   1
[2] closure    2   0        ; 2 upvalues
[3] move       0   0
[4] move       0   1
[5] setglobal  2   0        ; r
[6] close      0
[7] return     0   1
; end of function
```

p and **q** are local to the **do end** block, and they are upvalues as well. The global **r** is assigned an anonymous function that has **p** and **q** as upvalues. When **p** and **q** go out of scope at the end of the **do end** block, both variables have to be put somewhere because they are part of the environment of the function instantiated in **r**. This is where the CLOSE instruction comes in.

In the top-level function, the CLOSE in line [6] makes the virtual machine find all affected locals (they have to be open upvalues,) take them out of the stack, and place them in a safe place so that they do not disappear when the block or function goes out of scope. A RETURN instruction does an implicit CLOSE so the latter won't appear very often in listings.

Here is another example which illustrates a rather subtle point with CLOSE (thanks to Rici Lake for this nugget):

```
>do \
>>  local p \
>>  while true do \
>>    q = function() return p end \
>>    break \
>>  end \
>>end
; function [0] definition (level 1)
; 0 upvalues, 0 params, 2 stacks
.function  0 0 0 2
.local  "p"  ; 0
.const  "q"  ; 0

  ; function [0] definition (level 2)
  ; 1 upvalues, 0 params, 2 stacks
  .function  1 0 0 2
  .upvalue  "p"  ; 0
  [1] getupval   0   0        ; p
  [2] return     0   2
  [3] return     0   1
  ; end of function

[1] loadnil    0   0
[2] jmp        4           ; to [7]
[3] closure    1   0       ; 1 upvalues
[4] move       0   0
[5] setglobal  1   0       ; q
[6] jmp        1           ; to [8]
[7] jmp        -5          ; to [3]
[8] close      0
[9] return     0   1
; end of function
```

In the above example, a function is instantiated within a loop. In real-world code, a loop may instantiate a number of such functions. Each of these functions will have its own **p** upvalue. The subtle point is that the **break** (the JMP on line [6]) does not jump to the RETURN instruction in line [9]; instead it reaches the CLOSE instruction on line [8]. Whether or not execution exits a loop normally or through a **break**, the code within the loop may have caused the instantiation of one or more functions and their associated upvalues. Thus the enclosing **do end** block must execute its CLOSE instruction; if we always remember to associate the CLOSE with the **do end** block, there will be no confusion.

15 Digging Deeper

For studying larger snippets of Lua code and its disassembly, you can try ChunkSpy's various disassembly functions. Both vmmerge5 and ChunkSpy can merge source code lines into a disassembly listing. ChunkSpy can provide more detail, because it processes every bit of a binary chunk.

A good way of studying how any instruction functions is to find where its opcode appears in the Lua sources. For example, to see what MOVE does, look for OP_MOVE in `lparser.c` (the parser), `lcode.c` (the code generator) and `lvm.c` (the virtual machine.) From the code implementing OP_MOVE, you can then move deeper into the code by following function calls. I found this approach (bottoms up, following the execution path from generated opcodes to the functions that performs code generation) is a little easier than following the recursive descent parser's call graph. Once you have lots of little pictures, the big picture will form on its own.

I hope you have enjoyed, as I did, poking your way through the internal organs of this Lua thingy. Now that the Lua internals seem less magical and more practical, I look forward to some Dr Frankenstein experiments with my newfound knowledge...

16 Acknowledgements

The author gratefully acknowledges valuable feedback from Rici Lake and Klaas-Jan Stol.

17 ChangeLog & ToDos

Changes:

20050106 Typo. Fixed Size of Instruction field on page 7, to 4 bytes (was 8 bytes.)
20050118 Page 7: Added note on how a binary chunk is recognized by the loader.
20060221 *Corrected or clarified a few things, from Klaas-Jan Stol's feedback:*
Page 23: Mistake in CALL instruction description.
The line: If B is 2 or more, there are (B-1) return values.
Should be: If B is 2 or more, there are (B-1) parameters.
Page 28: Clarified the possible values of A for EQ, LT and LE. Added:
The Lua code generator produces either 0 or 1 for the boolean A.
Page 38: Mistake in formula for the starting index of SETLIST.
The formula Bx-Bx%FPF+i should be Bx-Bx%FPF+1.
Changes made using feedback from Rici Lake:
Page 14: Clarified the meaning of field A.
Page 26: Noted that only Lua functions can be tailcalled. Fixed explanation for the RETURN instruction after TAILCALL.
Page 45: Added a second example for CLOSE illustrating its subtleties.