LASM: The mysterious language under the hood of Lua

A quick introduction and tutorial using the Lua Assembly Tools LASM Parser

LASM (Lua Assembly) is what all Lua scripts eventually compile into. It is much more difficult to actually do anything in opposed to Lua, but you can do a lot more than in plain Lua. To learn about the opcodes and basic LASM code structure, please read the ANoFrillsIntroToLua51VMInstructions.pdf file by Kein-Hong man, available on LuaForge and in the etc\docs folder.

**LAT (Lua Assembly Tools) LASM**

LAT Lua Assembly is simple. There are three types of statements: Controls (beginning with a dot), Opcodes, and Comments. Constants can be automatically managed for you, strings are the same as in Lua, numbers can be in decimal, hexadecimal, binary, or octal, and can have underscores in them.

Constants can either be a string (such as ‘print’), **true**, **false**, a number, or **nil**/**null**. The way to use a constant is, if it’s not a number, to just put it in code. You can also use a function-call like structure using “k”, “const”, or “constant”, e.g. k(7), or const’str’. To use a function by name, use the “p” or “proto” function, like so: p’FuncName’ or proto(FunctionX).

Comments start with a semicolon ‘;’ and end with a newline.

The controls are as follows (an Ident is like a variable name):

[ ] = optional, < > = required

.const <Constant>

.name <String or Ident>

.options [Number upvalue count] [Number argument count] [Number vararg flag] [Number MaxStackSize]

.local <Name or Ident>

.upval or .upvalue <Name or Ident>

.stacksize or .maxstacksize <Number>

.vararg <Number>

.params or .args or .argcount <Number>

.func or .function [String name] – begins a new function in the current one

.end – ends a function

Opcode format:

<Opcode name> <Argument A> <Argument B> <Argument C>

<Opcode name> <Argument A> <Argument Bx>

<Opcode name> <Argument A> <Argument sBx>

Opcode names are not case sensitive. An argument can either be a Number, a Constant (see above for the many ways to automanage them), a proto (see above for how to use them), or a number that starts with ‘$’ or ‘R’ or ‘r’

You don’t need to explicitly define the RETURN opcode at the end of a function, the LAT parser will automatically insert the default one if none is specified.

That’s about it for LAT LASM! Pretty simple, huh? Now for some examples:

**Mandatory Hello World sample:**

getglobal 0 ‘print’ ; Loads the global variable ‘print’ into register 0

loadk 1 ‘Hello World’ ; Loads the constant ‘Hello world’ into register 1, using the magical constant manager)

call 0 2 1 ; Calls ‘print’, keeping no results

**AddTen function (included in samples/output dir):**

.func 'AddFunction' ; Defines a new function, called AddFunction

.options 0 1 ; Sets the upvalue count to Zero and the argument count to 1

add 0 0 k(10) ; Adds 10 into register 0, getting the value from register 0 (which is the argument), and then adding 10 into it using an auto-managed const

return 0 2 ; Returns the result of the ADD

.end

closure 0 p'AddFunction' ; Creates a Closure in register 0 from the Proto at index 0, whose name is ‘AddFunction’, by using the magical proto manager

getglobal 1 '\_G' ; Loads the global table (\_G) into register 1 (again, using automanaged constants for ‘\_G’)

settable 1 'AddTen' 0 ; Sets the key ‘AddTen’ into the table in register one (\_G), the value of register 0 – the closure