LuaJIT 2.0 Bytecode Instructions

Table of Contents

Introduction

The following document describes the LuaJIT 2.0 bytecode instructions. See `src/lj_bc.h` in the LuaJIT source code for details. The bytecode can be listed with `luajit -bl`, see the -b option.

A single bytecode instruction is 32 bit wide and has an 8 bit opcode field and several operand fields of 8 or 16 bit. Instructions come in one of two formats:

B	C	A	OP
D		A	OP

The figure shows the least-significant bit on the right. In-memory instructions are always stored in host byte order. E.g. 0xbbccaa1e is the instruction with opcode 0x1e (`ADDVV`), with operands A = 0xaa, B = 0xbb and C = 0xcc.

The suffix(es) of the instruction name distinguish variants of the same basic instruction:

- V variable slot
- S string constant
- N number constant
- P primitive type
- B unsigned byte literal
- M multiple arguments/results

Here are the possible operand types:

- (none): unused operand
- var: variable slot number
- dst: variable slot number, used as a destination
- base: base slot number, read-write
- rbase: base slot number, read-only
- uv: upvalue number
- lit: literal
- lits: signed literal
- pri: primitive type (0 = nil, 1 = false, 2 = true)
- num: number constant, index into constant table
- str: string constant, negated index into constant table
- tab: template table, negated index into constant table
- func: function prototype, negated index into constant table
- cdata: cdata constant, negated index into constant table
- jump: branch target, relative to next instruction, biased with 0x8000

Comparison ops

All comparison and test ops are immediately followed by a `JMP` instruction which holds the target of the conditional jump. All comparisons and tests jump to the target if the comparison or test is true. Otherwise they fall through to the instruction *after* the `JMP`.

OP	A	D	Description
ISLT	var	var	Jump if A < D
ISGE	var	var	Jump if A ≥ D
ISLE	var	var	Jump if A ≤ D
ISGT	var	var	Jump if A > D
ISEQV	var	var	Jump if A = D
ISNEV	var	var	Jump if A ≠ D
ISEQS	var	str	Jump if A = D
ISNES	var	str	Jump if A ≠ D
ISEQN	var	num	Jump if A = D
ISNEN	var	num	Jump if A ≠ D
ISEQP	var	pri	Jump if A = D
ISNEP	var	pri	Jump if A ≠ D

Q: Why do we need four different ordered comparisons? Wouldn't `<` and `<=` suffice with appropriately swapped operands?

A: No, because for floating-point comparisons `(x < y)` is *not* the same as `not (x >= y)` in the presence of NaNs.

The LuaJIT parser preserves the ordered comparison semantics of the source code as follows:

Source code	Bytecode
if x < y then	ISGE x y
if x <= y then	ISGT x y
if x > y then	ISGE y x
if x >= y then	ISGT y x
if not (x < y) then	ISLT x y
if not (x <= y) then	ISLE x y
if not (x > y) then	ISLT y x
if not (x >= y) then	ISLE y x

(In)equality comparisons are swapped as needed to bring constants to the right.

Unary Test and Copy ops

These instructions test whether a variable evaluates to true or false in a boolean context. In Lua only `nil` and `false` are considered false, all other values are true. These instructions are generated for simple truthness tests like `if x then` or when evaluating the `and` and `or` operators.

OP	A	D	Description
ISTC	dst	var	Copy D to A and jump, if D is true
ISFC	dst	var	Copy D to A and jump, if D is false
IST		var	Jump if D is true
ISF		var	Jump if D is false

Q: What do we need the test and copy ops for?

A: In Lua the `and` and `or` operators return the original value of one of their operands. It's generally only known whether the result is unused after parsing the full expression. In this case the test and copy ops can easily be turned into test ops in the previously emitted bytecode.

Unary ops

OP	A	D	Description
MOV	dst	var	Copy D to A
NOT	dst	var	Set A to boolean not of D
UNM	dst	var	Set A to -D (unary minus)
LEN	dst	var	Set A to #D (object length)

Binary ops

OP	A	B	C	Description
ADDVN	dst	var	num	A = B + C

SUBVN	dst	var	num	A = B - C
MULVN	dst	var	num	A = B * C
DIVVN	dst	var	num	A = B / C
MODVN	dst	var	num	A = B % C
ADDNV	dst	var	num	A = C + B
SUBNV	dst	var	num	A = C - B
MULNV	dst	var	num	A = C * B
DIVNV	dst	var	num	A = C / B
MODNV	dst	var	num	A = C % B
ADDVV	dst	var	var	A = B + C
SUBVV	dst	var	var	A = B - C
MULVV	dst	var	var	A = B * C
DIVVV	dst	var	var	A = B / C
MODVV	dst	var	var	A = B % C
POW	dst	var	var	A = B ^ C
CAT	dst	rbase	rbase	A = B .. ~ .. C

Note: The `CAT` instruction concatenates all values in variable slots B to C inclusive.

Constant ops

OP	A	D	Description
KSTR	dst	str	Set A to string constant D
KCDATA	dst	cdata	Set A to cdata constant D
KSHORT	dst	lits	Set A to 16 bit signed integer D
KNUM	dst	num	Set A to number constant D
KPRI	dst	pri	Set A to primitive D
KNIL	base	base	Set slots A to D to nil

Note: A single `nil` value is set with `KPRI`. `KNIL` is only used when multiple values need to be set to `nil`.

Upvalue and Function ops

OP	A	D	Description
UGET	dst	uv	Set A to upvalue D
USETV	uv	var	Set upvalue A to D
USETS	uv	str	Set upvalue A to string constant D
USETN	uv	num	Set upvalue A to number constant D
USETP	uv	pri	Set upvalue A to primitive D
UCLO	rbase	jump	Close upvalues for slots ≥ rbase and jump to target D

FNEW	dst	func	Create new closure from prototype D and store it in A

Q: Why does `UCLO` have a jump target?

A: `UCLO` is usually the last instruction in a block and is often followed by a `JMP`. Merging the jump into `UCLO` speeds up execution and simplifies some bytecode fixup steps (see `fs_fixup_ret()` in `src/lj_parse.c`). A non-branching `UCLO` simply jumps to the next instruction.

Table ops

OP	A	B	C/D	Description
TNEW	dst		lit	Set A to new table with size D (see below)
TDUP	dst		tab	Set A to duplicated template table D
GGET	dst		str	A = _G[D]
GSET	var		str	_G[D] = A
TGETV	dst	var	var	A = B[C]
TGETS	dst	var	str	A = B[C]
TGETB	dst	var	lit	A = B[C]
TSETV	var	var	var	B[C] = A
TSETS	var	var	str	B[C] = A
TSETB	var	var	lit	B[C] = A
TSETM	base		num*	(A-1)[D], (A-1)[D+1], ... = A, A+1, ...

Notes:

- The 16 bit literal D operand of `TNEW` is split up into two fields: the lowest 11 bits give the array size (allocates slots 0..asize-1, or none if zero). The upper 5 bits give the hash size as a power of two (allocates 2^hsize hash slots, or none if zero).
- `GGET` and `GSET` are named 'global' get and set, but actually index the current function environment `getfenv(1)` (which is usually the same as `_G`).
- `TGETB` and `TSETB` interpret the 8 bit literal C operand as an unsigned integer index (0..255) into table B.
- Operand D of `TSETM` points to a biased floating-point number in the constant table. Only the lowest 32 bits from the mantissa are used as a starting table index. MULTRES from the previous bytecode gives the number of table slots to fill.

Calls and Vararg Handling

All call instructions expect a special setup: the function (or object) to be called is in slot A, followed by the arguments in consecutive slots. Operand C is one plus the number of fixed arguments. Operand B is one plus the number of return values, or zero for calls which return all results (and set MULTRES accordingly).

Operand C for calls with multiple arguments (`CALLM` or `CALLMT`) is set to the number of fixed arguments. MULTRES is added to that to get the actual number of arguments to pass.

For consistency, the specialized call instructions `ITERC`, `ITERN` and the vararg instruction `VARG` share the same operand format. Operand C of `ITERC` and `ITERN` is always 3 = 1+2, i.e. two arguments are passed to the iterator function. Operand C of `VARG` is repurposed to hold the number of fixed arguments of the enclosing function. This speeds up access to the variable argument part of the vararg pseudo-frame below.

MULTRES is an internal variable that keeps track of the number of results returned by the previous call or by `VARG` instructions with multiple results. It's used by calls (`CALLM` or `CALLMT`) or returns (`RETM`) with multiple arguments and by a table initializer (`TSETM`).

OP	A	B	C/D	Description
CALLM	base	lit	lit	Call: A, ..., A+B-2 = A(A+1, ..., A+C+MULTRES)
CALL	base	lit	lit	Call: A, ..., A+B-2 = A(A+1, ..., A+C-1)
CALLMT	base		lit	Tailcall: return A(A+1, ..., A+D+MULTRES)
CALLT	base		lit	Tailcall: return A(A+1, ..., A+D-1)
ITERC	base	lit	lit	Call iterator: A, A+1, A+2 = A-3, A-2, A-1; A, ..., A+B-2 = A(A+1, A+2)
ITERN	base	lit	lit	Specialized ITERC, if iterator function A-3 is next()
VARG	base	lit	lit	Vararg: A, ..., A+B-2 = ...
ISNEXT	base		jump	Verify ITERN specialization and jump

Note: The Lua parser heuristically determines whether `pairs()` or `next()` might be used in a loop. In this case, the `JMP` and the iterator call `ITERC` are replaced with the specialized versions `ISNEXT` and `ITERN`.

`ISNEXT` verifies at runtime that the iterator actually is the `next()` function, that the argument is a table and that the control variable is `nil`. Then it sets the lowest 32 bits of the slot for the control variable to zero and jumps to the iterator call, which uses this number to efficiently step through the keys of the table.

If any of the assumptions turn out to be wrong, the bytecode is despecialized at runtime back to `JMP` and `ITERC`.

Returns

All return instructions copy the results starting at slot A down to the slots starting at one below the base slot (the slot holding the frame link and the currently executing function).

The `RET0` and `RET1` instructions are just specialized versions of `RET`. Operand D is one plus the number of results to return.

For `RETM`, operand D holds the number of fixed results to return. MULTRES is added to that to get the actual number of results to return.

OP	A	D	Description
RETM	base	lit	return A, ..., A+D+MULTRES-1
RET	rbase	lit	return A, ..., A+D-2

RET0	rbase	lit	return
RET1	rbase	lit	return A

Loops and branches

The Lua language offers four loop types, which are translated into different bytecode instructions:

- The numeric 'for' loop: `for i=start,stop,step do body end` => set start,stop,step `FORI` body `FORL`
- The iterator 'for' loop: `for vars... in iter,state,ctl do body end` => set iter,state,ctl `JMP` body `ITERC` `ITERL`
- The 'while' loop: `while cond do body end` => inverse-cond- `JMP` `LOOP` body `JMP`
- The 'repeat' loop: `repeat body until cond` => `LOOP` body cond- `JMP`

The `break` and `goto` statements are translated into unconditional `JMP` or `UCLO` instructions.

OP	A	D	Description
FORI	base	jump	Numeric 'for' loop init
JFORI	base	jump	Numeric 'for' loop init, JIT-compiled
FORL	base	jump	Numeric 'for' loop
IFORL	base	jump	Numeric 'for' loop, force interpreter
JFORL	base	lit	Numeric 'for' loop, JIT-compiled
ITERL	base	jump	Iterator 'for' loop
IITERL	base	jump	Iterator 'for' loop, force interpreter
JITERL	base	lit	Iterator 'for' loop, JIT-compiled
LOOP	rbase	jump	Generic loop
ILOOP	rbase	jump	Generic loop, force interpreter
JLOOP	rbase	lit	Generic loop, JIT-compiled
JMP	rbase	jump	Jump

Operand A holds the first unused slot for the `JMP` instruction, the base slot for the loop control variables of the `*FOR*` instructions (`idx` , `stop` , `step` , `ext idx`) or the base of the returned results from the iterator for the `*ITERL` instructions (stored below are `func` , `state` and `ctl`).

The `JFORL` , `JITERL` and `JLOOP` instructions store the trace number in operand D (`JFORI` retrieves it from the corresponding `JFORL`). Otherwise, operand D points to the first instruction after the loop.

The `FORL` , `ITERL` and `LOOP` instructions do hotspot detection. Trace recording is triggered if the loop is executed often enough.

The `IFORL` , `IITERL` and `ILOOP` instructions are used by the JIT-compiler to blacklist loops that cannot be compiled. They don't do hotspot detection and force execution in the interpreter.

The `JFORI` , `JFORL` , `JITERL` and `JLOOP` instructions enter a JIT-compiled trace if the loop-entry condition is true.

The `*FORL` instructions do `idx = idx + step` first. All `*FOR*` instructions check that `idx <= stop` (if `step >= 0`) or `idx >= stop` (if `step < 0`). If true, `idx` is copied to the `ext idx` slot (visible loop variable in the loop body). Then the loop body or the JIT-compiled trace is entered. Otherwise, the loop is left by continuing with the next instruction after the `*FORL`.

The `*ITERL` instructions check that the first result returned by the iterator in slot A is non-`nil`. If true, this value is copied to slot A-1 and the loop body or the JIT-compiled trace is entered.

The `*LOOP` instructions are actually no-ops (except for hotspot detection) and don't branch. Operands A and D are only used by the JIT-compiler to speed up data-flow and control-flow analysis. The bytecode instruction itself is needed so the JIT-compiler can patch it to enter the JIT-compiled trace for the loop.

Function headers

OP	A	D	Description
FUNCF	rbase		Fixed-arg Lua function
IFUNCF	rbase		Fixed-arg Lua function, force interpreter
JFUNCF	rbase	lit	Fixed-arg Lua function, JIT-compiled
FUNCV	rbase		Vararg Lua function
IFUNCV	rbase		Vararg Lua function, force interpreter
JFUNCV	rbase	lit	Vararg Lua function, JIT-compiled
FUNCC	rbase		Pseudo-header for C functions
FUNCCW	rbase		Pseudo-header for wrapped C functions
FUNC*	rbase		Pseudo-header for fast functions

Operand A holds the frame size of the function. Operand D holds the trace-number for `JFUNCF` and `JFUNCV`.

For Lua functions, omitted fixed arguments are set to `nil` and excess arguments are ignored. Vararg function setup involves creating a special vararg frame that holds the arguments beyond the fixed arguments. The fixed arguments are copied up to a regular Lua function frame and their slots in the vararg frame are set to `nil`.

The `FUNCF` and `FUNCV` instructions set up the frame for a fixed-arg or vararg Lua function and do hotspot detection. Trace recording is triggered if the function is executed often enough.

The `IFUNCF` and `IFUNCV` instructions are used by the JIT-compiler to blacklist functions that cannot be compiled. They don't do hotspot detection and force execution in the interpreter.

The `JFUNCF` and `JFUNCV` instructions enter a JIT-compiled trace after the initial setup.

The `FUNCC` and `FUNCCW` instructions are pseudo-headers pointed to by the `pc` field of C closures. They are never emitted and are only used for dispatching to the setup code for C function calls.

All higher-numbered bytecode instructions are used as pseudo-headers for fast functions. They are never emitted and are only used for dispatching to the machine code for the corresponding fast functions.

LuaJIT 2.0 Bytecode Dump Format

LuaJIT bytecode dump format is produced using `luajit -b` or `string.dump` function. It can be saved to file and loaded later, instead of storing plain Lua source, occupying more space and taking longer to load.

Details for the bytecode dump format can be found in `src/lj_bcdump.h` in the LuaJIT source code. Here's the concise format description:

```
dump   = header proto+ 0U
header = ESC 'L' 'J' versionB flagsU [namelenU nameB*]
proto  = lengthU pdata
pdata  = phead bcinsW* uvdataH* kgc* knum* [debugB*]
phead  = flagsB numparamsB framesizeB numuvB numkgcU numknU numbcU
         [debuglenU [firstlineU numlineU]]
kgc    = kgctypeU { ktab | (loU hiU) | (rloU rhiU iloU ihiU) | strB* }
knum   = intU0 | (loU1 hiU)
ktab   = narrayU nhashU karray* khash*
karray = ktabk
khash  = ktabk ktabk
ktabk  = ktabtypeU { intU | (loU hiU) | strB* }

B = 8 bit, H = 16 bit, W = 32 bit, U = ULEB128 of W, U0/U1 = ULEB128 of W+1
```

TODO: turn the description into human-readable text :-)

The dump starts with magic `\x1bLJ`. After the magic comes version number, which indicates the version of bytecode. Different versions are not compatible. At the time of writing, current version number is `1` and is defined by `BCDUMP_VERSION` macro in `src/lj_dump.h`. Next, `BCDUMP_F_{STRIP, BE, FFI}` bit flags (found in `src/lj_dump.h`) are encoded using ULEB128. If `BCDUMP_F_STRIP` flag is not set, next comes ULEB128-encoded chunk name's length and it itself right after length, otherwise this step is skipped.

TODO: what does `lj_bcwrite.c:370` `ctx->status = ctx->wfunc(ctx->L, ctx->sb.buf, ctx->sb.n, ctx->wdata);` do exactly?

TODO: more information about `GCproto`

Next, the `GCproto` objects are written which carry the the bytecode. Notice the plural *objects*, there's one object per function. Objects are written deepest, first first, i.e.:

```
function a()
 function b()
  print(1)
 end
 return b
end
a()()
```

First `b`, then `a` and then the rest of the scope is written.

At the end there is a `\0` byte, which signals EOF for `bcread_proto`.

BCDUMP_F_* flags

TODO

GCProto

TODO

Last edited by **Constantine (fst3a)**, 2017-04-02 23:09:41