Lua Bit Operations Module

Lua BitOp is a C extension module for Lua 5.1/5.2 which adds **bitwise operations** on numbers.

Features

- Supported functions: `bit.tobit, bit.tohex, bit.bnot, bit.band, bit.bor, bit.bxor, bit.lshift, bit.rshift, bit.arshift, bit.rol, bit.ror, bit.bswap`
- Consistent semantics across 16, 32 and 64 bit platforms.
- Supports different lua_Number types: either IEEE 754 doubles, int32_t or int64_t.
- Runs on Linux, *BSD, Mac OS X, Windows and probably anything else you can find.
- Simple installation on all systems. No bulky configure scripts.
- Embedded-systems-friendly.
- Internal self-test on startup to detect miscompiles. Includes a comprehensive test and benchmark suite.
- Compatible with the built-in bitwise operations in LuaJIT 2.0.
- It's as fast as you can get with the standard Lua/C API.

API Functions

This list of API functions is not intended to replace a tutorial. If you are not familiar with the terms used, you may want to study the Wikipedia article on bitwise operations first.

Defining Shortcuts

It's a common (but not a required) practice to cache often used module functions in locals. This serves as a shortcut to save some typing and also speeds up resolving them (only relevant if called hundreds of thousands of times).

```
local bnot = bit.bnot
local band, bor, bxor = bit.band, bit.bor, bit.bxor
local lshift, rshift, rol = bit.lshift, bit.rshift, bit.rol
-- etc...

-- Example use of the shortcuts:
local function tr_i(a, b, c, d, x, s)
   return rol(bxor(c, bor(b, bnot(d))) + a + x, s) + b
end
```

Remember that **`and`**, **`or`** and **`not`** are reserved keywords in Lua. They cannot be used for variable names or literal field names. That's why the corresponding bitwise functions have been named `band`, `bor`, and `bnot` (and `bxor` for consistency).
While we are at it: a common pitfall is to use `bit` as the name of a local temporary variable — well, don't! :-)

About the Examples

The examples below show small Lua one-liners. Their expected output is shown after `-->`. This is interpreted as a comment marker by Lua so you can cut & paste the whole line to a Lua prompt and experiment with it.
Note that all bit operations return signed 32 bit numbers (rationale). And these print as signed decimal numbers by default.
For clarity the examples assume the definition of a helper function `printx()`. This prints its argument as an unsigned 32 bit hexadecimal number on all platforms:

```
function printx(x)
   print("0x"..bit.tohex(x))
end
```

Bit Operations

y = bit.tobit(x)

Normalizes a number to the numeric range for bit operations and returns it. This function is usually not needed since all bit operations already normalize all of their input arguments. Check the operational semantics for details.

```
print(0xffffffff)               --> 4294967295 (*)
print(bit.tobit(0xffffffff))    --> -1
printx(bit.tobit(0xffffffff))   --> 0xffffffff
print(bit.tobit(0xffffffff + 1)) --> 0
print(bit.tobit(2^40 + 1234))   --> 1234
```

(*) See the treatment of hex literals for an explanation why the printed numbers in the first two lines differ (if your Lua installation uses a `double` number type).

y = bit.tohex(x [,n])

Converts its first argument to a hex string. The number of hex digits is given by the absolute value of the optional second argument. Positive numbers between 1 and 8 generate lowercase hex digits. Negative numbers generate uppercase hex digits. Only the least-significant 4*|n| bits are used. The default is to generate 8 lowercase hex digits.

```
print(bit.tohex(1))             --> 00000001
print(bit.tohex(-1))            --> ffffffff
print(bit.tohex(0xffffffff))    --> ffffffff
print(bit.tohex(-1, -8))        --> FFFFFFFF
print(bit.tohex(0x21, 4))       --> 0021
print(bit.tohex(0x87654321, 4)) --> 4321
```

y = bit.bnot(x)

Returns the bitwise **not** of its argument.

```
print(bit.bnot(0))          --> -1
printx(bit.bnot(0))         --> 0xffffffff
print(bit.bnot(-1))         --> 0
print(bit.bnot(0xffffffff)) --> 0
printx(bit.bnot(0x12345678)) --> 0xedcba987
```

y = bit.bor(x1 [,x2...])
y = bit.band(x1 [,x2...])
y = bit.bxor(x1 [,x2...])

Returns either the bitwise **or**, bitwise **and**, or bitwise **xor** of all of its arguments. Note that more than two arguments are allowed.

```
print(bit.bor(1, 2, 4, 8))               --> 15
printx(bit.band(0x12345678, 0xff))       --> 0x00000078
printx(bit.bxor(0xa5a5f0f0, 0xaa55ff00)) --> 0x0ff00ff0
```

y = bit.lshift(x, n)
y = bit.rshift(x, n)
y = bit.arshift(x, n)

Returns either the bitwise **logical left-shift**, bitwise **logical right-shift**, or bitwise**arithmetic right-shift** of its first argument by the number of bits given by the second argument.
Logical shifts treat the first argument as an unsigned number and shift in 0-bits.
Arithmetic right-shift treats the most-significant bit as a sign bit and replicates it.
Only the lower 5 bits of the shift count are used (reduces to the range [0..31]).

```
print(bit.lshift(1, 0))             --> 1
print(bit.lshift(1, 8))             --> 256
print(bit.lshift(1, 40))            --> 256
print(bit.rshift(256, 8))           --> 1
print(bit.rshift(-256, 8))          --> 16777215
print(bit.arshift(256, 8))          --> 1
print(bit.arshift(-256, 8))         --> -1
printx(bit.lshift(0x87654321, 12))  --> 0x54321000
printx(bit.rshift(0x87654321, 12))  --> 0x00087654
printx(bit.arshift(0x87654321, 12)) --> 0xfff87654
```

y = bit.rol(x, n)
y = bit.ror(x, n)

Returns either the bitwise **left rotation**, or bitwise **right rotation** of its first argument by the number of bits given by the second argument. Bits shifted out on one side are shifted back in on the other side.
Only the lower 5 bits of the rotate count are used (reduces to the range [0..31]).

```
printx(bit.rol(0x12345678, 12))  --> 0x45678123
printx(bit.ror(0x12345678, 12))  --> 0x67812345
```

`y = bit.bswap(x)`

Swaps the bytes of its argument and returns it. This can be used to convert little-endian 32 bit numbers to big-endian 32 bit numbers or vice versa.

```
printx(bit.bswap(0x12345678)) --> 0x78563412
printx(bit.bswap(0x78563412)) --> 0x12345678
```

Example Program

This is an implementation of the (naïve) Sieve of Eratosthenes algorithm. It counts the number of primes up to some maximum number.
A Lua table is used to hold a bit-vector. Every array index has 32 bits of the vector. Bitwise operations are used to access and modify them. Note that the shift counts don't need to be masked since this is already done by the BitOp shift and rotate functions.

```
local bit = require("bit")
local band, bxor = bit.band, bit.bxor
local rshift, rol = bit.rshift, bit.rol

local m = tonumber(arg and arg[1]) or 100000
if m < 2 then m = 2 end
local count = 0
local p = {}

for i=0,(m+31)/32 do p[i] = -1 end

for i=2,m do
  if band(rshift(p[rshift(i, 5)], i), 1) ~= 0 then
    count = count + 1
    for j=i+i,m,i do
      local jx = rshift(j, 5)
      p[jx] = band(p[jx], rol(-2, j))
    end
  end
end

print(string.format("Found %d primes up to %d\n", count, m))
```

Lua BitOp is quite fast. This program runs in less than 90 milliseconds on a 3 GHz CPU with a standard Lua installation, but performs more than a million calls to bitwise functions. If you're looking for even more speed, check out LuaJIT.

Caveats

Signed Results

Returning signed numbers from bitwise operations may be surprising to programmers coming from other programming languages which have both signed and unsigned types. But as long as you treat the results of bitwise operations uniformly everywhere, this shouldn't cause any problems.
Preferably format results with `bit.tohex` if you want a reliable unsigned string representation. Avoid the "%x" or "%u" formats for `string.format`. They fail on some architectures for negative numbers and can return more than 8 hex digits on others.
You may also want to avoid the default number to string coercion, since this is a signed conversion. The coercion is used for string concatenation and all standard library functions which accept string arguments (such as `print()` or `io.write()`).

Conditionals

If you're transcribing some code from C/C++, watch out for bit operations in conditionals. In C/C++ any non-zero value is implicitly considered as "true". E.g. this C code:

```
if (x & 3) ...
```

must not be turned into this Lua code:

```
if band(x, 3) then ... -- wrong!
```

In Lua all objects except `nil` and `false` are considered "true". This includes all numbers. An explicit comparison against zero is required in this case:

```
if band(x, 3) ~= 0 then ... -- correct!
```

Comparing Against Hex Literals

Comparing the results of bitwise operations (signed numbers) against hex literals (unsigned numbers) needs some additional care. The following conditional expression may or may not work right, depending on the platform you run it on:

```
bit.bor(x, 1) == 0xffffffff
```

E.g. it's never true on a Lua installation with the default number type. Some simple solutions:

- Either never use hex literals larger than 0x7fffffff in comparisons:
  ```
  bit.bor(x, 1) == -1
  ```
- Or convert them with `bit.tobit()` before comparing:
  ```
  bit.bor(x, 1) == bit.tobit(0xffffffff)
  ```
- Or use a generic workaround with `bit.bxor()`:
  ```
  bit.bxor(bit.bor(x, 1), 0xffffffff) == 0
  ```
- Or use a case-specific workaround:
  ```
  bit.rshift(x, 1) == 0x7fffffff
  ```

Operational Semantics and Rationale

Lua uses only a single number type which can be redefined at compile-time. By default this is a `double`, i.e. a floating-point number with 53 bits of precision. Operations in the range of 32 bit numbers (and beyond) are exact. There is no loss of precision, so there is no need to add an extra integer number type. Modern desktop and server CPUs have fast floating-point hardware — FP arithmetic is nearly the same speed as integer arithmetic. Any differences vanish under the overhead of the Lua interpreter itself.
Even today, many embedded systems lack support for fast FP operations. These systems benefit from compiling Lua with an integer number type (with 32 bits or more).
The different possible number types and the use of FP numbers cause some problems when defining bitwise operations on Lua numbers. The following sections define the operational semantics and try to explain the rationale behind them.

Input and Output Ranges

Bitwise operations cannot sensibly be applied to FP numbers (or their underlying bit patterns). They must be converted to integers before operating on them and then back to FP numbers.
It's desirable to define semantics that work the same across all platforms. This dictates that**all operations are based on** the common denominator of **32 bit integers**.
The `float` type provides only 24 bits of precision. This makes it unsuitable for use in bitwise operations. Lua BitOp refuses to compile against a Lua installation with this number type.
Bit operations only deal with the underlying bit patterns and generally ignore signedness (except for arithmetic right-shift). They are commonly displayed and treated like unsigned numbers, though.
But the Lua number type must be signed and may be limited to 32 bits. Defining the result type as an unsigned number would not be cross-platform safe. All bit operations are thus defined to **return results in the range of signed 32 bit numbers** (converted to the Lua number type).
Hexadecimal literals are treated as **unsigned numbers** by the Lua parser before converting them to the Lua number type. This means they can be out of the range of signed 32 bit integers if the Lua number type has a greater range. E.g. 0xffffffff has a value of 4294967295 in the default installation, but may be -1 on embedded systems.
It's highly desirable that hex literals are treated uniformly across systems when used in bitwise operations. **All bit operations accept arguments in the signed or the unsigned 32 bit range** (and more, see below). Numbers with the same underlying bit pattern are treated the same by all operations.

Modular Arithmetic

Arithmetic operations on n-bit integers are usually based on the rules of modular arithmeticmodulo 2^n. Numbers wrap around when the mathematical result of operations is outside their defined range. This simplifies hardware implementations and some algorithms actually require this behavior (like many cryptographic functions). E.g. for 32 bit integers the following holds: `0xffffffff + 1 = 0`
Arithmetic modulo 2^{32} is trivially available if the Lua number type is a 32 bit integer. Otherwise normalization steps must be inserted. Modular arithmetic should work the same across all platforms as far as possible:

> For the default number type of `double`, **arguments can be in the range of $\pm 2^{51}$** and still be safely normalized across all platforms by taking their least-significant 32 bits. The limit is derived from the way doubles are converted to integers.
> The function `bit.tobit` **can be used to explicitly normalize numbers** to implement**modular addition or subtraction**. E.g. `bit.tobit(0xffffffff + 1)` returns 0 on all platforms.
> The limit on the argument range implies that modular multiplication is usually restricted to multiplying already normalized numbers with small constants. FP numbers are limited to 53 bits of precision, anyway. E.g. $(2^{30}+1)^2$ does not return an odd number when computed with doubles.

BTW: The `tr_i` function shown here is one of the non-linear functions of the (flawed) MD5 cryptographic hash and relies on modular arithmetic for correct operation. The result is fed back to other bitwise operations (not shown) and does not need to be normalized until the last step.

Restricted and Undefined Behavior

The following rules are intended to give a precise and useful definition (for the programmer), yet give the implementation (interpreter and compiler) the maximum flexibility and the freedom to apply advanced optimizations. It's strongly advised not to rely on undefined or implementation-defined behavior.

> All kinds of floating-point numbers are acceptable to the bitwise operations. None of them cause an error, but some may invoke undefined behavior:
> -0 is treated the same as +0 on input and is never returned as a result.
> Passing **$\pm$Inf, NaN or numbers outside the range of $\pm 2^{51}$** as input yields an**undefined** result.
> **Non-integral numbers** may be rounded or truncated in an **implementation-defined** way. This means the result could differ between

different BitOp versions, different Lua VMs, on different platforms or even between interpreted vs. compiled code (as in LuaJIT).
Avoid passing fractional numbers to bitwise functions.
Use `math.floor()` or`math.ceil()` to get defined behavior.
Lua provides **auto-coercion of string arguments** to numbers by default. This behavior is**deprecated** for bitwise operations.