Test implementation of a logic unit in SpeedCrunc post-0.9

In order to overcome the current limitations of the parser in SpeedCrunch, a makeshift implementation of logic operations has been introduced. Following a suggestion of a user, all operations are accessible through function calls. This form is not particularly user friendly, as it forces you to type more, and the resulting expressions are not very readable:

(0xF3 & 0x76) | 0x55 vs. or(and(0xF3; 0x76); 0x55).

But the alternative is waiting until a better parser is available, and that seems to be worse.

So expect the user interface to be changing, possibly replacing the functions by operators in future versions. Here now comes a preliminary implementation of logic functions.

Logic unit

Internally, the math engine uses 256 bit signed integers for all logic operations. Hence the bounds are -2^{255} to $2^{255}-1$, or roughly +/- $5.789*10^{77}$. All input to logic functions is converted to this integer range prior to logic operations. That means,fractional parts of a number are simply cut off, and numbers >= abs(2^{256}) will produce NaN results.

In order to be able to enter a negative number in two's complement notation, unsigned 256 bit integers are accepted as well, but silently converted to a signed integer with the same bit pattern.

Input/Output

The automatic integer conversion lets you freely enter all numbers to logic functions, no matter whether real or integer. However, if an operand to a logic function results from an approximative real computation, you might occasionally receive surprising results. For instance, the evaluation of (1/3)*3 is displayed as a rounded result 1.0, but might internally be stored as something like 0.999...9999. If you enter such a value to a logic function, the fractional part is discarded, leaving 0 as operand, which could come as a surprise to the unwary. If there is a necessity to mix real and logic operations, you are advised to apply the ROUND function on operands to ensure proper computation.

Negative numbers are displayed and entered as integers with a - sign preceding them, no matter what display mode (hex, dec, oct or bin) is in effect. For example, the number -256 is entered as -0x100, -256, -0o400 or -0b10000000. This form of representation is sometimes referred to as `signed value notation'. Computer experts know, there is a second form called `two's complement notation', being their default data representation in integer arithmetic. A value like -256 in 256 bit size integer hex encoding shows in this mode as 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00.

There is one obvious downside to this kind of representation: Negative numbers need quite long input/output strings. With 256 bit integers, this is already beyond usability.

Nevertheless, you may enter the above string as input. What happens then, is that SpeedCrunch interprets this as an unsigned integer and converts it to a value being approximately $+1.157*10^{77}$. And that is what is used in mathoperations, so you e.g. may take the square root of it, and so on. Even though being too big for 256 bit signed integers, as an extension, the logic unit accepts positive values up to $2^{256}-1$ as operands, silently subtracting 2^{256} before processing such a parameter, and returning the expected result.

Although two's complement notation is unhandy with 256 bit numbers, as a matter of fact, computer

scientists, usually dealing with much less sized integers, are in desperate need of two's complement. So two convenience functions to convert from and to this representation are introduced:

mask(value; bitsize)

This function returns the lowest `bitsize' bits of the integer part of `value', clearing all the upper bits. `bitsize' is an integer restricted to 0 < bitsize < 256.

The result is always positive or zero.

This function is especially useful when applied to negative values, because it then returns the two's complement of `value':

mask(-1; 32) returns 0xFFFFFFFF, the 32 bit two's complement of -1.

But keep in mind: 0xFFFFFFFF is **not** -1, it is 4294967295.

In addition, you might want to use this function to mask out overflow bits of an arithmetic result. For instance, 0xFF + 0xFF is 0x1FE, but mask(0xFF + 0xFF; 8) returns 0xFE. Similar, 0x17*0x17 is 0x121, but mask(0x17*0x17; 8) gives 0x21 (modulo 256 arithmetic).

unmask(value; bitsize)

`bitsize' is an integer restricted to 0 < bitsize < 256.

This function takes the lowest `bitsize' bits of the integer part of `value', and sign-extends them to 256 bit. Sign-extending means, bit `bitsize' - 1 of `value' is copied to all upper bits. In many cases, this will revert the effect of a mask function, that's why it's called so.

This function comes in handy, if you want to enter a number in two's complement notation. As said above, the full 256 bit representation would require you to enter 64 hex digits. But mostly, numbers do not need the full 256 bits. Function calls like

unmask(0xFFFFFF00; 32), or unmask(0xFF00; 16), or even unmask(0x100, 9)

will all return -256. So unmask can be used as a shorthand for lengthy two's complement expressions.

logic operations

The logic unit supports common logic operations on signed 256 bit integers. Here's the list:

not(value)

returns the bitwise 1-complement of value. This will always change the sign bit, so a `not' creates negative numbers out of positive, and vice versa.

Example:

not(2) is -3.

In conjunction with mask you will receive the more familiar result

mask(not (2); 16) is 0xFFFD.

and(value1; value2 ...)

takes one or more parameters and combines them using the bitwise logical AND operation.

or(value1; value2 ...)

takes one or more parameters and combines them using the bitwise logical OR operation.

xor(value1; value2 ...)

takes one or more parameters and combines them using the bitwise logical exclusive OR operation.

shl(value; count)

shifts `value' to the left by `count' bits, if count >= 0. Zero bits are shifted in. If `count' is negative, a shr(value; -count) operation is executed.

-255 <= `count' <= 255

shr(value; count)

shifts `value' to the right by `count' bits, if count >= 0. The sign bit is duplicated and shifted in. If `count' is negative, a shl(value; -count) operation is executed.

-255 <= `count' <= 255

other arithmetic operations

Because logic operands have an arithmetic interpretation as integers as well, you may apply the usual arithmetic operations like +, - (unary or binary), * and so on in the ordinary fashion. Yet, there are a few limitations:

First, 256 bit arithmetic is usually taken modulo 2^{256}, avoiding overflow. The arithmetic unit does not obey this restriction, because it 'sees' no logic operand, but a real value. As a consequence, you must make sure, an arithmetic operation does not overflow into the 256th (or a higher) bit. This is guaranteed as long as your operands are representable as 128 bit signed integers, and as long as you apply a mask(result;128) call to intermediate results, and do an unmask(result; 128) call on the final result. This way, you effectively perform 128 bit arithmetic, and the upper 128 bits are just used as an overflow area that is cleared after an operation.

The second restriction refers to the division operator. A usual real value division on integers returns a full scale real valued result, possibly having a fraction close to 1 (like 0.999...99). You might occasionally encounter an off-by-one error, if you use this type of division in integer arithmetic. It is recommended to use the safe idiv function instead.

Arithmetic unit

In order to deal with integer arithmetic safely, the arithmetic unit has been enhanced as well. The following functions have been designed with integer arithmetic in mind, but you may submit real values as well.

operations

idiv(dividend;divisor)

Performs a division on both operands, and returns the integer part of the quotient. If the operands are known to be exact, the returned value is guaranteed to be the integer that results from truncating the true quotient towards zero. This holds especially for integers, because they are always exact.

For those familiar with C/C++: idiv is the / operator when applied to integers.

The idiv function differs from the ordinary division operator in

- that the result never has a fractional part;
- that the integer part of the true real-valued quotient always equals the result;
- that it returns a NaN whenever the quotient exceeds SpeedCrunch's integer range. By contrast, the usual division will continue to yield an approximative huge result.

Examples:

idiv(-17;4) is -4

idiv(387.334; 8.5443) is 45.

You cannot divide by zero; this returns NaN.

mod(value; divisor)

returns the remainder after finding the integer part of the quotient in a usual schoolbook division. This function has been designed with integer arithmetic in mind. But you may submit any real values as parameters, since you can perform schoolbook division on non-integers as well. This operation can be very costly, if both parameters differ in scale very much. The division loop is limited to some hundred steps; if the mod operation would require more, NaN is returned instead.

The mod function is related to the idiv operation in that the two relations

- value == idiv(value; divisor) * divisor + mod(value; divisor);
- abs(mod(value; divisor)) < divisor;

hold.

The result of mod is either 0 or has the same sign as its parameter `value'.

You cannot find the remainder if divisor is zero. This returns a NaN.

For those familiar with C/C++: mod is the % operator when applied to integers.

Examples:

mod(-17; 4) is -1

mod(387.334; 8.5443) is 2.8405.

Output

While it was already possible to display integers in hexagesimal, octal or binary representation, now all (real-valued) numbers are shown in these modes. A value like 0.75 may be alternatively displayed as 0x0.C, 0o0.6 or 0b0.11.

If the number is too big or too small to be displayed in a (hex, oct or bin) fix point format, the scientific notation is applied to these modes. For example, the huge value exp(1000) is displayed in hex as 0x6.79C8DE6BB5CEB6016(+0d360). This means, if you would write out this number in hex, its integer part would have 361 (add one to the exponent) hex digits, and start with the sequence 679C8DE6BB5CEB601... (we omit the last digit here, because it might be rounded).
Another way to look at this format is to interpret it as 0x6.79C8... * 16^{360}. (Note: the factor 16^{360} here is in decimal, while the significant 0x6.7... is in hex).
These extended display facilities may be useful to those who have to interpret binary encodings of

IEEE floating point formats directly.
Unfortunately, you currently cannot enter real-valued numbers in this way. This enhancement affects output only, where it presents more information than the former format.