CTQG

A Language for Synthesis of Reversible Circuits

USER MANUAL

Alexey Lvov

August 21, 2013

Contents

Abstract

CTQG is a compiler and a simulator for reversible logic circuits. The input file to CTQG contains a high level description of a reversible circuit and the output files are the circuit in ".qasm" format and an executable for simulation of the circuit.

Reversible circuits are used in all GFI quantum algorithms as subroutines (often called 'oracles'). By gate count reversible logic oracles constitute more than 90% of GFI algorithms. The main difference of GTQG from a general quantum circuits compiler is that it can do *simulation* of circuits, thus allowing continuous development cycle:
1. Write code - 2. Test (by simulation) - 3. Correct bugs (if any).

CTQG is used by quantum circuits compiler SCAFFOLD as a subroutine for compilation of reversible logic oracles and is a part of HLCB compilation flow. Also CTQG is designed as a stand alone tool and can be used as such.

Chapter 1

Installation

1.1 Installation

Download file ctqg.tar .

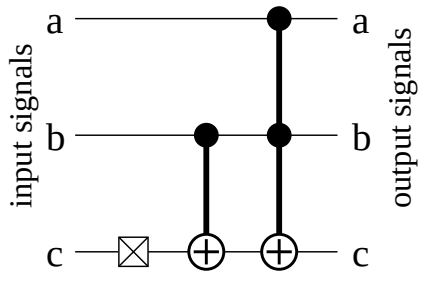
```
> tar -xvf ctqg.tar
> source source_me.setup
```

CTQG language is now fully installed on your machine.

1.2 Hello World

File "e00_hello_world.ctqg" :

```
module
main_module(
  qbit a, qbit b, qbit c
) {
  $ not(c);
  $ cnot(b, c);
  $ toffoli(a, b, c);
}
```



This simple circuit has just three input/output bit signals and three reversible logic gates.

This and all the other '.ctqg' examples used in this manual are included in the distribution. They do not have to stay at the location where CTQG is installed but can be moved to (and compiled from) any other directory.

1.3 Compilation

```
> ctqg e00_hello_world
```

Compiler creates three files:

1. "e00_hello_world.qasm" :

```
qubit a
qubit b
qubit c
X c
cnot b,c
toffoli a,b,c
```

This file contains the declarations of signals and the sequence of quantum gates in QASM format.

2. "e00_hello_world.signals" :

```
. a ~
. b ~
. c ~
```

This file contains the list of input/output signals:

3. Executable file "sim_e00_hello_world" simulates the circuit for any given initial values of input/output signals.

1.4 Simulation

The first column of file "e00_hello_world.signals" consists of '.' characters. To assign initial values to the signals for simulation replace the dots by '0' or '1' characters. This can be done either manually for a one time simulation, or automatically by any other program which uses the simulator as a subroutine. For example:

"e00_hello_world.signals" :

```
1 a ~
0 b ~
1 c ~
```

Run command

```
> sim_e00_hello_world  e00_hello_world.signals  e00_hello_world.out_signals
```

It produces a new file "e00_hello_world.out_signals" which contains the values of the output signals in the same format:

"e00_hello_world.out_signals" :

```
1 a ~
0 b ~
0 c ~
```

In order to invoke the simulator from a C or C++ program use

```
system("sim_e00_hello_world  e00_hello_world.signals  e00_hello_world.out_signals");
```

Chapter 2

CTQG By Example

2.1 Declaration of Modules

Similarly to a C program which consists of a number of functions, a CTQG description of a quantum circuit consists of a number of modules. Exactly one module must have name "main_module". Modules can instantiate (call) each other.

Unlike functions in a program, modules of a quantum circuit must form a strict hierarchy. Recursive instantiations and, more generally, any directed cycles in the dependencies graph of modules are not allowed. Because of this there is no need to separate declaration and definition of a module. Definition of modules in CTQG must immediately follow their declarations.

2.2 Basic Types

There are two basic types of signals in CTQG.

- **qbit**. A single bit.
- **qint**$[n]$, $1 \leq n \leq 2^{63} - 1$. A signed n-bit integer in standard binary representation.

Because CTQG can handle extremely large integers unsigned integer type is not needed. Our first examples use only "qbit" type; "qint" type will be explained in detail later.

2.3 Instantiation of Built in Modules

File "e01_swap_two_bits.ctqg" :

```
module
main_module(
  qbit a, qbit b
) {
  $ cnot(a, b);
  $ cnot(b, a);
  $ cnot(a, b);
}
```

See Chapter 3 for a complete list of built in modules and operators.

2.4 Instantiation of User Defined Modules

There is no difference between instantiations of built in and user defined modules.

File "e02_swap_two_triplets.ctqg" :

```
module
swap_two_bits(
  qbit a, qbit b
) {
  $ cnot(a, b);
  $ cnot(b, a);
  $ cnot(a, b);
}

module
main_module(
  qbit x1, qbit x2, qbit x3, qbit y1, qbit y2, qbit y3
) {
  $ swap_two_bits(x1, y1);
  $ swap_two_bits(x2, y2);
  $ swap_two_bits(x3, y3);
}
```

2.5 Arrays

CTQG supports multi index arrays of the basic types. The previous example can be rewritten as follows:

File "e03_swap_two_triplets2.ctqg" :

```
module
swap_two_bits(
  qbit a, qbit b
) {
  $ cnot(a, b);
  $ cnot(b, a);
  $ cnot(a, b);
}

module
main_module(
  qbit x[3], qbit y[3]
) {
  $ swap_two_bits(x[0], y[0]);
  $ swap_two_bits(x[1], y[1]);
  $ swap_two_bits(x[2], y[2]);
}
```

or

File "e04_swap_two_triplets3.ctqg" :

```
module
swap_two_bits(
  qbit a[2]
) {
  $ cnot(a[0], a[1]);
  $ cnot(a[1], a[0]);
  $ cnot(a[0], a[1]);
}

module
main_module(
  qbit x[3][2]
) {
  $ swap_two_bits(x[0]);
  $ swap_two_bits(x[1]);
  $ swap_two_bits(x[2]);
}
```

2.6 Parameterized Modules and Control Variables

Any module except for "main_module" can be parameterized. Parameters are always treated as signed 64-bit integers. A module can have any number of parameters. Module "main_module" can not have parameters because in that case in order to build a particular circuit the compiler would have to request the values of these parameters from the user at the time of compilation.

Declaration of a module with k parameters has form

```
module <parm1> <parm2> ... <parmK> name_of_module( ... signals decl. ... ) { ...
```

Instantiation of a module with k parameters and n arguments has form

```
$ [parm1_value] [parm2_value] ... [parmK_value] name_of_module(arg1, arg2, ..., argN);
```

File "e05_swap_two_bit_strings.ctqg" :

```
module
swap_two_bits(
  qbit a[2]
) {
  $ cnot(a[0], a[1]);
  $ cnot(a[1], a[0]);
  $ cnot(a[0], a[1]);
}

module <n>
swap_two_bit_strings(
  qbit x[n][2]
) {
  int i;
  for (i = 0; i < n; i++) {
    $ swap_two_bits(x[i]);
  }
}


module
main_module(
  qbit bs[100][2]
) {
  $ [100] swap_two_bit_strings(bs);
}
```

File "e06_swap_two_bit_strings2.ctqg" :

```
module
swap_two_bits(
  qbit a[2]
) {
  $ cnot(a[0], a[1]);
  $ cnot(a[1], a[0]);
  $ cnot(a[0], a[1]);
}

module <n>
swap_two_bit_strings(
  qbit x[n][2]
) {
  int i;
  for (i = 0; i < n; i++) {
    $ swap_two_bits(x[i]);
  }
}

#define bs_length 100

module
main_module(
  qbit bs[bs_length][2]
) {
  $ [bs_length] swap_two_bit_strings(bs);
}
```

Note that in both examples "`int i;`" is not a signal of the circuit. This variable is only used to control the process of building of the circuit. One can declare and use any number of control variables in any module exactly the same way one does it in C with regular local variables.

2.7 Array Subranges

In declaration of an array the ranges can be arbitrary integer expressions in the parameters of the module. The valid values for indexes are 0 to $(range\_size - 1)$ inclusively. For example:

```
module <m> <n> <a>
abc(
  qbit MATR[m][n],
  qbit DOUBLE_MATR[2 * m][2 * n],
  qbit VEC[m + n + a + 123]
) {
  $ not(DOUBLE_MATR[0][0]);
  $ not(DOUBLE_MATR[2 * m - 1][2 * n - 1]);
}
```

In an instantiation one has three options for mapping of each of the arrays dimensions: to specify a subrange, to specify a value of index or to omit the dimension at all. Specifying a value kills the dimension, omitting the dimension is equivalent to specifying the full range (that is the same range as in the declaration). Once one dimension is omitted all dimensions after it are automatically omitted. In particular omitting all dimensions maps the whole array, as it is done in main_module of example "e06_swap_two_bit_strings2".

In the example below all three options are used in the order they were listed:

```
module
aaa(
  qbit BOX_3D[10][10][5]
) {
  $ transpose_5_by_5_matrix(BOX_3D[2 .. 6][9]);
}
```

2.8 Type qint

CTQG has a built in set of integer arithmetic modules and a built in type "**qint**".
"**qint**[*size_in_bits*] my_qint" declares a signed *size_in_bits*-bit integer in standard binary representation.

File "e07_integer_arithm.ctqg" :

```
module
main_module(
  qint[32] a, qint[32] b, qint[32] c, qbit r
) {
  $ a += b;
  $ a -= b * c;
  $ r ^= a < b; /* r = r XOR (a < b) */
}
```

See Chapter 3 for a complete list of built in integer arithmetic modules and operators.

Although **qint** is made a separate type in CTQG, it is done so for better human readability only. All built in arithmetic modules and operators work in the absolutely same way with **qint** and with one dimensional arrays of **qbit**. More precisely: declarations

`qint[n] my_sig` and `qbit my_sig[n]`

are indistinguishable in CTQG and so are

`qint[n] my_sig[a1]` and `qbit my_sig[a1][n]`,

`qint[n] my_sig[a1][a2]` and `qbit my_sig[a1][a2][n]`,

etc.

2.9 Integer Arithmetic Operators

Unlike user defined modules each built in integer arithmetic module can be instantiated either in a regular fashion or as an operator:

```
$ [32] a_swap_b(a, b);                    $ a <=> b;
$ [32] assign_value_of_b_to_a(a, b);      $ a := b;
$ [32] a__eq__a_plus_b(a, b);             $ a += b;
   ...    ...    ...                       ...   ...
```

The values of the parameters have to be given in brackets in regular instantiations. Integer operators always have exactly one parameter – the size of the integer – and the compiler always can figure out the value of this parameter from the sizes of the arguments. Thus no parameter values should be given to operators.

See Chapter 3 for a complete list of built in integer arithmetic modules and operators.

2.10 Ancilla Signals

Assignment of a constant value to a signal is not a reversible operation and can not be used in a reversible circuit. To compensate for this inconvenience reversible circuits allow to request a number of bit signals preloaded with '0' or '1' constants before the execution of the circuit starts. Such signals are called "ancilla signals".

Depending on whether or not the circuit designer guarantees that after execution an ancilla signal will retain its original value the ancilla signal can be classified as *reusable* or *not reusable (garbage)*.

Both types of ancilla bits can be requested in CTQG as follows:

File "e08_ancilla_signals.ctqg" :

```
module
i_use_ancilla(
  qbit x
) {
  zero_to_garbage a;
  one_to_garbage  b[3];
  zero_to_zero    c[2][2];
  one_to_one      d;
}

module
main_module(
  qbit x1, qbit x2, qbit x3
) {
  $ i_use_ancilla(x1); /* Each of the 3 instantiations of "i_use_ancilla" module grabs    */
  $ i_use_ancilla(x2); /* 4 garbage ancilla signals and 5 reusable ancilla signals.       */
  $ i_use_ancilla(x3); /* Reusable ancilla signals are recycled after each instantiation. */
}                      /* Thus the circuit has 12 garbage and 5 reusable ancilla signals. */
```

There are four situations when CTQG generates ancilla signals automatically.

1. Each time the user passes a constant as an argument to a module the necessary amount of reusable ancilla signals is created by the compiler. See Section 2.14 for details.

2. Each use of if-else statement requires one garbage ancilla signal to store the value of the control expression.

3. Each use of `$ a := b;` operator adds *size_of*(a) garbage ancilla signals to the circuit.

4. Fixed point arithmetic operations add a lot of ancilla signals because such operations are not reversible by their nature. The most ancilla consuming operations are e^x, $\ln x$, $\sin x$ and $\cos x$ each of which uses $O(argument\_bit\_size^2)$ ancilla signals.

2.11 Constants

Constants can be passed as arguments to both built-in and user defined modules. All necessary ancilla bits are created and managed automatically by the compiler.

File "e09_bit_constants.ctqg" :

```
module
main_module(
  qbit a, qbit b
) {
  $ cnot('0', a);
  $ cnot('1', b);
}
```

If the callee does not conserve value of a constant, for example `$ not('0');` an error message will be issued during the simulation (but not during the compilation) of the circuit.

The instantiated module is allowed to temporarily change the values of constants. It is only required that the initial value of a constant signal is equal to its value at exit from the instantiated module. For example

```
module
n_cnot(
  qbit a, qbit b
) {
  $ not(a);
  $ cnot(a, b);
  $ not(a);
}

module
main_module(
  qbit r
) {
  $ n_cnot('0', r);
}
```

2.12 Bit String Constants

CTQG supports one dimensional arrays of bit constants:

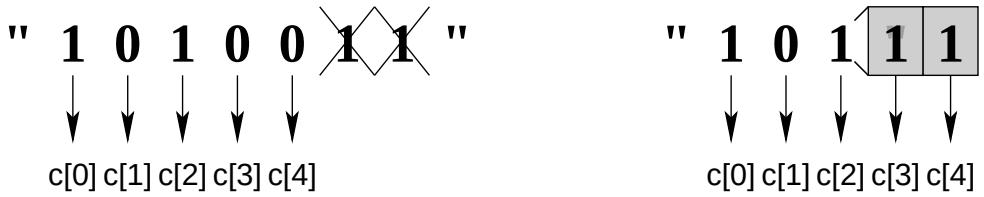
File "e10_bit_str_const.ctqg" :

```
module
cnot5(
  qbit c[5], qbit x[5]
) {
  int i;
  for (i = 0; i < 5; i++) {
    $ cnot(c[i], x[i]);
  }
}

module
main_module(
  qbit a[5]
) {
  $ cnot5("1010011", a);
  $ cnot5("10100", a);
  $ cnot5("101", a);
}
```

The length of the bit string is adjusted automatically to match the length of the corresponding argument of the callee. If the bit string constant is longer than needed it is truncated, if it is shorter than needed it is padded with the value of the last

(rightmost) bit.



Note that analogously to C strings the value of the element with index 0 is given by the leftmost character. This may be confusing when representing binary numbers, for example 8-bit binary string constant representing 19 is `"11001000"`.

2.13 Integer Constants

Signed integer constants of arbitrary size are supported.

File "e11_integer_const.ctqg" :

```
module
main_module(
  qint[24] x, qint[24] y, qint[24] z
) {
  $ [24] assign_value_of_b_to_a(x, 10);  /* Equivalent to (x, "01010")  */
  $ [24] assign_value_of_b_to_a(y, -27); /* Equivalent to (y, "101001") */
  $ [24] a__eq__a_plus_b_times_c(z, 0, -1020304050607080908070605040302 01);
}
```

If the bit width of the argument is sufficient to represent the constant then the argument receives exactly the signed integer value of the constant. Otherwise the argument receives a signed integer value which is equivalent to the constant modulo 2^{bit_width}. For example:

```
qint[8] signed_byte /* Can represent numbers from segment [-128 .. 127]. */
 ...  ...
$ signed_byte := -111; /* Now "signed_byte" holds signed binary value of -111.                 */
$ signed_byte := -222; /* Now "signed_byte" holds signed binary value of 34. 34 = -222 mod 2^8. */
```

2.14 Automatic Optimal Management of Reusable Ancilla Signals for Constants

When a GTQG program is compiled into a circuit all constants used in the program end up being stored in reusable ancilla signals of the circuit. CTQG minimizes the number of ancilla signals for constants as illustrated by the following examples.

```
module
main_module(
  qint[8] x
) {
  $ x += 25 * 51; /* "10011000" * "11001100" : 9 '0's and 7 '1's. */
  /* The circuit has 9 + 7 = 16 reusable ancilla signals.       */
  /* This number can not be reduced because all ancilla signals */
  /* are used simultaneously by the built-in multiplier module. */
}

module
main_module(
  qint[8] x
) {
  $ x += 25 * 51; /* "10011000" * "11001100" : 9 '0's and 7 '1's. */
  /* The 9 '0's and 7 '1's remain available. */
  $ x -= 26 * 55; /* "01011000" * "11101100" : 8 '0's and 8 '1's. */
  /* The circuit has MAX(9, 8) + MAX(7, 8) = 17 reusable ancilla signals. */
}
```

Analogously a module with, say, 100 instantiations, each of which uses a number of constants of total bit-weight w, will use the number of ancilla signals only slightly greater then w ($2w$ in the worst case of '0's and '1's distribution).

The total number of reusable ancilla signals in a circuit is equal to the maximum over all possible directed paths in the module dependencies graph of the sums of the reusable ancilla signals usages in the modules of each path. For example let the deepest call stack of a program contains 4 modules and each module in the program uses 100 instantiations with the bit-weight of constants 10. Then the number of reusable ancilla signals for constants in the circuit will be between $4 \cdot 10 = 40$ and $2 \cdot 4 \cdot 10 = 80$.

2.15 Comments

Regular C style comments are welcome.

```
/*COMMENT*/module/*COMMENT*/
main_module(
  qbit a
) {
  $/*COMMENT*/not/*COMMENT*/(/*COMMENT*/a/*COMMENT*/)/*COMMENT*/;
}
```

2.16 "If-Else"

The following circuit sorts an array of two integers in either increasing or decreasing order.

File "e12_sort_two_integers.ctqg" :

```
module
main_module(
  qbit     sort_in_increasing_order,
  qint[25] a[2]
) {
  $if (sort_in_increasing_order)

    $if (a[0] > a[1])
      $ a[0] <=> a[1];
    $endif

  $else

    $if (a[0] < a[1])
      $ a[0] <=> a[1];
    $endif

  $endif
}
```

The expression under "if" statement can contain any number of terms separated by "||" (or) operators, where each term can be *bit*, *!bit*, a comparison of two integers or a comparison between an integer and a constant. Hence 14 possible types of terms:

```
  $if (
        bit1  ||  !bit2  ||
        n < a  ||  n <= b  ||  n > c  ||  n >= d  ||  n == e  ||  n != f  ||
        a < 0  ||  b <= 1  ||  a > 2  ||  b >= 3  ||  a == 4  ||  b != 5
      )
    . . .    . . .
  $else
    . . .    . . .
  $endif
```

Operator "&&" (and) is not allowed in "if" expressions. Use repeated "if" instead. For example for $\big($not(A) or B$\big)$ and C use

```
$if (!A || B) $if (C)    . . .    $endif $endif
```

Independently of the length of the expression under "`$if`" each use of "`$if ... $endif`" or "`$if ... $else ... $endif`" adds one garbage ancilla signal to the circuit.

2.17 Low Level "If"

One can avoid garbage ancilla signals generated by "`$if`" statements by computing the control bit before the body of "`$if ... $endif`" and then uncomputing it to its original value. In each particular situation it requires some creative thinking and can not be done automatically by the compiler (although for high level "if-else" the compiler always does a part of the job by uncomputing all integer comparisons). The following example does if (A xor B) n += m; without any garbage ancilla signals.

```
module
main_module(
  qbit A, qbit B, qint[25] n, qint[25] m
) {
  $ cnot(A, B);  /* Now B contains the value of (A xor B).       */
  PUSH_CONTROL_SIGNAL(@B);
    $ n += m;
  POP_CONTROL_SIGNAL;
  $ cnot(A, B);  /* B has been uncomputed to its original value. */
}
```

It is not allowed to use `SIGNAL` anywhere between `PUSH_CONTROL_SIGNAL(@SIGNAL)` and `POP_CONTROL_SIGNAL`. This restriction applies only to the low level use of control bits. It is perfectly OK to use and modify any signals between high level `$if ... $else ... $endif`.

High level control and low level control can be mixed but must comply to the balanced parenthesis rule just the same way as "(", ")", "[", "]" must be balanced altogether in any C expression.

The stack of control signals must be empty at exit from any module.

2.18 Usage of Functions Written in C

CTQG allows to use the full power of language C for building reversible circuits.

Note that it is not possible to translate an arbitrary algorithm written in C into a reversible logic circuit and CTQG does not do it. CTQG only can execute users C program which tells it *how* to build the circuit.

For example: Build a circuit that inverts those bits of a 20-by-20 array of input signals which correspond to the Gaussian (integer complex) numbers with norm 100.

File "e13_usage_of_language_c.ctqg" :

```
typedef struct {
  int re;
  int im;
} gaussian_number;

int
norm(
gaussian_number x) {
  return x.re * x.re + x.im * x.im;
}

module
main_module(
  qbit lattice[20][20]
) {
  gaussian_number x;

  for (x.re = 0; x.re < 20; x.re++) for (x.im = 0; x.im < 20; x.im++) {
    if (norm(x) == 100) {
      $ not(lattice[x.re][x.im]);
    }
  }
}
```

2.19 Low Level Management of Signal Maps

Array subrange mapping mechanism is not flexible enough in certain situations. In parallel with the high level subrange mapping CTQG provides a low level signal mapping mechanism which allows to specify an arbitrary injection of the set of the signals of the callee into the set of the signals of the caller.

Example: Given a module for right cyclic bit shift of an array write a circuit which does a right cyclic bit shift of only those elements of b[30] which indexes are divisible by 3.

Solution:

```
module <n>
right_cyclic_shift(
  qbit a_of_right_cyclic_shift[n]
) {
  . . . < B O D Y > . . .
}

module
main_module(
  qbit b[30]
) {
  int i;

  INSTANTIATE_MODULE
    for (i = 0; i < 10; i++) {
      MAP_SIGNAL(%%a_of_right_cyclic_shift[i], @b[3 * i]);
    }
  right_cyclic_shift(
    LOCATION_INFO,
    10             /* The value of the first parameter of the callee is 10. */
  );
}
```

When the callee is a built-in module refer to Chapter 3 for the names of the module and its signals.

Example: Implement $b = b - a_1 * a_2$, where a1 is the integer represented by the even bits of a[100] and a2 is the integer represented by the odd bits of a[100].

File "e14_low_level_sig_map.ctqg" :

```
module
main_module(
  qint[50]  b,
  qint[100] a
) {
  int i;

  INSTANTIATE_MODULE
    for (i = 0; i < 50; i++) {
      MAP_SIGNAL( %%a[i] /* This is 'a' of built-in 'a-=b*c'. */,         @b[i]);
    }
    for (i = 0; i < 100; i++) {
      if (i % 2 == 0) {
        MAP_SIGNAL( %%b[i / 2] /* This is 'b' of built-in 'a-=b*c'. */,     @a[i]);
      } else {
        MAP_SIGNAL( %%c[i / 2] /* This is 'c' of built-in 'a-=b*c'. */,     @a[i]);
      }
    }
  a__eq__a_minus_b_times_c(
    LOCATION_INFO, /* Do not mind ''LOCATION_INFO'', it gives file name and line number for error messages. */
    50             /* The value of the first parameter of the callee is 10. */
  );
}
```

2.20 An Example of an Unrolled for() Loop

Generally neither conventional nor reversible circuits can have loops. However if number

$$M \stackrel{\text{def}}{=} \max_{\substack{all\ possible\ combina-\\ tions\ of\ input\ signals}} \#\ of\ iterations$$

is known the loop can be "unrolled" producing the amount of gates approximately equal to $M \cdot (\#\ gates\ in\ the\ body)$.

The following circuit computes $1+2+3+\ldots+n$ for a given $n \leq M$ in a brute force fashion. That is it does an equivalent of

```
sum = 0;
for (i = 1; (i <= n) && (i <= M); i++) {sum += i;}
```

File "e15_loop_unrolling.ctqg" :

```
#define M 100

module
main_module(
  qint[16] sum,
  qint[16] i,
  qint[16] n
) {
  int control_var_i;

  $ i := 1;
  $ sum := 0;

  for (control_var_i = 1; control_var_i <= M; control_var_i++) {
    $if (i <= n)

      /* --- BEGIN loop body --- */
      $ sum += i;
      /* ---  END  loop body --- */

    $endif
    $ i += 1;
  }
}
```

2.21 Fixed Point Arithmetic

CTQG provides a basic set of operations on real numbers in fixed point representation:

$e^x,\ \ln x,\ \sin x,\ \cos x,\ 1/x,\ xy,\ x+y,\ x-y,\ \lfloor x \rfloor,\ |x|,$ unary minus and comparisons.

Most of these operations produce extra ancilla signals.

There is no separate type for fixed point variables. One should declare them as integers and give the position of the fixed point at the time of instantiation.

File "e16_fixed_point_arithm.ctqg" :

```
module
main_module(
  qint[32] x, qint[32] y, qint[32] z,
  qbit b
) {
  /* "[21] [11]" means treat the arguments as fixed point numbers with */
  /* 21 bits before the point and 11 bits after the point.             */

  $ y := 6434;  /* This is very close to  pi * 2048. */
  $ [21] [11] fxp_sin(x, y); /* x := sin(y) */
  $ [21] [11] fxp_cos(x, y); /* x := cos(y) */

  $ z := 5567;  /* This is very close to  e * 2048.  */
  $ [21] [11] fxp_ln(y, z);  /* y := natural_logarithm(z) */
  $ [21] [11] fxp_exp(x, y); /* x := e^y                  */

  $ [21] [11] fxp_inverse(x, y);   /* x := 1 / y                       */
  $ [21] [11] fxp_mult(x, y, z);   /* x := y * z                       */
  $ [21] [11] fxp_floor(z);        /* z := (the greatest integer <= z) */
  $ [21] [11] fxp_abs(x);          /* x := |x|                         */
  $ [21] [11] fxp_invert_sign(x);  /* x := -x                          */

  /* The result of comparisons, addition and subtraction does not depend on */
  /* whether or not a fixed point is present in the arguments. Use regular  */
  /* integer operators for these functions.                                 */
  $ b ^= x != y;
  $ b ^= x == y;
  $ b ^= x >= y;
  $ b ^= x > y;
  $ b ^= x <= y;
  $ b ^= x < y;
  $ x += y;
  $ x -= y;
}
```

In case of an illegal operation on real numbers such as

```
  /* x happens to be 0 during simulation for some combinations of input signals. */
  $ [21] [11] fxp_ln(y, x);
```

the circuit produced by CTQG still computes 'y' to some value which depends only on the name of the operation and the values of the arguments. Also the circuit never changes the value of the arguments ('x' in this example) which are supposed to be constant.

Chapter 3

Built-in Modules, Operators and Keywords

Note: The arguments passed to any built-in module or operator must be pairwise different.
Instantiations like `$ a += a;` or `$ toffoli(x, y, x);` are not allowed.

3.1 $ not

not(qbit x)

$$x \rightarrow \text{NOT } x$$

const. to garbage ancilla signals	0
not	1

3.2 $ cnot

cnot(qbit c, qbit x)

$$\begin{aligned} c &\rightarrow c \\ x &\rightarrow c \text{ XOR } x \end{aligned}$$

const. to garbage ancilla signals	0
cnot	1

3.3 $ toffoli

toffoli(qbit c_1, qbit c_2, qbit x)

$$\begin{aligned} c_1 &\rightarrow c_1 \\ c_2 &\rightarrow c_2 \\ x &\rightarrow (c_1 \text{ AND } c_2) \text{ XOR } x \end{aligned}$$

const. to garbage ancilla signals	0
toffoli	1

3.4 $ [] a_swap_b and $ a <=> b

a_swap_b <n> (qint[n] a, qint[n] b)

$$\begin{aligned} a &\rightarrow b \\ b &\rightarrow a \end{aligned}$$

const. to garbage ancilla signals	0
cnot	$3n$

3.5 $ [] assign_value_of_b_to_a and $ a := b

assign_value_of_b_to_a <n> (qint[n] a, qint[n] b)

$$\begin{aligned} a &\rightarrow b \\ b &\rightarrow b \end{aligned}$$

const. to garbage ancilla signals	n
cnot	$3n$

3.6 $ [] a__eq__a_plus_b and $ a += b

a__eq__a_plus_b <n> (qint[n] a, qint[n] b)

$$\begin{aligned} a &\rightarrow a+b \\ b &\rightarrow b \end{aligned}$$

const. to garbage ancilla signals	0
cnot	$6n-3$
toffoli	$2n-2$

3.7 $ [] a__eq__a_minus_b and $ a -= b

a__eq__a_minus_b <n> (qint[n] a, qint[n] b)

$$\begin{aligned} a &\rightarrow a-b \\ b &\rightarrow b \end{aligned}$$

const. to garbage ancilla signals	0
cnot	$6n-3$
toffoli	$2n-2$

3.8 $ [] a__eq__a_plus_b_times_c and $ a += b * c

a__eq__a_plus_b_times_c <n> (
qint[n] a, qint[n] b, qint[n] c)

$$\begin{aligned} a &\rightarrow a+bc \\ b &\rightarrow b \\ c &\rightarrow c \end{aligned}$$

const. to garbage ancilla signals	0
toffoli	$6n^2-15n+18$
3ctrl toffoli	$2n^2-6n+6$

3.9 $ [] a__eq__a_minus_b_times_c and $ a -= b * c

a__eq__a_minus_b_times_c <n> (
qint[n] a, qint[n] b, qint[n] c)

$$\begin{aligned} a &\rightarrow a-bc \\ b &\rightarrow b \\ c &\rightarrow c \end{aligned}$$

const. to garbage ancilla signals	0
toffoli	$6n^2-15n+18$
3ctrl toffoli	$2n^2-6n+6$

3.10 $ [] a_less_than_b__as_signed and $ x ^= a < b

a_less_than_b__as_signed <n> (
qbit x, qint[n] a, qint[n] b)

$$\begin{aligned} x &\rightarrow (a<b) \text{ XOR } x \\ a &\rightarrow a \\ b &\rightarrow b \end{aligned}$$

const. to garbage ancilla signals	0
not	4
cnot	$12n-5$
toffoli	$4n-3$

3.11 $ [] a_less_than_or_eq_to_b__as_signed and $ x ^= a <= b

a_less_than_or_eq_to_b__as_signed <n> (
qbit x, qint[n] a, qint[n] b)

$$\begin{aligned} x &\rightarrow (a\leq b) \text{ XOR } x \\ a &\rightarrow a \\ b &\rightarrow b \end{aligned}$$

const. to garbage ancilla signals	0
not	5
cnot	$12n-5$
toffoli	$4n-3$

3.12 $ [] a_greater_than_b__as_signed and $ x ^= a > b

a_greater_than_b__as_signed <n> (
qbit x, qint[n] a, qint[n] b)

$x \rightarrow (a > b)$ XOR x
$a \rightarrow a$
$b \rightarrow b$

const. to garbage ancilla signals	0
not	4
cnot	$12n - 5$
toffoli	$4n - 3$

3.13 $ [] a_greater_than_or_eq_to_b__as_signed and $ x ^= a >= b

a_greater_than_or_eq_to_b__as_signed <n> (
qbit x, qint[n] a, qint[n] b)

$x \rightarrow (a \geq b)$ XOR x
$a \rightarrow a$
$b \rightarrow b$

const. to garbage ancilla signals	0
not	5
cnot	$12n - 5$
toffoli	$4n - 3$

3.14 $ [] is_a_eq_to_b and $ x ^= a == b

is_a_eq_to_b <n> (
qbit x, qint[n] a, qint[n] b)

$x \rightarrow (a = b)$ XOR x
$a \rightarrow a$
$b \rightarrow b$

const. to garbage ancilla signals	0
not	5
cnot	$24n - 10$
toffoli	$8n - 6$

3.15 $ [] is_a_not_eq_to_b and $ x ^= a != b

is_a_not_eq_to_b <n> (
qbit x, qint[n] a, qint[n] b)

$x \rightarrow (a \neq b)$ XOR x
$a \rightarrow a$
$b \rightarrow b$

const. to garbage ancilla signals	0
not	4
cnot	$24n - 10$
toffoli	$8n - 6$

3.16 $ [] [] fxp_exp

fxp_exp <m><k> (
qint[$m + k$] a, qint[$m + k$] b)

$a \rightarrow e^b$
$b \rightarrow b$

const. to garbage ancilla signals	$O((m+k)^2)$
not	$O((m+k)^3)$
cnot	$O((m+k)^3)$
toffoli	$O((m+k)^3)$

3.17 $ [] [] fxp_ln

fxp_ln <m><k> (
qint[$m + k$] a, qint[$m + k$] b)

$a \rightarrow \ln b$
$b \rightarrow b$

const. to garbage ancilla signals	$O((m+k)^2)$
not	$O((m+k)^3)$
cnot	$O((m+k)^3)$
toffoli	$O((m+k)^3)$

3.18 $ [] [] fxp_sin

fxp_sin <m><k> (
qint[m + k] a, qint[m + k] b)

$$a \rightarrow \sin b$$
$$b \rightarrow b$$

const. to garbage ancilla signals	$O((m+k)^2)$
not	$O((m+k)^3)$
cnot	$O((m+k)^3)$
toffoli	$O((m+k)^3)$

3.19 $ [] [] fxp_cos

fxp_cos <m><k> (
qint[m + k] a, qint[m + k] b)

$$a \rightarrow \cos b$$
$$b \rightarrow b$$

const. to garbage ancilla signals	$O((m+k)^2)$
not	$O((m+k)^3)$
cnot	$O((m+k)^3)$
toffoli	$O((m+k)^3)$

3.20 $ [] [] fxp_inverse

fxp_inverse <m><k> (
qint[m + k] a, qint[m + k] b)

$$a \rightarrow 1/b$$
$$b \rightarrow b$$

const. to garbage ancilla signals	$O((m+k)\log(m+k))$
not	$O((m+k)^2\log(m+k))$
cnot	$O((m+k)^2\log(m+k))$
toffoli	$O((m+k)^2\log(m+k))$

3.21 $ [] [] fxp_mult

fxp_mult <m><k> (
qint[m + k] a, qint[m + k] b, qint[m + k] c)

$$a \rightarrow bc$$
$$b \rightarrow b$$
$$c \rightarrow c$$

const. to garbage ancilla signals	$O((m+k))$
not	$O((m+k)^2)$
cnot	$O((m+k)^2)$
toffoli	$O((m+k)^2)$

3.22 $ [] [] fxp_floor

fxp_floor <m><k> (
qint[m + k] a)

$$a \rightarrow \lfloor a \rfloor$$

const. to garbage ancilla signals	$O((m+k))$
not	$O(m+k)$
cnot	$O(m+k)$
toffoli	$O(m+k)$

3.23 $ [] [] fxp_abs

fxp_abs <*m*><*k*> (
qint[$m+k$] *a*)

$a \rightarrow |a|$

const. to garbage ancilla signals	1
not	$O(m+k)$
cnot	$O(m+k)$
toffoli	$O(m+k)$

3.24 $ [] [] fxp_invert_sign

fxp_invert_sign <*m*><*k*> (
qint[$m+k$] *a*)

$a \rightarrow -a$

const. to garbage ancilla signals	0
not	$O(m+k)$
cnot	$O(m+k)$
toffoli	$O(m+k)$

3.25 List of Keywords and Special Characters