RKQC: RevKit For Quantum Computation

A Framework for the Synthesis of Reversible Circuitry

Adam Holmes

June 2016

Abstract

RKQC is a compiler for reversible logic circuitry. The framework has been developed to compile high level circuit descriptions down to assembly language instructions, primarily for quantum computing machines. Specifically, input files to the RKQC compiler contain descriptions of reversible circuits, and the output files are the assembly instructions for the circuit, in the ".qasm" format.

In many important quantum computing algorithms, a large portion of the modules use only classical reversible logic operations that can be decomposed into the universal set of NOT, CNOT, and Toffoli gates. Often these are referred to as "classical oracles." These oracles can also be simulated on a conventional computer.

RKQC is used by the SCAFFOLD [1] quantum circuits compiler as a subrouting for the compilation of purely classical reversible logic modules, or oracles. It has also been designed to operate as a stand alone tool, and can be used in this fashion. It was also developed as a full conversion of the RevKit platform [2].

Contents

Chapter 1

Installation

1.1 Installation

1.1.1 Obtain

The first step in installation is to obtain RKQC from the public repository. It can be cloned through the Git tool by:

```
mkdir rkqc
git clone https://github.com/ah744/RKQC.git rkqc/
```

1.1.2 Build and Compile

There are two steps to building and compiling RKQC. The first is to build and obtain the dependencies, and the second is to compile the system. These are both combined in the script "build.sh", and invoking this script from the main directory will build and compile the system.

```
./build.sh
```

RKQC is now built and compiled on your system.

1.2 First Example

The example circuits are located in:

```
src/examples
```

In order to compile and run any of these examples, the "revkit" script may be invoked, located in the main directory of RKQC.

The example circuit "e001_hello_world.cpp":

```
using namespace revkit;
int main( int argc, char ** argv )
{
```

```
    qbit a;
    qbit b;
    qbit c;

    NOT(a);
    cnot(b, c);
    toffoli(a, b, c);

    return 0;
}
```

can be compiled with RKQC by invoking the revkit script and the example circuit name, without the file extension, from the main directory of RKQC. This circuit has three input and output signals, and three reversible logic gates.

```
./revkit e001\_hello\_world
```

The compiler then creates the output ".qasm" file, containing the quantum assembly language instructions generated for this circuit. For "e001", the file "e001_hello_world.qasm" contains:

```
qubit w0
qubit w1
qubit w2
X w0
cnot w1, w2
toffoli w0,w1,w2
```

Chapter 2

Examples

2.1 Declaration of Circuits

The construction of RKQC is designed to be as similar as possible to C++ programs which contain a number of functions. To this end, circuits are specifiable through a main function, as well as through various submodules that are declared exactly as common C++ functions. Exactly one module must be named "int main", however this module, as well as any module, can call any other module at any time.

2.2 Signal Types

There is one major type of signal in RKQC, the **qint**. This type forms the basis for the five other types of signals that are derived from this main class:

- qbit
- zero_to_garbage ancilla
- zero_to_zero ancilla
- one_to_garbage ancilla
- one_to_one ancilla

All of the above signals can be declared and used in the specification of gates and circuit descriptions. The primary differences are that the "_to_garbage" ancilla are ancilla signals that are not guaranteed to maintain their original state during computation, while the two other ancilla types do make this guarantee.

As will be explained later on, all of these signals are of a base type **qint**, which allows for circuit description flexibility when creating and passing parameters to submodules.

2.3 Creating and Using Modules

At the circuit level, both user defined modules as well as built in modules can be instantiated in the same fashion.

In the example circuit "e008_swap.cpp", the built in function "cnot" is called, creating CNOT gates on the qubits passed in as parameters:

```
int main( int argc, char ** argv )
{
    qbit a;
    qbit b;

    cnot(a, b);
    cnot(b, a);
    cnot(a, b);

    return 0;
}
```

In exactly the same fashion, a circuit designer can create a submodule, and instantiate it within the main function, as in file "e009_swap_triples.cpp":

```
void swap\_bits( qint x, qint y){
    cnot(x, y);
    cnot(x, y);
    cnot(x, y);
}

int main( int argc, char ** argv ){
    qbit a0;
    qbit a1;
    qbit a2;

    qbit b0;
    qbit b1;
    qbit b2;

    swap\_bits(a0, b0);
    swap\_bits(a1, b1);
    swap\_bits(a2, b2);

    return 0;
}
```

Chapter 3

Libraries

Bibliography

[1] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: A framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2597917.2597939

[2] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: An open source toolkit for the design of reversible circuits," in *Reversible Computation 2011*, ser. Lecture Notes in Computer Science, vol. 7165, 2012, pp. 64–76, RevKit is available at www.revkit.org.