Scaffold: Quantum Programming Language

Ali Javadi Abhari, Arvin Faruque, Mohammad Javad Dousti, Lukas Svec,
Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt,
John Black, Fred Chong, Margaret Martonosi, Martin Suchara
Ken Brown, Massoud Pedram, Todd Brun

July 24, 2012

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20165. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

1 Introduction

Quantum computing is of significant research interest because of its potential to radically alter the performance and asymptotic complexity of certain computations [22, 21]. Over the years, physicists have explored possibilities for how to implement quantum bits and logic devices in hardware. At a higher level, mathematicians and algorithmicists have explored how to express computational problems from diverse fields—such as cryptography, chemical systems simulation, and database search—in terms of mathematical equations potentially implementable in quantum hardware.

For quantum computing to become potentially viable however, research must also make inroads regarding the implementation, compilation, and architectural issues that lie between high-level mathematical algorithm expressions and low-level physical implementations. In particular, we need tools and analysis techniques that can—for a given algorithm and potential physical implementation technology—answer questions like: how much would it cost (i.e., how much resource in terms of qubits, gates, time are required?) to implement the algorithm in this technology? What is its performance potential? Is it scalable? Are there more algorithms that offer such speedups over classical computers?

Our work is building a language and toolflow to answer such questions. This document describes the Scaffold programming language, its design goals, and related tools. Scaffold is a programming language for

expressing quantum algorithms. A quantum algorithm can consist of a wide variety of components (including classical and quantum routines) which will be defined using different coding techniques. As a quantum programming language (QPL), Scaffold was formulated to make it easy to express an algorithm with so many disparate components in a clean and efficient manner. It is from this notion of "putting things together" that Scaffold derives its name.

1.1 Language Design Goals

As mentioned, Scaffold is designed to facilitate expressing quantum algorithms in terms of operations and data types that can be compiled towards a target implementation, and for which resource estimation and other analyses can be expeditiously applied. Some of the primary design goals for the language are:

- **Completeness and Expressiveness:** Scaffold should enable programmers to express whole quantum applications. In addition to completeness, Scaffold should also provide usefully expressive features which make common QC techniques relatively convenient for the programmer to express. Since many quantum algorithms are specified as quantum gate sequences or imperative codes, the language must facilitate programmers expressing computations in these ways. For example, many quantum algorithms include a quantum oracle, which is usually expressed as classical code; such oracles must be efficiently representable in scaffold. In other cases, there may be a circuit representation for the functionality of the algorithm. Once again, the language must not present an obstacle for such methods.

- **Familiarity and Ease of use:** Quantum computing is difficult enough; the language should not make things worse. Users familiar with basic quantum computing and classical computer programming concepts should be able to learn and use Scaffold smoothly. Where it makes sense, the syntax and programming paradigms of this new language should mimic existing and widely-used programming constructs. In particular, Scaffold draws from the near-ubiquitous familiarity of C and C++ programming constructs in the computer science and engineering communities.

- **Integration:** The language should be designed such that it integrates well into a toolchain for quantum program analysis and processing. The flow from high-level mathematical expressions of algorithms down to the physical synthesis and realization must be done seamlessly and easily in this toolchain. This means that Scaffold must be able to express algorithms such that they can later be compiled by a custom compiler infrastructure for this language, generate low-level codes and pass on enough information to the lower levels to enable their conversion into physical qubits, quantum gates and other circuit entities for a wide range of physical implementations of quantum computers, and then be scheduled for execution by a specific execution model.

- **Leveraging existing compiler infrastructures:** In order to focus tool development time on the portions that are most novel and important to the project, we wish to leverage existing compiler infrastructures for portions of the language that are not central to the novelty of the research. In particular,

where feasible, we hope to exploit previous efforts in software compilation and hardware synthesis. Not only does this save on compiler development time, but it also makes full use of existing, mature concepts and tools. This means that Scaffold must retain syntax close to classical widely-used languages to foster use of existing compiler code. Of course, modifications must be done whenever important to the research goals; compiler reuse will not be a reason to hold back the development of the quantum programming language.

1.2 Scaffold Design decisions

Towards the realization of the design goals listed above, our language development has made design decisions that are briefly listed here. Later sections of this document elaborate on these.

- **Imperative Programming Model:** We explicitly preferred an imperative (rather than functional) programming approach for Scaffold. The applications Scaffold targets often have very natural expression in terms of loop nests, iterative calculations, and function calls with argument passing. Around any quantum calculation lies a framework of classical computation control, and for these it is often preferred to have familiar variable and control structures. Languages that influenced the design of Scaffold included popular classical high-level imperative programming languages (C/C++, Java) [16, 25, 11], hardware description languages (Verilog) [13], C-to-hardware languages (System-C) [14] and existing quantum programming languages (QCL) [23].

- **Variant of C and Verilog:** Scaffold syntax was chosen to be very similar to C (and to some extent Verilog HDL.) This reflects our belief that expressing computations in terms of familiar iterative and imperative control structures will pay off both in terms of programmer effort and in terms of compilability. Scaffold constructs — such as the declaration of modules, passing of arguments, input and return types — serve our goals regarding ease of use as well as exploitation of existing compiler infrastructure. Many of the classical paradigms that we used in this design, such as structures, unions, conditionals, arrays and different data types, although at times tailored distinctly for the QPL, nonetheless reflect this intention.

- **Aggressive Syntax Checking:** While Scaffold is a variant of C for familiarity's sake, we recognize that some of its features — e.g., pointers — are of modest use in our application scenarios, and yet may be used in ways that introduce bugs. To mitigate these issues, Scaffold minimizes the legal use of pointers (no pointer arithmetic or arbitrary dereferencing) and provides aggressive syntax and out-of-bounds checking at compile time. In this way, we benefit from C's straightforward familiarity, without paying for it in pointer and addressing errors.

- **C2QG:** A key feature of Scaffold is a Classical code to Quantum Gates sequence (C2QG) module. C2QG modules allow a programmer to express a desired quantum functionality using classical programming. Rather than viewing everything in terms of quantum gates, C2QG modules allow programmers to describe the functionality of some parts of the algorithm from a higher perspective.

This is somewhat like Hardware Description Languages, where this classical code is then converted into a sequence of gates. The inclusion of this feature greatly increases the ease of expression, and is used in several of our application implementations. Furthermore, C2QG modules come into play when allowing the expression of whole quantum applications, since many existing algorithms in the literature specify their code in terms of oracles that are typically consisting of imperative code.

- **Control primitives:** To further enhance the expressiveness of the language as well as to facilitate the programmer's job by allowing more coding options, a control primitive is included in Scaffold. This is a high-level feature allowing programmers to indicate different flows of execution based on the state of quantum bits. In other words, defining a set of complex controlled modules is facilitated. This is in accordance with the goal of providing ease of use.

- **Leveraging Open-Source Compilation:** Because Scaffold bears such similarity to C, it has the benefit that a high-level Scaffold compiler can use a high-level C compiler as its starting point. In particular, our work leverages the LLVM compiler infrastructure [19, 18] for much of the parsing, intermediate representation, and back-end. Our language design decisions greatly facilitate leveraging existing tools, since we can build off of previous efforts in parsing, optimization and specially reversible logic synthesis.

- **Modular Design:** One of the challenges of quantum compilation is that different ways of expressing the quantum algorithm will call for different ways of compiling it. For example, the inclusion of C2QG modules in Scaffold requires the conversion of these codes to reversible logic, suitable for use inside a quantum circuit model. Our modular approach facilitates this, by separating the classical parts of the code and feeding them into available reversible logic synthesis tools, thus creating an easy integration of these tools alongside each other in the toolchain.

1.3 Document overview

The remainder of this document is organized as follows: Section 2 presents the general structure of a Scaffold program. Section 3 describes Scaffold's main features, including the syntax for their use. Section 4 presents the details about C2QG modules, an integral part of reversible logic synthesis employed in the tool chain. Finally, Section 5 concludes the document.
Appendix A contains example programs that show the usage of the language. Appendix B presents the general envisioned tool flow and the integration of the QPL within such a toolchain. Appendix C contains elaboration on some of the syntax that is mostly common with C.

2 Scaffold: Language Overview

This section discusses the typical components comprising a program written in Scaffold. It shows how the language is structured and how programmers can organize their code.

On the most basic level, a Scaffold program contains code that describes a sequence of classical and quantum operations on classical and quantum data. The classical operations are specified in a syntax that resembles C code and the quantum operations (such as quantum gate applications on quantum registers) appear within this C-like code as special function calls on explicitly defined quantum data structures.
Figure 1 illustrates the components inside a Scaffold program and the relationships between them. Every program (if it is not written for being included inside another code) must contain a module called `main` , which tells the compiler the entry point of compilation. Other components include the following.

1. **Preprocessor directives** occur at the beginning of the program and can be used for declarations that are anticipated to be used throughout the rest of the code, such as constants, macros, libraries, etc.

2. **Gate prototypes** define placeholders for quantum gates. Later in the program, these quantum gates can be invoked using calls on the appropriate quantum (and sometimes classical) data.

3. **Modules** can occur throughout the code. They may in turn call other modules or C2QG modules when they are called. The module construct has been designed to be a cross between a C-function and a sub-circuit inside a quantum circuit. They take a series of arguments as input and can have a return value. Unlike gates, modules can be defined as more than just prototypes. We can specify what happens inside a module using code.

4. **Classical code to quantum gate sequences (C2QG) modules** are modules that the programmer specifies using purely classical data, but calls with appropriately sized quantum registers as inputs. Tools within our toolchain can compile these modules into reversible logic quantum circuits.

2.1 Scaffold Program Example

Program 1 begins our discussion by illustrating a very simple Scaffold program. Similar to the familiar "hello world" examples, this program has very little functionality, but is intended to show the simple beginnings of Scaffold programming and to illustrate a few key concepts. Note that actual Scaffold programs do not use line numbers; we include them here only to ease our description of the program.
On line 1, the program includes a `gates.h` file which gives the prototypes for a standard set of gates. Then line 2 is where the main module begins. Every free-standing program must contain

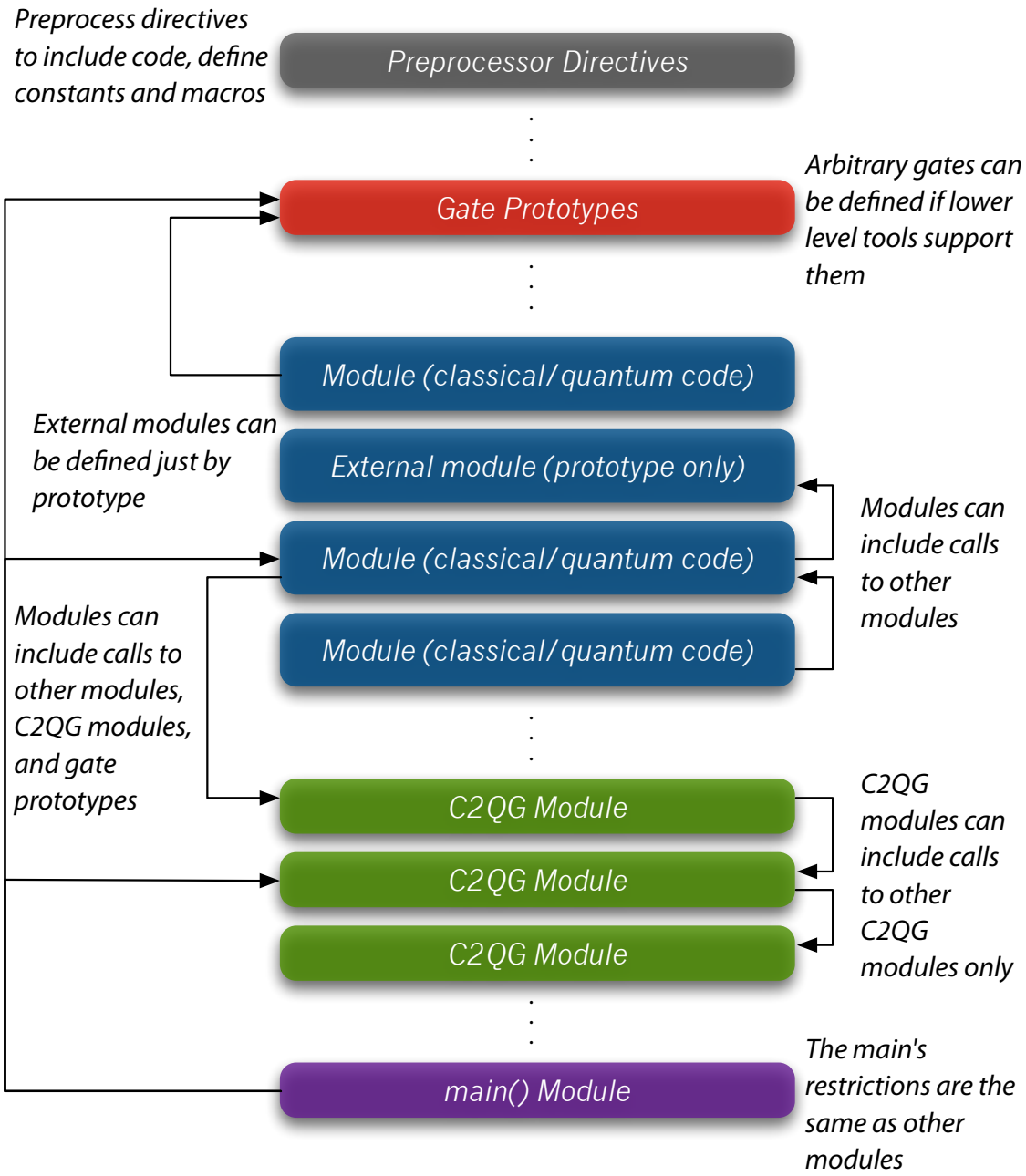

Figure 1: Structure of a Scaffold program

```
1    #include "gates.h"
2    module main ( ) {
3    int i=0;
4    qreg extarget[4];
5    qreg excontrol[4];
6    forall(i=0; i<4; i++) {
7       CNOT(extarget[i],excontrol[i]);
8       }
9    }
```

Program 1: Example: A very simple Scaffold program.

a module called main, which tells the compiler the entry point of compilation. Lines 3-5 define several program variables of different types. Scaffold programs support both classical and quantum variables. In this case, lines 4 and 5 each define a register comprised of 4 quantum bits. Lines 6-8 are a very simple loop construct. The loop body—in line 7—computes a CNOT in which the i-th qubit in extarget is CNOT-ed with the i-th qubit in excontrol as the control bit. Because each iteration of this loop is independent (there are no loop-carried dependencies) the iterations could be executed in parallel. To express this succinctly, the programmer may choose to use the `forall` construct as shown in line 6.

Scaffold lets the user specify when and where quantum operations get applied (sometimes based on classical conditions). It does not, however, force the user to dictate what happens "inside" a quantum operation (i.e. the actual unitary matrix of the gate). Such functionality is applied in a later technology-mapping stage. This is akin to the idea that a classical computer has a known instruction set at the assembly language level, and does not require each programmer to describe the specific functionality of an add or multiply instruction set.

In Scaffold, users can refer to particular gates through "prototypes" of the gate, and the toolflow maps accordingly. Scaffold's compiler will pass information about gate prototypes (including their input qubits and classical parameters) and their usage circumstances down to the lower-level format that it compiles to. Because of the gate prototypes, the Scaffold compiler does not check whether each gate operation is unitary (it cannot, since it does not have information about the gate) but it does enforce rules that ensure the gates are applied in a valid quantum mechanical manner. For example, it will not allow the user to feed the same qubit into both the control and target ports of a CNOT gate.

The next section is devoted to describing these components as well as other features of the language in detail.

3 Scaffold Language Features

Although Scaffold provides a coding style similar to known classical programming languages, it also incorporates special distinct features which makes it suitable for coding quantum algorithms. These features are discussed in this section.

3.1 Data Types

3.1.1 Quantum Registers

Quantum registers, represented as `qreg` , are provisioned in Scaffold for grouping individual qubits together. They can be defined and initialized as follows, where `n` is the size of the quantum register, and `example_reg` is the name.

```
qreg example_reg[n];
```

Note that `n` is a constant number and for runtime safety reasons no variable length quantum register can be declared. After the quantum register has been declared, a particular m'th qubit within this qreg can be accessed using

```
example_reg[m]
```

Similar to C, the qubits are numbered starting from 0. To access a range of qubits

```
example_reg[a..b]
```

can be invoked. Furthermore,

```
length (example_reg)
```

returns the length of the declared register back to the user.

It is also possible to form quantum registers (or expand existing ones) by means of concatenation. The following syntax describes this procedure:

```
qreg reg1[p];
qreg reg2[q];
qreg reg = {reg1,reg2};
```

Individual qubits are also treated as qregs of size 1 and must be defined and named as such.

```
qreg example_qubit[1];
```

Since quantum registers store qubits which must abide by the no-cloning theorem, call-by-reference is used when passing them as arguments to either gates or modules.

3.1.2 Classical variables

It is possible for a Scaffold programmer to define classical variables and pass them as parameters in his code. A classical variable is defined using the following notation:

```
datatype variablename;
```

where "datatype" is the type of the variable and "variablename" is the name of the variable. Valid primitive datatypes include the same datatypes commonly used in C, such as int, char, float, etc.

A classical variable is referenced using its name. Users may also elect not to choose these standard data types, and define their own custom-length type. The syntax will be as follows:

```
int<n> example_data;
```

Valid casts between datatypes can be done using the C casting notation (parentheses around a datatype preceding a variable). For example, the following code returns a floating point version of the number `data` .

```
(float) data;
```

All casts must be made explicit by the programmer using this notation, otherwise an "inconsistent type" error will be displayed if the utilized types do not match.
The inclusion of these various data types in Scaffold is with the intention of providing users a similar experience to C, and keep their hands open at specifying classical code. It is important, however, for the programmer to remember that these types are only usable when dealing with classical data. For those data that will be mapped to qubits, many of them are not allowed since they run the risk of inducing great overhead. This is discussed in 4.3 when C2QG modules are introduced.

3.2 Arrays

Scaffold also provides support for arrays of variables. We can use the following syntax to define an array called `example_array` , consisting of `n` elements of type `datatype` :

```
datatype example_array[n];
```

Datatype can be any of the previously mentioned classical types. We can use the brackets operator to access elements in the array based on their index number.

```
example_array[a];
example_array[a..b];
```

We can also statically initialize the data such as integer, float, and double inside arrays to constants. For example, an integer array can be initialized as:

```
int integer_array[3] = {1, 2, 3};
```

The built in function `size()` returns the number of elements in the array.

```
size(integer_array)
//returns 4.
```

To add or remove elements in the array, one can use the `append` and `delete` functions.

```
append{example_array1, example_array2};
delete{example_array1[a]}
```

3.3 Structs and Unions

In order to provide a convenient option for grouping relevant variables together, and thus facilitating the coding of algorithms by programmers, Scaffold contains support for structures and unions. Furthermore, while structures can be defined and manipulated, we do not provide any C-like syntax for accessing them via pointers, or for dereferencing pointers. This represents a compromise solution that gives programmers the expressiveness of composite variables like structs, without the challenges of pointers. In particular, disallowing pointers allows us to provide compiler analysis related to bounds-checking on arrays and structures without needing to handle arbitrarily complex memory disambiguation scenarios.

In addition, since programs cannot retain or manipulate pointers indeterminately, the Scaffold compiler can analyze at compile-time when the program is done with data. This is possible whether the data is quantum or classical. The compiler can mark such "deadpoints" in the code that it generates. The decision of how to handle such deadpoints will depend on other implementation factors, but can include static or dynamic resource collection in most cases. This will help in reducing the resource consumption of the quantum computer.

3.3.1 Classical structs

These structures can only contain variables in their definition (i.e. no functions are allowed). It is possible to define a structure using the following syntax:

```
struct example_structure {
    datatype1 member_variable_1 = default_value_1;
    datatype2 member_variable_2 = default_value_2;
    ...
```

```
    datatypen member_variable_n = default_value_n;
};
```

where "example_structure" is the name of the struct. To define a new instance of this struct called "instance1", the following constructors can be used:

```
// to use the default values of the struct
example_structure instance1 = new example_structure();

// to use custom initialized values
example_structure instance1 = new example_structure(
member_variable_1 = value_1,
member_variable_1 = value_2,
       ...
member_variable_n = value_n
);
```

To access a particular variable "member_variable_m" of this instance, the following syntax is used:

```
instance1.member_variable_m
```

3.3.2 Quantum structs and unions

In addition to structs of classical variables, Scaffold also supports a quantum struct and quantum union data type. This allows the user to hierarchically organize several quantum registers into one structure and reference each of the relevant pieces of the register as necessary. Instantiating a quantum struct will be the same as instantiating the quantum registers inside it and quantum structs will be implicitly passed by reference in the same manner as quantum registers. A quantum union datatype is also included , allowing users to reference different pieces of a quantum register by different names (similar to a C union).
The following shows an example of the struct type with two 20-qubit quantum structs named qstruct1 and qstruct2:

```
qstruct struct1 {
    qreg first[10];
    qreg second[10];
};

qstruct struct2 {
    qreg a[5];
    qreg b[5];
```

```
    qreg c[5];
    qreg d[5];
}
```

We can access elements of these structs as:

```
struct1 example_qstruct_1;

// index = 0-9
example_qstruct_1.first[index];
// index = 0-9
example_qstruct_1.second[index];

// treat struct as a register, index = 0-19
example_qstruct_1[index];
```

A quantum union is similar to a quantum structure except that now all the variables share the same qubits. It can be defined and accessed as follows

```
// Union of two 20-qubit structs, still 20 qubits
qunion example_union {
    struct1 example1;
    struct2 example2;
};

// Instance of union
qunion example_union instance1;

// Accessing elements from struct1, index = 0-9
instance1.example1.first[index]
instance1.example1.second[index]

// Accessing elements from struct2, index = 0-4
instance1.example2.a[index]
instance1.example2.b[index]
instance1.example2.c[index]
instance1.example2.d[index]
```

3.4 Quantum Gates

Now that the previous subsection has laid out the data types available in Scaffold programs, we turn to computation. In particular, this subsection discusses quantum gates, and the following two subsections cover control structures and modules.

Within a Scaffold program, gates are either defined as prototypes whose implementation is specified elsewhere or via specific libraries. It is important to note that prototypes can only be used when later stages of the toolchain support them. In the case that a gate is defined from outside standard libraries, the compiler will pass it on to lower levels, which are assumed to contain means of interpretation and implementation for them. This is useful, for example, when the programmer wants to code for a specific PMD[1], knowing the set of gates specific to that particular technology. In such cases, errors due to non-existent gates must be flagged by the backend of the compilation toolchain, since that is where the gate prototypes are instantiated into PMD-based implementations. When users must adhere to gates inside libraries for non-supportive toolchains, they have the option of defining more complex behaviors inside modules.

3.4.1 Gate prototypes

Firstly, gates are defined using gate prototypes. Gate inputs include quantum registers (assumed to be passed by reference) and classical variables (which can be passed by value or by refernce). The following syntax defines a gate called "gatename" with n parameters:

```
gate gatename(type_1 parameter_1,..., type_n parameter_n);
```

Each input parameter is specified using a type and a keyword. Valid parameter types are:

- **Quantum Registers:** Quantum registers can be passed to well defined gates as follows:

  ```
  qreg reg[n]
  ```

- **Constant classical values:** It is also possible to specify constant integer, float, and double parameters as inputs to a gate. Such inputs are prefaced by the keyword "const" and must have constant values at compile time. The following is an example of a constant floating point parameter called "angle", which can be used as the angle of a desired rotation gate:

  ```
  const float angle
  ```

[1] Physical Machine Description

- **Classical variables**: Classical variables of signed and unsigned int, char, float and double can be passed by reference into the gate. They must be passed as single variables and not as arrays.

Later, inside modules, a gate defined by a gate prototype can be called as follows (similar to a C function call):

```
gatename(parameter_1,...,paramenter_n);
```

3.4.2 Gate libraries

Secondly, the user has the option of using a standard library of gates by using the "gates.h" header file. This header file contains commonly used gates that are already defined similar to the prototypes above. They are recognized by the compiler after the inclusion of the header file as if they had been prototyped. Program 2 contains the list of standard gates in this library, showing the name and the arguments of each one.

```
// Pauli X, Pauli Y, Pauli Z, Hadamard, S, and T gates
gate X(qreg input[1]);
gate Y(qreg input[1]);
gate Z(qreg input[1]);
gate H(qreg input[1]);
gate S(qreg input[1]);
gate T(qreg input[1]);

// Daggered gates
gate Tdag(qreg input[1]);
gate Sdag(qreg input[1]);

// CNOT gate defined on two 1-qubit registers
gate CNOT(qreg target[1], qreg control[1]);

// Toffoli (CCNOT) gate
gate Toffoli(qreg target[1], qreg control1[1], qreg control2[1]);

// Rotation gates
gate Rz(qreg target[1], float angle);        //Arbitrary Rotation

// Controlled rotation
gate controlledRz(qreg target[1], qubit control[1], float angle);

// One-qubit measurement gates
gate measZ(qreg input[1], bit data);
gate measX(qreg input[1], bit data);

//One-qubit prepare gates: initializes to 0
gate prepZ(qreg input[1]);
gate prepX(qreg input[1]);

//Fredkin (controlled swap) gate
gate fredkin(qreg targ[1], qreg control1[1], qreg control2[1])
```

Program 2: "gates.h" header file: The standard library of gates recognized by the compiler

3.5 Loops and Control Constructs

3.5.1 The Basics

Program execution can be sequenced using fairly familiar conditional and iteration constructs. These include:

- *Simple Conditionals:* This is a C-like "if" or "if-else" statement. EXPRESSION must evaluate to a bool or an int/char/bit (0 = false, nonzero = true). If curly brackets are omitted following the statement the conditional code will be assumed to be the first line after the if statement. If EXPRESSION tests the value of a quantum variable, a measurement has occurred.

```
if(EXPRESSION) {CODE}

if(EXPRESSION) {CODE0} else {CODE1}
```

- *Switch Statement:* Again, this control statement is very C-like in nature. Code associated with the case where $EXPRESSION == VALUE$ is true will execute. If specified, the default code will execute. If `EXPRESSION` includes quantum variables, a measurement will occur.

```
switch(EXPRESSION) {
      case VALUE1: CODE1; break;
      ...
      case VALUEn: CODEn; break;
      default: CODE_DEFAULT; break;
   }
```

- *Loops:* Scaffold supports standard C-like loop constructs including `for` and `while` . In addition, Scaffold provides a `forall` loop. This has syntax similar to a `for` loop, but it allows the programmer to assert that all the loop iterations will be independent and thus can be performed in parallel. The compiler can then select (based on hardware and timing constraints) whether to unroll the loop to be fully or partially parallel. Section 3.5.2 describes this in more detail.

In addition, all familiar C-style classical operators are available:

- **Operators and assignment:** Basic arithmetic and logical operations as described in Appendix C.

3.5.2 Parallel "forall" loop

One of the most important features that a quantum computer must have is the ability to execute independent operations in parallel, thereby reducing the execution time. This is envisioned in the execution model of this project as well. For example, in the specifications of the Machine Control Language (MCL), each statement is preceded by a time stamp, a non-negative integer representing the time at which the statement is to begin execution. This scheduling is done by tools residing at lower levels in the toolchain, but the user may want to specify his own preferences too. In order to facilitate the generation of parallel code inside the toolchain, and to provide means of parallel execution without conflict during runtime, a parallel `forall()` primitive is defined and can be used inside modules. It allows the programmer to identify independent portions of the code, and gives him specific control over which parts of the code he wants to be executed in parallel. The code snippet below shows an example of this concept.

```
qreg test[8];
    forall(i=0; i<8; i++){
    H(test[i]);
}
```

Here, Hadamard gates are applied separately to the 8 qubits that constitute the qreg `test` . They do not depend on each other, and so during compile time this code gets unrolled producing more instances of the same gate.

One cannot use forall when an iteration in the loop depends on the results of other iterations. For example:

```
\\using (regular) for
qreg test[8];
for(i=0; i<8; i++){
    H(test[i]);
    X(test[8-i]);
}

\\using (parallel) forall
qreg test[8];
forall(i=0; i<8; i++){
    H(test[i]);
    X(test[8-i]);
}
```

Program 3: Usage of the parallel forall

The code snippets above show the same loop with two different implementations: regular and parallel. They yield different results in each case, since earlier iterations of the loop modify qubits that are acted upon in later iterations. This highlights the programmers role in making a choice regarding the use of this primitive.

3.5.3 Quantum Control Primitives

Quantum computers often need to take different actions (i.e. follow different execution paths) based on specific conditions. These conditions can generally be expressed in 3 ways:

(a) *Runtime conditions:* In this case, the compiler passes the conditions in classical assembly format to the lower level, where the classical computer communicates the result of condition evaluation at runtime to the quantum core. An example might be the case of a classically controlled gate, where the control line is the result of a measurement performed at runtime. The lower level tools treat this similar to a multiplexer.

(b) *Static classical conditions:* These conditions — as in Section 3.5.1 — will be evaluated by the compiler, and the corresponding quantum gates will be generated to satisfy the conditional circuit. This is similar to unrolling loops in that it generates the circuit statically.

(c) *Qubit state conditions:* For these conditions, Scaffold supports "Quantum Control Primitives" which facilitate the programmer's job of calling an arbitrary module using a series of quantum bits as control lines. Without them, programmers must generate their own quantum triggers based on different evaluations of the condition.

Some quantum-controlled unitaries are so fundamental that they are defined as standard gates, such as the CNOT or Toffoli (CCNOT). However, programmers may want want more complicated controlled structures, either in terms of the modules used or the conditions controlling the use of those modules. Scaffold uses a similar syntax to the classical if-else construct to specify more general controls. The code snippet below illustrates what the basic syntax of this primitive looks like.

The compiler will use controlled versions of such modules to realize the circuits. Figure 2 illustrates this for the program above. On a lower level, all the gates inside the individual modules will be translated to controlled versions of themselves by the compiler; and using the decomposition of them to simpler gates, the final fault-tolerant circuit is produced. .Theoretically, it is possible to decompose any controlled gate to a series of 1-qubit non-controlled gates [22]. Well-known quantum identities are useful at this stage for performing the decomposition and possible optimizations. An example of this is illustrated in Figure 2 [8].

```
//Module prototypes. They are defined elsewhere
module U (qreg input[4], int n);
module V (qreg input[4]);
module W (qreg input[4], float p);

//Quantum Control Primitive
module control_example (qreg input[4]) {
if (control_1[0]==1 && control_2[0]==1){
     U(input);}
else if (control_1[0]==1 && control_2[0]==0){
     V(input);}
else{
     W(input);}
}
```

Program 4: Example quantum control primitive for controlled execution of 3 different modules, U, V and W

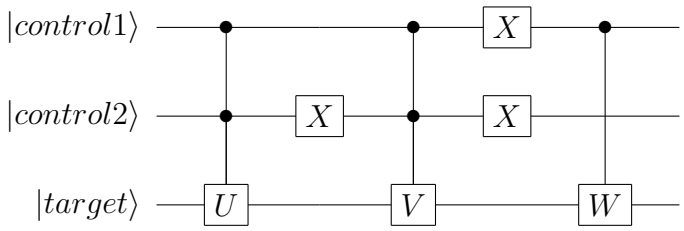

Figure 2: Circuit representation of Program 4

Two key restrictions of this feature is that none of the control lines will be allowed to be mapped to quantum register inputs of the modules, and the code comprising the body of each module must be purely quantum (and unitary).

Each of the control conditions is an expression that evaluates to a boolean. This expression can be made up of comparisons (==, !=) between individual quantum bits as well as boolean and, or, and not operators between subexpressions. As mentioned earlier, code inside the body of this construct cannot use any of the qubits used in the condition expressions.

3.6 Modules

To improve the organization, readability, compilability, and error-checking for Scaffold programs, computations are encapsulated into function-like structures called modules. Each module defines

a piece of the quantum algorithm, and a program typically consists of several modules, with the ”main” module serving as the program's entry point. Modules take quantum registers and classical variables as input and are defined using a C-like syntax. These modules can be defined externally and linked to the current program. While some are written with quantum variables and gates, along with classical or quantum control structures, others are written via C2QG approaches. In C2QG, modules are written as classical C-like code, with the intent of synthesizing to reversible logical circuits that are compatible with a quantum circuit. These variants will be explained in detail in the sections that follow.

3.6.1 Module definition

The syntax for defining a module is:

```
return_type module modulename(type_1 param_1,..., type_n param_n) {
     // Module body
}
```

The return type can be void or an int, float, double, char or struct. The list of input parameters following the module name has syntax identical to the parameter list for a gate prototypes and also the same datatype rules and restrictions.

A module is meant to be called inside a module body using the following syntax (similar to a C function call):

```
modulename(parameter_1,...,parameter_n);
```

A module can also be defined to external (as a prototype) by prefacing the definition with the keyword “external” and omitting the module body, which tells the compiler to look for the module in another file. This file must be included at the beginning of the program:

```
external return_type module name(type_1 param_1,..., type_n param_n);
```

3.6.2 Module body

The body of a module consists of C-like statements. These statements can consist of the following

- **Local quantum register definitions:** These include quantum registers that have scope limited to the body of the module unless they are passed as parameters to other gates or modules. To define a local quantum register called ”localreg” of n qubits, the following syntax is used:

  ```
  local qreg localreg[n];
  ```

- **Local classical variable definitions:** Local classical variables of type int, char, float, double, and struct can be defined within a function. Unlike the local quantum registers, they are not prefaced by the local keyword.

- **Classical code and calls to quantum gates and modules:** In addition to the classical variable and quantum register definitions described earlier, module bodies can include classical code mixed with calls to quantum gates and modules. The syntax of the classical code consists of the following elements (in addition to the variable definitions):

- **Calls to gates and modules:** Calls to gates and modules can be included within the code. The restrictions on the placement of these calls is identical to the restrictions of the placement of functions inside a C program. Recursive calls to modules are allowed.

- **Quantum code (quantum gate calls) interspersed with classical code** A quantum gate call has syntax nearly identical to a classical function call in C and inherits all the same restrictions, in addition to others that are necessary to enforce valid use.

- **Built-in functions**

  A number of built-in functions are available for use within the code. These are supported only for classical datatypes. These include familiar math.h functions such as `sin`, `cos`, etc. or file input/output functions such as printf() and scanf(). The Scaffold compiler can check that the invocations and inputs for such functions are legal. That is, the compiler can check that such built-ins are only invoked in classical regions of code, and using solely classical variable types.

3.7 Possible Future Language Extensions

Although all efforts have been made to ensure the comprehensiveness of this language, there is always capacity for improvement in future releases. One area of possible expansion is the provision of "reverse" primitives. They can be added to allow a user to call the reverse of a module, as a way to $uncompute$ its action. This will require several restrictions. One strategy is to forbid the destruction of quantum data (measurements, creation of local ancilla qubits) inside the modules that are called using this primitive. Another issue that needs to be addressed is the reversibility of modules that include dynamically generated gate sequences. One possible solution to this problem is to provide a primitive to store the sequence of gates generated during a single call of the module and to provide a mechanism to apply this sequence in normal or reversed order later on. The following code snippet illustrates the basic idea behind this concept:

```
// Module instance
instance m1;

// Call module capturing gate sequence
module1(...) {
   ...
   m1;
   ...
   }

// Call the instance of gates
call m1;
// Call the instance backwards
reverse m1;
```

Program 5: psuedocode representing the idea behind the quantum reverse primitive

Note that this would require runtime support for capturing gate sequences, some of which may be largely dependent on the algorithm.

Another possible extension to the language is the expansion of libraries. For example, simple math functions can be implemented to act upon quantum registers. While general implementations of such functions would likely be very resource-intensive, specialized and optimized versions could be offered as library functions for use when needed.

4 C2QG Modules

4.1 Motivation

In writing code to represent quantum operations and algorithms, often the programmer finds it easier to describe a higher-level functionality of the circuit in a somewhat behavioral manner, instead of expressing those operations at the level of individual gates. In some cases, this can improve the compactness and readability of the code. In other cases, it is more straightforward for handing instances where quantum algorithms include components that are defined to act upon qubits but whose logic is specified using classical sequential code consisting of statements such as comparisons, loops, and arithmetic on data. While these parts of the program are expressed as classical code, they are intended to execute as a sequence of quantum gates that operate on some quantum states. Scaffold is designed to support the specification of modules for such algorithm components, which is referred to as Classical-code-to-Quantum-Gate-sequence (C2QG).

As an example, we can look at the case of the Toffoli gate, also known as the CCNOT gate. This is only a simple example for illustration purposes, and may not be of practical significance since Toffoli gates are included as part of the standard library themselves.

The programmer has the option of specifying this 3-qubit gate either in a normal module with purely quantum code, using its decomposition into individual known 1-qubit or 2-qubit gates such as $CNOT, T, T^{\dagger}$ and H, or in a C2QG module using the description of the behavior. These two approaches are depicted in Figure 3 and their Scaffold code is presented in Program 6. The succinct simplicity of the latter choice highlights the advantage of using C2QG modules.

```
// Normal module representing a Toffoli gate
module Toffoli(qreg target,qreg control1,qreg control2){
   H(target);
   CNOT(target,control2);
   T(control2)
   Tdg(target);
   CNOT(target,control1);
   CNOT(control2,control1);
   Tdg(control2);
   T(target);
   CNOT(control2,control1);
   CNOT(target,control2);
   Tdg(target);
   CNOT(target,control1);
   T(target);
   T(control1);
   H(target);
}

// C2QG module representing a Toffoli gate
c2qg Toffoli(qint<1> target, qint<1> control1,qint<1> control2){
   if(control1 == 1 && control2==1) {
     target= ~target;
     }
   else {
     target=target;
     }
}
```

Program 6: comparison between the usage of C2QG modules versus normal modules

Other instances of such convenience can be seen in oracle modules, which often play an integral part in the development of quantum algorithms. These oracles offer an abstract functionality, which

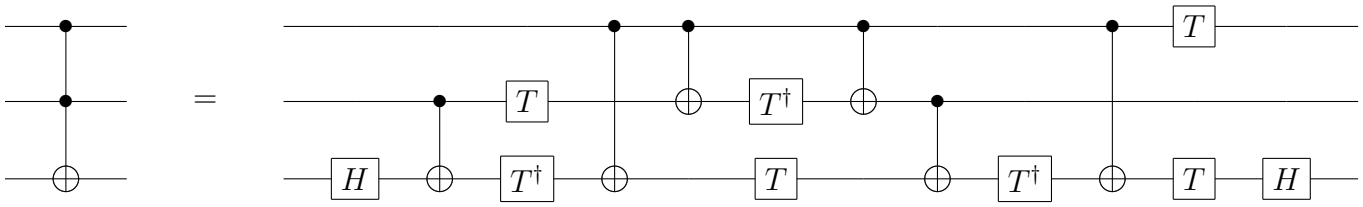

Figure 3: Decomposition of a Toffoli gate into fault-tolerant gates

```
1    #include "gates.h"
2    c2qg qtoy (qint<4> example) {
3        example = example *2
4    }
5
6    module main(){
7        qreg example[4];
8        qtoy(example);
9    }
```

Program 7: Example: A very simple program containing a C2QG module

can best be described from a high level. In some cases, such as the Binary Welded Tree algorithm of the GFI, the circuit level synthesis of such module is also known and straightforward, albeit long. In other cases however, the decomposition into gates might not be readily available.

4.2 Simple C2QG Program

Similar to the previous section, we begin the discussion of C2QG modules with a very simple example, shown in Program 7. This program contains a toy C2QG module — called qtoy to emphasize this — which is defined on line 2. It functions as a multiplier by two, as described by the body of the code on line 3. The body is a classical function, which will be synthesized into quantum gates. Notice on line 2 how the type of the arguments are unique to C2QG modules (i.e. beginning with "q"). In the "main" module, on line 7, we define a 4-qubit quantum register, which will then be passed to our toy module. Every C2QG module takes as input some qubits, and alters the state of the same qubits. This is in accordance with the nature of these modules which are to be synthesized to reversible circuits. As a result, there are no return types associated with these modules.

4.3 Functionality and Syntax Specifications

These modules are similar in concept to classical C-to-HDL tools such as System-C. They basically provide for quantum logic all the functionality that verilog modules provide for classical logic. As mentioned, the programmer specifies these modules in a C-like syntax using purely classical data, but calls the modules with appropriately sized quantum registers as inputs. Figure 4 illustrates the concept of C2QG modules using a quantum circuit diagram. During the compilation process, a reversible logic synthesis tool generates reversible logic (i.e. quantum gates) out of these modules even if classically, they are not reversible. The programmer must be aware that using large irreversible modules may result in large ancilla overheads at this stage.

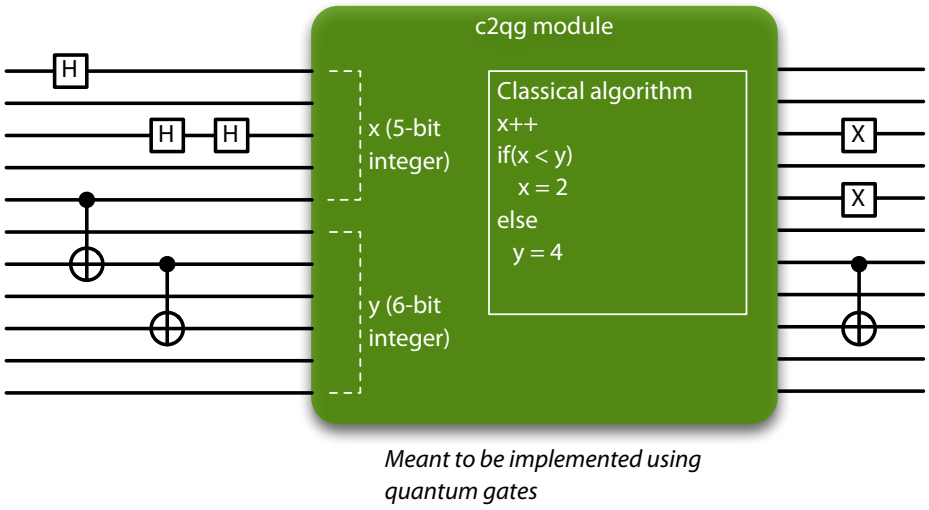

Figure 4: Classical code to quantum gate (C2QG) module shown inside a quantum circuit.

Variables that will map to quantum registers inside C2QG modules must be explicitly labeled using the "q" prefix. No classical variables (int, float, etc.) are allowed to be passed to these modules, because the circuits are generated statically. Only classical constants can be input in addition to the *qvariables*.

The quantum-mapped variables that will be allowed as arguments in these modules are variable-bit. The reason for this is that classical datatype conventions have been found suitable for binary computation and have historically been implemented in classical computers, as they align well with their microarchitecture (e.g. register sizes, memory systems). However, when the classical code within C2QG modules get converted to quantum bits and gates, the inclusion of unnecessary overhead might severely degrade the performance of the quantum computer. Therefore, Scaffold

uses the variable-bit integer type to allow the programmer to define integers with the precise sizes needed, which will later be mapped to quantum registers. In this regard, there are two types for the arguments: qint and qfix. The former allows for declarations of variable-bit integers, where the latter gives users the ability to declare custom fixed-point numbers. The syntax is as follows:

```
qint<n> integer_name: n-bit integer
qfix<n,m> fixed_point_name: n-bit fixed point with m bits after the radix
```

No floating point arithmetic is allowed in these modules. Furthermore, if there are classical datatypes used, they must only be defined locally within the body, and be fully analyzable classically at during code generation, to allow for the easy evaluation of the underlying reversible quantum circuit. The C2QG is implicitly of type "void", and thus does not need any type declarations. Here is the correct syntax for this declaration:

```
c2qg module_name (qint<n>, ..., qfix<n,m>, ...,const c) {
    ...
    ...
    }
```

Indexing into these types will be similar to that defined for qregs above:

```
qint<8> example;
example<0> //The 0th qubit
example<0..3> //The first four qubits
```

The code inside the body of a C2QG module includes the following:

1. **Variable definitions:** Definitions of variables that are valid for the C2QG module, as mentioned above.

2. **Operators:** Normal C operators may be used between variables. The user must make a distinction between variables that will be mapped to qubits (qvariables) and normal classical variables. Mixing them will cause a compiler error.

3. **Conditional and loop constructs**: C conditional and iteration constructs are allowed. They must be analyzable at compile time to allow the generation of reversible logic. The loops will be passed to lower levels in a compact form, using some high level classical assembly, to reduce code size.

4. **Calls to other c2qg modules**. Within a C2QG module, only calls to other C2QG modules are allowed, as is natural for the way these modules are synthesized. Between C2QG modules, classical variables may be passed; this is done so by reference.

5 Conclusion

5.1 Summary

This document provided descriptions of Scaffold, a quantum programming language developed for efficient representation of quantum algorithms. The requirements for the first phase of this project are to develop and demonstrate a language capable of expressing the quantum algorithms provided and to design the toolbox that surrounds it, which can ultimately be used to estimate the resource overhead requirements for various algorithms and physical implementations.

In this documentation, we presented the incentives for having such a language, as well as the motivation behind each of its features. In includes acceptable syntax definitions and example usages, and emphasizes the distinctions between classical and quantum data and operations. We discuss the context of the language's uses, the programmers' benefits and pitfalls and the errors they might expect in each case. We also discuss the process of compilation and synthesis that goes on concealed from the programmer. Furthermore, in the appendix we provide several small algorithms that illustrate Scaffold's functionality and capabilities.

A Example Scaffold Programs

This section presents several examples to show what users can achieve in terms of implementing quatnum algorithms using the Scaffold language. It draws on portions of real and useful algorithms but avoids lengthy programs in order to preserve their pedagogical value. The goal is to bring to light the diverse features of the language and their usage.

A.1 Binary Welded Tree

This subsection shows the Scaffold implementation for a particular function, often found in a *Quantum Random Walk on a Binary Welded Tree* algorithm. The algorithm is explained in more detail below. The pseudocode representation is as follows:

Algorithm 1 PARSENODEROOT (a, root, even, *n*)

for $index = n$ **to** 1 **by** -1 **do**
  if $|scratch[index]\rangle = |0\rangle$ ***and*** $|a[index]\rangle = |1\rangle$ **then**
    $|scratch[index-1]\rangle \leftarrow |scratch[index-1] \oplus 1\rangle$
  endif
  if $|scratch[index]\rangle = |1\rangle$
    $|scratch[index-1]\rangle \leftarrow |scratch[index-1] \oplus 1\rangle$
  endif
end for

if $|scratch[0]\rangle = |0\rangle$ **then**
  $|root\rangle \leftarrow |root \oplus 1\rangle$
  $|even\rangle \leftarrow |even \oplus 1\rangle$
endif

for $index = 1$ **to** n **by** 1 **do**
  if $|scratch[index]\rangle = |1\rangle$ **then**
    $|scratch[index-1]\rangle \leftarrow |scratch[index-1] \oplus 1\rangle$
  endif
  if $|scratch[index]\rangle = |0\rangle$ ***and*** $|a[index]\rangle = |1\rangle$ **then**
    $|scratch[index-1]\rangle \leftarrow |scratch[index-1] \oplus 1\rangle$
  end if
end for
return

Algorithm Description: On a general level, the Quantum Random Walk on a Binary Welded Tree algorithm is concerned with finding the exit node in the conjunction of two binary trees, given the

entry node and a specification of the graph layout (i.e. welding function). The nodes in the graph are numbered, and a quantum register is initialized to represent the number corresponding to the *entrance* node. After a series of evolutions within the framework of this algorithm, the state of the qubits must have evolved to the *exit* node.

This particular function, called the *Parse Node Root* algorithm, is part of the classical oracle specification, and is supposed to decide whether a given node number is that of a root or not, and if so set some flags. Here, *n* is a constant, equal to the depth of the tree. *a* is a $(n+1)$-qubit register which acts as the input to this algorithm, and holds the number of the node in question. Nodes are numbered 1 to $2^{n+1}-1$. The flags to be set are two qubits called *root* and *even*.

In the algorithm a temporary $(n+1)$-qubit qreg *scratch* is defined, and is initialized to 0. We know that the root node is numbered as $100...00$. To detect it, a loop traverses over all the qubits of a and $scratch$ starting from the LSB. If it detects a state of $|1\rangle$ in any of the qubits of *a*, it performs a NOT on the next qubit of the *scratch* register, and therefore sets it to $|1\rangle$. This has a ripple effect, and all the qubits in *scratch* will have the state $|1\rangle$ from that point on. The only case where this does not happen is for the root node, where none of the n least significant qubits of *a* have a state of $|1\rangle$. Therefore, it is sufficient to check the state of $|a[0]\rangle$ at the end of this loop and set the flags accordingly. The last loop of the algorithm is only included to revert *scratch* to its previous state.

Scaffold Implementation: The implementation for a tree height of $n = 4$ is shown in Program 8 [4]. It shows an example of a module where classical and quantum code can be used hand in hand. The usage of parallel forall() is also evident, where we can take advantage of it to perform parallel initialization of qubits, but cannot use it for later loops where consecutive iterations depend on each other.

```
#include "gates.h"

const int n = 4;        //tree depth

module PARSENODEROOT(qreg a[5], qreg root[1], qreg even[1], int n) {
        qreg scratch[5];
        qreg ancl[1];
        int index;
        int i;

        forall(i=0;i<=4;i++){
                prepZ(scratch[i]);
        }

        prepZ(ancl[0],1);

        for(index=n;index>=1;index--){
                X(scratch[index]);
                Toffoli(scratch[index-1],scratch[index],a[index]);
                X(scratch[index]);
                CNOT(scratch[index-1],scratch[index]);
        }

        X(scratch[0]);
        CNOT(root[0],scratch[0]);
        CNOT(even[0],scratch[0]);
        X(scratch[0]);

        for(index=1;index<=n;index++){
                CNOT(scratch[index-1],scratch[index]);
                X(scratch[index]);
                Toffoli(scratch[index-1],scratch[index],a[index]);
                X(scratch[index]);
        }
}

module main() {
        //define the quantum registers in question
        qreg a[5];
        qreg root[1];
        qreg even[1];

        //initialize to 0
        prepZ(root[0]);
        prepZ(even[0]);
        PARSENODEROOT(a,root,even,n);
}
```

Program 8: Parse Node Root Algorithm Scaffold code

A.2 Quantum Fourier Transform

Quantum Fourier Transform is an important cornerstone of many quantum algorithms. It is a linear transformation on the state of n qubits, where the state $\sum_{i=1}^{n-1} x_i|i\rangle$ is transformed to $\sum_{i=1}^{n-1} y_i|i\rangle$ according to the formula: $y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega^{jk}$. It can be computed on a quantum computer with only $O(n^2)$ quantum gates.

The circuit-level realization of the QFT is presented in Figure 5 [22, 24]. We present a possible Scaffold implementation in Program 9.

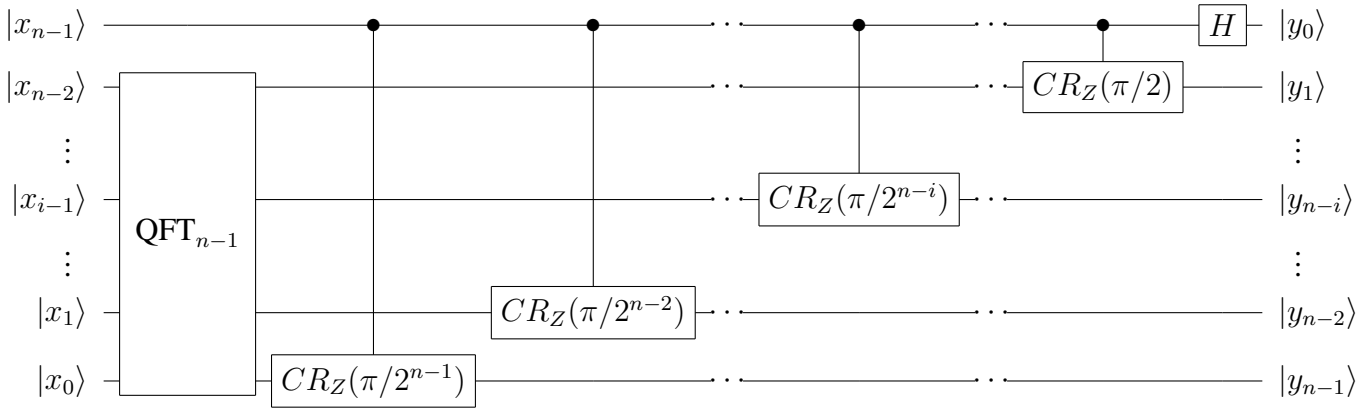

Figure 5: n-qubit Quantum Fourier Transform circuit

Algorithms Description: In the recursive implementation of the QFT, the circuit can be synthesized using a similar QFT block on a reduced number of qubits, plus a series of controlled rotations on the outcome of that lower-order QFT. The algorithm therefore is simple: it requires only a series of well-defined gates in the quantum domain, and a recursive function call in the classical domain.

Scaffold Implementation: The Scaffold implementation is a direct result of looking at the above circuit. Each invocation generates one Hadamard as well as a number of controlled rotation gates. The invocations will stop once the algorithm reaches the last qubit. As mentioned, loops like this will be unrolled statically to represent individual gates, and therefore increasing the size of input would easily yield a huge number of gates generated by the compiler.
Because we need rotations with angles dependent on an integer loop counter, it is convenient to define a module which takes in the integer as an argument an constructs the angle. We call this the `controlledRd` module, with d being the integer.

```
#include "gates.h"
#include "math.h"

const float PI = 3.142;
const int n = 10;

//Controlled rotation about the Z axis with angle=pi/2^d
module controlledRd (qreg target[1], qreg control[1], int d) {
        float angle;
        angle = PI/pow(2,d);

        //standard rotation gate from the library
        controlledRz(target[0], control[0], angle);
}

//Recursive QFT
module QFT(qreg data[n]) {

        // Termination condition
        if(length(data) == 1) {
        return;
        }

        // Recursively call
        QFT(data[0..length(data) - 2]);

        // Hadamard
        H(data[length(data)-1]);

        // Rotation chain
        int i=0;

        for(i = 0; i < length(data)-1; i++) {
        controlledRd(data[i], data[length(data)-1],  length(data)-i-1);
        }
}

//Main module
module main() {
     qreg input [n];
     QFT (input);
}
```

Program 9: Recursive QFT Scaffold code

A.3 Zero-state measurement

In this subsection we demonstrate Scaffold's capability to provide runtime environment support for programs. The following is a simple example of a classical subroutine and shows how these subroutines can naturally be leveraged in quantum algorithms. Furthermore, it illustrates the usage of classically conditioned modules.

The `allZero` subroutine is a classical function, very similar to those written in C. It takes as input an array of "bits", and checks each of the them in sequence. It returns `true` if and only if all of the bits are zero.

In the `main` module, we define two distinct quantum registers: `data` and `test` . The former is used for storing the arbitrary states of 5 (non-entangled) qubits, while the latter is a used to as a test qubit for performing an arbitrary operation based on the states of those 5 qubits. In this case, we seek to invert the state of `test` about the X axis if all the `data` qubits are found to be in the $|0\rangle$ state. Assuming that we do not care about preserving the state of those 5 qubits, we can achieve this by performing measurements on them.

Both quantum registers are local to the `main` module, since the function call can only take place on classical arguments, implying that the qregs they are not passed. On the other hand, we define a classical array of type `bit` and pass it to the classical function the same way we would do in C. This will then return to us a boolean evaluation of the measured qubit states. Now we can use the classical control functionality of Scaffold to apply the X gate if the condition has evaluated to `true` . From a practical point of view, this process involves the communication of the quantum core and classical processor during runtime (execution), where the measurement results are fed from the quantum core to the classical processor to allow for feedback on the use of X gate, similar to the "select" signal of a multiplexer. The Scaffold code corresponding to this algorithm is shown in Program 10.

```
#include "gates.h"

// Purely classical function for 5-bit data
//returns true if all bits are zero

bool module allZero(bit *data) {

    // Result of the result
    bool result = false;

    // Variable to loop
    int i = 0;

    // Toggle for every 1 detected
    for(i = 0; i < 5; i++) {
        if(data[i] == 1) {
            result = true;
        }
    }

    // Return the result
    return result;
}

module main() {

    local qreg data[5];
    bit result[5];
    local qreg test[1];

    // Measurements and all zero
    int i = 0;
    forall(i = 0; i < 5; i++) {
            measX(data[i], result[i]);
    }

    // See if all zeros
    bool measuredZeros = allZero(result);

    //Classically controlled unitary
    if(measuredZeros){
            X(test[0]);
        }
}
```

Program 10: Zero-state measurement Scaffold code

B Envisioned Tool Flow

As mentioned in the design goals of Scaffold, this language is meant to be employed in a larger toolchain that can compile various quantum algorithms for execution on a physical platform. This section contains information about the possible overall toolchain design. It illustrates and describes how different levels in this chain interact with one another.

B.1 General View

Figure 6 illustrates the different layers in the frontend of the toolbox. The inputs to the toolchain are programs coded using the Scaffold language. A "source code" is written for this purpose to represent the functionality of an algorithm which is of importance to the programmer. Examples of this are the Grover Search Algorithm, Shor's Algorithm, Binary Welded Tree Problem and the Quantum Fourier Transform. These codes must abide by the specifications and syntax presented throughout this document.

The source code will pass through a compiler which is responsible for interpreting the high-level code and converting it into a low-level language to be used by later tools before feeding into the quantum computer. This compiler uses the LLVM infrastructure and extends the modular framework it presents to bring in Scaffold's functionality. The first step in the compilation process is parsing, where the grammatical structures of the code's text stream are extracted and analyzed. This begins by a lexical analysis of the program where meaningful tokens are extracted, followed by a syntactic analysis where these tokens are checked to be forming valid expressions. The syntax and grammar specified for the language play an integral role at these stages. At the end of the parsing phase, an Abstract Syntax Tree (AST) is generated representing the syntactic structure of the source code and the dependency between different syntactic elements [3].

The goal for the compiler infrastructure is to create a low-level representation of the source code, in a language called Logical Quantum Assembly (logical QASM). This language is universal for all targets and thus independent of the actual implementation technology, and describes the circuit in terms of abstract quantum assembly gates or simple classical assembly code. The inclusion of classical assembly is important to reduce the size of code generated by the compiler, since unrolling all loops and hierarchical structures can result in huge and possibly infeasible program sizes. The backend computer will be able to step through this code and generate the set of individual quantum operations for execution on the quantum core.

Since Scaffold allows for the use of both classical (reversible) and quantum code, the compiler

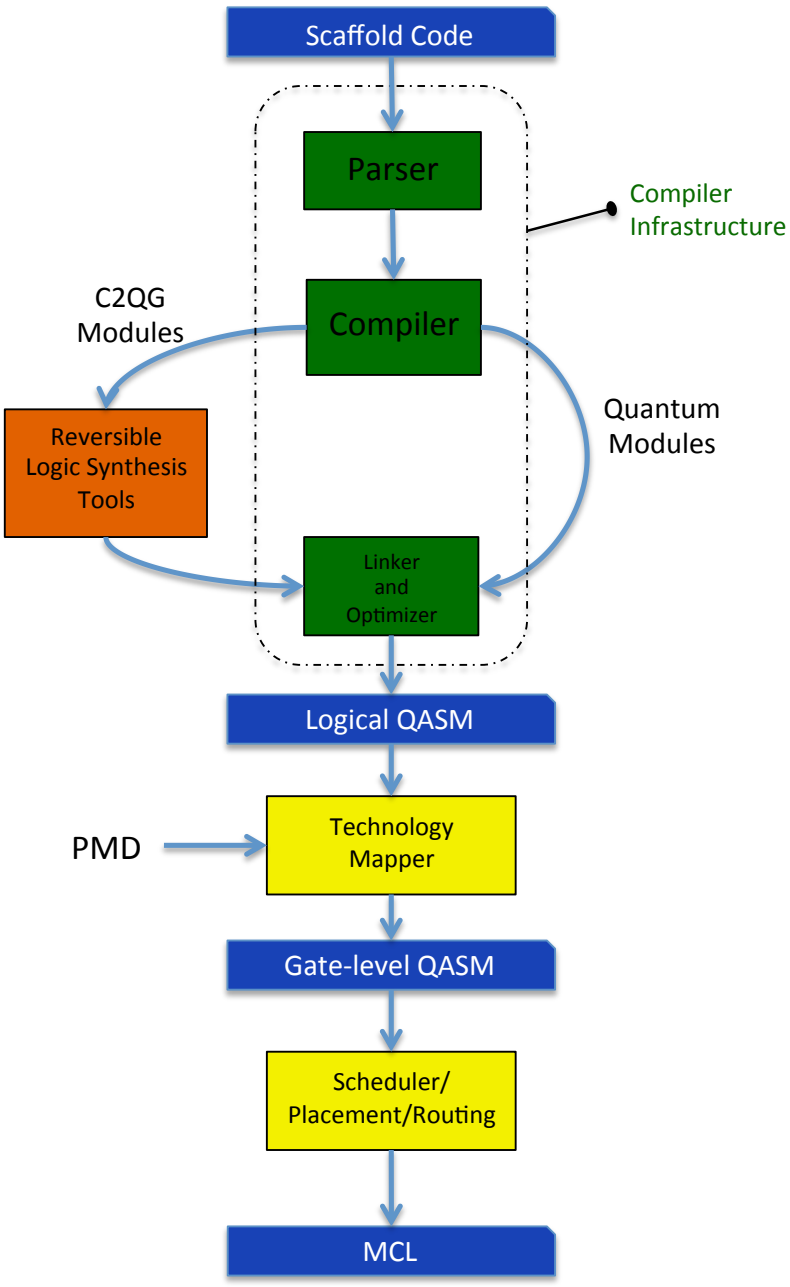

Figure 6: This flowchart shows details on the front end of the QC toolchain, from Scaffold to an intermediate representation such as Logical QASM. The backend of the toolflow would vary with different synthesis approaches, but we show an example toolflow here. Other documents give more details on particular backend toolflows being implemented.

takes different approaches in interpreting each one. Codes which are already specified in reversible logic format inside normal modules (using quantum gates, quantum registers, conditional and iteration statements, etc.) are directly converted to their corresponding logical QASM representation via techniques such as loop unrolling. However, in the case of purely classical code — as specified in C2QG modules — the compiler does not directly make this conversion; rather the parsed output is separated from the rest of the code, and fed into existing tools for reversible logic synthesis (an example of such tool is Reed-Muller Decision Diagram Synthesis (RMDDS) [20]). The interface to each tool is different: some accept a PLA format declaration of corresponding inputs and outputs, while others accept the non-reversible logic in forms of AND, NOT, OR, etc. specifiers. In any case, these tools convert the non-reversible logic into reversible logic suitable for quantum computers, and specify them with gates such as NOT, CNOT and Toffoli. This process often involves the creation of ancilla qubits to accommodate the needs of reversible logic. As mentioned, this might incur a large overhead, which the programmer can try to reduce by making C2QG modules smaller in size. Finally, to construct the final logical QASM output, the output of the two different flows are merged together.

From this step onwards, technology mapping takes place, in which given a set of legal gates as specified in the Physical Machine Description (PMD), the logical QASM is converted to gate-level QASM. The target language which is understood by the quantum core is the Machine Control Language (MCL), which contains a very specific set of instructions targeted to its relavent technology, and other primitives such as the time stamp for instruction execution. This stage is very dependent on the technology that is targeted, and each physical implementation has its own set of allowed (implementable within that technology) circuit primitives [5]. Therefore, unlike the technology-oblivious logical QASM, the gate-level QASM can be described as a netlist of quantum gates that are available in the target PMD. One important step in the transformation from logical to gate-level QASM is the approximation that is often needed in arbitrary quantum operations. For example, the gate library allows for qubit state rotations of arbitrary angles. There are several approaches for synthesizing this efficiently – to a good degree of precision – using fault-tolerant gates [12, 15]. These approaches include *phase kick-back* [17], *the Solovay-Kitaev algorithm* [6], and *Fowler's algorithm* [10].

The rest of the toolchain flow takes place at the backend, which deals with issues such as placement and routing of quantum primitives, and is not the focus of this document [1, 7].

Figure 7: LLVM's 3-phase compiler structure

B.2 LLVM Compiler Infrastructure

LLVM is an umbrella project designed as a set of reusable libraries with well-defined interfaces, which can be used to implement compilers for a wide variety of source and target languages [18]. LLVM implements these different compilers by taking a 3-phase approach: The Frontend, the Optimizer, and the Backend. Figure 7 shows this approach. Independent modifications can be performed on the frontend and the backend to tailor for various source languages and targets.

The middle phase, the optimizer, is somewhat isolated, and compile-time optimizations can be done without being constrained by the requirements of either language, although it has to serve both from top and bottom well. The representation of code at this stage is known as the Intermediate Representation (IR), and is an important feature of LLVM. It is a low-level RISC-like instruction set capable of representing full codes, although it does not act exactly as an instruction set architecture (it assumes an infinite number of registers for example). This isolation means that optimizations can be done on the IR textual representation, with relatively small knowledge of the other parts of the compiler. On the other hand, to write the front-end and back-end requires only knowledge of the features of the IR. These are the reasons why LLVM provides great flexibility for novel compiler designs.

Another important aspect of LLVM compiler design is that it consists of a set of libraries, rather than being a monolithic command line compiler (like GCC). This is a huge benefit when designing custom compilers for new applications. Depending on the requirements of the compiler at hand, one can elect to pick and choose different libraries at different orders. For example, various libraries with different functionality on each pass exist at optimization stage (e.g. arithmetic identity reduction, expression re-association, inliner, etc.) and code generation stage (e.g. instruction selection, register allocation, scheduling, etc.).

Due to the reasons above, the quantum compiler for Scaffold draws upon these features in its

implementation. In simple forms, in an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR (usually, but not always, by building an AST and then converting the AST to LLVM IR). This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code.

C Scaffold Syntax Specifications

Scaffold shares much of its syntax with the C language [2, 16], and thus we do not elaborate on all the possible details. This section is an adjunct to the previous text and examples; it elaborates on some particularly relevant aspects.

C.1 Valid data types and casts

Classical variables in Scaffold contain the following valid data types, which are common with C:

```
int: 32-bit signed integer
unsigned int: 32-bit unsigned integer
char: 8-bit signed integer/character
unsigned char: 8-bit unsigned integer
float: signed 32-bit IEEE 754-2008 floating point number
double: signed 64-bit IEEE 754-2008 floating point number
bit: bit value, can either be 0 or 1
bool: boolean value, can be either "true" or "false"
```

When working with classical variables in scaffold, the following casts between data types are allowed:

```
bool: to signed/unsigned int, char, float, double (0 = false, 1 = true)
signed/unsigned int to: float, double, signed/unsigned char, bool
signed/unsigned char to: float, double, signed/unsigned float, bool
float to: double, signed/unsigned char, signed/unsigned int
double to: int, signed/unsigned char, signed/unsigned int
```

C.2 Operators

In both normal and C2QG modules, by operators between variables we mean the following:

```
1. Assignment (=):
    - all datatypes (copy by value)
2. Basic arithmetic (+,-,*,/) and
    compound variants (+=,-=,*=,/=)
    - all datatypes
3. Increment/decrement (++,--)
    - signed/unsigned int/char
4. Bitwise operations (!,|,&,^,<<,>>) and
    compound variants (!=, |=, &=, ^=, >>=, <<=)
```

```
    - signed/unsigned int/char (note ^ is an xor)\\
5. Logical (!!,&&,||)
    - singed/unsigned int/variable bit int/char \\
```

C.3 Preprocessor directives

Preprocessor directives and macros can be used as in normal C code. In particular, this allows us to include standard files of gates prototypes.

References

[1] A. Abdollahi and M. Pedram, "Analysis and synthesis of quantum circuits by using quantum decision diagrams," *Proceedings of the Conference on Design, Automation and Test in Europe*, 2006.

[2] A. Allain, "C programming tutorials," www.cprogramming.com.

[3] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[4] A. Chakrabarti, C. Lin, and N. K. Jha, "Analysis of the binary welded tree algorithm," June 2012.

[5] E. Chi, S. A. Lyon, and M. R. Martonosi, "Tailoring quantum architectures to implementation style: A quantum computer for mobile and persistent qubits," *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[6] C. M. Dawson and M. A. Nielsen, "The Solovay-Kitaev Algorithm," *Quantum Information and Computation*, February 2008.

[7] M. J. Dousti and M. Pedram, "Minimizing the latency of quantum circuits during mapping to the ion-trap circuit fabric," *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012.

[8] B. Eastin and S. T. Flammia, "Q-circuit tutorial," arXiv:quant-ph/0406003v2.

[9] A. I. Faruque *et al.*, "Scaffold: Quantum Programming Language v1.0," January 2012.

[10] A. G. Fowler, "Constructing Arbitrary Steane Code Single Logical Qubit Fault-tolerant gates," *Quantum Information and Computation*, 2011.

[11] J. Gosling *et al.*, *The Java Language Specification*, Oracle America Inc., February 2006.

[12] A. Harrow, "Quantum compiling," May 2001, Bachelor's Thesis, Massachussets Institute of Technology.

[13] *IEEE Standard for Standard Verilog Hardware Description Language*, IEEE Computer Society, April 2006.

[14] *IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Standards Association, January 2012.

[15] N. C. Jones *et al.*, "Simulating chemistry efficiently on fault-tolerant quantum computers," April 2012, arXiv:1204.0567v1.

[16] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, Inc., 1988.

[17] A. Y. Kitaev, A. H. Shen, and M. N. Vyalyi, *Classical and Quantum Computation*, 1st ed. American Mathematical Society, 2002.

[18] C. Lattner, *The Architecture of Open Source Applications*, vol. I, ch. 11.

[19] C. Lattner and V. Adve, ""LLVM: A compilation framework for lifelong program analysis and transformation"," *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[20] C. Lin and N. K. Jha, "RMDDS: Reed-Muller Decision Diagram Synthesis of Reversible Logic Circuits," *ACM Journal on Emerging Technologies in Computing Systems*.

[21] T. S. Metodi, A. I. Faruque, and F. T. Chong, *Quantum Computing for Computer Architects, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[22] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press.

[23] B. Ömer, "Quantum Programming in QCL," January 2000, Master's Thesis, Technical University of Vienna.

[24] G. Paradisi and H. Randriam, "A presentation of the Quantum Fourier Transform from a recursive viewpoint," October 2004, arXiv:quant-ph/0411069v1.

[25] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 2000.