Chisel-Q: Designing Quantum Circuits with a Scala Embedded Language

Xiao Liu and John Kubiatowicz
Computer Science Division
University of California, Berkeley
Email: {xliu, kubitron}@eecs.berkeley.edu

***Abstract*—We introduce *Chisel-Q*, a high-level functional language for generating quantum circuits. Chisel-Q permits quantum computing algorithms to be constructed using the meta-language features of Scala and its embedded DSL *Chisel*. With Chisel-Q, designers of quantum computing algorithms gain access to high-level, modern language features and abstractions. We describe a synthesis flow that transforms Chisel-Q into an explicit quantum circuit in the Quantum Assembly Language (QASM) format. We also discuss several optimizations to reduce the generated hardware cost. The Chisel-Q tool includes resource and performance estimation which can be used to compare different implementations of the same functionality. We compare the output of the generic Chisel-Q synthesis flow with hand-tuned versions of well-known quantum circuits.**

Keywords*—*Quantum Computing, Computer Aided Design

I. Introduction

Quantum computing [1, 2] has great potential to speed up certain computations, such as factorization [3] and quantum mechanical simulation [4]. Although practical quantum computers are still on the horizon, research progress is steady: over the last decade, physicists have investigated a number of approaches to implementing quantum circuits [5, 6], computer architects have investigated architectures for quantum computers [7, 8], and mathematicians have explored how to express difficult computational problems as instances of quantum computing [9, 10]. Unfortunately, techniques for *expressing* quantum algorithms are mostly limited to high-level mathematical expressions or low-level sequences of quantum gates [11]. More traditional programming languages have not yet surfaced that are capable of expressing and handling the idiosyncrasies of quantum computing. As a result, many of the time-honored techniques for abstraction, design, and debugging of classical algorithms are not available to the writer of quantum algorithms.

Since many proposed quantum computing architectures express algorithms using a *quantum circuit model* [2], *i.e.* a netlist-like sequence of quantum gates operating on quantum bits (or "qubits" for short), one approach would be to provide an improved, programmatic interface for generating quantum circuits. Hardware Design Languages (HDLs) such as Verilog [12] immediately come to mind. However, quantum computing circuits have their own challenges stemming from their need to be reversible[1]: temporary state bits, called *ancillas*, must often be introduced to turn irreversible computations into reversible ones. To decouple these ancillas from the final output bits, parts of the circuit must often be reversed at the end of a computation to return ancillas back to their original state. In fact, classical design methodologies utilizing *state elements* introduce a need for tracking the *history* of the state in order to retain enough information to revert ancillas at the end of the computation. Fortunately, as we discuss in Sections III and IV, much of the ancilla reversal process can be handled with automatic transformations and should not be something that a designer must consider. Thus, we seek a new *domain specific language* (DSL) which focuses a designer's attention on the important aspects of quantum circuit design and which can be compiled into correct quantum circuits.

[1]The quantum equivalents of classical gates are unitary operators which must be reversible by definition.

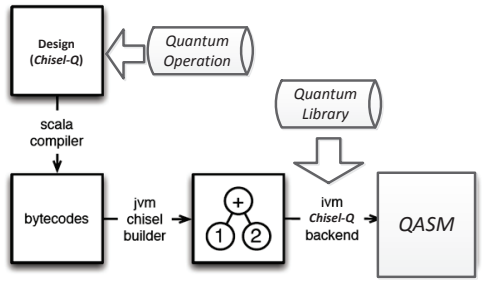

Fig. 1. Proposed Quantum Circuit Design Flow. Quantum circuits are described in a new Scala embedded language, called Chisel-Q. The output of the synthesis tool flow is QASM (the Quantum Assembly Language) [11].

To develop our language, we started with *Chisel* [13], a new hardware description language that supports *classical* hardware design with parameterized generators and layered domain-specific hardware syntax. Chisel is embedded in the Scala programming language, and raises the design abstraction level by providing object orientation, functional programming, parameterized types, and type inference. In fact, all of the meta-level language features of Scala are available to the hardware designer in Chisel. As shown in [13], Chisel permits compact descriptions of hardware circuits using high levels of abstraction, after which the Chisel backend generates low-level Verilog (for synthesis) or C simulators (for design verification). Chisel is gaining a rapid following and has already been used to fabricate a complete RISC processor with a vector unit.

In this work, we introduce Chisel-Q, a quantum hardware description language (QHDL) and compilation environment that permits the expression of quantum circuits using Chisel syntax. As shown in Figure 1, Chisel-Q takes a *classical* digital circuit description, including both combinational and state elements, and produces a *quantum* circuit with similar functionality. Although not required, designers may choose to include quantum operators in their circuit descriptions to help direct the compilation process. The output of Chisel-Q is the defacto-standard quantum netlist format, called "QASM" (for Quantum Assembly Language) [11]. Chisel-Q includes a resource and performance estimation tool that reports the hardware cost, parallelism and latency of the produced netlist.

By supporting the existing Chisel syntax, we gain two important benefits: First, the fact that Chisel-Q is embedded in Scala means that quantum computing algorithms can be designed in a high-level, modular fashion, using modern

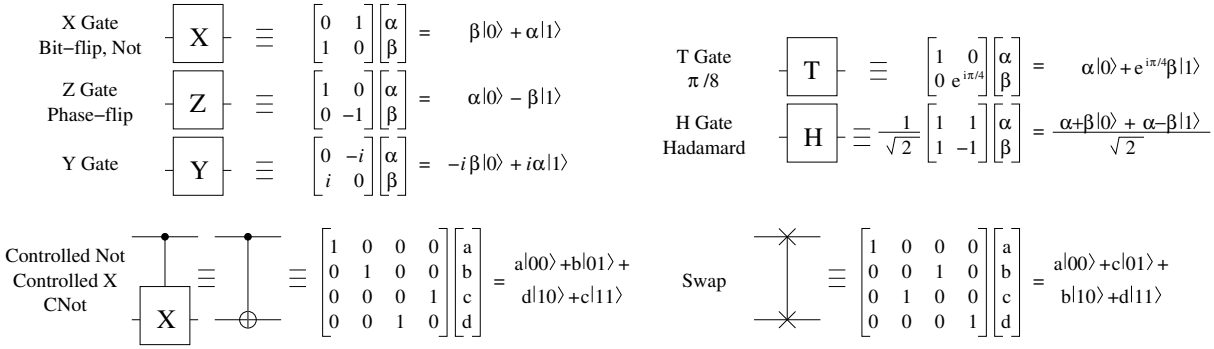

Fig. 2. Examples of Quantum Gates. Horizontal lines represent quantum bits (or "qubits"). Qubits are considered to be in a *superposition* of 0 and 1, written as $\psi = \alpha|0> + \beta|1>$, where α and β are complex constants. This figure shows one and two qubit gates. Only the *CNOT* gate (a quantum equivalent of the classical XOR gate) and *SWAP* gates operate on two input bits, while the others operate on a single bit (altering the phase and/or sign between |0> and |1>). Not shown here is a *measurement* gate which takes a qubit input and produces a classical binary result.

language features – an important step forward to the process of describing such algorithms. Second, with Chisel-Q, we can harness the skills of a variety of classical circuit designers to produce quantum computing circuits. Consider, for instance, a quantum floating-point unit derived from a classical design – something completely possible with Chisel-Q.

Since Chisel-Q supports an extended syntax consisting of quantum operators such as CNOT, a quantum circuit designer can introduce hand-optimized versions of common modules– thereby choosing exactly when and how ancillas will be generated and where circuit reversal will be performed. A clever designer can often produce smaller and more efficient implementations of structured functions (such as addition or multiplication) than can be produced by Chisel-Q. The modular nature of Chisel permits such operators to be introduced selectively and reused by many designs.

II. Preliminaries and Motivation

In this section, we introduce both quantum computing and Chisel. Even though quantum computing is radically different from classical computing in a number of ways, we can abstract most of its interesting properties into a quantum circuit model [2] that mirrors classical circuits reasonably closely. It is this rough congruence that permits us to successfully exploit a classical design tool (Chisel).

A. Quantum Computing and Quantum Circuits

Quantum Computing exploits quantum effects such as quantum superposition and entanglement (once called "spooky action at a distance" by Einstein) to perform certain computations more efficiently than possible with a classical computer. While classical circuit designers attempt to reduce quantum effects (*e.g.* as CMOS technology scales into the tens of nanometer range), quantum circuit designers strive to enhance these effects.

A single quantum bit is referred to as a *qubit* and is in a *superposition* of 0 and 1, written as $\psi = \alpha|0> + \beta|1>$, where α and β are complex constants such that $|\alpha|^2 + |\beta|^2 = 1$. This superposition means that each qubit carries more information than a classical bit (which can only be in *either* a 0 *or* a 1 state). The act of measuring a qubit will return either a 0 (with probability $|\alpha|^2$) or a 1 (with probability $|\beta|^2$). After a qubit has been measured, the result is a normal binary value that can be processed with normal, classical computing circuitry

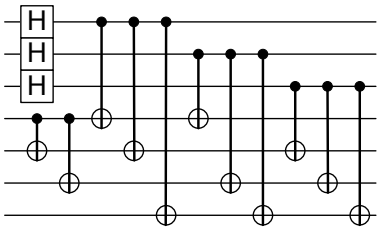

Fig. 3. Example of a Quantum Circuit. Horizontal lines are individual qubits. This circuit shows single qubit operations (H) and two-qubit operations (CNOT). Time advances from left to right, and operations are done in order. Gates that do not share a qubit may occur in parallel.

Many quantum computing algorithms can be constructed as *quantum circuits* which consist of a set of qubits operated upon by quantum gates – similar to what occurs in the classical realm with two important differences: First, quantum gates must be reversible, since they represent *unitary* transformations on data. Second, according to the no-cloning theorem [14], qubits cannot be duplicated, which prevents direct implementation of circuits with fan-out. Section III-B revisits the issue of fan-out.

Generally, a quantum circuit is constructed from a set elementary quantum gates, as shown in Figure 2. A standard universal set of one or two qubit quantum gates includes the *Controlled NOT* (CNOT) gate that acts like reversible XOR gate in classical circuit, the Hadamard/H gate that converts the qubit value to a phase value and vice versa, the $\pi/8$ gate, also known as the T gate, and the phase gate. Not shown in Figure 2 is the *measurement* gate that produces classical values from qubits. Figure 3, shown above, illustrates a quantum circuit constructed from qubits and quantum gates. Further, with the above gates, we can construct a 3-bit *Toffoli* gate which computes $c \oplus (a \wedge b)$, sometimes called the *Controlled-Controlled-NOT* (CCNOT) gate. The 3-bit Toffoli gate is universal and any reversible classical circuit can be constructed from Toffoli gates, something we exploit in Section III-B.

Quantum circuits can be represented by a netlist format that has become a de facto standard in the quantum computing community, namely QASM [11]; QASM allows the definition of qubits and sequences of operations between them. Note that we can manipulate quantum circuits very similarly to classical circuits – they have "wires" (*i.e.* qubits) and "gates" (with interconnections between them). We can perform transformations on these circuits without ever needing to deal with the quantum nature of the "wires", other than ensuring the reversibility of the circuit (which is a "classical" property).

```
def innerProductFIR[T <: Num](w: Array[Int], x: T) =
   foldR(Range(0, w.length).map(i => Num(w(i)) *
                                delay(x, i)), _ + _)

def delay[T <: Bits](x: T, n: Int): T=
    if( n == 0) x else Reg(delay(x,n-1))

def foldR[T <: Bits] (x: Seq[T], f: (T, T) => T): T =
    if (x.length == 1) x(0)
    else f(x(0), foldR(x.slice(1, x.length), f))
```

Fig. 4. FIR Digital Filter Module expressed in Chisel.

B. Quantum Oracles

As discussed in the previous section, qubits exist in a superposition of states – having properties of both one and zero at the same time. When you put N qubits together into an N-bit register, you gain a state element that can hold all 2^N combinations of bits at the same time. It is this exponential amount of state that leads, under certain circumstances, to powerful quantum computing algorithms.

Although the details of such algorithms are beyond the scope of this paper[2], it is important for our purposes to understand that most quantum computing algorithms have a core that is often called an *oracle*. Oracles are portions of the algorithm that can be regarded as "black boxes" and are often specified by classical functions, such as addition or modular exponentiation. These functions take as input quantum values (such as our N-bit register, above), and produce superposed outputs.

An oracle described as a classical digital circuit can be transformed into a quantum circuit by replacing irreversible operations such as "AND" or "OR" with reversible equivalents in the set of quantum gates. The extra *ancilla* qubits that are introduced to make the circuit reversible can be restored to their initial states at the end of the computation through selective reversal of the forward computation[3]. Consequently, a classical circuit such as an adder that takes classical values as input (*i.e.* values *without* superposition), can be transformed into a quantum adder with quantum inputs (*i.e.* values *with* superposition) by performing the correct transformation: introducing reversibility and logic for ancilla restoration. *By automating this process, we simplify the design of complex quantum oracles.*

C. Chisel: a Scala Embedded Language for Hardware

The Chisel [13] Domain Specific Language (DSL) is embedded in the powerful Scala language [15]. Chisel raises the design abstraction level by providing functional programming, parameterized types, and object orientation. It exploits Scala libraries to define hardware data types and routines to convert hardware data into low-level Verilog for logic synthesis.

Since Chisel is embedded in Scala, the Chisel programmer may use all of the meta-programming features of Scala to describe their circuit. Consider, for instance, a parameterized inner-product FIR digital filter, mathematically described as:

$$y[t] = \sum_j \omega_j * x_j[t-j] \quad (1)$$

[2]You might start with the excellent book by Nielsen and Chuang [2], for an introduction to quantum computing.

[3]If ancilla remain entangled with a result at the end of a computation, they will introduce errors into the result when they are recycled.

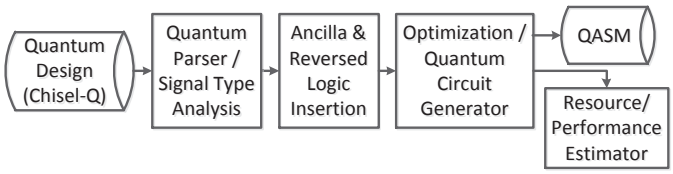

Fig. 5. Proposed Chisel-Q Design Flow. Quantum circuits are described in Chisel-Q, then passed through parsing, conversion, optimization, and circuit generation. The output of our synthesis tool flow is either QASM or statistics about the resulting circuit (such as gate count or level of parallelism).

The programmer can describe this design with Chisel in a compacted manner as shown in Figure 4. Here, function *delay* creates an n-cycle delayed copy of its input, *foldR* describes a reduction circuit given a function f and it creates summation circuit. Based on above, *innerProductFIR* is introduced to combine the multiplication and addition together.

Internally, Chisel constructs a netlist-like graph of operations that represents the output circuit. By walking this netlist, backend generators can transform this graph into whatever format is desired. For instance, Chisel includes modules to output Verilog as well as a high-level C simulator of the circuit. The Chisel architecture makes it particularly easy to add new backends – a feature that we exploit to transform classical circuit descriptions into reversible quantum circuits. The following two sections discuss how we perform this transformation.

III. Chisel-Q Architecture

In this section, we discuss the basic flow of Chisel-Q, as illustrated by Figure 5. We focus on the transformation of circuits *without* state (*i.e.* combination circuits) and save the discussion of circuits with state elements for Section IV. As mentioned earlier, the classical Chisel framework builds a dataflow graph of circuit elements from modules expressed in the Chisel language; to enable fine tuning of the output, we supplement the Chisel syntax with quantum operators.

A basic summary of Chisel-Q compilation is as follows: First, we traverse the dataflow graph to identify circuit elements and separate quantum from classical signals; this operation identifies portions of the circuit that are intended to handle quantum data (*i.e.* the *quantum datapath*). Next, we map classical irreversible gates into quantum reversible gates — introducing ancillas as necessary. We construct a reversed computation to return ancillas to their original states, thereby decoupling them from the computation. Finally, after some simple optimizations, we output QASM for the quantum datapath, along with performance and parallelism statistics.

A. Signal Type Analysis

To separate classical signals and circuits from quantum ones, we utilize a combination of user annotations and dataflow analysis. Our signal identification mechanism permits designers to transform part of the design (*e.g.* the data path) while aspects of the design remains classical (*e.g.* the control path). By default, signals in Chisel are labeled as "classical". The user can highlight signals that will carry quantum data with an **`isQuantum`** annotation. These annotations are typically placed in the top-level module. Further, quantum operators (as discussed in Section III-C) provide implicit labeling of their outputs as "quantum" in nature.

Classical Gate	Quantum Gate
AND	Toffoli
OR	Toffoli, X
NOT	X
XOR	CNOT

TABLE I. MAPPING BETWEEN CLASSICAL GATES & QUANTUM GATES.

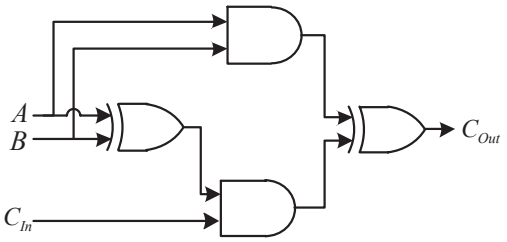

Fig. 6. Classical circuit to compute a "Carry".

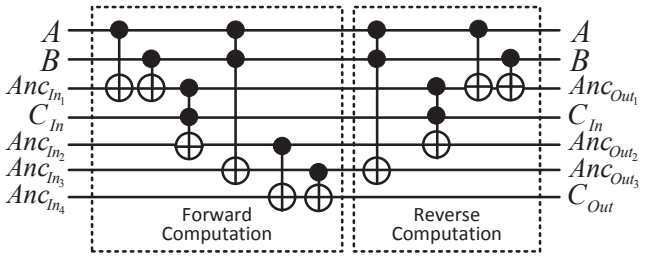

Fig. 7. Quantum version of the "Carry" circuit from Figure 6. The *Forward Computation* computes output C_{out} using four ancillas and gate substitution according to Table I. Three of the ancillas are returned to their initial state by the *Reverse Computation* logic, such that $Anc_{In_1} = Anc_{Out_1}$, $Anc_{In_2} = Anc_{Out_2}$, and $Anc_{In_3} = Anc_{Out_3}$. The fourth ancilla is consumed for the output.

Chisel-Q traces signals forward through the datapath, labeling signals as "quantum". Classical signals that interact with quantum ones are "upgraded" to quantum signals by selecting appropriately initialized ancilla bits. While, in principle, quantum signals can be "downgraded" to classical signals through measurement, our focus on quantum oracles places this operation outside the scope of this paper[4]. Quantum labeling is, by nature, an inter-module operation: when module inputs are labeled as quantum, then the module itself must be implemented with quantum operators and provide quantum outputs. Each time quantum conversion is detected at an output in a submodule, we restart analysis of the calling module. The process is also conducted iteratively to deal with sequential loops, until no more signals in the design can be further converted.

B. Ancilla Insertion and Reverse Logic Construction

To transform a classical circuit into a quantum equivalent, Chisel-Q walks through the dataflow graph. Each node in the graph represents a gate-level logical operation (*e.g.* AND) or an abstract arithmetic operation (*e.g.* summation). We start by mapping the classical gates to quantum gates, as summarized by Table I. To resolve the fan-out problem mentioned in Section II-A, ancilla qubits are introduced for each gate level operation. Chisel-Q handles literal values with initialized ancillas. Read-only memories (ROMs) with quantum addresses can be handled by implementing the ROM as a large sum-of-products (similar to a PLA).

[4]Measurement is typically part of the enclosing quantum algorithm. In the future, we plan to allow Chisel-Q to express complete quantum algorithms, in which case we will revisit the role of measurement in Chisel-Q.

Quantum Gate	Operator	Example
Toffoli	`#&&`	`c_p := c #&& (a, b)`
CNOT	`#^`	`c := a #^ b`
Pauli-X	`!` `X()`	`c := !a` `c := X(a)`
Pauli-Y	`Y()`	`c := Y(a)`
Pauli-Z	`Z()`	`c := Z(a)`
Hadamard	`H()`	`c := H(a)`
Phase	`P()` `P()...angle()`	`c := P(a)` `c := P(a) angle(n,d)`
C-phase	`#@` `#@...angle()`	`c := a #@ b` `c := a #@ b angle(n,d)`

TABLE II. SYNTAX OF QUANTUM GATES IN CHISEL−Q.

Chisel-Q implements some abstract operators with built-in implementations. For instance, by default, it utilizes a hand-tuned parameterized quantum adder [16] for addition and comparison[5]. Integer comparisons are based on the adder. Chisel-Q also supports quantum logical operators and shift with constant or varied steps. Of course, Chisel-Q can always be extended by developing new operators as Chisel-Q modules.

To illustrate the transformation process, we consider the circuit in Figure 6, a classical "Carry" circuit. Figure 7 shows the corresponding quantum version derived by gate mapping. In particular, the *Forward Computation* portion of the circuit utilizes four ancillas, four CNOT gates, and two Toffoli gates to produce its output, C_{out}. The output value is implemented by transforming an input ancilla (here labeled Anc_{In_4}) in order to leave the input values untouched.

The remainder of the transformation involves restoring temporary ancillas to their initial states. Since the transformed circuit is reversible by construction, restoring temporary ancillas merely requires walking backward through the dataflow graph, reverting any computation that was performed on these ancillas[6]. This process can also restore input bits to their original values if they were altered. The *Reverse Computation* in Figure 7 performs reversal of Anc_{In_1}, Anc_{In_2}, and Anc_{In_3}.

C. Optional Use of Explicit Quantum Operators

To allow developers to make full use of their quantum knowledge, Chisel-Q supports an optional native syntax for quantum circuit design. Table II shows the quantum operators available in Chisel-Q. Highlights include Toffoli, CNOT, Pauli, Hadamard, Phase and Controlled Phase (C-phase) gates. Without the angle() modifier, Phase and C-phase gates perform a $\pi/2$ phase rotation. With the angle operator, designers can specify any rational fraction of π. Most of these operators are self-reversing, although phase and C-phase gates must be reversed by applying a negative angle. It should be noted that designers can use annotation **`IsReversed = false`** to disable generation of reversal logic when appropriate.

Since quantum circuits differ from classical circuits in many aspects, the quantum development feature provided by Chisel-Q permits clever designers to implement a variety of efficient quantum designs. Example usage of Toffoli and CNOT gates can be found in Figure 8, while Hadamard and C-phase gates can be found in Figure 12.

[5]Chisel-Q can call out to external generators when desired.

[6]We must also develop reversed versions of external circuit generators.

```
class Ripple_AddIO(width_n: Int) extends Bundle {
    val in1  = Bits(INPUT, width_in)
    val in2  = Bits(INPUT, width_in)
    val out  = Bits(OUTPUT, width_in)
}
class Ripple_Add(width_in :Int = 4) extends Component {
  val io = new Ripple_AddIO(width_in)
  val c = Vec(width_in){Bits(width =1)}
  val sum = Vec(width_in){Bits(width =1)}

  c(0) :=  io.in1(0) && io.in2(0)
  for(k<-1 to width_in-1) {
    c(k) := (io.in1(k-1) && io.in2(k-1)) ^ (io.in1(k-1)
        && c(k-1)) ^ (io.in2(k-1) && c(k-1))
  }
  sum(0) :=  io.in1(0) ^ io.in2(0)
  for(k<-1 to width_in-1) {
    sum(k) :=  io.in1(k) ^ io.in2(k) ^ c(k)
  }
  io.out :=sum.toBits
}
class Ripple_Add_Q(width_in :Int = 4) extends Component {
  val io = new Ripple_AddIO(width_in)
  val c = Vec(width_in){Bits(width =1)}
  val c_p = Vec(width_in){Bits(width =1)}
  val sum = Vec(width_in){Bits(width =1)}

  c(0) :=  io.in1(0) && io.in2(0)
  for(k<-1 to width_in-1) {
    c(k) := io.in1(k-1) && io.in2(k-1)
    c_p(k) := (c(k) #&& (io.in1(k-1), c(k-1))) #&&
              (io.in2(k-1), c(k-1))
  }
  sum(0) :=  io.in1(0) #^ io.in2(0)
  for(k<-1 to width_in-1) {
    sum(k) :=  io.in1(k) #^ io.in2(k) #^ c_p(k)
  }
  io.out :=sum.toBits
}
```

Fig. 8. Parameterized Classical and Quantum Ripple-carry [16] adder modules. Both modules utilize the same IO bundle (`Ripple_AddIO`). This example illustrates the selective use of quantum operators.

D. Chisel-Q Optimization Approach

Optimization of Chisel-Q output is extremely important, given the cost of implementing quantum circuits (from error correction, scarcity of resources, *etc.*). Chisel-Q performs simple optimizations to reduce the number of generated ancillas, as detailed below. In addition, we assume that the generated QASM will be subsequently fed through synthesis tools [17], physical design tools [18, 19], or other optimization tools since it is in a standard format.

For nodes with a single-level of fan-out (*e.g.* direct assignments or NOT operations), we avoid introducing new ancillas. For nodes with more than one qubit bandwidth and multiple fan-outs, we avoid introducing new ancillas when the qubits from that node are disjointedly connected to other nodes. Further, for quantum operators, we avoid introducing ancillas entirely, leaving it up to the designers to guarantee the correctness of their circuits. After applying the above techniques, we conduct a back-trace of signal names to keep the correctness of the design, since some nodes are reduced and the output signal names should be mapped to their inputs. The above name-tracing process is repeated until no more reduction is possible.

IV. Transforming Circuits with State

Classical circuit designers introduce state for a variety of reasons, including pipelining, reuse of circuit elements, and controlled sequencing. The presence of state complicates translation for at least two reasons: First, sequential circuits may exhibit a data-dependent control structure. Since the *control* of quantum elements is usually classical, data-dependent control is problematic when the data is quantum in nature[7]. Second, classical latches *erase* information at every clock edge, making it impossible to clean ancilla state that depends on previous contents. We tackle both problems in the following sections.

A. Transforming Pipelines

Pipelines present a straightforward application of state. Because QASM treats idle bits as if they are stored in a latch, Chisel-Q can replace pipeline latches with multi-bit identity elements in QASM. The result signals that all bits must be available at the input before firing gates at the output—retaining the ability to overlap multiple computations simultaneously.

B. Removing Data-Dependent Control

When a circuit includes one or more sequential loops, Chisel-Q must remove any data-dependent looping behavior before transforming to the quantum domain. The simplest situation is one in which the number of cycles in the loop is *fixed* or *classically computable*. In this case there is no data-dependent looping, and the designer simply specifies the number of iterations with an **`Iteration_Count_Quantum`** annotation.

A more complex situation occurs when the number of loop iterations is dependent on input data (which will be quantum), but there is still a classically-determined *maximum* iteration count. A simple example would be a multiplication module that stops iterating when it detects that the remaining significant bits are zero. Chisel-Q requires the designer to specify a maximum iteration count with an **`Iteration_Count_Quantum`** annotation and identify a signal that will serve as a completion signal, via a **`Done=signal`** annotation. In this case, Chisel-Q performs a classical transformation as shown in Figure 9.

This transformation adds two new state elements, a *Data Latch* and a *Done Latch*. We assume that the **`Done`** signal will become true at some point in the computation, after which the output data will be latched in the *Data Latch* and stay there—even if the original circuit is iterated beyond the intended number of iterations. We replace the original **`Done`** signal with a classically-derived signal (**`Done'`**), that becomes true after the maximum number of iterations. This new circuit has no data-dependent looping and is now the same as our first case.

Consider what happens when this circuit is transformed to the quantum domain. When the data inputs to the circuit (not shown) contain a superposition of values, then each of these values may cause the original **`Done`** signal to become true after a different number of iterations. After the maximum iteration count, **`Output'`** will contain a superposition of the output values corresponding to the original input superposition[8].

[7] Again, we leave *measurement* out of the scope for now.

[8] Since quantum circuits are *linear*, we can reason about this by separating the superposition of input values into individual binary inputs, trace each such "classical" input through the circuit, then sum the outputs back together using the complex coefficients that appeared on the inputs.

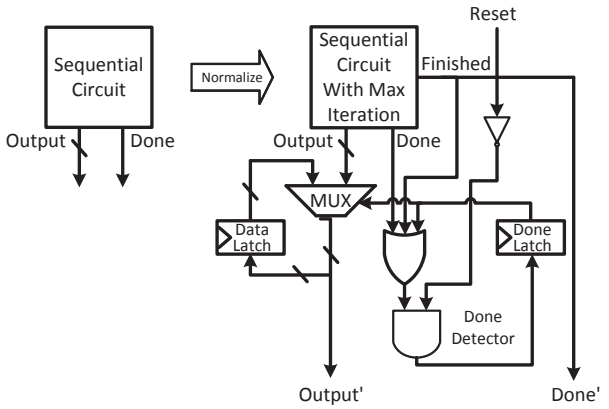

Fig. 9. Transforming sequential circuits with a data-dependent number of iterations into circuits with a fixed (maximum) number of iterations. This normalization process is entirely classical. When eventually transformed to a quantum circuit (see Figure 10), the result is that (1) control is not dependent on a quantum value (`Done'` is classical) and (2) the final output (`Output'`) can hold a superposition of output values. Input bits not shown for simplicity.

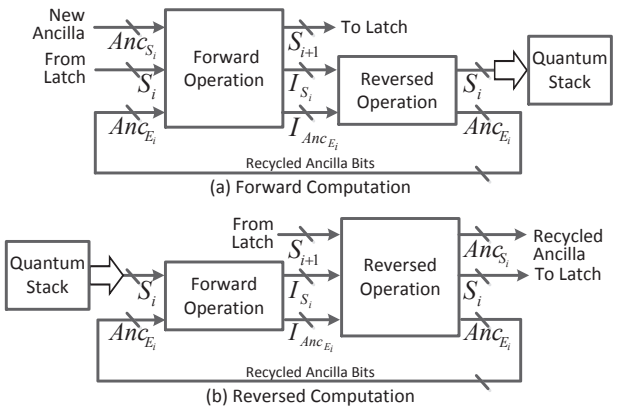

Fig. 10. Transforming classical sequential circuits to quantum sequential circuits while cleaning ancilla bits. The *Forward Computation* runs for a fixed number of iterations (it must be normalized as in Figure 9), while saving the *history* of states (of the "Latch") on the stack. Then, the *Reverse Computation* uses this history to reverse the computation and erase data on the stack. Some ancillas can be recycled each iteration. Input bits not shown for simplicity.

Finally, if the loop has no maximum iteration count or no clear completion signal, then Chisel-Q cannot handle it. Chisel-Q alerts the designer to potential problems by emitting a warning when it detects a sequential loop without annotations.

C. Circuit Reuse and Fixed Iterative Structure

After removing data-dependent looping as described above, we have two options. First, we could eliminate state elements by unrolling the loop. This choice reduces our circuit to a combinational one. While straightforward, unrolling greatly increases the size of the circuit emitted to QASM.

Alternatively, we can retain the structure of the implementation and emit a looping construct to QASM. In this case, the latches represent points in the circuit where state is overwritten. To revert ancilla at the end of the computation, we must retain the *history* of data stored the latches. Figure 10 illustrates how to utilize a *quantum stack* for this purpose. The quantum stack stores quantum state in *last-in-first-out* (LIFO) order and may be implemented with more primitive elements[9].

In Figure 10(a), each state transition, $S_i \Rightarrow S_{i+1}$, saves S_i on the stack for later use. The no-cloning theorem prevents us from sending S_i to both the *Forward Operation* block and the stack; instead, we reconstruct S_i after computing S_{i+1} with the *Reversed Operation* block. Some ancillas (Anc_{E_i}) are restored and recycled. After completion of the *Forward Computation*, the *Reversed Computation* (Figure 10(b)) runs the state machine backward ($S_{i+1} \Rightarrow S_i$) to erase data stored on the stack[10]. Figure 10 suggests a space/time tradeoff: instead of reconstructing S_i and Anc_{E_i} with each iteration, we *could* simply push and pop intermediate results on the stack (*i.e.* I_{S_i} and $I_{Anc_{E_i}}$). This alternative is twice as fast, at the cost of a large increase in ancilla consumption and stack space.

D. Circuits with Memory

When memories are *read-only* from the standpoint of the quantum datapath (*e.g.* constant or written by classical portions of the circuit), then the quantum lookup can be implemented by using bits of the address to drive a tree of MUXes. The result nicely handles an address that is a superposition of values.

On the other hand, when memories are *written* by the quantum data path, we must be much more careful. Whenever we read from such a memory, the no-cloning theorem forces us to treat it as a destructive read and reconstruct the value after using it (similar to an ancilla). During a write, if the *address* and *write-enable* signal are classical, we can push the previous value and its address on the stack for a later erasure step. However, if the *write-enable* signal is quantum, we introduce data-dependent control of the stack. Even worse, the meaning of a quantum *address* during a write is not at all clear. We leave the handling of these later two situations for future work.

V. Experimental Results

In this section, we examine resource and performance statistics for a variety of generated circuits and compare with hand-tuned versions. We also transform circuits for a RISC processor—circuits designed by classical circuit designers.

A. Chisel-Q Resource & Performance Evaluation

Chisel-Q produces resource and performance statistics during compilation. It scans the generated QASM to obtain a count of qubits and various quantum gates. For hierarchical designs, the cost of submodules is mapped to the calling module. It also estimates parallelism (minimum, maximum, and average) and latency for a design by using a breadth-first search on the dataflow graph in combination with knowledge of the parallelism and latency for each type of operation node.

B. Mathematical Oracles for Shor's Factoring

Table III shows resource estimation for 32-bit versions of components of Shor's factoring algorithm [3]. Chisel-Q code for these circuits is described in the Appendix. As shown by these results, our current optimization solution (Section III-D) specifically targets consumption of new ancillas and CNOT gates (by reducing their use for intermediate results). On average, our technique reduces 36.0% ancilla qubits and 35.2% CNOT gates for all the designs in Table III.

[9]For some quantum computing technologies with ballistic movement (*e.g.* Ion Traps) [5], this structure may have a very efficient physical implementation.

[10]Until erasure is complete, data on the stack is entangled with the result.

Circuit	Before Opt.				After Opt.			
	# of ancilla qubits	# of Toffoli	# of CNOT	# of X	# of ancilla qubits	# of Toffoli	# of CNOT	# of X
Adder	1032	188	2094	0	778	188	1586	0
Adder-Q	1001	188	2032	0	32	188	126	0
Mul_WT	17764	6582	37478	124	11101	6582	24152	124
Mul_Booth (Seq)	3704	4860	3811	4428	3598	4860	3387	4428
Exp_MulWT	572411	229018	1174488	36994	365826	229018	761318	36994
Shors_ExpMulWT	573192	229018	1176050	36994	366417	229018	762500	36994

TABLE III. RESOURCE ESTIMATION OF QUANTUM DESIGNS.

For the adder described with quantum operators, however (See "Adder-Q" in Row 2 of Table III), our solution reduces up to 96.8% ancilla qubits and 93.8% CNOT gates. The original circuit generated by Chisel-Q included a set of expensive concatenation operators that were avoided in hand-written quantum designs, and we enhanced our optimization techniques to reduce the above structure. In the end, our generated Adder-Q has the same resource cost as the hand-written design by [16], demonstrating the effectiveness of Chisel-Q.

Although our optimization heuristics do not currently reduce other quantum gates, such as X and Toffoli gates, it is important to remember that Chisel-Q facilitates the transformation of quantum circuits with high levels of abstraction into a *standard* gate-level netlist format (QASM). The result can be fed into other quantum development tools for further optimization.

Table IV shows circuit latency and parallelism for the circuits from Table III. We observe that the Wallace-tree multiplier (Denoted by "Mul_WT" in Row 3) provides significant parallelism: on average 46.4 operations can be conducted concurrently, and the maximum value is up to 2048. Further, its latency is within a factor of 3 of addition. The Booth multiplier ("Mul_Booth", Row 4) is iterative, so exhibits high latency but utilizes only 32.4% ancillas compared to "Mul_WT." Finally, we see that Chisel-Q preserves the parallelism of "Mul_WT" for calling modules: As shown by Rows 5–6, by constructing from this multiplier, the exponentiation module and Shor's factorization module easily preserve this high parallelism.

C. Mapping of a Classical RISC Processor

Table V, shows the results of compiling elements of a RISC processor developed in Chisel. These components were developed by classical circuit designers without any quantum knowledge. Without no additional design effort, we can generate quantum versions of an ALU, several arbiters, the flush unit, FPU decoder and FPU comparator. More optimization is clearly needed, but the important point is that existing well developed classical circuits can be easily converted to cost-effective quantum ones, meeting one of the primary goals of this work.

VI. CONCLUSION

We introduced Chisel-Q, a high-level quantum circuit design language that permits quantum oracles to be constructed by classical circuit designers using the meta-language features of Scala and its embedded DSL "Chisel". Sophisticated designers can incorporate quantum operators in select portions of the circuit for additional control over the synthesized output. We discussed how Chisel-Q translates both combinational and stateful circuits, as well as optimization techniques to increase the quality of the synthesized output. For future work, we plan to extend Chisel-Q to a full-blown language for constructing quantum-computing algorithms, as well as introducing additional optimization heuristics to better match the quality of quantum circuits produced by human designers.

Circuit	Latency	Parallelism		
		Min	Max	Ave
Adder	448	1	190	4.9
Adder-Q	268	1	32	2.2
Mul_WT	756	1	2048	46.4
Mul_Booth (Seq)	39680	1	236	10.4
Exp_MulWT	23543	1	3968	48.9
Shors_ExpMulWT	23792	1	3968	48.4

TABLE IV. PERFORMANCE EVALUATION OF QUANTUM DESIGNS.

Component	# of ancilla qubits	# of Toffoli	# of CNOT	# of X
ALU	27785	38492	15528	54056
Arbiter	132	95	35	162
Mem. Arbiter	1032	390	1714	488
Locking Arbiter	6856	10800	2776	14626
Flush Unit	357	638	546	474
FPU Decoder	9364	25948	21152	8226
FPU Comparator	271	1100	1037	329

TABLE V. RESOURCE ESTIMATION OF QUANTUM COMPONENTS IN RISC PROCESSOR.

ACKNOWLEDGEMENT

This work was supported by the IARPA QCS program (document numbers D11PC20165 and D11PC20167). 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.

APPENDIX

In this appendix, we present some of the Chisel-Q circuits evaluated in Section V. These examples are inspired by Shor's factorization as introduced in [3] and implemented in [8]. All designs shown here are parameterized. Consequently, we can obtain large scale quantum designs by setting the input bandwidth variable (e.g., **`width_in`** in Figure 13).

Figure 8, shown earlier, includes two implementations of ripple-carry adders designed both classically and with quantum annotations. In the latter case, quantum gate operators were used to tune the circuit as in Draper [16]. Figure 11 shows a Booth multiplier. To convert the classical design into quantum circuit, mapping of abstracted operators (e.g., summation "+" and equal "===") are utilized for this design. Since this is an iterated structure, an **`Iteration_Count_Quantum`** annotation is given. Figure 12 shows a Quantum Fourier transform (QFT), described in a purely quantum manner: only Hadamard gates and C-phase gates are used. Since all the qubits in this module carry result information, there is no need to generate a reversed circuit and we use annotation **`IsReversed = false`**. Finally, Figure 13 utilizes a few lines of code to construct a complete Shor's factorization circuit from these modules.

```
class Mul_IO(width_in: Int) extends Bundle {
  val a     = Bits(INPUT,  width_in)
  val b     = Bits(INPUT,  width_in)
  val prod  = Bits(OUTPUT, 2*width_in)
  val start = Bits(INPUT,  1)
  val done  = Bits(OUTPUT,  1)
}
class Mul_Booth (mulwidth :Int = 4)  extends Component {
  val io  = new Mul_IO(mulwidth)
  val A = Reg(){ Bits(width = mulwidth) }
  val Q = Reg(){ Bits(width = mulwidth) }
  val Q_1 = Reg(Bits(width = 1) )
  val Count = Reg(resetVal = UFix(0, log2Up(mulwidth)))
  val sum = A.toUFix + io.a.toUFix
  val difference = A.toUFix - io.a.toUFix
  Iteration_Count_Quantum = mulwidth

  when (io.start === Bits(1)) {
    Q := io.b
    A := Bits(0)
    Q_1 := Bits(0)
    Count := UFix(0)
  }
  when (io.start === Bits(0)) {
    Count := Count + UFix(1)
    when (Q(0) === Bits(0) && Q_1 === Bits(1)) {
      Q_1 := Q(0)
      Q := Cat(sum(0),Q(mulwidth-1,1))
      A := Cat(sum(mulwidth-1),sum)
    }
    when (Q(0) === Bits(1) && Q_1 === Bits(0)) {
      Q_1 := Q(0)
      Q := Cat(difference(0),Q(mulwidth-1,1))
      A := Cat(difference(mulwidth-1),difference)
    }
    when ((Q(0) === Bits(0) && Q_1 === Bits(0)) ||
          (Q(0) === Bits(1) && Q_1 === Bits(1))) {
      Q_1 := Q(0)
      Q := Cat(A(0),Q(mulwidth-1,1))
      A := Cat(A(mulwidth-1),A)
    }
  }
  io.done := Count >= UFix(mulwidth)
  io.prod := Cat(A,Q)
}
```

Fig. 11. Parameterized Multiplier with Booth's Algorithm. This version of the multiplier is a sequential circuit with a fixed iteration count.

```
class QFT_IO(width_in: Int) extends Bundle {
  val in  = Bits(INPUT, width_in)
  val out = Bits(OUTPUT, width_in)
}
class QFT(width_in :Int = 4 ) extends Component {
  val io = new QFT_IO(width_in)
  val dummy_p = Vec(width_in){Bits(width =1)}
  IsReversed = false

  dummy_p(0) := H(io.in(width_in-1))
  for(k<-1 to width_in-1) {
    val dummy = Vec(k+1){Bits(width =1)}
    dummy(0) := io.in(width_in-1-k)
    for(i<-0 to k-1)
      dummy(i+1) := dummy(i) #@ dummy_p(i) angle(1<<k-i)
    dummy_p(k) := H(dummy(k))
  }
  io.out :=dummy_p.toBits
}
```

Fig. 12. Parameterized Quantum Fourier Transform Module.

```
class Shors_IO(width_in: Int) extends Bundle {
    val in  = Bits(INPUT,  width_in)
    val out = Bits(OUTPUT, width_in)
}
class Shors_ExpMulWT(width_in :Int = 4) extends
     Component {
    val io = new Shor_IO(width_in)
    val exp = new Exp_MulWT(width_in)
    val qft = new QFT(width_in)
    val c = new Bits()

    exp.io.in := io.in
    c := exp.io.out.toBits
    qft.io.in := c
    io.out := qft.io.out
}
```

Fig. 13. Parameterized Factorization Module for Shor's Algorithm. This version uses exponentiation module including Wallace-tree multiplier, Mul_WT.

References

[1] T. S. Metodi, A. I. Faruque, and F. T. Chong. In *Quantum Computing for Computer Architects*. Morgan & Claypool, 2011.

[2] M. A. Nielsen and I. L. Chuang. In *Quantum Computation and Quantum Information*. Cambridge University Press, 2011.

[3] P.W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev.*, 52(3), 1995.

[4] C. Zalka. Simulating quantum systems on a quantum computer. *Phys. Rev.*, 454(1969):313–322, 1998.

[5] D. Leibfried et al. Experimental demonstration of a robust, high-fidelity geometric two ion-qubit phase gate. *Nature*, 422(6930):412–415, 2003.

[6] J. M. Taylor et al. Relaxation, dephasing, and quantum control of electron spins in double quantum dots. *Phys. Rev.*, 76, 2007.

[7] Tzvetan S. Metodi et al. A Quantum Logic Array Microarchitecture: Scalable Quantum Data Movement and Computation. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2005.

[8] M. G. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz. A Fault Tolerant, Area Efficient Architecture for Shor's Factoring Algorithm. In *Proceedings of the International Symposium on Computer Architecture*, pages 383–394, 2009.

[9] A. M. Childs et al. Exponential algorithmic speedup by a quantum walk. In *Proceedings Annual ACM Symposium on Theory of Computing*, pages 59–68, 2003.

[10] Dominic Berry, Graeme Ahokas, Richard Cleve, and Barry Sanders. Efficient quantum algorithms for simulating sparse hamiltonians. *Communications in Mathematical Physics*, 270:359–371, 2007.

[11] S. Balensiefer, L. Kregor-Stickles, and M. Oskin. An Evaluation Framework and Instruction Set Architecture for Ion-Trap based Quantum Microarchitectures. In *Proceedings international symposium on Computer Architecture*, pages 186–196, 2005.

[12] Z. Navabi. *Verilog Digital System Design (Professional Engineering)*. McGraw-Hill Inc., 1999.

[13] J. Bachrach et al. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 1212–1221, 2012.

[14] W.K. Wootters and W.H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, 1982.

[15] M. Odersky. Scala Programming Language. http://www.scala-lang.org/.

[16] T.G. Draper. Addition on a Quantum Computer. In *Arxiv preprint quant-ph/0008033*, 2000.

[17] V.V. Shende, S.S. Bullock, and I.L. Markov. Synthesis of quantum-logic circuits. 25(6):1000–1010, 2006.

[18] D. Maslov, S. M. Falconer, and M. Mosca. Quantum Circuit Placement. 27(4):752–763, 2008.

[19] M. J. Dousti and M. Pedram. Minimizing the Latency of Quantum Circuits during Mapping to the Ion-Trap Circuit Fabric. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 840–843, 2012.