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.

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

---

\(^1\)The quantum equivalents of classical gates are unitary operators which must be reversible by definition.
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, quantum circuit designers strive to enhance these effects.

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 $\psi = \alpha |0\rangle + \beta |1\rangle$, where $\alpha$ and $\beta$ 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 either 0 or 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. 2. Examples of Quantum Gates. Horizontal lines are individual qubits.

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 of 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 $c \oplus (a \land b)$, 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).
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.
Quantum Gate | Classical Gate
---|---
Toffoli | AND
X | OR
CNOT | NOT

TABLE I. MAPPING BETWEEN CLASSICAL GATES & QUANTUM GATES.

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\(_{\text{out}}\). The output value is implemented by transforming an input ancilla (here labeled Anc\(_{\text{In1}}\)) 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\(_{\text{In1}}\), Anc\(_{\text{In2}}\), and Anc\(_{\text{In3}}\).

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() modifier, designers can specify any rational fraction of \(\pi\). 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 \texttt{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_Add(width_in :Int = 4) extends Component {
    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))
    }
    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_AddIO(width_n: Int) extends Bundle {
    val in1 = Vec(width_n){Bits(width =1)}
    val in2 = Vec(width_n){Bits(width =1)}
    val sum = Vec(width_in){Bits(width =1)}
}

for(k<-1 to width_in-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 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))
        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
}

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. 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.

*Again, we leave measurement out of the scope for now.
*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.
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 on 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.

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 (\( \text{Anc}_{E_i} \)) 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 \( \text{Anc}_{E_i} \) with each iteration, we could simply push and pop intermediate results on the stack (i.e. \( I_{S_i} \) and \( I_{\text{Anc}_{E_i}} \)). 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)

---

\(^9\)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.
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.

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.

<table>
<thead>
<tr>
<th>Circuit</th>
<th># ancilla qubits</th>
<th># CNOT</th>
<th># X</th>
<th># CNOT</th>
<th># X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>1032</td>
<td>188</td>
<td>2094</td>
<td>0</td>
<td>778</td>
</tr>
<tr>
<td>Adder-Q</td>
<td>1001</td>
<td>188</td>
<td>2032</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Mul_WT</td>
<td>1758</td>
<td>6582</td>
<td>37478</td>
<td>124</td>
<td>11101</td>
</tr>
<tr>
<td>Mul_Booth</td>
<td>3704</td>
<td>4860</td>
<td>3811</td>
<td>4428</td>
<td>3598</td>
</tr>
<tr>
<td>Mul_Booth (Seq)</td>
<td>3968</td>
<td>23792</td>
<td>1586</td>
<td>0</td>
<td>1714</td>
</tr>
<tr>
<td>Exp_MulWT</td>
<td>573211</td>
<td>229018</td>
<td>1174488</td>
<td>36994</td>
<td>365826</td>
</tr>
<tr>
<td>Shors_ExpMulWT</td>
<td>573192</td>
<td>229018</td>
<td>1176050</td>
<td>36994</td>
<td>366417</td>
</tr>
</tbody>
</table>

| Table III. RESOURCE ESTIMATION OF QUANTUM DESIGNS. |

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Latency</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Adder</td>
<td>448</td>
<td>1</td>
</tr>
<tr>
<td>Adder-Q</td>
<td>268</td>
<td>1</td>
</tr>
<tr>
<td>Mul_WT</td>
<td>756</td>
<td>1</td>
</tr>
<tr>
<td>Mul_Booth</td>
<td>39680</td>
<td>1</td>
</tr>
<tr>
<td>Exp_MulWT</td>
<td>23541</td>
<td>1</td>
</tr>
<tr>
<td>Shors_ExpMulWT</td>
<td>23792</td>
<td>1</td>
</tr>
</tbody>
</table>

| Table IV. PERFORMANCE EVALUATION OF QUANTUM DESIGNS. |

<table>
<thead>
<tr>
<th>Component</th>
<th># ancilla qubits</th>
<th># CNOT</th>
<th># X</th>
<th># CNOT</th>
<th># X</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>27758</td>
<td>38492</td>
<td>15528</td>
<td>54056</td>
<td></td>
</tr>
<tr>
<td>Arbiter</td>
<td>132</td>
<td>95</td>
<td>35</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>Mem. Arbiter</td>
<td>1032</td>
<td>390</td>
<td>1714</td>
<td>488</td>
<td></td>
</tr>
<tr>
<td>Locking Arbiter</td>
<td>6856</td>
<td>10800</td>
<td>2776</td>
<td>14626</td>
<td></td>
</tr>
<tr>
<td>Flush Unit</td>
<td>357</td>
<td>638</td>
<td>546</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>FPU Decoder</td>
<td>9364</td>
<td>25948</td>
<td>21152</td>
<td>8226</td>
<td></td>
</tr>
<tr>
<td>FPU Comparator</td>
<td>271</td>
<td>1100</td>
<td>1037</td>
<td>329</td>
<td></td>
</tr>
</tbody>
</table>

| Table V. RESOURCE ESTIMATION OF QUANTUM COMPONENTS IN RISC PROCESSOR. |
class Mul_Booth (mulwidth :Int = 4) extends Component {
val A = Reg(){ Bits(width = mulwidth) }
val Q = Reg(){ Bits(width = mulwidth) }
val Q_1 = Reg(){ Bits(width = 1) }
val Count = Reg(){ UFix(0, log2Up(mulwidth)) }
val sum = A.toUFix + io.a.toUFix
val difference = A.toUFix - io.a.toUFix
val IsReversed = false
}

class Mul_IO(width_in: Int) extends Bundle {
val io = new Mul_IO(mulwidth)
val Q := io.b
val A := io.a
val Q_1 := Q(0)
val A := Cat(A(0),Q(mulwidth-1,1))
val Q := Cat(difference(0),Q(mulwidth-1,1))
val A := Cat(sum(0),sum)
val Count := Count + UFix(1)
}

class Mul_Booth (mulwidth :Int = 4) extends Component {
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 Shors_ExpMulWT(width_in :Int = 4) extends 
    Component {
val io = new Shors_ExpMulWT(width_in)
val dummy_p = Vec(width_in){Bits(width =1)}
val dummy_p(0) := H(io.in(width_in-1))
for(k<-1 to width_in-1) {
    dummy_p(k) := dummy_p(k) @ dummy_p(0) angle(1<<k)
}
}

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 Shors_ExpMulWT(width_in)
val qft = new QFT(width_in)
val exp = new Exp_MulWT(width_in)
c := exp.io.out.toBits
}

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(width_in)
val dummy_p = Vec(width_in){Bits(width =1)}
val dummy_p(0) := H(io.in(width_in-1))
for(k<-1 to width_in-1) {
    dummy_p(k) := dummy_p(k) @ dummy_p(0) angle(1<<k)
}
}

class QFT(IO(width_in: Int) extends Bundle {
val in = Bits(INPUT, width_in)
val out = Bits(OUTPUT, width_in)
}

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

Fig. 12. Parameterized Quantum Fourier Transform Module.

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

REFERENCES
    Scalable Quantum Data Movement and Computation. In Proceedings 
    of International Symposium on Microarchitecture (MICRO), 2005.
    Area Efficient Architecture for Shor’s Factoring Algorithm. In Proceedings 
[10] Dominic Berry, Graeme Ahokas, Richard Cleve, and Barry Sanders. 