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

Motivation

• Why Quantum Computers?
  - Great potential to speed up certain computations, such as factorization and quantum mechanical simulation
  - Fascinating exploration of physics

• Slow but constant research progress
  - New technologies, Computer Architectures, Algorithms
  - Still cannot quite build a large quantum computer

• Unfortunately, techniques for expressing quantum algorithms are limited:
  - High-level mathematical expressions
  - Low-level sequences of quantum gates

• Let’s see if we can find a better form of expression!

Structure of Quantum Algorithms

• Quantum Algorithms contain two pieces:
  - Enclosing Algorithm
    • Quantum measurement, control structures, I/O
  - Quantum “Oracle” (black-box function of quantum state)
    • Often specified as classical function, but must handle inputs/outputs that are superpositions of values

• Much of the implementation complexity in the Oracles

Example: Shor’s Algorithm

• Oracles: operate on 2048 or 4096-bit values
  - Modular exponentiation (with embedded operations)
  - Quantum Fourier Transform (QFT)
**Compilation Target**: The Quantum Circuit Model

- Quantum Circuit model - graphical representation of quantum computing algorithm
  - Time Flows from left to right
  - Single Wires: persistent qubits
  - Double Wires: classical bits
  - Measurement: turns quantum state into classical state
- Quantum gates typically operate on one or two qubits
  - Universal gate set: Sufficient to form all unitary transformations

**How to express Quantum Circuits/Algorithms?**

- Graphically: Schematic Capture Systems
  - Several of these have been built
- QASM: the quantum assembly language
  - Primitives for defining single Qubits, Gates
- C-like languages
  - Scaffold: some abstraction, modules, fixed loops
- Embedded languages
  - Use languages such as Scala or Ruby to build Domain Specific Language (DSL) for quantum circuits
  - Can build up circuit by overriding basic operators
  - Backend generator can add ancilla bits and erasure of information at end of computation for reversibility

**Starting Point: Berkeley Chisel**

- Scala-based language for digital circuit design
  - High-level functional descriptions of circuits as input
  - Many backends: for instance direct production on Verilog
  - Used in design of new advanced RISC pipeline
- Features
  - High-level abstraction:
    - higher order functions, advanced libraries, flexible syntax
  - Abstractions build up circuit (netlist)
    - E.g.: Inner-Product FIR Digital Filter: \[ y[t] = \sum_j w_j x[t-j] \]

**Quantum Version: Berkeley Chisel-Q in Nutshell**

- Augmented Chisel Syntax, New Backend
  - Generate reversible versions of classical circuits
  - Classical \(\Rightarrow\) Quantum translation:
    - Map classical gates to quantum gates
    - Add ancilla bits when necessary for reversibility
    - Erase ancilla state at end (decouple ancilla from answer)
    - State machine transformations
  - Supplemental quantum syntax for tuning output
- Output: Quantum Assembly (QASM)
  - Input to other tools!
- Goal: Take classical circuits designed in Chisel and produce quantum equivalents
  - Adders, Multipliers
  - Floating-Point processors
**Chisel-Q Design Flow**

- Chisel-Q piggybacks on basic Chisel design flow
  - Maintains basic parsing infrastructure
  - Internal dataflow format
  - Output hooks for generating simulators/HDL (e.g. Verilog)
- **Chisel-Q additions:**
  - Quantum/Classical Signal Type Analysis
  - Parsing of Quantum Operators
  - Reversible Circuit Generation, Ancilla Erasure
  - State Machine analysis
- **Output:** QASM and statistics about the resulting circuit
  - Gate count, level of parallelism, predicted latency

---

**Signal Type Analysis**

- Annotations and dataflow analysis to distinguish classical and quantum signals
  - Use “isQuantum” annotation on inputs or outputs to indicate quantum datapath
  - Quantum annotations traced through rest of datapath
    - Traced through design hierarchy, sequential loops, ...
    - Combined quantum and classical signal $\Rightarrow$ quantum signal
  - Classical signals automatically upgraded to quantum
- **Advantages**
  - Combine classical control and quantum datapath in same design
  - Classical designs easily transformed to quantum designs simply by annotating enclosing module (subject to some restrictions)

---

**Easy Case: Combination Circuits (Ancilla Insertion and Reversal)**

- **Gate Level operator mapping:**
  - Simple, one for one substitution
  - Addition of ancilla as necessary
- **Reversed circuit generation**
  - Leveling the nodes in the dataflow graph
  - Output the nodes in a reversed order
  - In reversed circuit, each node is replaced by the reversed operation from original one
    - Gets tricky only with rotation operators
- **Example:** transformation of carry circuit

---

**Chisel-Q Optimization Approach**

- Optimization on a case-by-case basis:
  - More like "peep-hole optimization" than "logical optimization"
- **Some examples** which get a lot of mileage:
  - For nodes with single-level of fan-out (e.g. direct assignments or NOT operations), avoid introducing new ancillas
  - For nodes with more than one qubit bandwidth and multiple fanouts, we avoid introducing new ancillas when the qubits from that node are disjointedly connected to other nodes
  - For quantum operators, we avoid introducing ancillas
- **Lots of room for improvement!**
  - Assume that Chisel-Q output will feed into QASM-compatible optimization toolset
**Easy Case: Pipeline (Acyclic Dataflow Graph with State)**

- When left alone, qubits act like registers
  - Except for fact that state decays if left too long
- Classical circuit with pipelined structure
  - With registers
  - Without loops (acyclic dataflow graph)
- Easy to identify/handle this type of structure
  - Pipeline registers replaced by multi-input identity elements for synchronization
  - Transformation is similar to combinational circuit
    - Gate mapping, ancilla additions, reversal, ...

**Hard Case: State Machine (Sequential Circuits with State)**

- Sequential loops: Very important
  - Widely used by classical designers
  - Includes: state machines, iterative computations, ...
- Sequential circuits are challenging:
  1. Quantum circuits achieved via classical control \( \Rightarrow \) cannot handle iteration count based on quantum info
  2. Cannot erase state information: must restore ancilla at end of computation
- Two options for Chisel-Q
  - Only handle easy cases: Combinational Logic, Pipelines
  - Try to handle at least some sequential circuits

**Handling Looped Structure**

- Fixed (classically computable) iteration count
  - No data-dependent looping!
  - Use *Iteration_Count_Quantum* annotation
- Specified Quantum Completion Signal:
  - Transformation into Fixed iteration count (first case)
  - Use *Iteration_Count_Quantum* and *Done* annotations
  - Cannot currently handle unspecified (and/or unbounded) termination condition!

**Reversing Ancilla State with Fixed Iterative Structure**

- Save state values before they are overwritten, then use to erase state at end of computation
  - “Quantum Stack”: LIFO physical structure for holding qubits
  - Natural implementation in, e.g. Ion-trap quantum computer
- Transformation discussed in paper:
Designing with quantum operators

Syntax of Quantum Gates in Chisel-Q

<table>
<thead>
<tr>
<th>Quantum Gate</th>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toffoli</td>
<td>X(a, b)</td>
<td>c := X(a)</td>
</tr>
<tr>
<td>CNOT</td>
<td>X(c, d)</td>
<td>c := a # b</td>
</tr>
<tr>
<td>Pauli-X</td>
<td>Y(a)</td>
<td>c := Y(a)</td>
</tr>
<tr>
<td>Pauli-Y</td>
<td>Z(a)</td>
<td>c := Z(a)</td>
</tr>
<tr>
<td>Hadamard</td>
<td>H(a)</td>
<td>c := H(a)</td>
</tr>
<tr>
<td>Phase</td>
<td>P(λ)</td>
<td>c := P(λ)</td>
</tr>
<tr>
<td>C-phase</td>
<td>λ(a)</td>
<td>c := a # b</td>
</tr>
<tr>
<td></td>
<td>λ#...angle</td>
<td>c := λ#...angle(n,d)</td>
</tr>
</tbody>
</table>

- Native syntax for quantum circuit design
  - Insert “just enough” quantum knowledge to improve generated results
  - Specify a complete quantum circuit without intervention from Chisel-Q backend
  - Annotation \( \text{IsReversed} = \text{false} \) to block the generation of reversed circuit

Parameterized Quantum Fourier Transform (QFT) Module

- Annotation \( \text{IsReversed} = \text{false} \) to block the generation of reversed circuit

Mathematical Benchmarks

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder_Ripple</td>
<td>Ripple-carry adder designed in classical way.</td>
</tr>
<tr>
<td>Adder_Ripple_Q</td>
<td>Ripple-carry adder designed with quantum gate operators were used; Using designer intuition to recognize very specific quantum operators.</td>
</tr>
<tr>
<td>Adder_CLA</td>
<td>Carry-lookahead adder designed in classical way.</td>
</tr>
<tr>
<td>Mul_Booth</td>
<td>Multiplier using Booth's algorithm designed in classical way; Quantum annotation are used to describe the iterative operation.</td>
</tr>
<tr>
<td>Mul_WT</td>
<td>Multiplier using Wallace tree structure.</td>
</tr>
<tr>
<td>Exp_Mul_Booth</td>
<td>Exponentiation module with multipliers using Booth's algorithm.</td>
</tr>
<tr>
<td>Exp_Mul_WT</td>
<td>Exponentiation module with multipliers using Wallace tree structure.</td>
</tr>
<tr>
<td>QFT</td>
<td>Quantum Fourier transform module described in a purely quantum manner; Annotation IsReversed is used to avoid reversed circuit generation.</td>
</tr>
<tr>
<td>Shor_Exp_Mul</td>
<td>Factorization module with Shor’s algorithm; including submodule Exp_Mul_WT and QFT.</td>
</tr>
</tbody>
</table>

Resource Estimation for Simple Benchmarks

- Parse the generated QASM
- Count the required resource
  - Ancilla qubit, different gates, ...

<table>
<thead>
<tr>
<th>Circuit</th>
<th># of Ancilla Qubits</th>
<th># of Toffoli</th>
<th># of CNOT</th>
<th># of X</th>
<th># of Ancilla Qubits</th>
<th># of Toffoli</th>
<th># of CNOT</th>
<th># of X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>1032</td>
<td>188</td>
<td>2094</td>
<td>0</td>
<td>778</td>
<td>188</td>
<td>1586</td>
<td>0</td>
</tr>
<tr>
<td>Adder_Q</td>
<td>1001</td>
<td>188</td>
<td>2032</td>
<td>0</td>
<td>32</td>
<td>188</td>
<td>126</td>
<td>0</td>
</tr>
<tr>
<td>Mul_WT</td>
<td>11764</td>
<td>6582</td>
<td>37478</td>
<td>124</td>
<td>11101</td>
<td>6582</td>
<td>24152</td>
<td>124</td>
</tr>
<tr>
<td>Mul_Booth</td>
<td>33704</td>
<td>4860</td>
<td>3811</td>
<td>4428</td>
<td>3598</td>
<td>4860</td>
<td>3387</td>
<td>4428</td>
</tr>
<tr>
<td>Exp_MulWT</td>
<td>572411</td>
<td>229018</td>
<td>175488</td>
<td>36994</td>
<td>365826</td>
<td>229018</td>
<td>76918</td>
<td>36994</td>
</tr>
<tr>
<td>Shors_ExpMulWT</td>
<td>573992</td>
<td>229018</td>
<td>175488</td>
<td>36994</td>
<td>365417</td>
<td>229018</td>
<td>76918</td>
<td>36994</td>
</tr>
</tbody>
</table>
Performance Evaluation for Simple Benchmarks

- Parallelism: How many quantum operations can be conducted concurrently?
- Latency: How many steps are required to complete all the operations?

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Latency</th>
<th>Parallelism Min</th>
<th>Parallelism Max</th>
<th>Parallelism Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>448</td>
<td>1</td>
<td>190</td>
<td>4.9</td>
</tr>
<tr>
<td>Adder-Q</td>
<td>268</td>
<td>1</td>
<td>32</td>
<td>2.2</td>
</tr>
<tr>
<td>Mul_WT</td>
<td>756</td>
<td>1</td>
<td>2048</td>
<td>46.4</td>
</tr>
<tr>
<td>Mul_Booth (Seq)</td>
<td>39680</td>
<td>1</td>
<td>236</td>
<td>10.4</td>
</tr>
<tr>
<td>Exp_MulWT</td>
<td>23543</td>
<td>1</td>
<td>3968</td>
<td>48.9</td>
</tr>
<tr>
<td>Shors_ExpMulWT</td>
<td>23792</td>
<td>1</td>
<td>3968</td>
<td>48.4</td>
</tr>
</tbody>
</table>

Resource Estimation of Components of a RISC Processor

<table>
<thead>
<tr>
<th>Component</th>
<th># of Ancilla Qubits</th>
<th># of Toffoli</th>
<th># of CNOT</th>
<th># of X</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>27785</td>
<td>38492</td>
<td>15528</td>
<td>54056</td>
</tr>
<tr>
<td>Arbiter</td>
<td>132</td>
<td>95</td>
<td>35</td>
<td>162</td>
</tr>
<tr>
<td>Mem. Arbiter</td>
<td>1032</td>
<td>390</td>
<td>1714</td>
<td>488</td>
</tr>
<tr>
<td>Locking Arbiter</td>
<td>6856</td>
<td>10800</td>
<td>2776</td>
<td>14626</td>
</tr>
<tr>
<td>Flush Unit</td>
<td>357</td>
<td>638</td>
<td>546</td>
<td>474</td>
</tr>
<tr>
<td>FPU Decoder</td>
<td>9364</td>
<td>25948</td>
<td>21152</td>
<td>8226</td>
</tr>
<tr>
<td>FPU Comparator</td>
<td>271</td>
<td>1100</td>
<td>1037</td>
<td>329</td>
</tr>
</tbody>
</table>

Conclusion

- **Chisel-Q**: a high-level quantum circuit design language
  - Powerful Embedded DSL in Scala
  - Classical circuit designers can construct quantum oracles
- Translation of combinational logic straightforward
  - Direct substitution of operations and introduction of ancilla bits
  - Generation of reversed circuits to restore ancilla after end of computation
- Sequential circuits more challenging
  - Must identify maximum number of iterations and completion signals
  - Must save state for later use in restoring ancilla (and erasing information).
- For future work, we plan to
  - Extend Chisel-Q to a full-blown language for constructing quantum-computing algorithms
  - Additional optimization heuristics

Extra Slides
Quantum Bits (Qubits)

- Qubits can be in a combination of "1" and "0":
  - Written as: \( \Psi = C_0|0> + C_1|1> \)
  - The \( C \)'s are complex numbers!
  - Important Constraint: \( |C_0|^2 + |C_1|^2 = 1 \)
- If measure bit to see what looks like,
  - With probability \( |C_0|^2 \) we will find \(|0>\) (say "UP")
  - With probability \( |C_1|^2 \) we will find \(|1>\) (say "DOWN")
- An \( n \)-qubit register can have \( 2^n \) values simultaneously!
  - 3-bit example:
    \[
    \Psi = C_{000}|000> + C_{001}|001> + C_{010}|010> + C_{011}|011> + C_{100}|100> + C_{101}|101> + C_{110}|110> + C_{111}|111>
    \]

Parameterized Ripple-carry adder

- Ripple-carry adder designed in classically and with quantum additions

Parameterized Multiplier using Booth's algorithm

- Iterative operation annotation
  - \texttt{Iteration\_Count\_Quantum}
  - \texttt{Done\_Signal\_Name\_Quantum}

Parameterized Factorization module with Shor's algorithm

- Connection of EXP & DFT modules