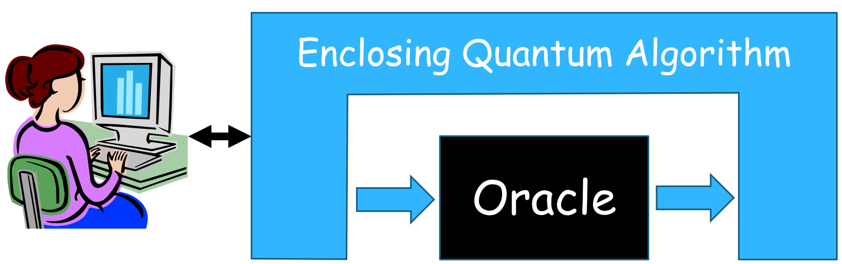
Chisel-Q: Designing Quantum Circuits with a Scala Embedded Language

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

Motivation

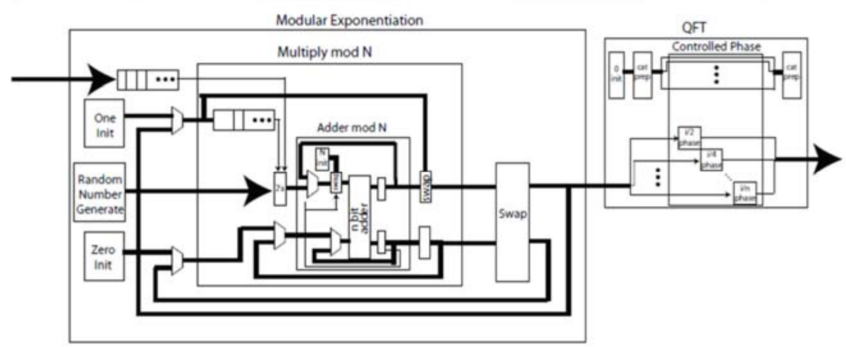
- Why Quantum Computers?
  - Great potential to speed up certain computations, such as factorization and quantum mechanical simulation
  - Fascinating exploration of physics
- Slow but constant research progress
  - New technologies, Computer Architectures, Algorithms
  - Still cannot quite build a large 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

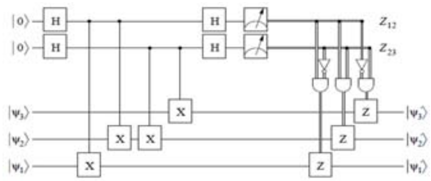
Example: Shor's Algorithm



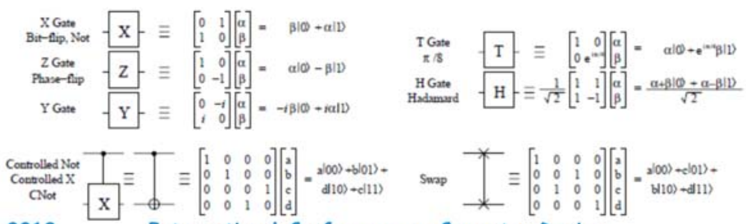
- Oracles: operate on 2048 or 4096-bit values
  - Modular exponentiation (with embedded operations)
  - Quantum Fourier Transform (QFT)

"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

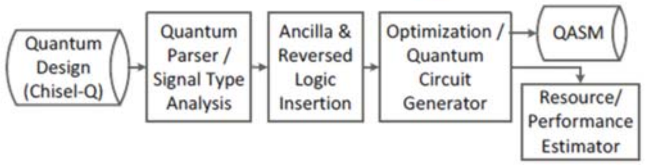
- Scala-based language for digital circuit design
  - High-level functional descriptions of circuits as input
  - Many backends: for instance direct production on Verilog
  - Used in design of new advanced RISC pipeline
- Features
  - High-level abstraction:
    - higher order functions, advanced libraries,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

```
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))
```


Quantum Version: Berkeley Chisel-Q in Nutshell

- Augmented Chisel Syntax, New Backend
  - Generate reversible versions of classical circuits
  - Classical ⇒ 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 Erasuree
  - State Machine analysis
- Output: QASM and statistics about the resulting circuit
  - Gate count, level of parallelism, predicted latency

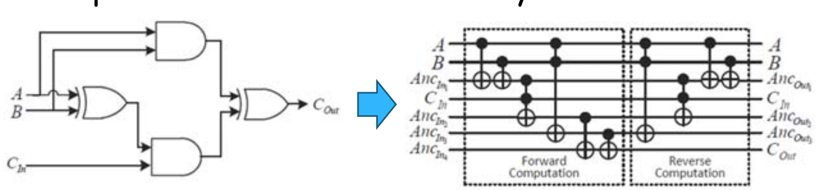
Signal Type Analysis

- 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

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



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, …

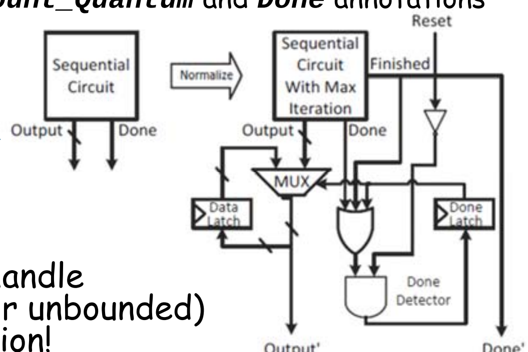
Hard Case: State Machine (Sequential Circuits with State)

- 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 ⇒ 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

Loop Fixing Transformation

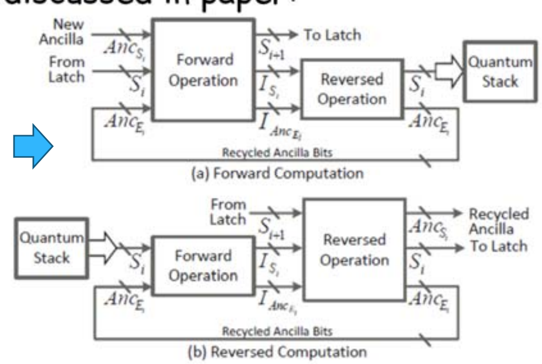


- Cannot currently handle unspecified (and/or unbounded) termination condition!

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:

State Hoarding Transformation



Designing with quantum operators

Syntax of Quantum Gates 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)

- 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 ***IsReversed = false*** to block the generation of reversed circuit

Parameterized Quantum Fourier Transform (QFT) Module

- Annotation *IsReversed = false* to block the generation of reversed circuit

```
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
}
```


Mathematical Benchmarks

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

Resource Estimation for Simple Benchmarks

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

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

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?

Circuit	Latency	Parallelism Min	Parallel ism Max	Parallelism Average
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

Resource Estimation of Components of a RISC Processor

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

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: $\Psi = C_0|0\rangle + C_1|1\rangle$
    - 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 $|0\rangle$ (say "UP")
  - With probability $|C_1|^2$ we will find $|1\rangle$ (say "DOWN")
- An *n*-qubit register can have 2^n values simultaneously!
  - 3-bit example:

$$\Psi = C_{000}|000\rangle + C_{001}|001\rangle + C_{010}|010\rangle + C_{011}|011\rangle + C_{100}|100\rangle + C_{101}|101\rangle + C_{110}|110\rangle + C_{111}|111\rangle$$

Parameterized Ripple-carry adder

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

```
class Adder_Ripple (width_in :Int = 4) extends Component
    {
  val io = new Bundle {
    val in1  =  Bits(INPUT, width_in)
    val in2  =  Bits(INPUT, width_in)
    val out  = Bits(OUTPUT, 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 Adder_Ripple_Q (width_in :Int = 4) extends
    Component {
  val io = new Bundle {
    val in1  =  Bits(INPUT, width_in)
    val in2  =  Bits(INPUT, width_in)
    val out  = Bits(OUTPUT, 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;
}
```

Parameterized Multiplier using Booth's algorithm

- Iterative operation annotation
  - *Iteration_Count_Quantum*
  - *Done_Signal_Name_Quantum*

```
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)
}
```


Parameterized Factorization module with Shor's algorithm

- Connection of EXP & DFT modules

```
class Shor_Exp_MulIO(n: Int) extends Bundle {
    val in  = Bits(INPUT,  n)
    val out  = Bits(OUTPUT, n)
}
class Shor_Exp_Mul(width_in :Int = 4) extends Component {
    val io = new Shor_Exp_MulIO(width_in)
    val exp = new Exp_Mul_WT(width_in);
 // val exp = new Exp_Mul_Booth(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;
}
```