A Gate-Level Approach To Compiling For Quantum Computers

IGSCC, 9AM Oct. 22, 2018

Hank Dietz
Professor and Hardymon Chair,
Electrical & Computer Engineering

University of Kentucky

Abstract

Programming language constructs generally operate on data words, and so does most compiler analysis and transformation. However, individual word-level operations often harbor pointless, yet resource and power hungry, lower-level operations. By transforming complete programs into gate-level operations on individual bits, and optimizing operations at that level, it is possible to dramatically reduce the total amount of work needed to execute the program's algorithm. This gate-level representation can be in terms of any complete set of logic gate types; earlier work targeted conventional multiplexor gates, but the work reported here centers on targeting CSWAP (Fredkin) gates without fanout – a form that can be implemented on a quantum computer. This paper will overview the approach, describe the current state of the prototype compiler, and suggest some ways in which compiler automatic parallelization technology might be extended to allow ordinary programs to take advantage of the unique properties of quantum computers.

University of Kentucky

Green And Sustainable?

- Try to manage power more efficiently
  - Whole-system power modeling
  - Scheduling, Throttling, ...
- Use inherently more efficient circuitry
  - Adiabatic, Quantum, ...
- Reduce the number of gate-level operations needed to implement each computation

Optimizing / Parallelizing Compilers

- Programming languages like C and Fortran
- Lots of analysis and transformations!
- Speedup-oriented automatic parallelization
  - Recognize parallelizable loops, etc.
  - Rewrite `for` as `parfor`, etc.
- Many optimizations, mostly at the word level: Common subexpression elimination, folding, register allocation, code scheduling, ...

University of Kentucky

Compiler Should Eliminate Unnecessary Work

- What don't we need to do?
  - Algorithms with too high O() complexity
  - Common subexpressions; recomputation
  - Excessive data motion
  - ...

These things also happen at the **gate level**, so **compilers should optimize at the gate level**

University of Kentucky

Many Early Machines Were Bit Serial, But Machines Got Wider

- 1958 *EDSAC 2* used microcoded bit-slicing; Various *PDP-11* were 4-bit; then 8, 16, 32, 64
- Massively-parallel microcoded bit-slicing in *DAP*, *STARAN*, *MPP*, *CM*, *CM2*, *GAPP*; *MP-1* was 4-bit; then 32 and 64
- Widening done to speed sequential code... assuming not enough parallelism is available

University of Kentucky

From Bits To Words, And Back Again

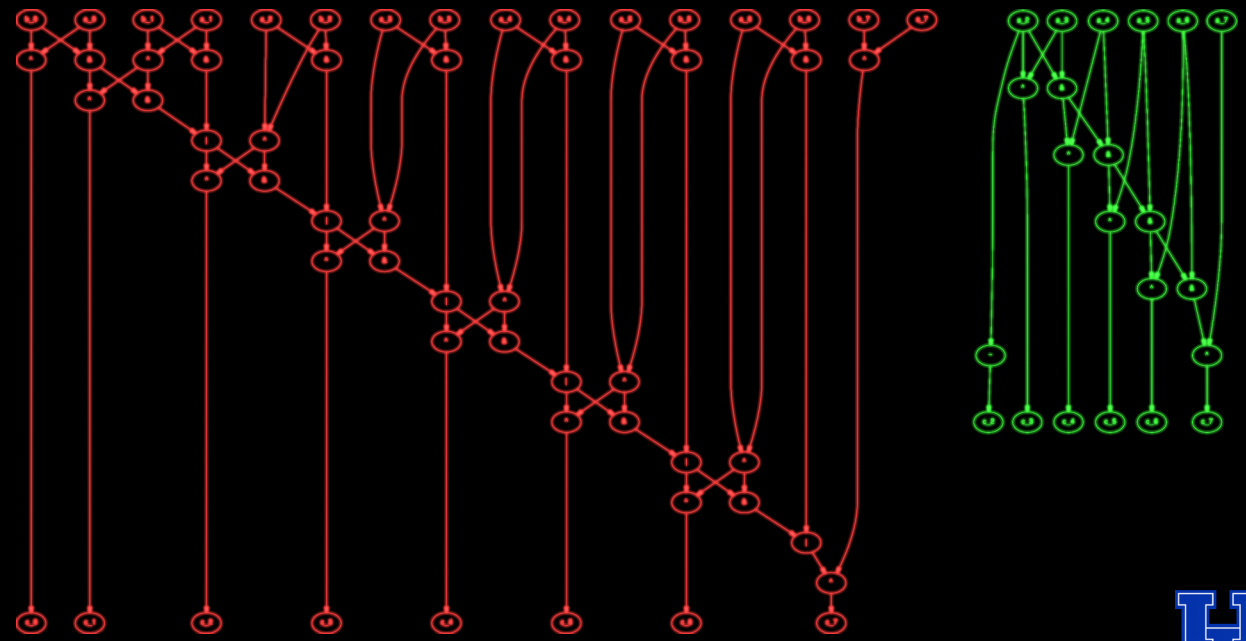
- Why go back to what is essentially bit-slicing?
  - Sequential code is *handled elsewhere*
  - Lots of parallelism available
- Fewer gates active per computation, e.g.:
  - 32 ripple carry 32-bit Adds in 32 clocks
  - To get one 32-bit Add in 1 clock, need *additional hardware* for carry lookahead...

i.e., **Lower power per computation!**

University of Kentucky

True Bit-Level Optimization

- Bit-slice systems were generally microcoded to implement a simple word-level ISA
- Word-level operations can imply useless work
  - E.g., using an Add to add 4 to a register:



University of Kentucky

True Bit-Level Optimization

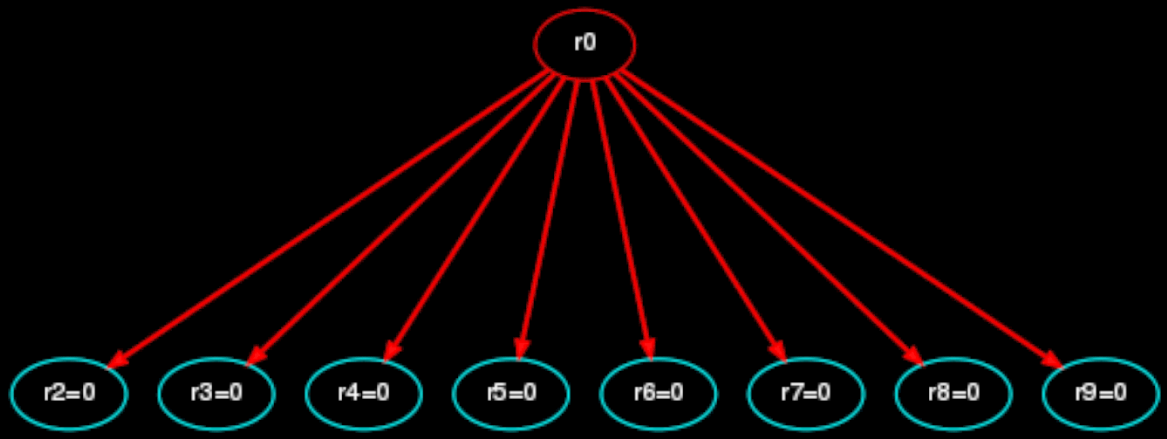
```
int:8 a, b, c;
a = (c * c) ^ 70;
a = ((a >> 1) & 1);
a = b + (c * b) + a;
a = a + ~(b * (c + 1));
```

University of Kentucky

True Bit-Level Optimization

```
int:8 a, b, c;
a = (c * c) ^ 70;
a = ((a >> 1) & 1);
a = b + (c * b) + a;
a = a + ~(b * (c + 1));

Total of 206669 ITEs created, 8 kept
```



University of Kentucky

Language Support For Bit-Level Specification

- How big is an `int`?
  - C has types like `int_fast8_t`
  - Only supports 8, 16, 32, or 64 bits
  - PCC: 2,882 `int`, 174 `unsigned`, but just 44 specifying 8, 16, 32, or 64 bits!
- Allow syntax like `int:10`
- Can also use for floats, although we prefer specifying accuracy rather than precision

University of Kentucky

Language Support For Explicit Quantum Algorithms

- Allowing quantum values has very little impact on gate-level logic design optimization
- Could allow a `q` ***attribute*** for quantum bits
  - `q int:5 a;` would be a 5-qubit integer
  - `int:5 *q p;` would be a qubit pointer to a randomly selected 5-bit signed integer
- Could allow `?` to be Hadamard bits
  - `a=?;` sets `a` to all possible 5-bit values

University of Kentucky

Basic Compilation To Bit-Level

- Bit-serial machines used world-level ISAs
- **SWARC** (SIMD within a register C):
  - The model behind MMX, SSE, AVX...
  - `int:5[6]` packed in 32-bit `int`; `a=b+c` is

```
a=((b&0x1ef7bdef)+(c&0x1ef7bdef))
   ^(0x21084210&(b^c))
```

- **BitC** language & compiler for nanocontrollers:
  - Word ops ⇒1-bit multiplexor ops, SITEs
  - Transformation to normal form (Karplus) and heavy gate-level optimization

University of Kentucky

Basic Compilation In Prototype "Hardly Software" Compiler

- Similar to BitC, but able to convert a complete program into a single combinatorial circuit
  - Implements *any state* in the state machine
  - Word level ⇒ vector of bit-level DAGs
  - AND/OR/NOT/XOR DAG optimized by scalable gate-level compiler methods (not Quine-McClusky nor Espresso)
  - Back-end generating CSWAPs
- A "research toy" testing ideas for a better compiler to follow...

Issues In The Prototype "Hardly Software" Compiler

- No range nor precision analysis
- No code generation for **array references** – perhaps a conventional memory interface?
- Seamless handling of function calls, including recursion, not yet implemented (needs arrays)
- No support for **cracking basic blocks** – a single very complex basic block can increase the size of the combinatorial logic for all states

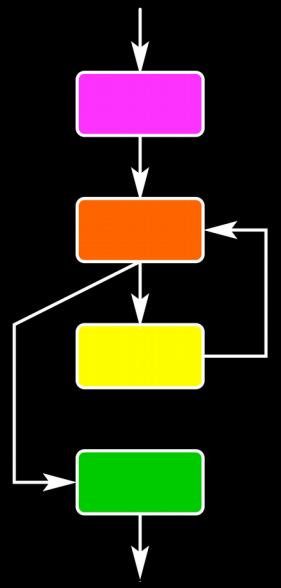
University of Kentucky

Basic Compilation Example

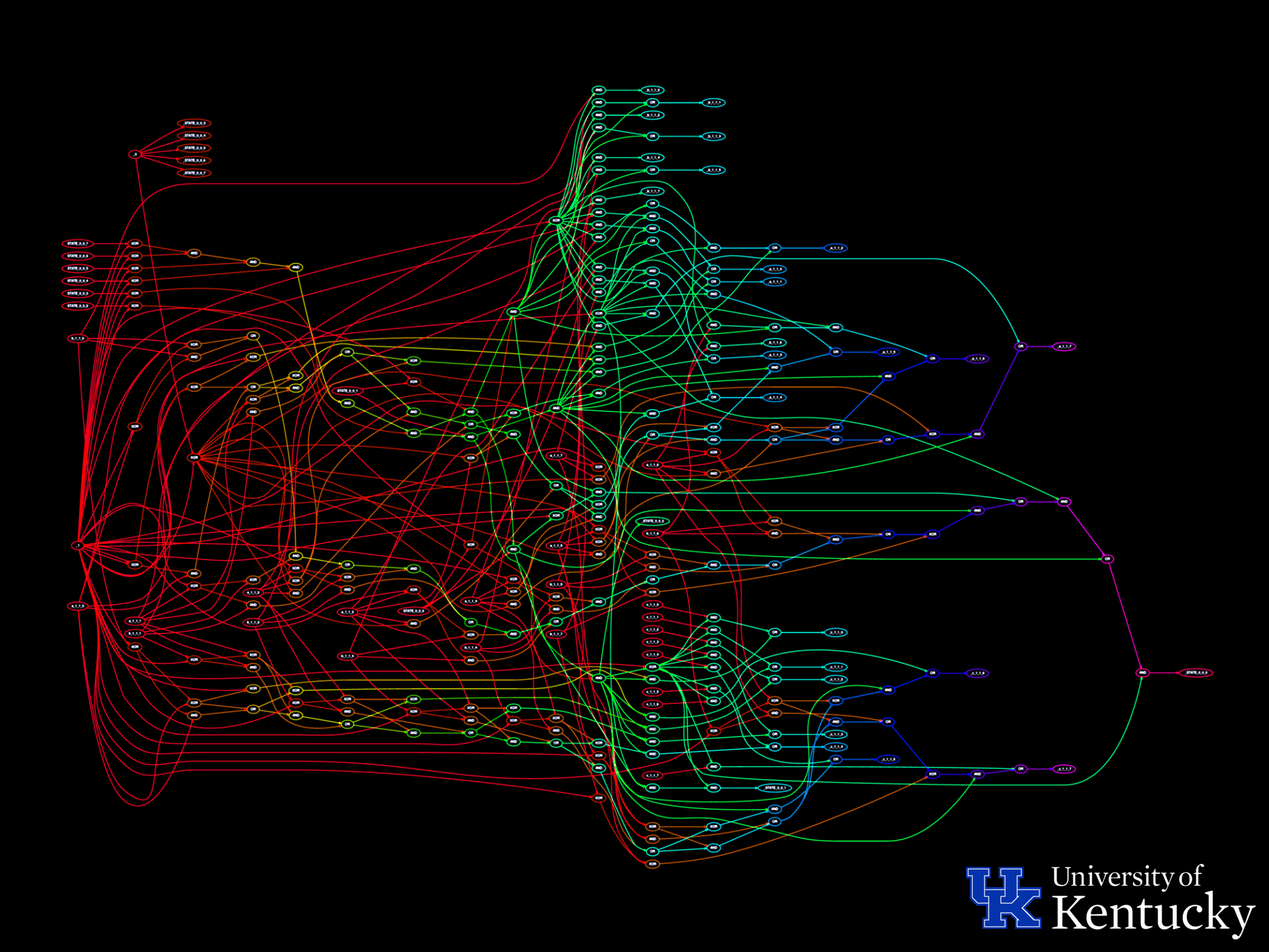
- Consider a trivial (8-bit default `int`) program:

```
int a, b, c;

main()
{
  b = 42; a = 100;
  while (a > b) a = a - 1;
  c = a - b;
}
```

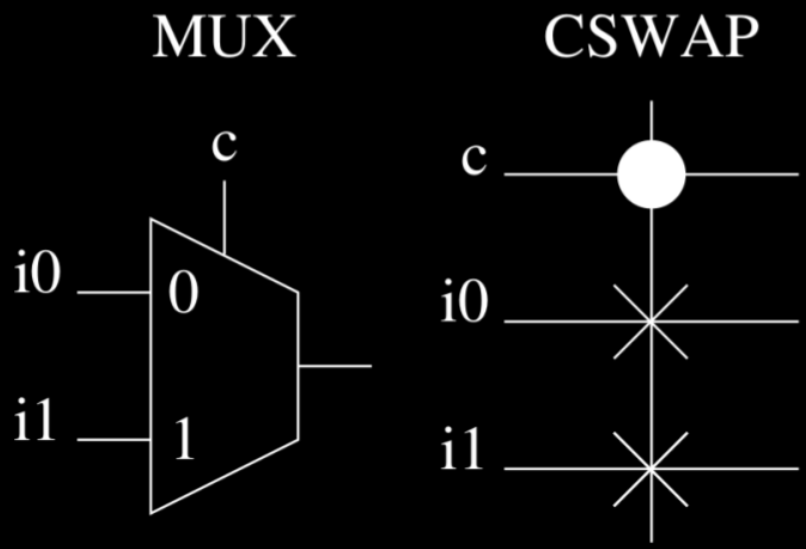


University of Kentucky


University of
Kentucky

CSWAP (Fredkin) Logic

- "Billiard-ball model" **adiabatic** gate
- All signals must be **unit-fanout**
- Efficient quantum implementation (2016)



c	i0	i1	MUX	CSWAP
0	0	0	0	0 0 0
0	0	1	0	0 0 1
0	1	0	1	0 1 0
0	1	1	1	0 1 1
1	0	0	0	1 0 0
1	0	1	1	1 1 0
1	1	0	0	1 0 1
1	1	1	1	1 1 1

University of Kentucky

CSWAP Output From Prototype "Hardly Software" Compiler

- Unit-fanout CSWAP generation:
  1. AND/OR/NOT/XOR ⇒ mutiplexors (MUX)
  2. MUX ⇒ CSWAP, inserting duplication gates wherever there is fanout
  3. Search to use alternate CSWAP outputs
  4. Order CSWAPs to sequence use of control pass-thru outputs, remove duplicate gates
- Considering Genetic Algorithm restructuring to minimize CSWAP complexity...

University of Kentucky

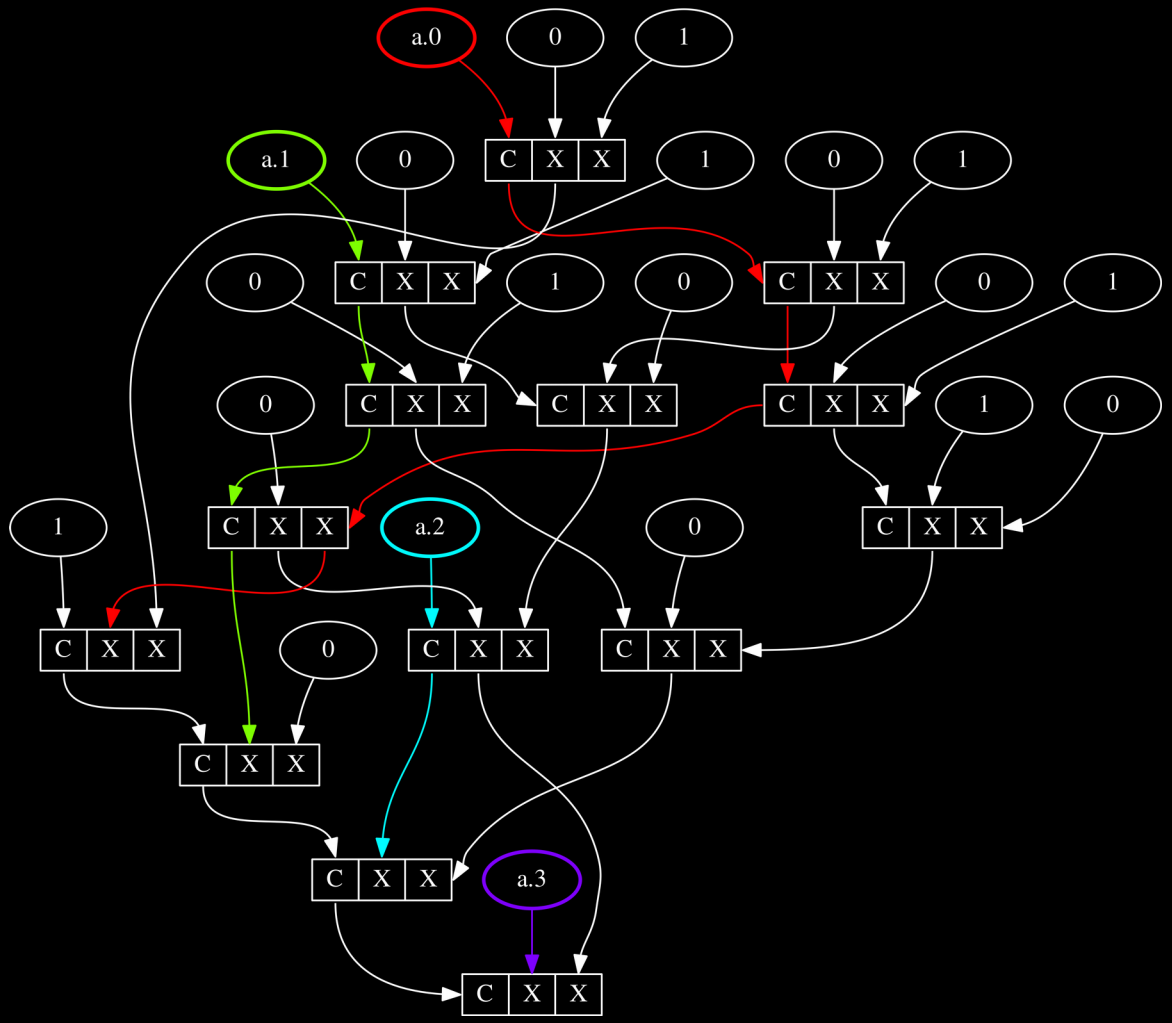
After This Paper Was Submitted...

- Reimplementation using code from BitC
- New SITE ⇒ CSWAP algorithm
  - Incrementally creates duplicates as needed
  - Tracks “lanes” and routes new values to same lane the target variable began in
- Output as Verilog code, text “lane” diagram, gate list, and circuit diagram

University of Kentucky

int:4 a; a=a*a;

```
0:       -----X---X---X
0:       ------X-------
0:       --X---C-------
0:       -------X----X-
0:       ----X--C------
0:       -X------XX----
0:       --------X-----
0:       ---X----C-----
0:       X---------X---
0:       ----------X--
1:       ------XX------
1:       --X-----------
1:       ----X---------
1:       -X------------
1:       ---X----------
1:       ----------CCCC
1:       X-------------
a.0:     CCC--X----X---
a.1:     ---CCC-----X--
a.2:     ---------C--X-
a.3:     -------------X
```



University of Kentucky

Use Of Entangled Qubit Quantum Computation?

- Could express quantum algorithms using ? Hadamard values... by writing new code
- Compiling ordinary C code results in CSWAP logic that ***never*** uses entangled qubits?
  - Could substitute quantum operations for basic math functions, e.g., `sqrt()`
  - Could **recognize parallelizable loops** that produce a single result and "parallelize" them using Hadamard inputs

University of Kentucky

Conclusions

- Reduce power by using fewer gate-level ops
- Complete state machines can be implemented with minimal (if any) reconfiguration
- Gate-level compiler optimization of whole C programs to unit-fanout CSWAPs is feasible
- More to do to make use of entangled qubits, improve optimization

University of Kentucky