A Gate-Level Approach To Compiling For Quantum Computers

***Purdue ECE**, 12:30 Feb. 15, 2019*

Hank Dietz
Professor and Hardymon Chair,
Electrical & Computer Engineering

University of Kentucky

A Gate-Level Approach To Compiling For Quantum Computers

A lot of interesting and exotic physics go into making a quantum computer – but this talk isn't about that. Our goal is simply to leverage understanding of conventional computing to take advantage of the benefits offered by quantum computation.

Programming language constructs generally operate on data words, and so does most compiler analysis and transformation. By instead 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 computational resources needed to execute a program’s algorithm. Such a gate-level representation also can be transformed to use other types of logic gates, including ones that efficiently can be implemented by quantum computers. We have created a simple prototype of such a system, which compiles C code into adiabatic CSWAP (Fredkin) gates without fanout. This talk will briefly present a computer engineer's view of quantum computing, overview our 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 better advantage of the unique properties of quantum computers.

University of Kentucky

A Gate-Level Approach To Compiling For Quantum Computers

Extended Dance Remix Version!

University of Kentucky

What Limits Computer Performance?

- No parallel programs!
  - Compiler finds stuff to execute in parallel
  - Parallel languages & libraries & tools
  - Actually, a lot is *"embarrassingly parallel"*

University of Kentucky

What Limits Computer Performance?

- No parallel programs!
  - Compiler finds stuff to execute in parallel
  - Parallel languages & libraries & tools
  - Actually, a lot is *"embarrassingly parallel"*
- How should it be done in parallel?
  - *"All the wires, all the time"*
  - Pipeline, SIMD, VLIW, MIMD, GPU, …

What Limits Computer Performance?

- No parallel programs!
  - Compiler finds stuff to execute in parallel
  - Parallel languages & libraries & tools
  - Actually, a lot is *"embarrassingly parallel"*
- How should it be done in parallel?
  - *"All the wires, all the time"*
  - Pipeline, SIMD, VLIW, MIMD, GPU, …
- **Not enough power!**

University of Kentucky

It's Really All About Power

- I'm one of the folks who started the cluster supercomputing revolution...
- A few years ago, I realized:
  - My lab has 30 tons cooling, 1.5kA power
  - My lab heats half the Marksbury building
  - My lab could **not** power **1 high-end rack**!
  - Big systems have **thousands of racks**

University of Kentucky

It's Really All About Power

- Try to manage power more efficiently
  - Whole-system power modeling
  - Scheduling, Throttling, …
- Reduce the number of gate-level operations needed to implement each computation
- Use inherently more efficient circuitry
  - Adiabatic $\Rightarrow$ near zero power per gate
  - Quantum $\Rightarrow$ exponentially less circuitry

University of Kentucky

It's Really All About Power

- Try to manage power more efficiently
  - Whole-system power modeling
  - Scheduling, Throttling, …
- Reduce the number of gate-level operations needed to implement each computation
- Use inherently more efficient circuitry
  - Adiabatic $\Rightarrow$ near zero power per gate
  - Quantum $\Rightarrow$ exponentially less circuitry

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 don't just happen at the word level, but also at the **gate level**...
compilers should optimize at the gate level

University of Kentucky

A Word About Words

- Most programming languages treat data objects as ***indivisible, atomic, entities***
- The programmer specifies type and size

  Fortran: `REAL*8 A`

  C: `int i; long long j;`
- Compiler anaylsis should ***look inside***
  - Eliminate processing meaningless bits by using smaller words or packed fields
  - Optimize algorithms at the bit level

Not All The Bits,
Not All The Time

- Integer precision / value range
- Floating point accuracy (not precision)
- Packing of smaller data

University of Kentucky

Integer Precision / Value Range

- 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 use compiler range analysis to set types… which was demonstrated as early as 1965!

University of Kentucky

Benefits Of Integer Ranging

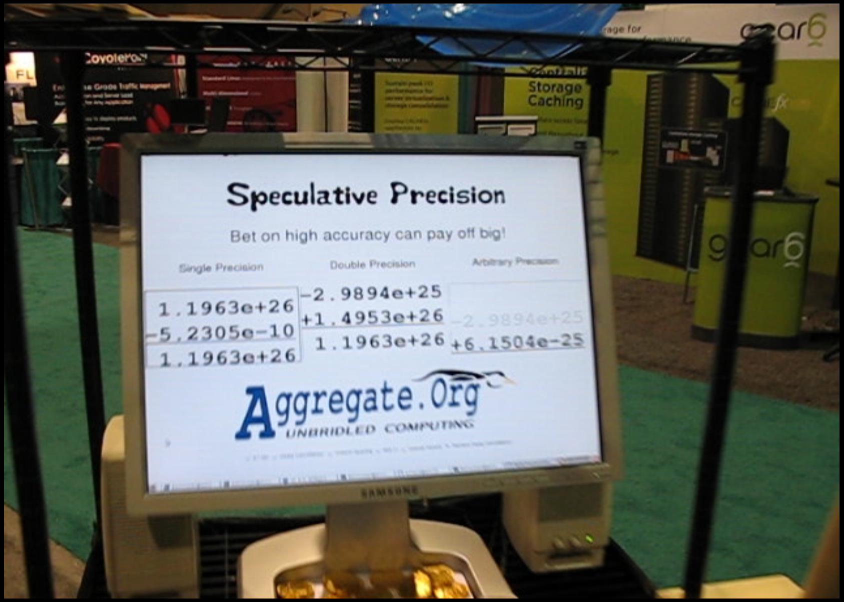
- Can ignore the bits that aren't active, e.g., only access low 16-bits of an `int` in [0..999]
  - Disable some wires and circuitry
  - Scatter/gather values (e.g., RISC-V AVS)
- Can use smaller storage space, thus reducing power use by:
  - Keeping more objects in registers/cache
  - Moving fewer bits/object

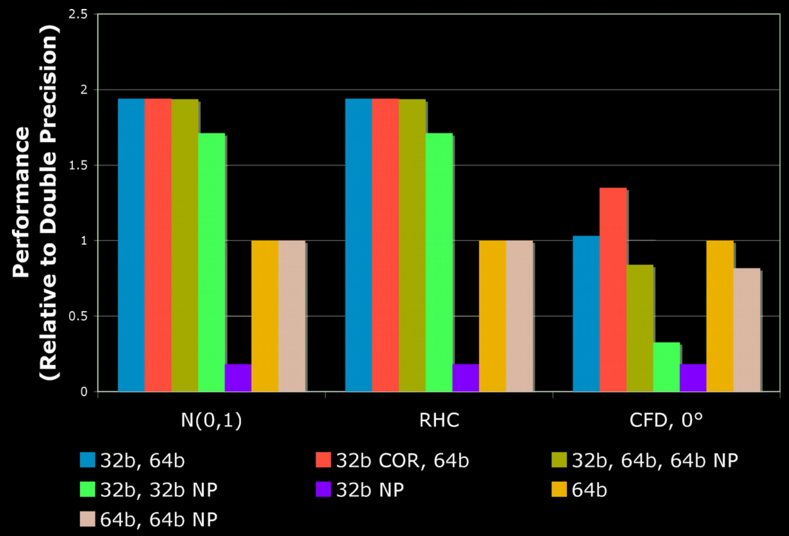
FP Accuracy, Not Precision

- Normally specify precision of floating-point (and could specify precisions in bits)
- Accuracy analysis is very difficult
- Accuracy analysis is very conservative; analysis often finds ***no*** significant digits, while computations typically have plenty
- Language constructs can help…

University of Kentucky

The Loosest Slots In Reno





- The first 32 terms of the Taylor series for $e^{-2\pi}$
- *Heavy Cancellation* sums $1 + ... + 1 + 1 \times 10^{-18} + 1 \times 10^{-18} - 1 - ... - 1$
- 32 *Uniform Spacing* values between 1.0 and 2.0
- $N(0,1)$adds Gaussian random numbers with $\mu = 0, \sigma = 1$
- *Inverse Square* is $\sum_{i=1}^{32} \frac{1}{i^2}$
- *Random Heavy Cancellation* is $\sum_{i=1}^{32} \pm 10^{x_i}$, where x_i is Gaussian with $\mu = 0, \sigma = 35$, but clipped to [-35, +35]

- 32-bit usually ok; 64-bit sometimes isn't!

Specifying FP Accuracy

```
#faildef exit();
#specdef fd(float, double)
#speculate fd
fd a=x; double b=sqrt(a);
if (!mytest(b, x)) {
#fail
} y=b;
#commit
```

University of Kentucky

Benefits For Floating-Point

- Huge performance gains for low precision
  - AMD RADEON INSTINCT MI25 GPU:
    64-bit: 0.768 TFLOPS
    32-bit: 12.3 TFLOPS
    16-bit: 24.6 TFLOPS
  - Memory footprint & bandwidth
- Potential to use LNS or scaled integer

University of Kentucky

Packing Smaller Data

- SWAR (SIMD Within A Register)
  - Originally, to obtain vector-like parallelism
  - More efficient use of memory & datapaths
- Virtualized in RISC-V AVS
- Compiler can ***pack unstructured things***: Common Subexpression Induction (CSI)

Computer Architectures Operate On Words, Not Bits

- 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
- This was done to speed sequential code… assuming not enough parallelism is available

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

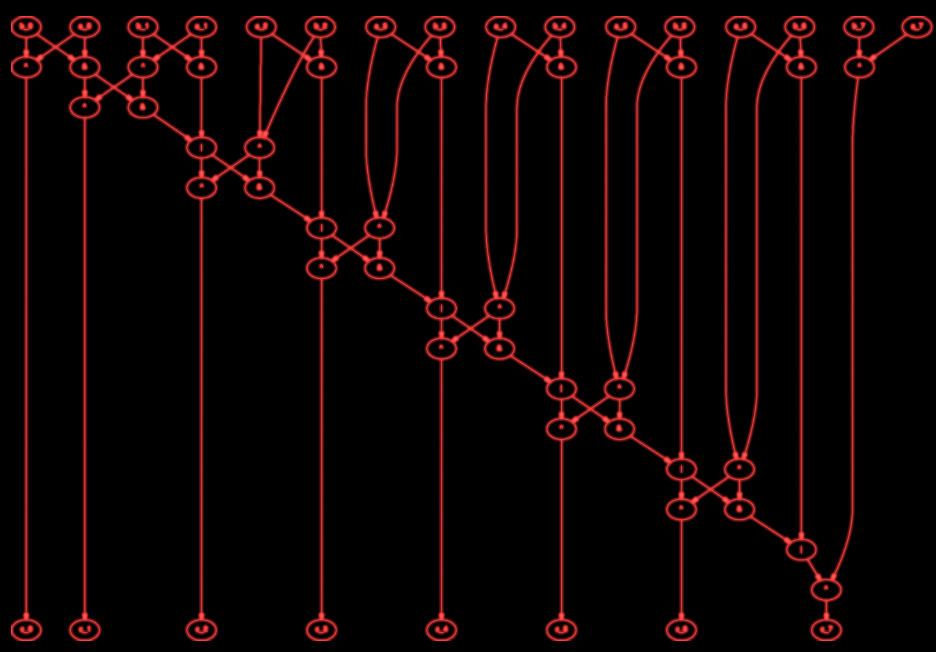
True Bit-Level Optimization

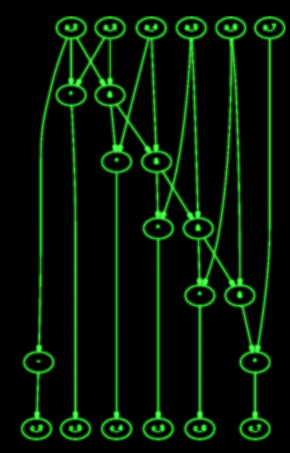
- How do we optimize gate-level designs?
  - Karnaugh maps?
  - Quine-McClusky algorithm?
  - Espresso?
  - Pattern matching with fixed modules?
- BitC language & compiler for nanocontrollers
  - Karplus algorithm for BDD normal form
  - Transformations to reduce execution cost

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

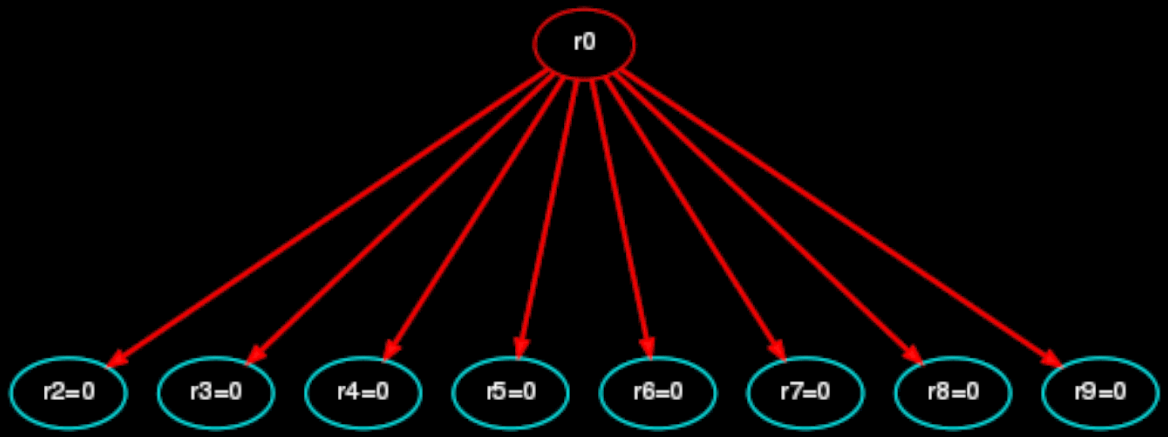
True Bit-Level Optimization

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

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

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

Whole-Program Gate-Level Optimization

- Similar to BitC, but able to convert a complete program into a single combinatorial circuit
  - Implements *any state* in the state machine
  - Word level $\Rightarrow$ 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…

University of Kentucky

Issues In The Prototype "Hardly Software" Compiler

- No range nor precision analysis
- No code generation for **array references** (it's an open problem for quantum mahines)
- 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

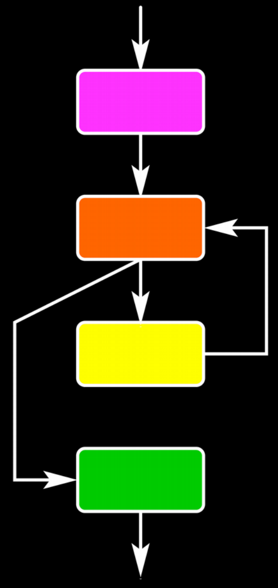
University of Kentucky

Basic Compilation Example

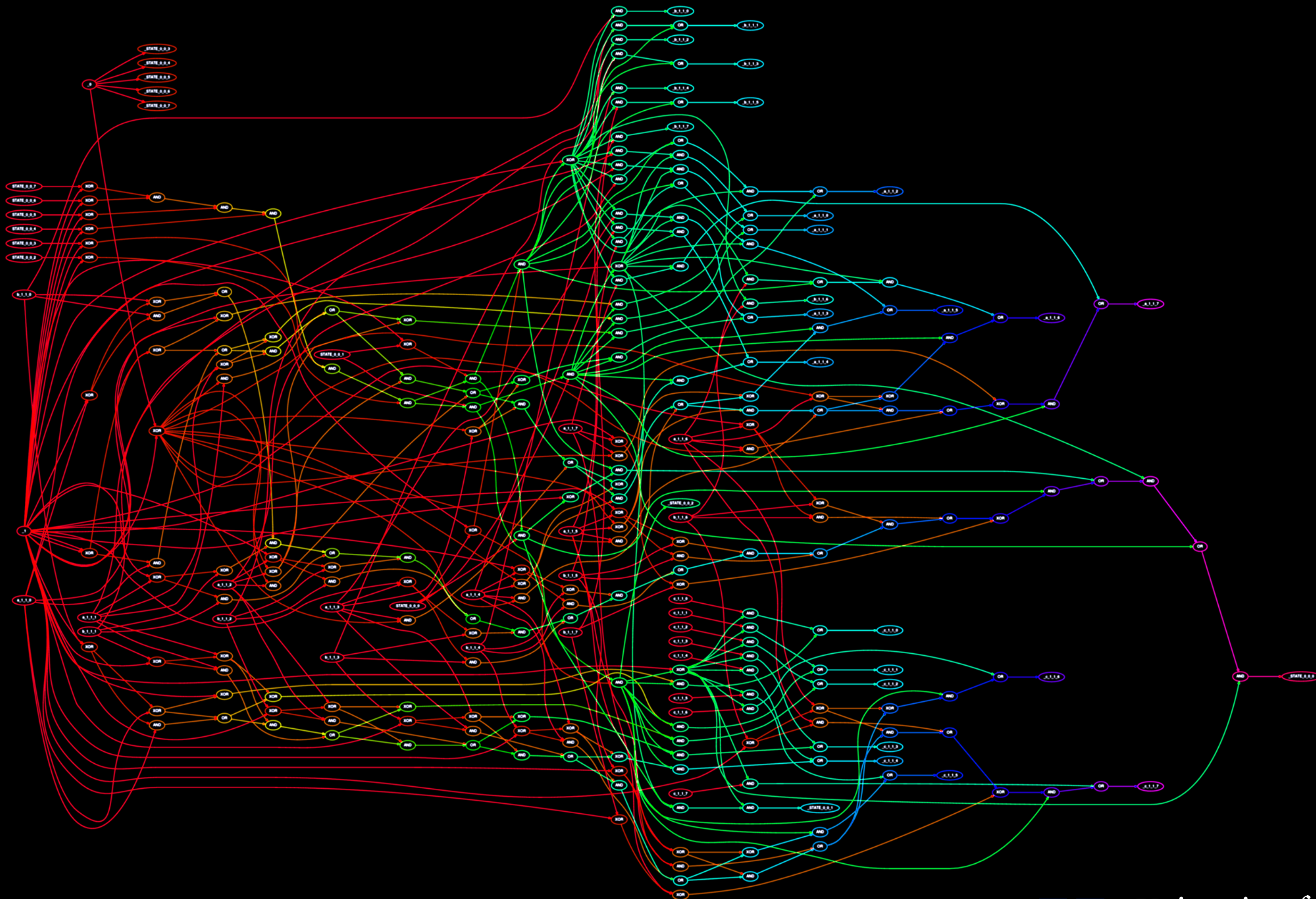
- Consider a trivial (8-bit default **`int`**) program:

```
int a, b, c;

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



University of Kentucky



University of Kentucky

Cool... But Isn't This Talk About Quantum Computing?

- Yes, it is!
- Quantum computers are gate-level systems
- Rather than using strange, new, programming methods, why not leverage the conventional?

What Is A Quantum Computer?

Parallel processing ***without*** parallel hardware.

- **Qubits** instead of bits
  - Each qubit can be 0, 1, or ***superposed***
  - ***Entangled*** qubits maintain values together
  - Measuring a qubit's value picks 0 or 1
- Quantum computers are *not state machines*; all they implement is ***combinatorial logic***
- Gates implemented *in sequence*

One OF IBM's Q Quantum Computers

University of Kentucky

KREQC: Kentucky's Rotationally Emulated Quantum Computer

- 6 qubits encode up to 2^6 6-bit values

"Spooky action at a distance via USB and servos"

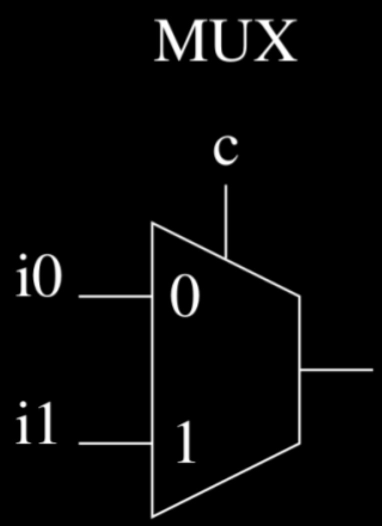
Run it at

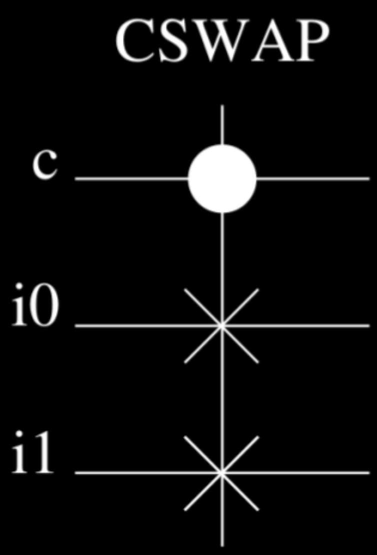
`http://aggregate.org/KREQC/`

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

CSWAP Full Adder

{carry, parity} = p + q + carry

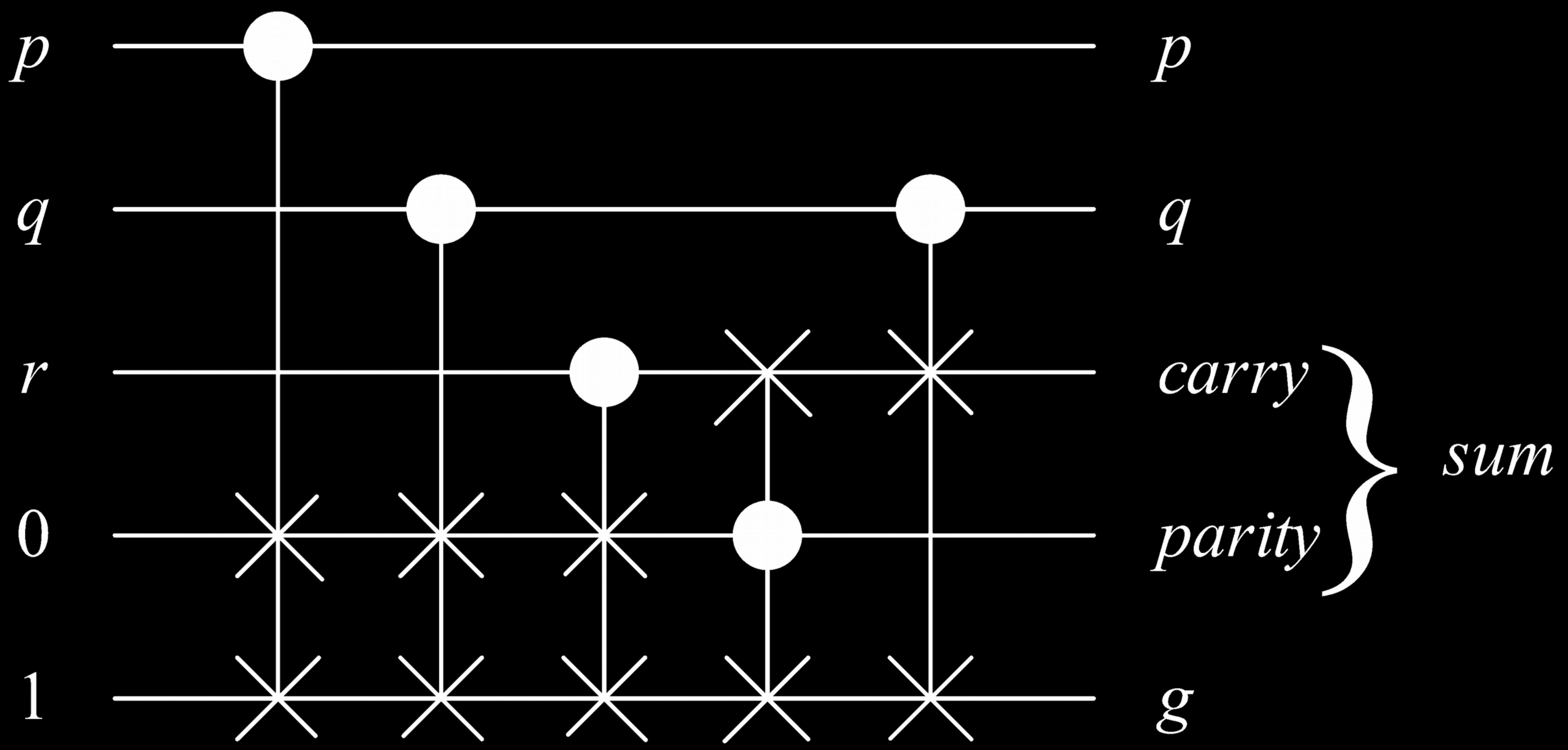


Figure from Bret Mulvey, Wikipedia page on Fredkin gates

KREQC Program

```
// 1-bit full adder
p=1;
q=1;
carry=0;
parity=0;
g=1;
CSWAP(p, parity, g);
CSWAP(q, parity, g);
CSWAP(carry, parity, g);
CSWAP(parity, carry, g);
CSWAP(q, carry, g);
```

Simulation Output

```
QUBIT                      g   parity   carry       q        p
               32 |     64 |      0 |      0 |     64 |     64 |
CSWAP             |        x-------x-------|-------|-------@
               32 |      0 |     64 |      0 |     64 |     64 |
CSWAP             |        x-------x-------|-------@       |
               32 |     64 |      0 |      0 |     64 |     64 |
CSWAP             |        x-------x-------@       |       |
               32 |     64 |      0 |      0 |     64 |     64 |
CSWAP             |        x-------@-------x       |       |
               32 |     64 |      0 |      0 |     64 |     64 |
CSWAP             |        x-------|-------x-------@       |
               32 |      0 |      0 |     64 |     64 |     64 |
                  1        0       0       1       1       1

                           g   parity   carry       q        p
   64/64                   0       0       1       1       1
```

KREQC Program

```
// 1-bit full adder
p=1;
q=0;
carry=?;
parity=0;
g=1;
CSWAP(p, parity, g);
CSWAP(q, parity, g);
CSWAP(carry, parity, g);
CSWAP(parity, carry, g);
CSWAP(q, carry, g);
```

Simulation Output

```
QUBIT                        g   parity   carry       q        p
                 32 |    64 |     0 |    32 |     0 |    64 |
CSWAP               |       x-------x-------|-------|-------@
                 32 |     0 |    64 |    32 |     0 |    64 |
CSWAP               |       x-------x-------|-------@       |
                 32 |     0 |    64 |    32 |     0 |    64 |
CSWAP               |       x-------x-------@       |       |
                 32 |    32 |    32 |    32 |     0 |    64 |
CSWAP               |       x-------@-------x       |       |
                 32 |    32 |    32 |    32 |     0 |    64 |
CSWAP               |       x-------|-------x-------@       |
                 32 |    32 |    32 |    32 |     0 |    64 |
                    0       1       0       1       0       1

                             g   parity   carry       q        p
   32/64                     0        1       0       0        1
   32/64                     1        0       1       0        1
```

University of Kentucky

KREQC Program

```
// 1-bit full adder
p=?;
q=?;
carry=?;
parity=0;
g=1;
CSWAP(p, parity, g);
CSWAP(q, parity, g);
CSWAP(carry, parity, g);
CSWAP(parity, carry, g);
CSWAP(q, carry, g);
```

Simulation Output

```
QUBIT                  g  parity   carry       q       p
            32 |    64 |     0 |    32 |    32 |    32 |
CSWAP          |       x-------x-------|-------|-------@
            32 |    32 |    32 |    32 |    32 |    32 |
CSWAP          |       x-------x-------|-------@       |
            32 |    32 |    32 |    32 |    32 |    32 |
CSWAP          |       x-------x-------@       |       |
            32 |    32 |    32 |    32 |    32 |    32 |
CSWAP          |       x-------@-------x       |       |
            32 |    48 |    32 |    16 |    32 |    32 |
CSWAP          |       x-------|-------x-------@       |
            32 |    32 |    32 |    32 |    32 |    32 |
               1       1       0       0       0       0

                       g  parity   carry       q       p
    8/64               0       0       1       1       1
    8/64               0       1       0       0       1
    8/64               0       1       0       1       0
    8/64               0       1       1       1       1
    8/64               1       0       0       0       0
    8/64               1       0       1       0       1
    8/64               1       0       1       1       0
    8/64               1       1       0       0       0
```

CSWAP Output From Prototype "Hardly Software" Compiler

- Unit-fanout CSWAP generation:
  1. AND/OR/NOT/XOR $\Rightarrow$ mutiplexors (MUX)
  2. MUX $\Rightarrow$ 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

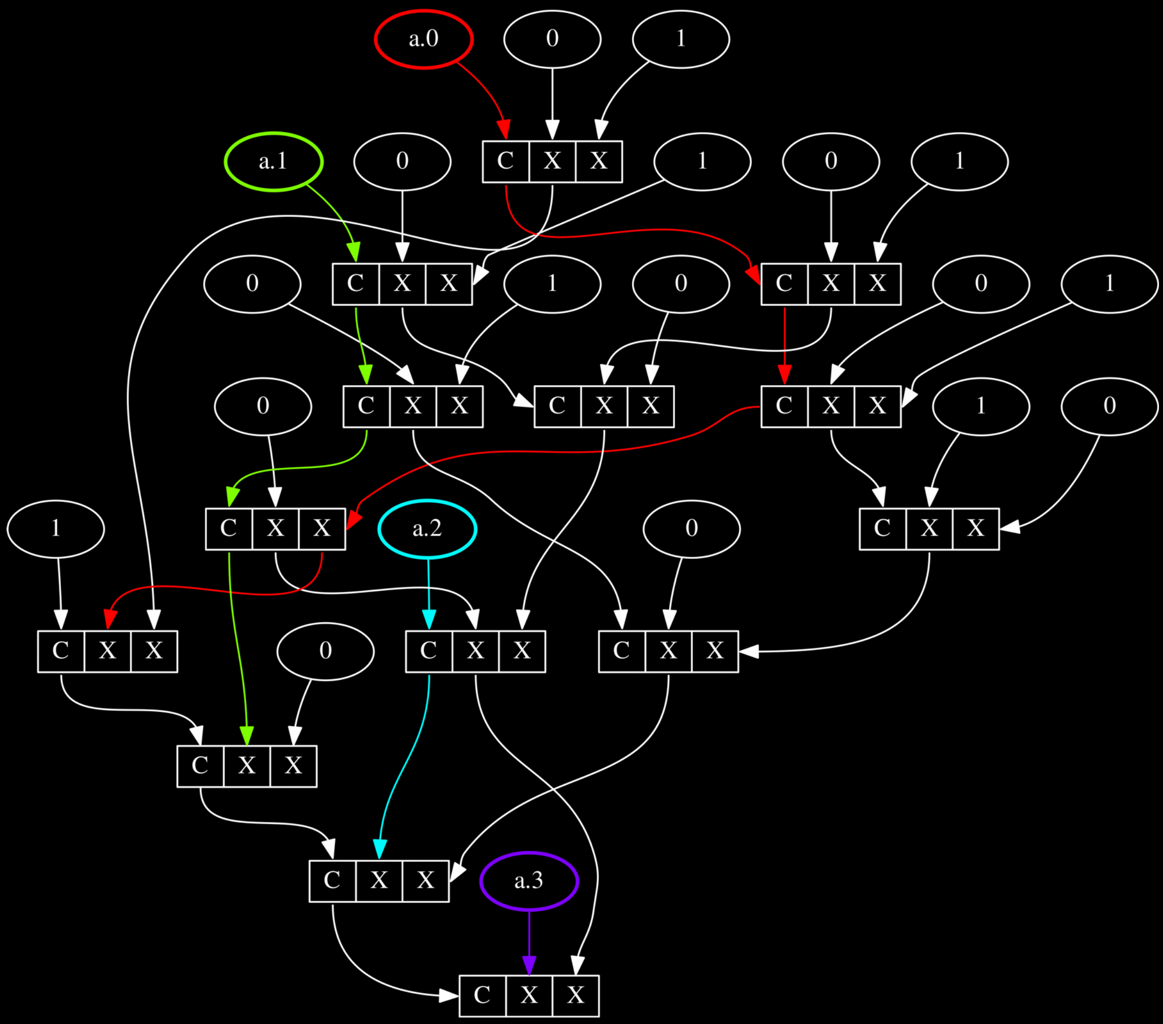
Second Prototype Compiler

- Reimplementation using code from BitC
- New SITE $\Rightarrow$ 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

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 superpositioned bits
  - **`a=?;`** sets **`a`** to all possible 5-bit values

University of Kentucky

Use Of Superposed-Qubit Quantum Computation?

- Could express quantum algorithms using `?` superposed 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 superposed inputs

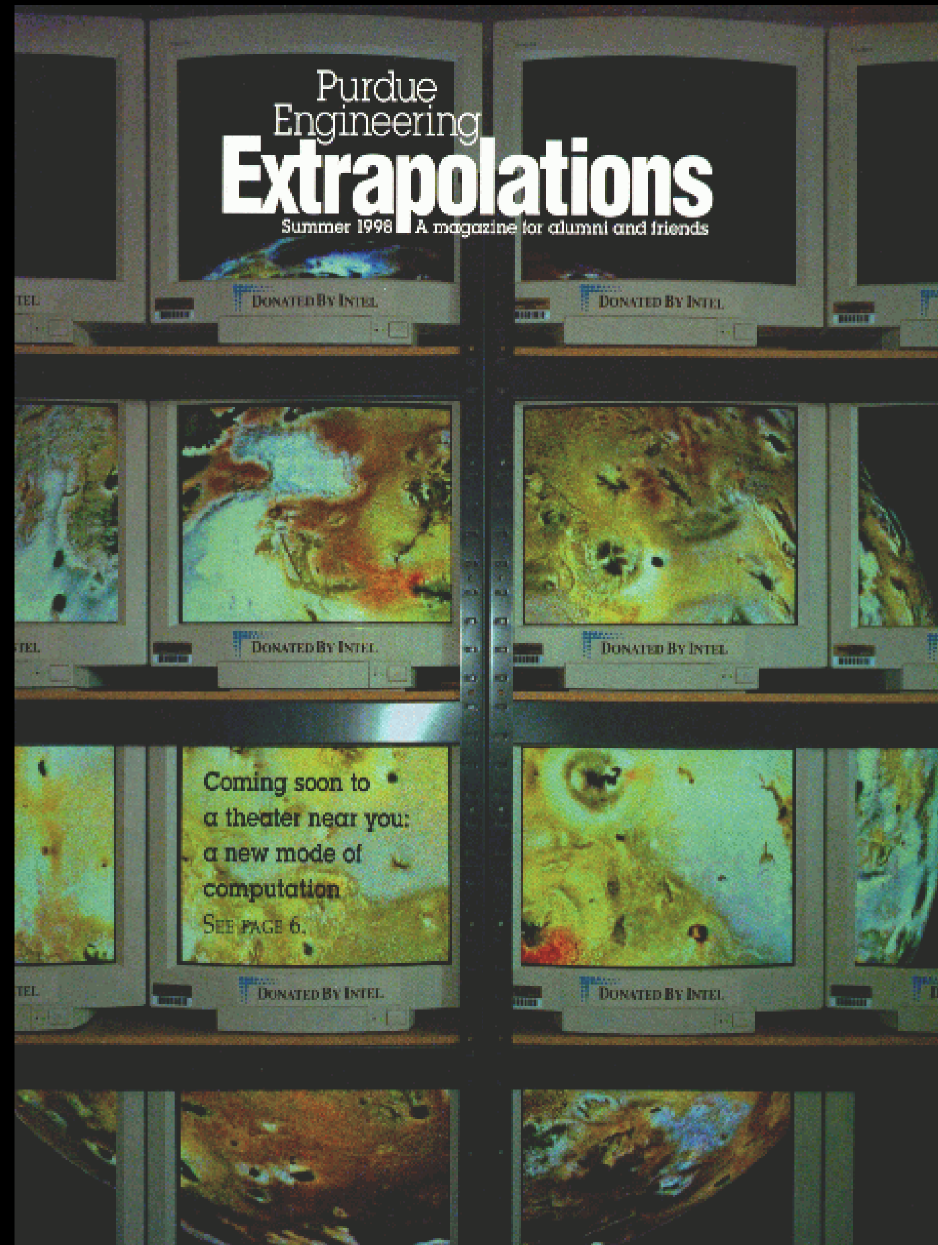
University of Kentucky

Conclusions

- Reduce power by using fewer gate-level ops
- Can implement using a quantum computer:
  - State machines can be implemented with minimal (if any) reconfiguration
  - Gate-level compiler optimization of whole C programs to CSWAPs is feasible
- Future work: use superposed qubits, improve optimization, & build quantum computers ;-)

Aggregate.Org
UNBRIDLED COMPUTING

University of Kentucky


Purdue Engineering
Extrapolations
Summer 1998
A magazine for alumni and friends
DONATED BY INTEL
Coming soon to a theater near you: a new mode of computation
SEE PAGE 6.