Parallel Bit Pattern Computing

***Computing with Unconventional Technologies, IGSCC**, Oct. 21, 2019*

Henry (Hank) Dietz
Professor and Hardymon Chair,
Electrical & Computer Engineering
University of Kentucky

I Want It All.

- **Reduce power / computation...** while getting **speedup**... while **leveraging** standard engineering practices... **which requires CUT**
- UTs we propose to use include:
  - Implement using low-power gates
  - Operate only on active bits
  - Apply compiler optimization at gate level
  - Amortize control logic overhead
  - *N*-way parallel without O(*N*) hardware

University of Kentucky

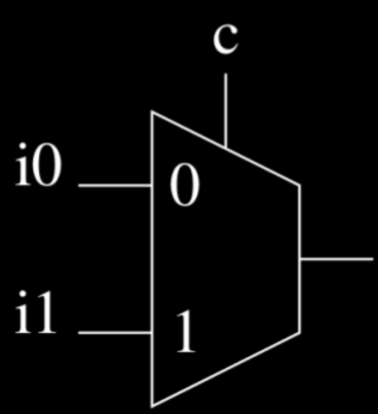
Implement Using Low-Power Gates

- Traditional digital logic wastes power
  - Inputs are absorbed
  - Outputs are generated from Vcc/Gnd
- Adiabatic (thermodynamically reversible) logic could avoid that waste
  - Can recover energy
  - Can use "billiard ball conservancy"

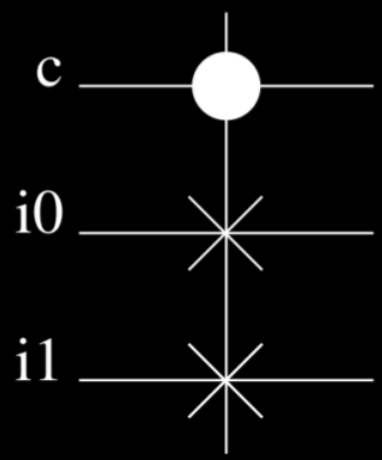
E.g., CSWAP (Fredkin) Gate

- Functionally complete adiabatic gate
- All signals must be **unit-fanout**
- Efficient circuit & quantum implementations

MUX



CSWAP



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

Operate Only On Active Bits

- How big is an `int`?
  - Typically, 32 bits
  - E.g., a value ∈ [0..100] only needs 7 bits
  - Why store, copy, & process inactive bits?
- SWARC (SIMD Within A Register C) operates on packed fields, saving space & operations
- BitC targets bit-serial nanocontrollers
- Specify floating-point accuracy, not precision

University of Kentucky

Apply Compiler Optimization At The Gate Level

- Compiler optimization applied at word level:

```
a=4; b=a-3; c=c+b; d=a*c; e=c*a*b;
```

Becomes:

```
a=4; b=1; ++c; d=c<<2; e=d;
```

- At gate level, improvement can be huge

```
unsigned int:4 a, b;
c = a + b;
```

- Unoptimized, **35** single-gate operations:

```
c0=(a0^b0); c1=((a0&b0)^(a1^b1));
c2=(((a1&b1)|((a0&b0)&(a1^b1)))^
    (a2^b2));
c3=(((a2&b2)|(((a1&b1)|((a0&b0)&
    (a1^b1)))&(a2^b2)))^(a3^b3));
c4=((a3&b3)|(((a2&b2)|(((a1&b1)|
    ((a0&b0)&(a1^b1)))&(a2^b2)))&
    (a3^b3)));
```

University of Kentucky

```
unsigned int:4 a, b;
c = a + b;
```

- Optimized, **17** single-gate operations:

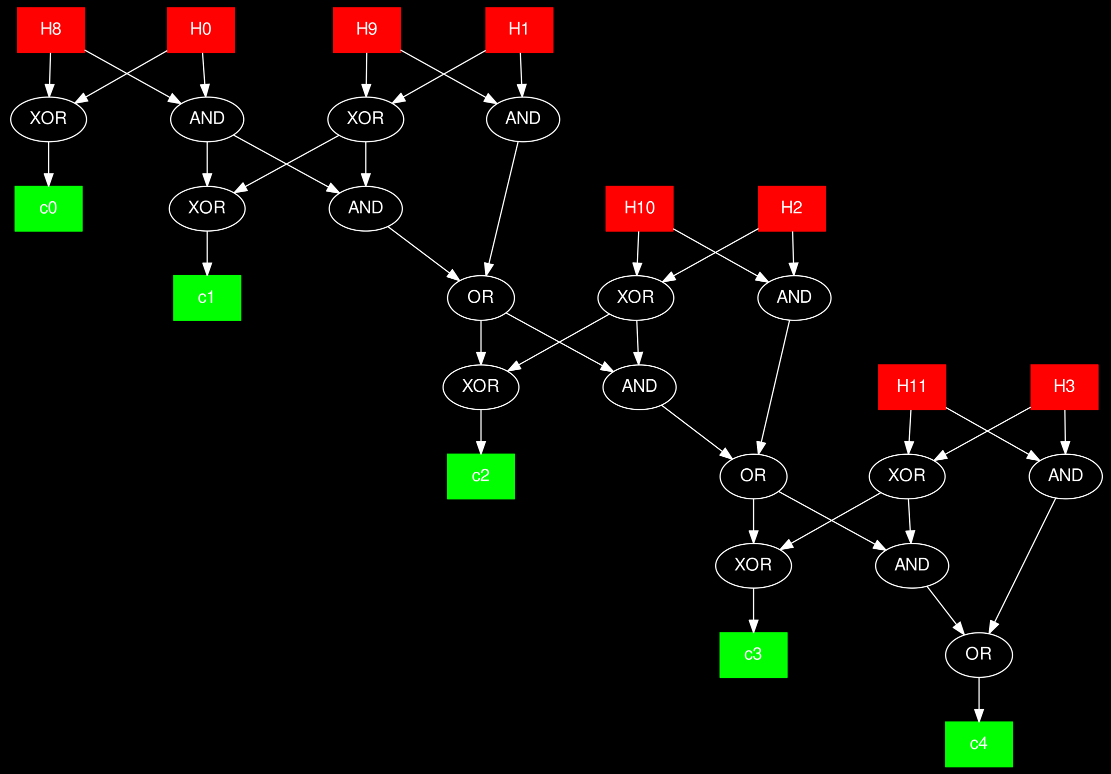
```
c0=(a0^b0); t0=(a0&b0); t1=(a1^b1);
c1=(t0^t1); t2=(a1&b1); t3=(t0&t1);
t4=(t2|t3); t5=(a2^b2); c2=(t4^t5);
t6=(a2&b2); t7=(t4&t5); t8=(t6|t7);
t9=(a3^b3); c3=(t8^t9);
c4=((a3&b3)|(t8&t9));
```

- By common subexpression elimination (CSE)

University of Kentucky

```
unsigned int:4 a, b;
c = a + b;
```

- Optimized, **17** single-gate operations:



University of Kentucky

```
unsigned int:4 a, b;
b = 1; c = a + b;
```

- Optimized, 7 single-gate operations:

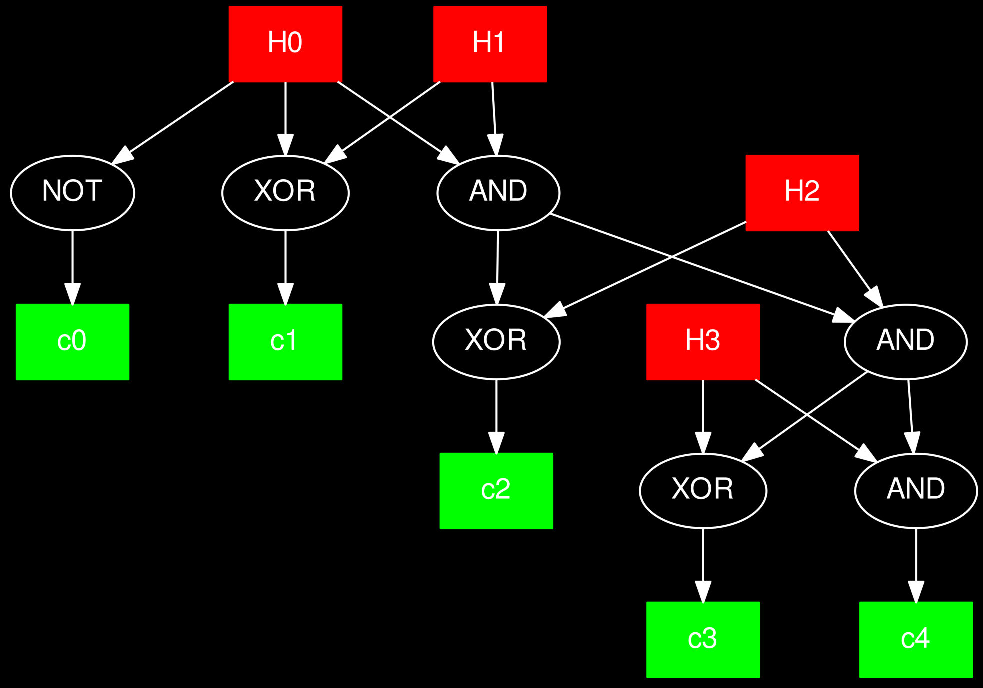
```
c0=~ a0; c1=(a0^a1); t0=(a0&a1);
c2=(a2^t0); t1=(a2&t0); c3=(a3^t1);
c4=(a3&t1);
```

- By value forwarding, constant folding, algebraic simplification, and CSE… ***standard compiler optimizations!***

University of Kentucky

```
unsigned int:4 a, b;
b = 1; c = a + b;
```

- Optimized, **7** single-gate operations:



University of Kentucky

```
unsigned int:4 a, b;
b = a; c = a + b;
```

- Optimized, **ZERO** single-gate operations:

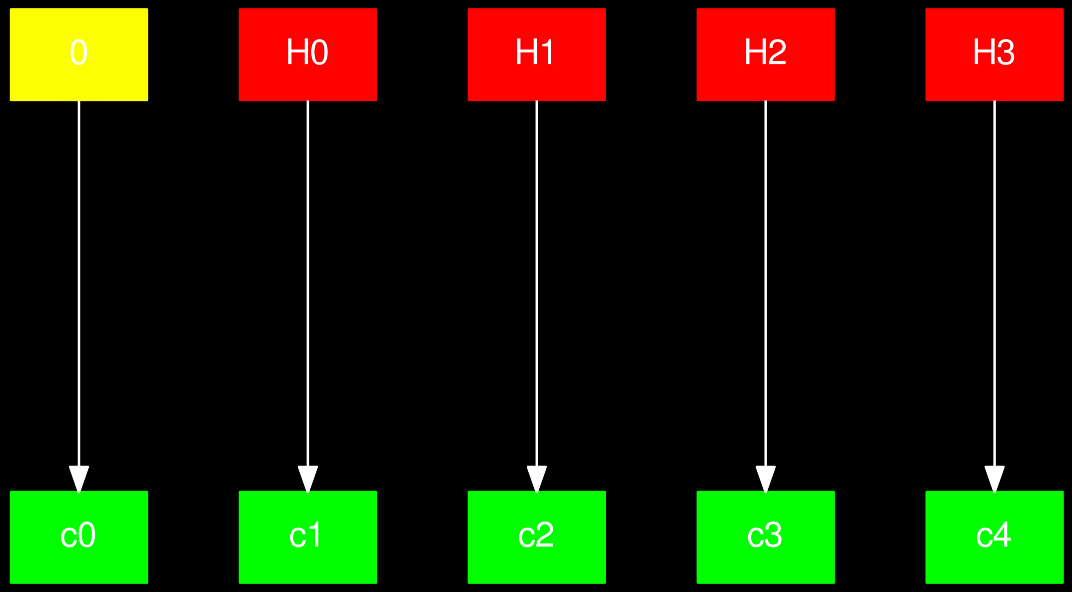
```
c0=0; c1=a0; c2=a1; c3=a2; c4=a3;
```

- Compiler simplified addition into a shift
- Shift left by one is literally changing where each bit of `c` is found, doesn't even copy bits

University of Kentucky

```
unsigned int:4 a, b;
b = a; c = a + b;
```

- Optimized, **ZERO** single-gate operations:



Amortize Control Logic Overhead

- Much of a conventional machine's power is spent implementing control logic
  - Dominates in conventional processors
  - Here, e.g., 1-bit ALU vs. increment the PC
- Virtualized SIMD can hide overhead
  - M-bit wide machine given $M \times N$ work
  - Cycling thru N can hide O(N) overhead
  - Same trick used in CM1/2/200, GPUs, ...

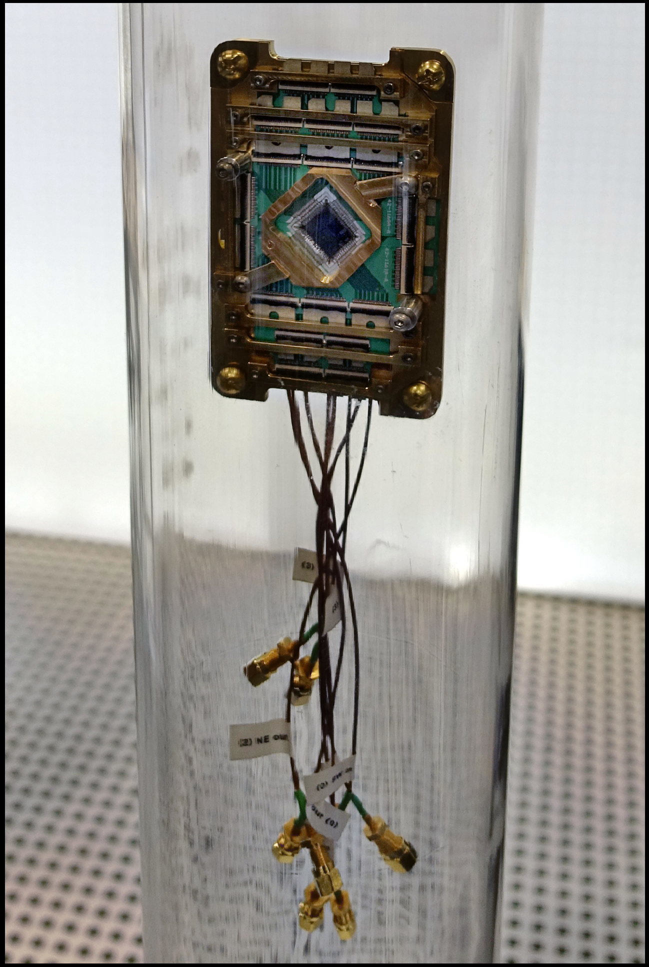
University of Kentucky

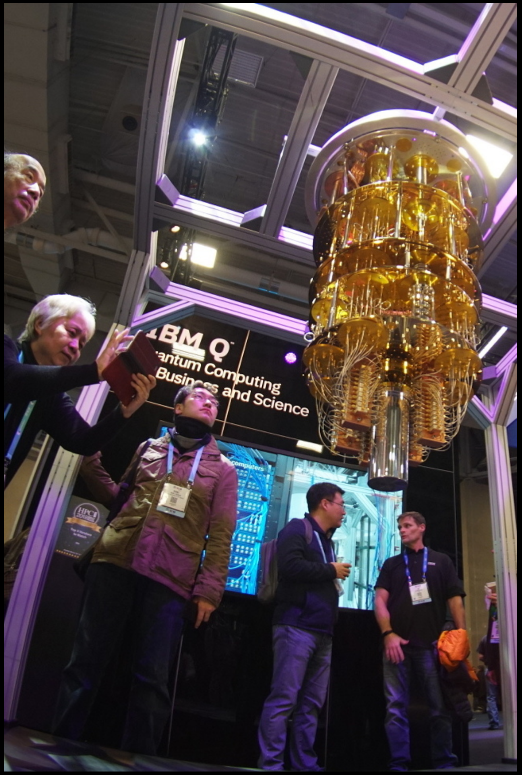
N-way Parallel Execution Without *N* Units Of Hardware

- Parallel processing has been the primary way to obtain speedup, but *N*-way parallelism conventionally implies O(*N*) hardware... and **O(*N*) power consumption**
- The best known way to avoid this is Quantum computing... which is **NOT** what we're doing, but understanding it clarifies our method

University of Kentucky

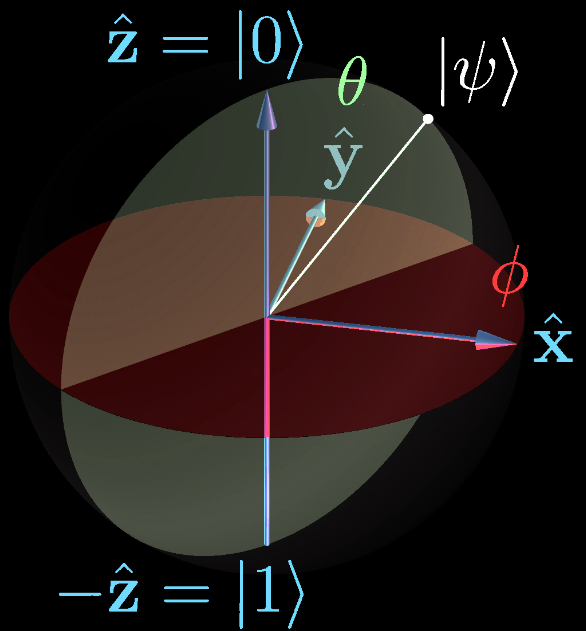
Quantum Computers at SC18 (Left: D-Wave, Center, right: IBM Q)





University of Kentucky

Bloch Sphere Qubit Model



$$|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\phi}\sin(\theta/2)|1\rangle$$

$$= \cos(\theta/2)|0\rangle + (\cos\phi + i\sin\phi)\ \sin(\theta/2)|1\rangle$$

where $0 \leq \theta \leq \pi$ and $0 \leq \phi < 2\pi$

- Value of a **Qubit** is a wave function
- Probability by coordinates on sphere surface

University of Kentucky

What Does That Mean?

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

- **Qubits** instead of bits
  - Each qubit can be 0, 1, or ***superposed***
  - A “gate” operates on superposed values
  - ***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 are implemented *in sequence*

KREQC: Kentucky's Rotationally Emulated Quantum Computer

- 6 qubits simultaneously encode 2^6 6-bit values

"Spooky action at a distance via USB and servos"

Run it at

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

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
```

University of Kentucky

How Does KREQC Work?

On **pattern bits**, `pbit`s, not exactly qubits...

- Superposition:
  Each `pbit` is an ***ordered*** set of bit values
- Each gate is applied to all bits in the set
- *N*-way entanglement:
  The set of 2^N bit values is ordered such that each ***position*** is entangled across `pbit`s

Addition Of Two 2-`pbit` Values

0	1	2	3
0	1	0	1
0	0	1	1

+

3	1	3	0
1	1	1	0
1	0	1	0

=

3	2	5	3
1	0	1	1
1	1	0	1
0	0	1	0

- Superposed state of a **`pbit`** is a set of bits
- *N*-way entangled **`pbit`** is ordered 2^N bits

University of Kentucky

Parallel Bit Pattern Computing

- Provides `pint` interface as well as `pbit`
- Ordered bit set is compressed by coding a generative Regular Expression (RE):
  $\{0,0,1,1,0,0,1,1\} \rightarrow (0^2\ 1^2)^2$
- Finding minimal RE is hard, but can just use Run Length Encoding (RLE) subset of REs:
  $\{1,0,0,0,0,1,1,1\} \rightarrow 1^1\ 0^4\ 1^3$
- **Operates on REs without expanding them**

University of Kentucky

Parallel Bit Pattern Computing

- Can use symbols larger than 1 bit; prototype uses 4096-bit symbols (12-way entanglement) in patterns with up to 32-way entanglement
- Duplicate symbols are recognized (FBP – Factored Bit Parallel chunks)
  - Only keep one copy (with reference count)
  - Applicative caching of chunk operations
- Just-in-time **optimizing compiler** translates `pint` ops into optimized `pbit` ops

4 Fully-Entangled `pbits`

0 1 0 1	0 1 0 1	0 1 0 1	0 1 0 1
1 1 0 0	1 1 0 0	1 1 0 0	1 1 0 0
0 0 0 0	1 1 1 1	0 0 0 0	1 1 1 1
0 0 1 0	0 0 1 0	0 0 1 0	1 0 1 0

0 1 0 1 *4*			
1 1 0 0 *4*			
0 0 0 0 *1*	1 1 1 1 *1*	0 0 0 0 *1*	1 1 1 1 *1*
0 0 1 0 *3*	1 0 1 0 *1*		

University of Kentucky

4 Fully-Entangled `pbits`

0101^{4}

1100^{4}

0000^{1} 1111^{1} 0000^{1} 1111^{1}

0010^{3} 1010^{1}

0101^{1}

1100^{1}

0000^{1} 1111^{1}

0010^{3} 1010^{1}

University of Kentucky

Complexity Of FBP Operations

Operation Name	Operation Functionality	O( )
H, Hadamard	a=superposition of 0/1	R
NOT	Adiabatic a =NOT a	N
SWAP	Adiabatic exchange values of a and b	1
CCNOT, Toffoli	Adiabatic if a AND b, c =NOT c	N
CSWAP, Fredkin	Adiabatic if c, SWAP(a, b)	N
Duplicate	$a = b$	R
AND	$a = b$ AND c	N
OR	$a = b$ OR c	N
XOR	$a = b$ XOR c	N
All	Reduction, true if a is 1	1
Any	Reduction, true if a contains a 1	1
Population	Reduction, count of 1s in a	R
Simplify	Internal simplify regular expression	R

- *R* is # of symbols in the regular expression
- *N* is # of bits in the entangled value

University of Kentucky

An Example: Find `sqrt(29929)`

- Initialize `pbit` (thus, also `pint`) system
  ```
  pbit_init();
  ```
- Create a 16-`pbit` value of 29929
  ```
  pint a = pint_mk(16, 29929);
  ```
- Create 8-way entangled Hadamard value, the superposition of 0, 1, 2, ... 255
  ```
  pint b = pint_h(8, 0xff);
  ```
- Square all 256 possible values
  ```
  pint c = pint_mul(b, b);
  ```

An Example: Find `sqrt(29929)`

- Make an entangled value that's 1 only where the squared value is equal to 29929
  ```
  pint d = pint_eq(c, a);
  ```
- Multiply entangled values of original guesses by the mask so only the solution is not 0
  ```
  pint e = pint_mul(d, b);
  ```
- Measure the result, printing all unique values
  ```
  pint_measure(e);
  ```

An Example: Find `sqrt(29929)`

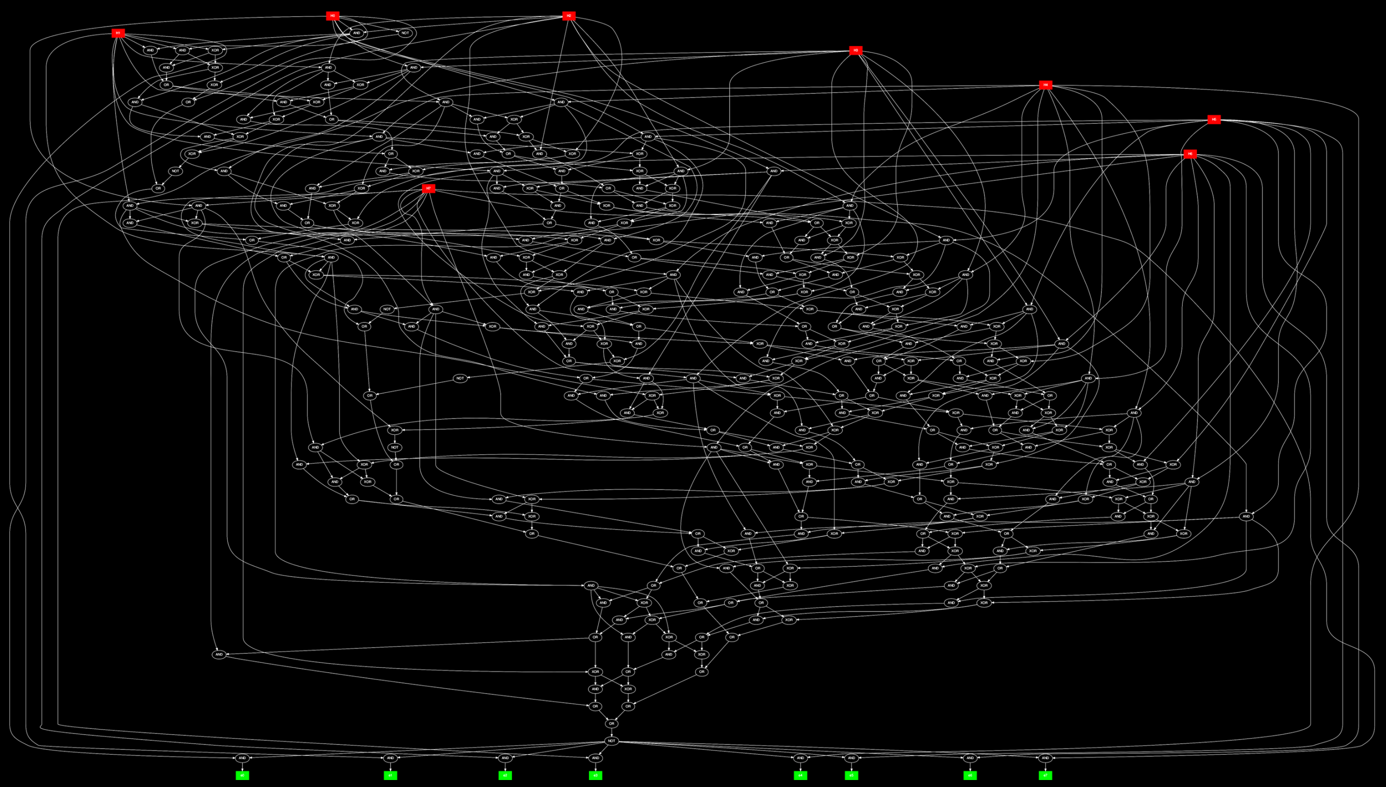
- The complete C program, prints `0 173`

```
int main(int argc, char **argv) {
  pbit_init();
  pint a = pint_mk(16, 29929);
  pint b = pint_h(8, 0xff);
  pint c = pint_mul(b, b);
  pint d = pint_eq(c, a);
  pint e = pint_mul(d, b);
  pint_measure(e);
}
```

University of Kentucky

An Example: Find `sqrt(29929)`

- **310** single-gate operations:



University of Kentucky

Conclusions

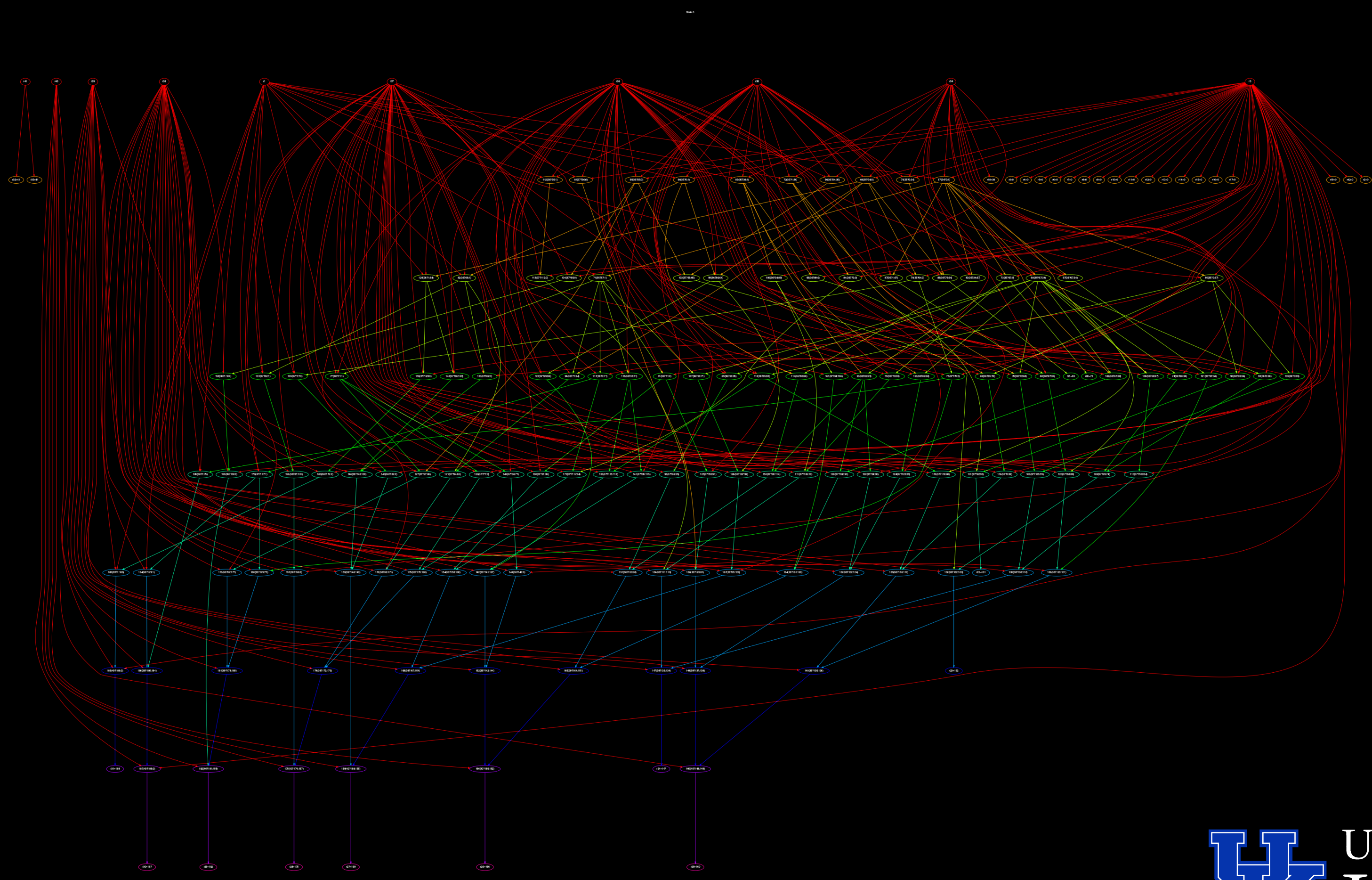
- Green & sustainable computing isn't just
  - Better power management
  - More efficient gates ...
- Use UTs to ***reduce power/computation***
- **Parallel Bit Pattern Computing** might work
  - C laptop prototype: 32-way, ≥1024 **`pbit`**
  - Lots to fully implement & improve: architecture, C++ wrappers, **`pfloat`**, etc.

Aggregate.Org UNBRIDLED COMPUTING

University of Kentucky

An Example: Find `sqrt(29929)`

- Only **159** single-gate operations for BitC:



University of Kentucky