How Low Can You Go?

The key to high performance is no longer finding and using parallelism, but maximizing the return on investment of power. Our goal is to ***minimize the total number of gate-level operations required*** while still obtaining speedup through parallel execution – which requires rethinking language concepts, compiler optimization and parallelization, and hardware architecture.

Most programming languages treat data objects as indivisible, atomic, entities. Programmers typically specify both type and size of each datum, but the compiler should save power by **not using all the bits, not all the time.**

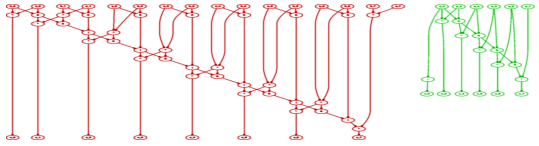
Integer range analysis: most programmers and languages are either lazy or paranoid about specifying how big an integer must be. How big is an `int`? Do you really use all 2^{32} values for things declared as 32-bit integers? Do you declare non-negative variables `unsigned`? Why not allow specifications like `int:8` for an 8-bit integer and also use compiler range analysis to minimize active bits?

Floating point accuracy, not precision: programs specify ***precision***, but what we want is ***accuracy***. As we proposed in 2006, why not have language constructs that provide an accuracy acceptance test so the compiler can speculatively define precision, automatically re-executing with higher precisions as needed? Use of LNS (Log Number System) or scaled integers is also possible.

Packing of smaller data: originally, we did this for **SWAR** (SIMD Within A Register) to obtain vector-like parallelism, but techniques like our **CSI** (Common Subexpression Induction) allow a compiler to pack unstructured things to more efficiently use datapaths and memories.

Over time, machine word size has grown. From the **EDSAC 2** in 1958, microcoded bit slicing was used in many computers, including the massively-parallel **DAP**, **STARAN**, **MPP**, **CM**, **CM2**, and **GAPP**. Over time, slice width increased; the **MP-1** was 4-bit, and GPUs went from 8 to 32 or even 64-bit words. This was done to speed sequential code – assuming not enough parallelism is available. Now, sequential code is handled elsewhere and there is lots of parallelism available. **Bit-level operations require fewer active gates per computation.**

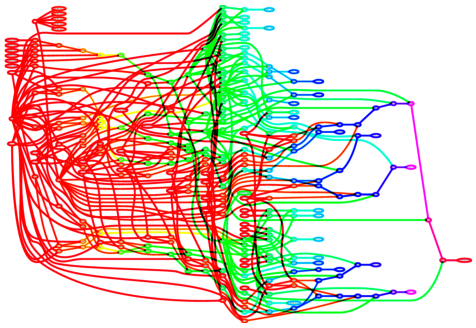
Consider executing 32 ***ripple carry*** 32-bit additions in 32 clock cycles using just 32 one-bit full adders. To execute one 32-bit addition in one clock cycle requires much more active circuitry (e.g., to implement ***carry lookahead***), more power, and probably a slower clock. Even greater savings are possible by optimizing bit-level operations rather than microcoding a simple word-level ISA (as bit-slice processors generally did). For example, consider an 8-bit add vs. what is truly needed to add 4 to an 8-bit number:



Optimizing whole programs at the gate level is hard, but our **BitC** langauge and compiler for mux-based nanocontrollers proved it is possible using our **MSC** (Meta-State Conversion), **BDD** (Binary Decision Diagram) normalization, and other optimizations. The following BitC code would become 206,669 mux operations implementing each word-level operation directly, but instead the compiler uses just 8 instructions to set each bit of `a` to 0:

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

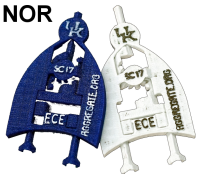
In Spring 2017, we built a C compiler that generates optimized gate-level hardware as a single combinatorial logic circuit with a state variable. A trivial example is:



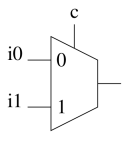
```
int:8 a=10;
int:8 b=3;
int:8 c;

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

Initial targets included **Verilog** code and **GPU**s. We are looking at various new targets, such as **adiabatic** (thermodynamically reversible) logic. **CSWAP** (aka, **FredKin**) gates are adiabatic gates that do not allow fanout, making them a difficult target, but a design expressed using them is efficiently executable by a quantum computer....

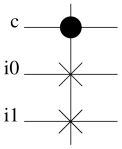
NOR



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

```
@techreport{sc17gates,
author={Henry Dietz},
title={{How Low Can You Go?}},
institution={University of Kentucky},
address={http://aggregate.org/WHITE/sc17gates.pdf},
month={Nov}, year={2017}}
```

SC17 Denver, CO hpc connects.

Aggregate.Org UNBRIDLED COMPUTING