# Wordless Integer and Floating-Point Computing

Henry Dietz

*LCPC, 9:30-10:00 October 14, 2022*

University of Kentucky
Electrical & Computer Engineering

LCPC 2022

Aggregate.Org UNBRIDLED COMPUTING

University of Kentucky

# Word.

*"A basic unit of data in a computer, typically 16 or 32 bits long"*

# Word.

*“A basic unit of data in a computer, typically 16 or 32 bits long”*

*“Exclamation used to express agreement”*

University of Kentucky

# Word.

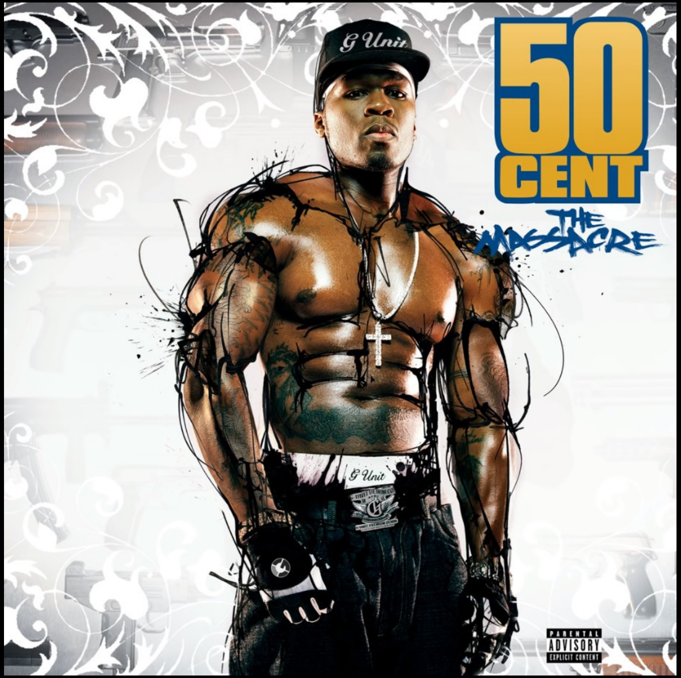
*“A basic unit of data in a computer, typically 16 or 32 bits long”*

*“Exclamation used to express agreement”*

Well, I *don’t* agree: THE basic unit is a **bit**

University of Kentucky

# All I really need is a lil bit...




University of Kentucky

# LCPC 2017:
# *How Low Can You Go?*

- Now it's all about power / computation
- Work only on active bits (bit-serial)
- Aggressive gate-level optimization
- Potential exponential benefit from Quantum?

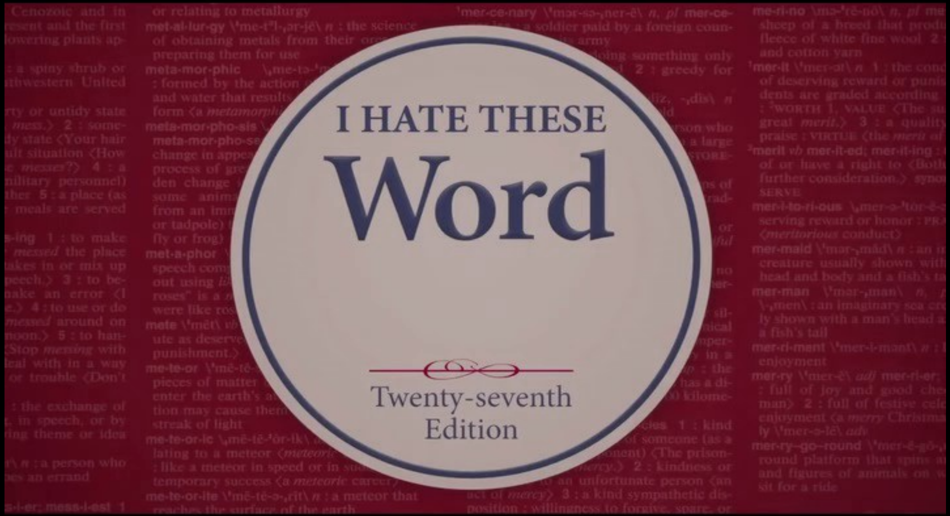
University of Kentucky

# LCPC 2022:
# *How Low Do We Go Now?*

- Now it's all about power / computation – Yes!
- Work only on active bits (bit-serial) – Yes!
- Aggressive gate-level optimization – Yes!
- Potential exponential benefit from Quantum?
  – No, but **SIMD** using **Parallel Bit Pattern**...

University of Kentucky

# I hate these word crimes!


University of Kentucky

# Reduce gates / word operation

```
int a,b,c; c=a+b;
```

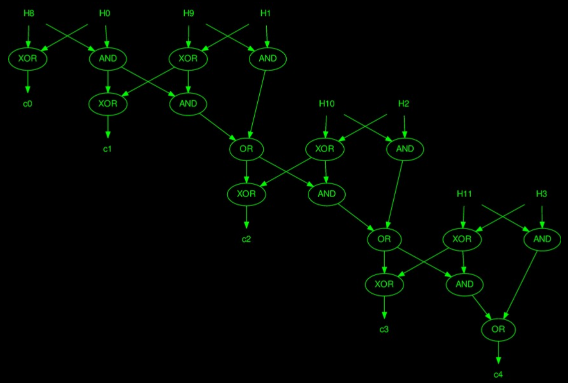
- Fast 32-bit Carry Lookahead:
  ~645 gate actions, ~12 gate delays
- 32-bit Ripple Carry, get *throughput* by **SIMD**:
  ~153 gate actions, ~91 gate delays (3 per FA)

University of Kentucky

# Only operate on active bits

`int:4 a,b; int:5 c; c=a+b;`

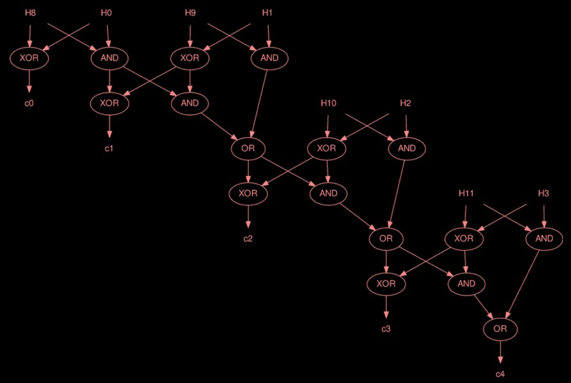
- 32-bit Ripple Carry:
  ~153 gate actions
- 4-bits active Ripple Carry:
  17 gate actions

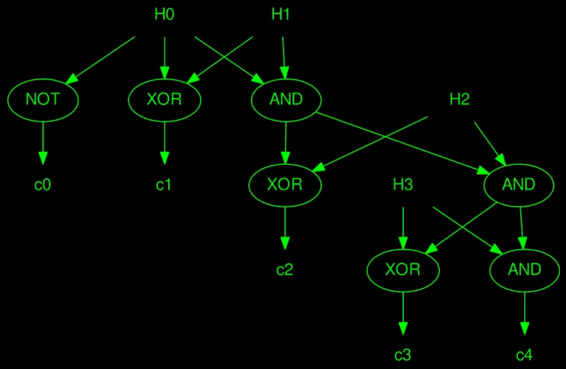


# Gate-level optimization

```
int:4 a,b; int:5 c; b=1; c=a+b;
```

- 17 gates becomes 7 when optimized




University of Kentucky

# Minimizing Number of Bits

- Use types like `uint8_t` instead of `int`
- Use compiler analysis to infer types (done as early as 1964 Klerer-May System)
- Specify accuracy requirements rather than precision for floating-point
- Pack smaller representations into fixed-size memory locations or registers

# Minimizing Gate-Level Ops

- Bit-slice hardware
- Bit-serial processing in SIMD supercomputers
  - DAP, STARAN, MPP, CM1/CM2, GAPP, …
  - 32 SIMD 1-bit full adders vs. one 32-bit adder: same throughput over 32 clocks uses fewer gates and a much faster clock

University of Kentucky

# No more words!


University of Kentucky

# Our approach

- Wordless integer & floating-point variables†
- Dynamic optimization at the bit level using the **Parallel Bit Pattern** (PBP) execution model
- C++ classes & preliminary performance results

† We still use words for scalars.

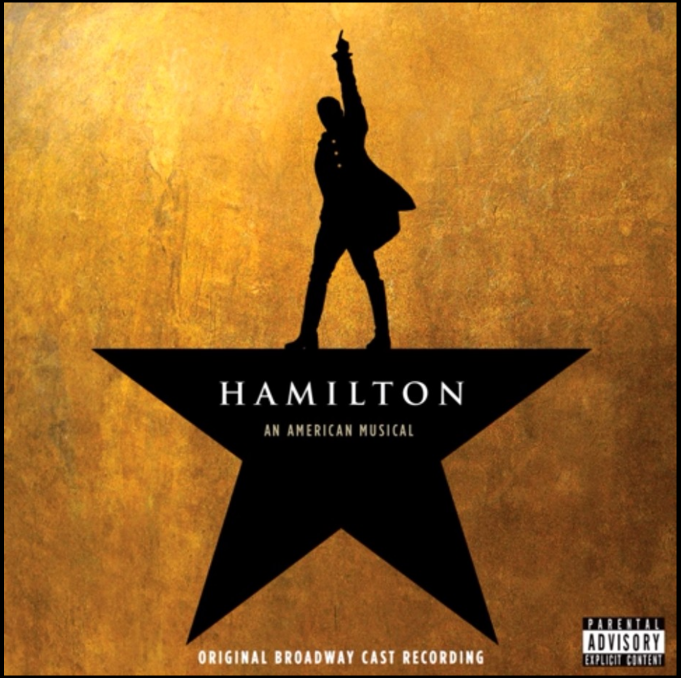
University of Kentucky

# Dynamic precision

- There are languages with *static* bit precisions: Verilog, VHDL, and even C `struct` bitfields
- There are *dynamic* “Big Number” libraries: GMP, BigDigits, ArPALib, etc.
- We want ***dynamic precision at the bit level***…

University of Kentucky

# That would be enough




University of Kentucky

# Dynamically resizing an `int`

- Suppose an `int` has the value 4:
  4 is an unsigned 3-bit integer, `100`
- Now decrement to the value 3:
  3 is an unsigned 2-bit integer, `11`
- Take 2's complement to make the value -3:
  2's complement of X is (~X)+1, so...
  -3 is a signed 3-bit integer, `101`

# Pattern (PBP) `int`: `pint`

- Ordered set of $k$ bit positions, $b_{k-1}, b_{k-2}, \ldots, b_1, b_0$
- Each $b_i$ corresponds to a bit-index value, $X_i$, across `nproc` SIMD PEs as PE[`iproc`].mem[$X_i$]

```
bool has_sign;       //signed?
uint8_t prec;        //k
pbit bit[PINTBITS];  //X
```

University of Kentucky

# Manipulating `pint` precision

- *Iff* PE[`iproc`].mem[$X_{k\text{-}1}$]==PE[`iproc`].mem[$X_{k\text{-}2}$] ∀ `iproc`, then $X_{k\text{-}1}$==$X_{k\text{-}2}$ and bit $k$-1 is redundant
- Primitive precision operations:
  - `pint Minimize() const;`
  - `pint Extend(const int p) const;`
  - `pint Promote(const pint& b) const;`

University of Kentucky

# Pattern (PBP) `float`: `pfloat`

- **Sign**: a one-`pbit` `pint`, 0 if non-negative
- **Exponent**: `pint` power-of-2 multiplier
  - No fixed minimum/maximum value
  - No bias nor reserved values
- **Mantissa**: `pint` fractional part
  - Fixed maximum precision
  - No implicit leading 1

University of Kentucky

# pfloat denormals

Mantissa==0 is exempt from normalization

**Table 1.** The pfloat value representations not subject to normalization.

| Decimal Value | Sign | Exponent | Mantissa (8 bit precision) |
|---|---|---|---|
| 0.0 | 0 | 0 | 0 |
| NaN | 1 | 0 | 0 |
| Infinity | 0 | 1 | 0 |
| Negative Infinity | 1 | 1 | 0 |

University of Kentucky

# `pfloat` normalization rules

**Table 2.** Some pfloat value representations, MSB normalized.

| Decimal Value | Sign | Exponent | Mantissa (8 bit precision) |
|---|---|---|---|
| 1.0 | 0 | 0 | 10000000 |
| 2.0 | 0 | 1 | 10000000 |
| 5.0 | 0 | 10 | 10100000 |
| 0.5 | 0 | −1 | 10000000 |
| -42.0 | 1 | 101 | 10101000 |

**Table 3.** Some pfloat value representations, LSB normalized.

| Decimal Value | Sign | Exponent | Mantissa (8 bit maximum precision) |
|---|---|---|---|
| 1.0 | 0 | 0 | 1 |
| 2.0 | 0 | 1 | 1 |
| 11.0 | 0 | 0 | 1011 |
| 0.5 | 0 | −1 | 1 |
| -42.0 | 1 | 1 | 10101 |

University of Kentucky

# Runtime optimizations


University of Kentucky

# Compiler-like optimizations

- Primarily done on `pbit` descriptors, which are always unique (*single assignment*)
- **Constant folding**: 0, 1 are descriptors 0, 1
- **Algebraic simplifications**: 42 AND 1 $\Rightarrow$ 42
- **Common subexpressions**: applicative cached

University of Kentucky

# Optimizations *across* PEs

- A **classical SIMD** idles “disabled” PEs
- A **GPU** can skip “all disabled” ***warps*** of PEs
- **Bit-serial SIMD using PBP** can:
  - Like GPU, skip “all disabled” ***chunks*** of PEs
  - Skip chunk compuations that have been performed before on ***any*** chunk of PEs
  - Examine global chunk properties

University of Kentucky

# An example of chunk handling

- For **`nproc`**=32, **`iproc`** is:
  (apparently 5x8x4=160 bits to store)
- For 8-bit chunks, this is:
  (only 5 chunks used, so **just 5x8=40 bits stored**)
- To add 1 to **`iproc`**, we add:
  (**only chunk operations *with unique operands* happen**)

```
10101010 10101010 10101010 10101010
11001100 11001100 11001100 11001100
11110000 11110000 11110000 11110000
11111111 00000000 11111111 00000000
11111111 11111111 00000000 00000000
```

```
chunk(2) chunk(2) chunk(2) chunk(2)
chunk(3) chunk(3) chunk(3) chunk(3)
chunk(4) chunk(4) chunk(4) chunk(4)
chunk(1) chunk(0) chunk(1) chunk(0)
chunk(1) chunk(1) chunk(0) chunk(0)
```

```
chunk(1) chunk(1) chunk(1) chunk(1)
```

University of Kentucky

# Putting a dream into action


University of Kentucky

# Initial implementation

- PBP library for `pint` and `pfloat` classes
  - PBP was lazy C; now 3,644 lines eager C++
  - `pint` operations include: all the usual C++ operators; value range initialization, scatter & gather; reductions & scans; sorts
  - `pfloat` operations *also* include: reciprocal and various transcendentals (exponentiation, logarithm, sine, etc.)
- Targets 32/64-bit processors, up to **4G** 1-bit PEs

# Preliminary performance

- Surprisingly competitive with native code
- Instrumented active gate counts for `pint` library validation suite as words vs. PBP model:

**Table 4.** Active gate counts for 32-bit word operations vs. proposed PBP model.

| *nproc* | Chunk bits | Gates (Words) | Gates (PBP) | Ratio |
|---|---|---|---|---|
| 65536 | 256 | 12279113318 | 3209523 | 3826:1 |
| 262144 | 256 | 55522282700 | 3141452 | 17674:1 |
| 262144 | 512 | 55520002048 | 6563379 | 8459:1 |
| 1048576 | 256 | 252845228032 | 3135360 | 80643:1 |
| 1048576 | 1024 | 252876370739 | 13902438 | 18189:1 |
| 4194304 | 2048 | 1154496017203 | 29179904 | 39565:1 |
| 16777216 | 4096 | 5277432676352 | 61104947 | 86366:1 |
| 67108864 | 8192 | 24153849174425 | 128459571 | 188027:1 |

University of Kentucky

# Better than words!


University of Kentucky

# Conclusion

- Preliminary results are very promising: **4-6 orders of magnitude** reduction in active gates!
- Bit-serial SIMD PBP has great potential, but we **need PBP hardware** to test power reduction
- Currently, no garbage collection on chunks...