Parallel Bit Pattern (part 1)

EE599-001 & EE699-010, Spring 2026

Hank Dietz

http://aggregate.org/hankd/

University of Kentucky

To pbits And Beyond

- In part 0, we explained how an AoB can represent an entangled superposition…
  and how layers up to RE avoid bit-level work
- The layers here are the programming interface presented by a **C++ library**
  - `pbit` – a dynamically managed RE reference
  - `pint` – variable precision, signed/unsigned
  - `pfloat` – variable precision (accuracy)

The C++ Library: `pbp.h`

- Any C++ program can `#include <pbp.h>`
  - **Warning**: research-grade implementation!
  - Defines `pbit`, `pint`, and `pfloat` classes
  - These classes function like built-in types with operators overloaded to handle them
- Library code lowers to optimized AoB ops
  - Currently, entirely done at runtime (like qiskit)
  - Actual execution is implicitly triggered when needed, some versions use lazy evaluation

The C++ Library Parameters

- Library is highly portable
  - 32/64 bit processors, SWAR support
  - For ESP32, includes C++ subset interpreter
- Key library parameters:
  - **`REWAYS`**: max ways entangled (≤32)
  - **`AOBWAYS`**: ways per chunk (5..9)
  - **`REREGS`**: how many RE registers
  - **`AOBREGS`**: how many AOB registers
  - **`ACSIZE`**: size of applicative cache

Using the PBP library

The PBP library is written as highly portable, self-contained, C++ code. All that is needed to use it is inclusion of the header file with **`REWAYS`** set to the desired maximum entanglement (default 10).

```
#include "pbp.h"
```

Sample `pint` Layer Algorithms

It is easy to compute the square root of an 8-bit number by exhaustive search. For example, **`sqrt(169)`** will find 13.

```
void pintsqrt(int val){
 pint a(val); // 8-bit number
 pint b = H(4); // all possible roots
 pint c = (b * b); // square them
 pint d = (c == a); // select answer
 int pos = d.First();
 printf("Square root of %d is %d\n",
  val, pos);
}
```

A less obvious algorithm factors an 8-bit number. Here, possible 4-bit factors are assigned different entanglement channel sets so the multiply produces an 8-way entangled answer rather than 4-way. For example, **`factor(143)`** will find 11 and 13.

```
#include "pbp.h"

void pintfactor(int val) {
 pint a(val); // 8-bit number
 pint b = H(4,0x0f); // 4-bit
 pint c = H(4,0xf0); // 4-bit
 pint d = b * c; // multiply 'em
 pint e = (d == a); // which were val?
 pint f = e * b; // zero non-answers
 int spot = f.First(); // factors
 int one = c.Meas(spot);
 int two = b.Meas(spot);
 printf("%d, %d are factors of %d\n",
  one, two, val);
}
```

As above, algorithms written for PBP tend to use abilities that quantum computers do not have, most notably entanglement channel-based operations and the fact that measurement is not destructive. *PBP also can be used for traditional SIMD computation.*

Sample `pbit` Layer Algorithm

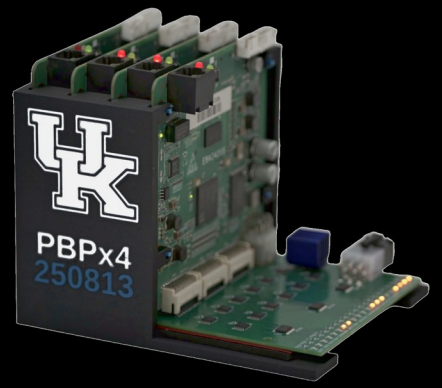
There is little point in directly using the **`pbit`** layer for PBP programs. However, quantum computer algorithms at the **Qubit** level can be programmed using the **`pbit`** layer. The following is a 4-bit ripple carry adder, adding 1 to all 4-bit values, as per **Cuccaro** et al, **`arXiv:quant-ph/0410184v1`**

```
void pbitripple() {
 pbit a0(0), a1(0), a2(0), a3(0);
 pbit b0(1), b1(0), b2(0), b3(0);
 pbit z(0), x(0);
 H(a0, 0); // unlike Qubits,
 H(a1, 1); // must specify groups of
 H(a2, 2); // entanglement channels
 H(a3, 3); // for Hadamard gates
 CNOT(a1,b1); CNOT(a2,b2);
 CNOT(a3,b3); CNOT(a1,x);
 CCNOT(a0,b0,x); CNOT(a2,a1);
 CCNOT(x,b1,a1); CNOT(a3,a2);
 CCNOT(a1,b2,a2); CNOT(a3,z);
 CCNOT(a2,b3,z); NOT(b1);
 NOT(b2); CNOT(x,b1);
 CNOT(a1,b2); CNOT(a2,b3);
 CCNOT(a1,b2,a2);
 CCNOT(x,b1,a1);
 CNOT(a3,a2); NOT(b2);
 CCNOT(a0,b0,x); CNOT(a2,a1);
 NOT(b1); CNOT(a1,x);
 CNOT(a0,b0); CNOT(a1,b1);
 CNOT(a2,b2); CNOT(a3,b3);
 SETMEAS(); // pick random channel
 printf("a=%d b=%d\n",
  MEAS(a0)+(MEAS(a1)<<1) +
  (MEAS(a2)<<2)+(MEAS(a3)<<3),
  MEAS(b0)+(MEAS(b1)<<1)+
  (MEAS(b2)<<2)+(MEAS(b3)<<3));
}
```

PBP References (oldest & FPGA)

H. Dietz, "**How Low Can You Go?**," In: Rauchwerger, L. (eds) Languages and Compilers for Parallel Computing. LCPC 2017. Lecture Notes in Computer Science(), vol 11403. Springer. 10.1007/978-3-030-35225-7_8

H. Dietz, P. Eberhart and A. Rule, "**Basic Operations And Structure Of An FPGA Accelerator For Parallel Bit Pattern Computation**," 2021 International Conference on Rebooting Computing (ICRC), 2021, pp. 129-133. 10.1109/ICRC53822.2021.00029



Parallel Bit Pattern Computing

C++ Library version 251114

`http://aggregate.org/PBP`

Professor Henry (Hank) Dietz
Electrical and Computer Engineering Department
University of Kentucky
Lexington, KY 40506-0046
hankd@engr.uky.edu

Parallel bit pattern computing is a quantum-inspired model of computation. **Superposition** and ***n*-way entanglement** are modeled by each **`pbit`** (pattern bit) having an ordered set of 2^n single-bit values. Each position in the ordered set is an ***entanglement channel***. E.g., the 2-way entangled **`pbit`** values {0,1,1,1} and {0,1,0,1} could represent {0,3,2,3}, with probabilities of 25% 0, 25% 1, and 50% 3. These ordered bit sets are not directly stored, but encode as compressed patterns, with duplicate sub-patterns factored. Applicative caching avoids recomputation of sub-pattern operations. Overall, PBP can exponentially reduce both memory footprint and total number of gate-level operations.

Unlike quantum systems, users are encouraged to program parallel bit pattern computations at a relatively high level. This **CC BY 4.0** C++ library provides automatically-managed pattern bits (**`pbit`**) and variable-precision integer (**`pint`**) layers. Compiler optimizations are applied dynamically at runtime to further simplify the bit-level operations.

`pint` Layer

A pattern integer, or **`pint`**, is an array of 1-32 **`pbit`** treated as a signed/unsigned integer. The precision and signedness of **`pint`** are variable at runtime, so that the minimum possible number of bits are active.

The usual C/C++ operators work as expected on **`pint`** values. Simple assignments between **`int`** and **`pint`** convert; other conversions must be explicit.

- **`pint()`**, **`pint(v)`**, **`pint(v,p)`**
  Create a **`pint`** initialized to an integer value: NaN, the **`int`** value ***`v`***, or ***`v`*** with precision ***`p`***
- **`H(w)`**, **`H(w,m)`**
  Create a **`pint`** **Hadamard** pattern ***`w`*** ways entangled using entanglement channels specified by mask ***`m`***
- ***`p`*****`.Valid()`**
  True *iff* **`pint`** ***`p`*** has a valid value (is not NaN)
- ***`p`*****`.Minimize()`**
  Create value of ***`p`*** with fewest **`pbit`** possible
- ***`p`*****`.Extend(b)`**
  Create value of ***`p`*** with ***`b`*** **`pbit`** precision
- ***`p`*****`.Promote(q)`**
  Create value of ***`p`*** with minimum **`pbit`** precision that covers both ***`p`*** and ***`q`*** values and signedness
- ***`p`*****`.Logic()`**
  Create **`pint`** with single **`pbit`** logic value of ***`p`***
- ***`p`*****`.Rot(e)`**
  Create value of ***`p`*** rotated by ***`e`*** entanglement channels (a simple phase shift)
- ***`p`*****`.Reset(e)`**, ***`p`*****`.Set(e)`**
  Create value of ***`p`*** with entanglement channel ***`e`*** reset or set
- ***`p`*****`.Dom(e)`**
  Create value of ***`p`*** with bits dominoed (inverted) from entanglement channel ***`e`*** downward
- ***`p`*****`.Meas(e)`**, ***`p`*****`.Meas()`**, ***`i=p`***
  Create **`int`** value of ***`p`*** from entanglement channel ***`e`*** or a random sample
- ***`p`*****`.First()`**
  Create **`int`** value of first entanglement channel in ***`p`*** that holds a 1; returns 2^{ways} if none
- ***`p`*****`.Ones()`**
  Create **`int`** value number of entanglement channels in ***`p`*** that holds a 1
- ***`p`*****`.Min(q)`**, ***`p`*****`.Max(q)`**
  Create **`pint`** with minimum/maximum value from ***`p`*** or ***`q`*** for each entanglement channel
- ***`p`*****`.Abs()`**
  Create **`pint`** with absolute value of ***`p`***
- ***`p`*****`.Signed()`**, ***`p`*****`.UnSigned()`**
  Create **`pint`** forcing signed/unsigned interpretation of the **`pbits`** in ***`p`***
- ***`p`*****`.Mul(q)`**, ***`p`*****`.Mul(q,b)`**
  Create **`pint`** product of ***`p`*** and ***`q`***, but limit result precision to ***`b`*** **`pbits`** to save effort
- ***`p`*****`.Any()`**, ***`p`*****`.All()`**
  Create **`int`** value that is 1 *iff* any/all entanglement channels in ***`p`*** are non-zero
- ***`p`*****`.Summary()`**, ***`p`*****`.Show()`**
  Print debugging info for **`pint`** ***`p`*** value: either **`pbit`** summary or complete bit patterns

`pbit` Layer

A pattern bit, or **`pbit`**, is logically a vector of 2^{ways} bits, but is generally stored and operated upon in a heavily compressed form – a 32-way entangled **`pbit`** can take as little as 16 bits of storage space. A **`pbit`** is similar to a **Qubit** in a quantum computer, but **`pbit`** values are automatically allocated, maintain their value forever, and allow arbitrary fan-out; thus, they are not restricted to reversible gate operations. The basic operations include:

- **`pbit()`**, **`pbit(v)`**
  Create a **`pbit`** initialized to NaN or **`pbit`** register ***`v`***: 0 is 0, 1 is 1, 2 is **H0**, 3 is **H1**, etc.
- ***`p`*****`.Valid()`**
  True *iff* **`pbit`** ***`p`*** has a valid value (is not NaN)
- ***`p`*****`.And(q)`**, ***`p`*****`.Or(q)`**, ***`p`*****`.Xor(q)`**, ***`p`*****`.Not()`**
  Bitwise AND, OR, XOR, and NOT
- ***`p`*****`.Rot(e)`**, ***`p`*****`.Flip(a)`**
  Phase operators; ***`p`*** rotated by ***`e`*** entanglement channels or dimensionally flipped on ***`a`***
- ***`p`*****`.Reset(e)`**, ***`p`*****`.Set(e)`**, ***`p`*****`.Tog(e)`**
  Create value of ***`p`*** with entanglement channel ***`e`*** reset, set, or toggled
- ***`p`*****`.Dom(e)`**
  Create value of ***`p`*** with bits dominoed (inverted) from entanglement channel ***`e`*** downward
- ***`p`*****`.Meas(e)`**, ***`p`*****`.Meas()`**
  Create **`int`** 0/1 value of ***`p`*** from entanglement channel ***`e`*** or a random sample
- ***`p`*****`.First()`**
  Create **`int`** value of first entanglement channel in ***`p`*** that holds a 1; returns 2^{ways} if none
- ***`p`*****`.Ones()`**
  Create **`int`** value number of entanglement channels in ***`p`*** that holds a 1
- ***`p`*****`.Any()`**, ***`p`*****`.All()`**
  Create **`pbit`** value that is 1 *iff* any/all entanglement channels in ***`p`*** are non-zero
- ***`p`*****`.Show()`**
  Print debugging info for **`pbit`** ***`p`*** value: complete bit patterns

The following **`pbit`** operations are provided solely for porting Qubit-level quantum algorithms:

- **`NOT(q)`**
  Pauli X gate; replaces ***`q`*** with ***`~q`***
- **`CNOT(c,t)`**
  Controlled not gate; where ***`c`***, replaces ***`t`*** with ***`~t`***
- **`CCNOT(a,b,c)`**
  Toffoli gate; where ***`a`*** and ***`b`***, replaces ***`c`*** with ***`~c`***
- **`SWAP(i0,i1)`**
  Swap values of ***`i0`*** and ***`i1`***
- **`CSWAP(c,i0,i1)`**
  Fredkin gate; where ***`c`***, swap ***`i0`*** and ***`i1`***
- **`H(q,c)`**
  Hadamard gate; replaces ***`q`*** with ***`q`*** ^ **Hadamard** entanglement pattern ***`c`***
- **`SETMEAS()`** and **`SETMEAS(m)`**
  Set measurement of **`rand()`** channel or ***`m`***
- **`MEAS(q)`**
  Measure and collapse state of ***`q`***, returns 0/1

`RE`, `AC`, and `AoB` Layers

The Regular Expression, Applicative Caching, and Array-of-Bits layers are not described here; they are considered internal, and my be changed without notice. Although the **`pint`** and **`pbit`** layers dramatically reduce gate-level operations per computation, these lower layers provide up to exponential reduction in both gate operations and in storage requirements. Performance of these layers can be summarized by calling **`re.Stats()`**.

The `pbit` Layer

- `pbit p;` allocates `p` and initializes to NaN
- `pbit p(0);` allocates `p` and initializes to 0
- `NOT(p);` applies Pauli X to `p`
- `CNOT(c,t);` where `c`, `t=~t`
- `CCNOT(a,b,c);` where `a` AND `b`, `c=~c`
- `SWAP(i0,i1);` swap values of `i0` and `i1`
- `CSWAP(c,i0,i1);` where `c`, swap values
- `H(p,c);` replace `p` with `p^H` pattern `c`
- `SETMEAS(m);` measure entanglement chan. `m`
- `MEAS(p);` measure chan. `m` of `p`, collapse `p`

The `pbit` Layer

- `p.Valid()` checks for not NaN
- `p.And(q)`, `p.Or(q)`, `p.Xor(q)`, `p.Not()`
- `p.Rot(e)` rotates `e` channels
- `p.Flip(a)` flips dimensions spec'd by `a`
- `p.Reset(e)`, `p.Set(e)`, `p.Tog(e)`
- `p.Dom(e)` domino channels from `e` downward
- `p.Meas(e)`, `p.Meas()`
- `p.First()` lowest channel with value 1
- `p.Ones()` population count of 1 values in `p`
- `p.Any()`, `p.All()`, `p.Show()`

4-pbit Quantum Adder

```
void
pbitripple()
{
 // 4-bit wide pbitripple-carry adder per Cuccaro et al arXiv:quant-ph/0410184v1
 pbit a0(0), a1(0), a2(0), a3(0), b0(1), b1(0), b2(0), b3(0), z(0), x(0);
 H(a0, 0); H(a1, 1); H(a2, 2); H(a3, 3);
 CNOT(a1,b1); CNOT(a2,b2);
 CNOT(a3,b3); CNOT(a1,x);
 CCNOT(a0,b0,x); CNOT(a2,a1);
 CCNOT(x,b1,a1); CNOT(a3,a2);
 CCNOT(a1,b2,a2); CNOT(a3,z);
 CCNOT(a2,b3,z); NOT(b1);
 NOT(b2); CNOT(x,b1);
 CNOT(a1,b2); CNOT(a2,b3);
 CCNOT(a1,b2,a2);
 CCNOT(x,b1,a1);
 CNOT(a3,a2); NOT(b2);
 CCNOT(a0,b0,x); CNOT(a2,a1);
 NOT(b1); CNOT(a1,x);
 CNOT(a0,b0); CNOT(a1,b1);
 CNOT(a2,b2); CNOT(a3,b3);
 SETMEAS();
 printf("a=%d b=%d\n",
        MEAS(a0)+(MEAS(a1)<<1)+(MEAS(a2)<<2)+(MEAS(a3)<<3),
        MEAS(b0)+(MEAS(b1)<<1)+(MEAS(b2)<<2)+(MEAS(b3)<<3));
}
```

Notes

- Phase kickback doesn't happen here for the same reason that measurement need not collapse a superposition… but PBP does have phase operations
- In a quantum computer, measurement is of a random entanglement channel; here, the channel is selected as `m` in `SETMEAS(m);` and should be consistently used after selection

Unphased

- How do we implement Deutsch's Algorithm without Phase kickback?
- E.g., use `Ones()` to count 1s in superposition; same count as H(0) means balanced

```
pbit x(0); H(0, x);
pbit y(0); H(1, y);
CNOT(x, y);
cout << (y.Ones()==x.Ones()) << "\n";
```

The `pint` Layer

- A `pint` is automatically dynamically sized
  - Superposition with 1<<`REWAYS` int values
  - Precision is 1..`REWAYS` pbits
  - Signed, but only magnitude if non-negative
- `p.Minimize()` minimizes pbits for `p`'s value
- `p.Logic()` makes `p` have just 1 pbit
- `p.Extend(b)` makes `p` have `b` pbits
- `p.Promote(q)` makes `p` big enough to hold `q`
- `p.Valid()` false if NaN

The `pint` Layer

- All the usual C++ `int` operators are overloaded
- `H(w)`, `H(w,mask)` create `w`-way H pattern
- `p.Rot(e)`, `p.Flip(a)`, etc. as for `pbit`
- `p.Min(q)`, `p.Max(q)` min/max in each channel
- `p.Abs(q)` make unsigned value
- `p.Signed(e)`, `p.UnSigned()` treat pbits as...
- `p.Mul(q)` the multiply used for *
- `p.Mul(q,b)` multiply constrained to `b` pbits out
- `p.Summary()`, `p.Show()`

The `pint` Layer

- `p.Reduce`***`Op`***`()` reduces superposition to scalar
- `p.Scan`***`Op`***`()` parallel-prefix of superposition
- `p.Scatter(ip,n,e)` converts an array into entangled superposition
- `p.Cover(min,max,dim)` make superposition that covers the specified range
- `p.Range(min,max,dim)` make superposition that tightly covers, with 0 padding
- `p.Gather(ip,n)` converts into an array
- `p.SortUp()`, `p.SortDown()` bitonic sort

`pint` Square Root

- Square root by squaring…
  For example, `pintsqrt(169)` is `13`

```
void
pintsqrt(int val)
{
 // Compute square root of val
 pint a(val); // 8-bit number
 pint b = pint(0).Had(4); // possible sqrts
 pint c = (b * b); // square them
 pint d = (c == a); // which were 169?
 int pos = d.First(); // first non-0 is answer
 printf("Square root of %d is %d\n", val, pos);
}
```

`pint` Prime Factor

- Prime factoring by multiply…
  For example, `pintfactor(143)` is `11` and `13`

```
void
pintfactor(int val)
{
 // Factor val
 pint a(val); // 8-bit number
 pint b = pint(0).Had(4); // 4-bit possible 1st factor
 pint c = pint(0).Had(4,4); // 4-bit possible 2nd factor
 pint d = b * c; // multiply 'em
 pint e = (d == a); // which were 143?
 pint f = e * b;
 int spot = f.First(); // factors
 int one = c.Meas(spot);
 int two = b.Meas(spot);
 printf("%d, %d are factors of %d\n", one, two, val);
}
```

The `pfloat` Layer

- **Buggy!**
- A `pfloat` contains three `pint` values:
  - Sign is always a single `pbit`
  - Exp is allowed to grow/shrink as needed
  - Frac with choice of two normalization rules: to MSB or to LSB
- Standard C++ `float` operators and library functions (e.g., `p.ArcTan()`)
- All the `pint` extensions

π in C using `int` sampling

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
 int i, j, intervals = atoi(argv[1]);
 int w = intervals * intervals, sum = 0;
 for (i=0; i<intervals; ++i) {
  int h = w / (intervals + ((i*i) / intervals));
  for (j=0; j<intervals; ++j) {
   if (h > j) ++sum;
  }
 }
 printf("Pi is roughly %f\n", (4.0 * sum) / w);
 return(0);
}
```

π in PBP, `pint` sampling

```
#include "pbp.h" // PBP classes and support

int main(int argc, char **argv) {
 int bits = atoi(argv[1]); // number of pbits
 pint intervals(1 << bits); // intervals in pbits
 pint w(1 << (2 * bits)); // int scaling factor
 pint x = pint(0).Had(bits); // all x values
 pint y = pint(0).Had(bits,bits); // all y values
 pint h = w / (((x * x) >> bits) + intervals);
 pint r = (h > y); // r is 1 where below curve
 // count 1s; quantum would sample probability
 double pi = (4.0 * r.Pop()) / (1 << REWAYS);
 printf("Pi is roughly %f\n", pi);
 return(0);
}
```