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

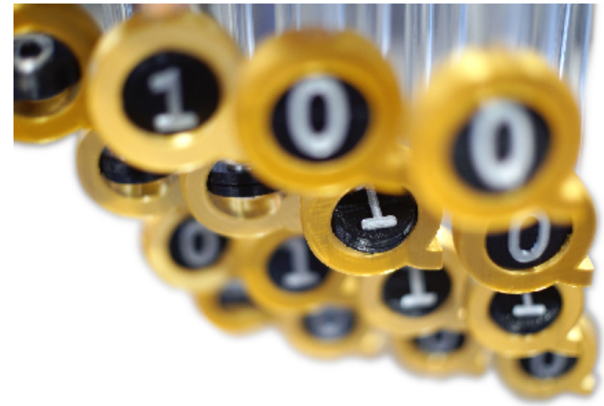
There is little point in directly using the **pbit** layer for PBP programs. However, quantum computer algorithms at the **Qubit** level can be programmed using the **pbit** layer. The following is a 4-bit ripple carry adder, adding 1 to all 4-bit values, as per **Cuccaro** et al, **arXiv:quant-ph/0410184v1**

```
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 & newest)

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 20220704

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()`** and **`pint(v)`** and **`pint(v,p)`**
  Create a **`pint`** initialized to an integer value: NaN, the **`int`** value ***v***, or ***v*** with precision ***p***
- **`H(w)`** and **`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)`** and **`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)`** and **`p.Meas()`** and **`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)`** and **`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()`** and **`p.UnSigned()`**
  Create **`pint`** forcing signed/unsigned interpretation of the **`pbits`** in ***p***
- **`p.Mul(q)`** and **`p.Mul(q,b)`**
  Create **`pint`** product of ***p*** and ***q***, but limit result precision to ***b*** **`pbits`** to save effort
- **`p.Any()`** and **`p.All()`**
  Create **`int`** value that is 1 *iff* any/all entanglement channels in ***p*** are non-zero
- **`p.Summary()`** and **`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()`** and **`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.Rot(e)`**
  Create value of ***p*** rotated by ***e*** entanglement channels (a simple phase shift)
- **`p.Reset(e)`** and **`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)`** and **`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()`** and **`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()`**.