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 number of entanglement dimensions.

```
#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 = pint(0).Had(4); // dim 0-3
 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 = pint(0).Had(4); // dim 0-3
 pint c = pint(0).Had(4,4); // dim 4-7
 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

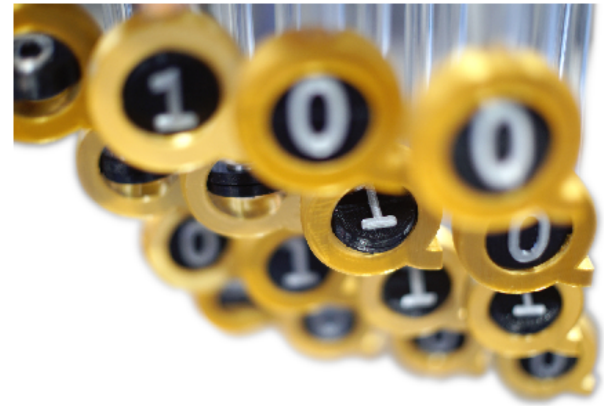
There is little point in directly using the **pbit** layer for PBP programs. However, quantum computer algorithms at the **Qubit** level can be programmed using the **pbit** layer. The following is a 4-bit ripple carry adder, adding 1 to all 4-bit values, as per **Cuccaro** et al, **arXiv:quant-ph/0410184v1**

```
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 20220730

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. provides automatically-managed pattern bits (**pbit**) as well as variable-precision pattern integers (**pint**) and floats (**pfloat**). Many optimizations are applied dynamically at runtime to reduce the total number of bit-level operations.

`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*.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 (logically inverted) in entanglement channels 0..***e***
- ***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 or all entanglement channels in ***p*** are non-zero
- ***p*.Show()**
  Print debugging info for **`pbit`** ***p*** value: complete bit patterns

There analogous operations for all the above at the **`pint`** and **`pfloat`** levels. For example, ***p*.Set(*e*,*v*)** sets entanglement channel ***e*** within the **`pint`** or **`pfloat`** value ***p*** to the value ***v***.

Additional reversible **`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 for dimension ***c***
- **SETMEAS()**, **SETMEAS(*m*)**
  Set measurement of **`rand()`** channel or ***m***
- **MEAS(*q*)**
  Measure and collapse state of ***q***, returns 0/1

`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 following C++ operators are implemented:

`= *= /= %= += -= >>= <<= &= |= ^=`

`&& || & | ^ == != > < >= <= >> <<`

`+ - * / % ! ~ ++ --`

The **`pint`** functions implemented include:

- **`pint()`**, **`pint(v)`**, **`pint(v,p)`**
  Create a **`pint`** initialized to an integer value: NaN, the **`int`** value ***v***, or ***v*** with precision ***p***
- ***p*.Had(*w*)**, ***p*.Had(*w*,*d*)**
  XOR ***p*** with **Hadamard** pattern ***w*** ways entangled starting with dimension ***d***
- **Cover(*lo*,*hi*,*d*)**, **Range(*lo*,*hi*,*d*)**
  Create a **`pint`** starting with dimension ***d*** and covering [***lo***..***hi***], or range padded with 0s
- **Gather(int*a,*n*)**
  Decode **`pint`** superposition into ***a***[0..***n***-1]
- **Scatter(int*a,*n*)**
  Encode ***a***[0..***n***-1] as a superposed **`pint`**
- ***p*.Mul(*q*,*b*)**
  Create **`pint`** product of ***p*** and ***q***, but limit result precision to ***b*** **`pbit`** to save effort
- ***p*.Reduce*Op*()**, ***p*.Scan*Op*()**
  Reduce entangled superposition to one **`int`** value or to parallel prefix (scan) **`pint`**; ***Op*** can be **And**, **Or**, **XOr**, **Add**, **Mul**, **Min**, or **Max**

`pfloat` Layer

A pattern float, or **`pfloat`**, contains separate **`pint`** values for the sign, exponent, and fraction of a floating-point value. The precision and exponent range of **`pfloat`** are variable at runtime so that the minimum possible number of bits are active.

The following C++ operators are implemented:

`= *= /= += -= >>= <<= &= |= ^=`

`&& || == != > < >= <= >> <<`

`+ - * / ! ++ --`

Boolean operations on **`pfloat`** produce **`pint`** results with 1 for true and 0 for false, but any non-0 is true. The **`pfloat`** functions implemented include:

- **`pfloat(f)`**, **`pfloat(f,b)`**
  Create a **`pfloat`** initialized to a **`float`** value ***f*** with ***b*** bit maximum precision
- **Range(*lo*,*hi*,*b*,*d*)**
  Create a **`pfloat`** starting with dimension ***d*** and covering values [***lo***..***hi***] with ***b*** bits precision
- ***p*.Recip(*i*)**
  Compute 1/***p*** with ***i*** Newton-Raphson iterations
- ***p*.Exp()**, ***p*.Log()**, ***p*.Sqrt()**, ***p*.Cos()**, ***p*.Sin()**, ***p*.Tan()**, ***p*.ArcTan()**
  The usual math functions using base e, radians
- ***p*.Reduce*Op*()**, ***p*.Scan*Op*()**
  As for **`pint`**, but ***Op*** is **Add**, **Mul**, **Min**, or **Max**
- **Scatter(float*a,*n*,*b*)**
  Gather works as for **`pint`**, but **Scatter** needs specification of ***b***-bit precision for **`pfloat`** values

RE, AC, and AoB Layers

Users should avoid these layers, but you can use **`re.Stats()`** to summarize performance.