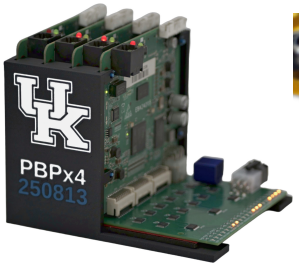
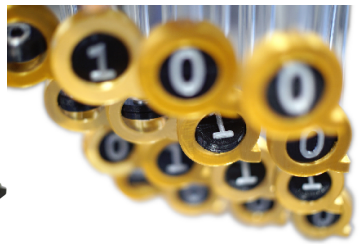
PBP Programming & Execution Models

Parallel bit pattern (PBP) computing is a quantum-inspired computation model. ***Superposition*** and ***n-way entanglement*** are conceptually 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 are compressed using a register number per unique sub-pattern. Applicative caching avoids recomputation of sub-pattern operations. Overall, despite using no quantum phenomena, PBP can exponentially reduce both memory footprint and total number of gate operations.

This document briefly describes the current state of the still evolving PBP programming and execution models.





User-Level Programming Model

The PBP programming model seen by users is implemented using a C++ library that defines classes allowing PBP types to behave as first-class types in programs. Changes being made since Summer 2026 are highlighted in **green**. From the highest level downward, they are:

Pattern Float, `pfloat`. These are vectors of floating-point values with variable-size exponents and mantissas.

Pattern Log Number System, `plns`. These are vectors of log number system values with variable precision. LNS has a variety of characteristics that make it more effective than floating-point, especially when using low precisions.

Pattern Integer, `pint`. These ae vectors of variable-precision integer values. Both signed and unsigned values can be represented, but a sign bit only appears when some vector element is negative.

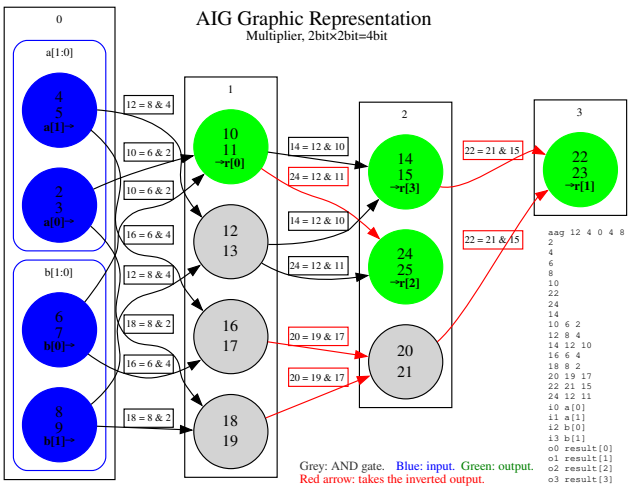
Pattern Bit, `pbit`. A pbit is the most fundamental unit of user-visible PBP data.

A simple example of the use of the PBP programming model is the following C++ code to find the square root of an 8-bit number by exhaustive search. For example, **`sqrt(169)`** will find 13.

```
#include “pbp.h”
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 New AIG-based Execution Model

The PBP execution model begins by lowering all operations to gates applied to pbits. Before lowering each operation, promotion was applied if operands differed in precision. Fairly obvious algorithms were then applied to generate and simplify each implementation using various types of gates. The new PBP will instead directly produce pre-optimized AIGs (And-Inverter graphs) that work without first promoting operands; an AIG model also will be applied to implement operations on unique sub-patterns.



Yosys and the ABC logic synthesizer were used to construct a library of aggressively pre-optimized AIG representations for higher-level operations (one example of which is shown in the above diagram). The preliminary results suggest a 3-5X reduction in pbit-level gate counts for typical operations over the original PBP implementation.

LNS Execution without Table Lookup

LNS arithmetic is very efficient for operations like multiplication: **`log(a*b) = log(a) + log(b)`**. However, addition is disturbingly nonlinear: **`log(a+b) = log(a) + log(1 + pow(base, log(b) – log(a)))`**. Conventional systems use lookup tables in computing LNS addition and subtraction, but lookup tables are not efficient mechanisms for Quantum or PBP systems.

For 16-bit LNS, Boolean rewriting and SAT-based local optimization reduced the complete add/subtract circuit from 2,416 to 1,107 AIG gates, a 54.2% reduction from initial synthesis. Sharing arithmetic logic and searching alternative circuit structures reduced the combined multiply/divide circuit from 813 to just 264 AIG gates, a 67.5% reduction.

```
@techreport{sc26models,
author={Henry Dietz and Paul Grubbs and
  Jacob Dayley and Amr Bittar},
title={{PBP Programming \& Execution Models}},
institution={{University of Kentucky}},
url={{http://aggregate.org/WHITE/sc26models.pdf}},
month={Nov}, year=2026}
```

University of Kentucky

Aggregate.Org UNBRIDLED COMPUTING