Parallel Bit Pattern (part 0)

EE599-001 & EE699-010, Spring 2026

Hank Dietz

http://aggregate.org/hankd/

University of Kentucky

Here's a qubit:

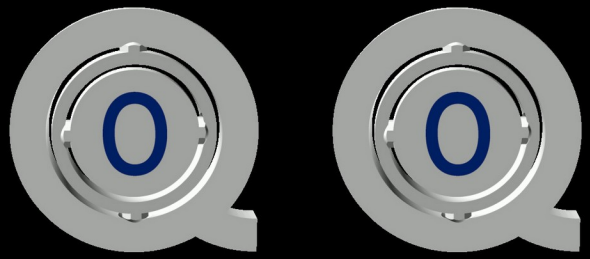
It can have a value of 0, 1, or both

Both?

- |0> is 100% 0, or p(0)=1 and p(1)=0
- |1> is 100% 1, or p(0)=0 and p(1)=1
- $\sqrt{a}\,|0\rangle + \sqrt{(1-a)}|1\rangle$ is p(0)=a and p(1)=(1-a)

- How ***accurately*** do I need to know a?
  - Hadamard gives √½ |0> + √½ |1>
    which needs an accuracy of 1 part in 2
  - Do other probabilities need to happen?

Two Entangled:



- The above happen to be in phase, so
  $\sqrt{1/2}\,|00\rangle + \sqrt{1/2}\,|11\rangle$
- Out of phase (using CNOT) gives
  $1/2\,|00\rangle + 1/2\,|01\rangle + 1/2\,|10\rangle + 1/2\,|11\rangle$
- Which corresponds to parts per 4, i.e., per 2^2
- Do other probabilities need to happen?

E-way Entangled

- Which corresponds to parts per 2^E
- Do other probabilities need to happen?

E-way Entangled

- Which corresponds to parts per 2^E
- Do other probabilities need to happen?

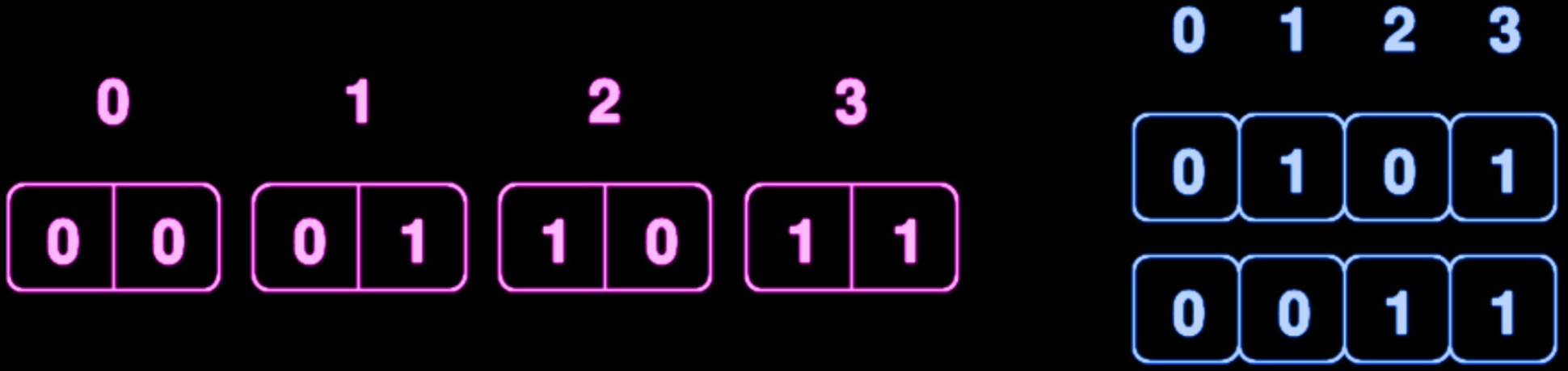
- ***If they do***, we can always add dimensions; for example, doubling the precision would just be using 2^{2*E}

pbits, not qubits

- Our representation does NOT duplicate all properties of qubits
  - Values can be cloned (fan out)
  - Gates need not use reversible logic
  - Values do not decohere over time
  - Measurement doesn't collapse superposition

- Our qubit analog is a **pbit** (**pattern bit**)

Encoding *E*-way Entanglement



- Fundamentally two approaches:
  - Array of Values (AoV): array of 2^E k-bit values
  - Array of Bits (AoB): k 2^E-bit arrays
- Converting between is corner-turning

The AoB Representation

- **AoB**: **Array of Bits** per **pbit** (**pattern bit**)

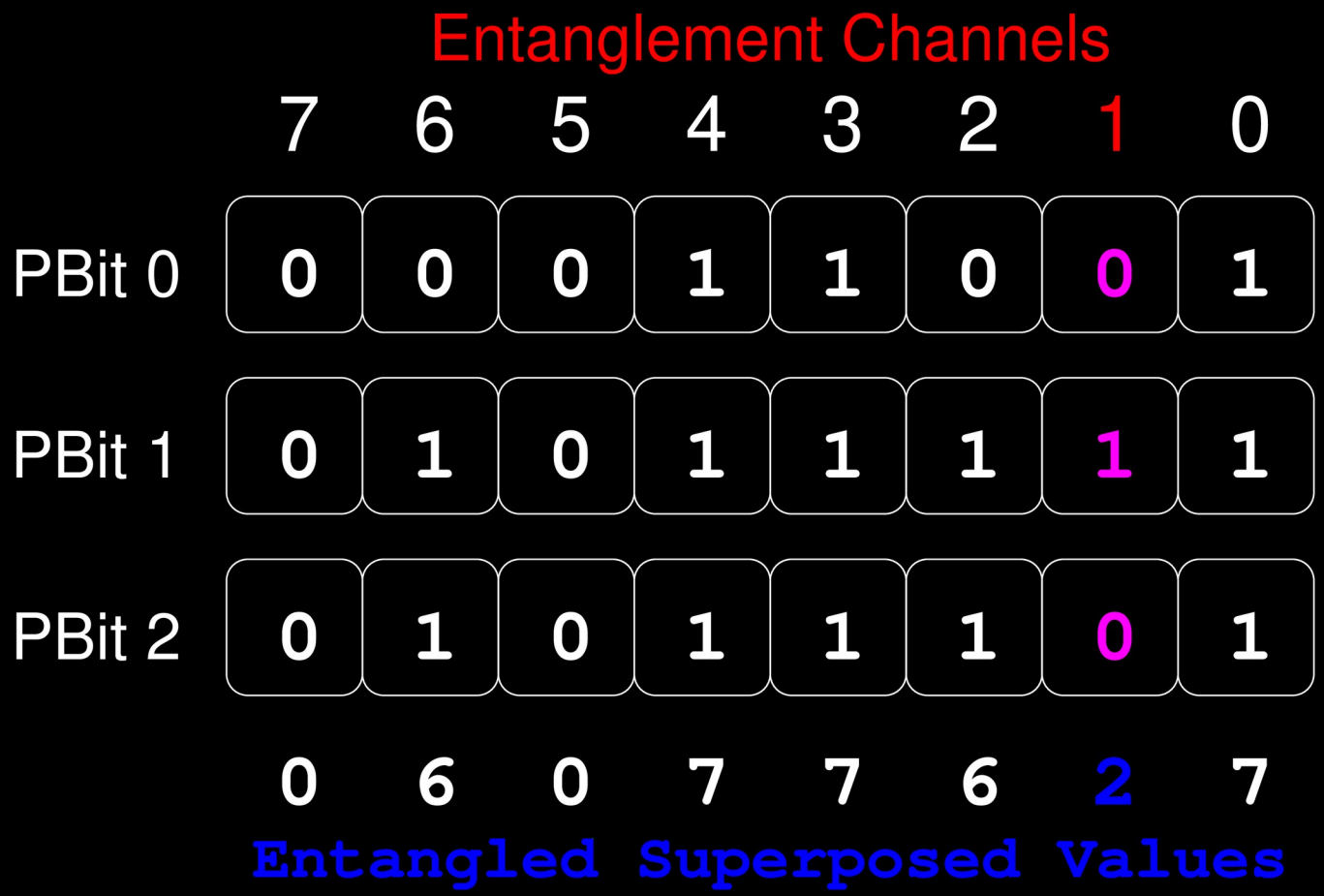
- An *E*-way entangled pbit's value is a vector of 2^E bits, equivalent to an *E*-dimensional cube
  - A single pbit value is a 2-bit vector
  - 2-way each pbit value is a 4-bit vector
  - 32-way each pbit value is a 4G-bit vector

- We call each bit position in such a vector an **entanglement channel**

What About Phase?

- One qubit has a 2-bit AoB with 4 possibilities:
  - {0,0} is the state |0>
  - {1,1} is the state |1>
  - {0,1} and {1,0} are both √½ |0> + √½ |1>; the difference between them is phase
- In big endian, {1,0} is the default phase and a 180° phase shift produces {0,1}
- A 180° phase shift in entanglement dimension *d* rotates by 2^{d-1} bit positions within 2^{d} bit blocks

AoB for 3-way Example

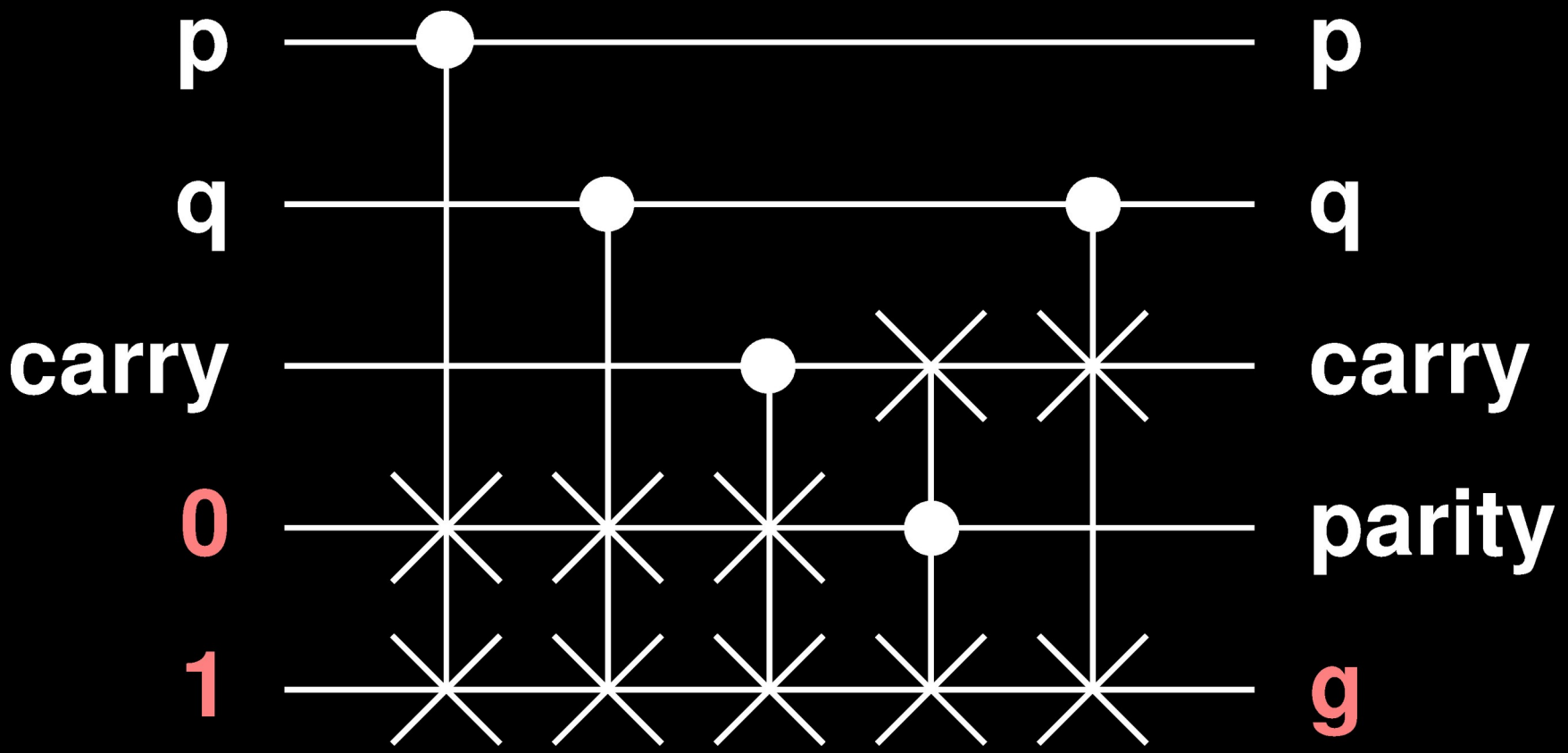


Probability	
0	2/8
1	0/8
2	1/8
3	0/8
4	0/8
5	0/8
6	2/8
7	3/8

KREQC AoB Implementation

- **KREQC**: **Kentucky's Rotationally Emulated Quantum Computer**, SC18
- 6 fully-entangled "Q-bits" that rotate to show probability (no phase display), C program run on laptop does actual computation
- Information and discussion at
  https://aggregate.org/KREQC/
- Web form simulator at
  https://aggregate.org/cgi-bin/kreqc.cgi

Let's Build a 1-bit Full Adder



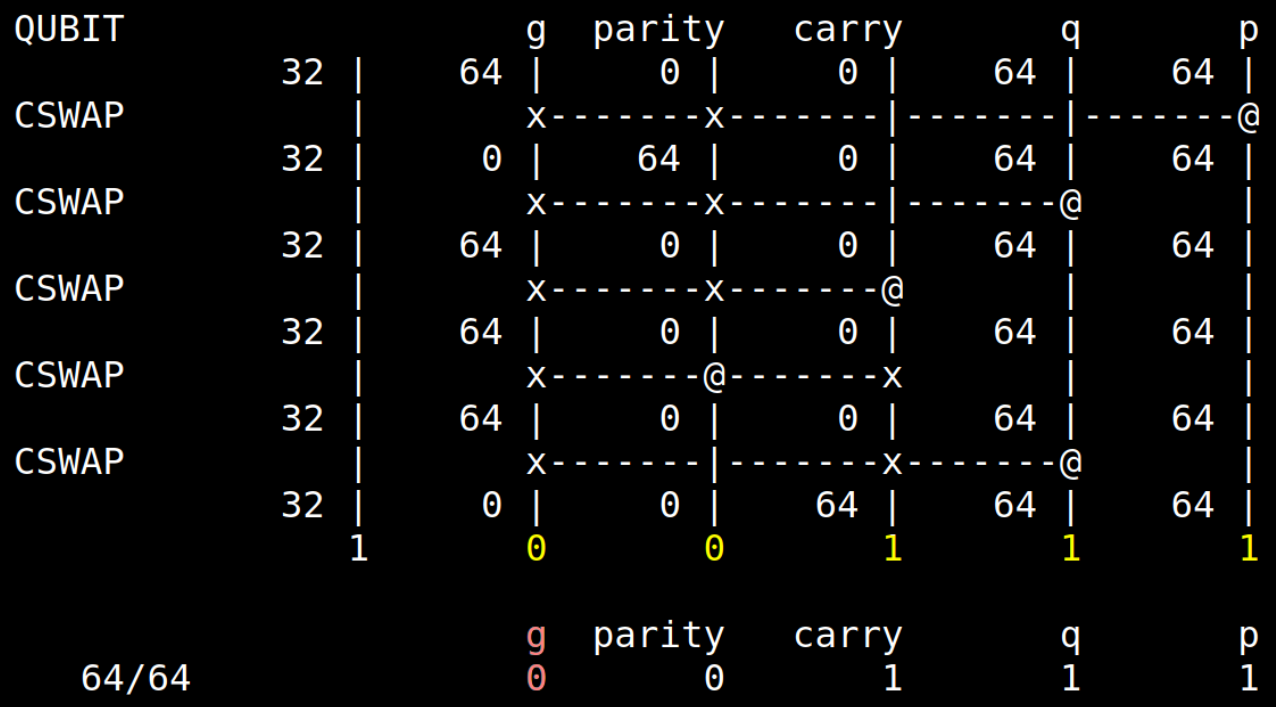
{carry, parity} = p + q + carry

Let’s Build a 1-bit Full Adder

KREQC Program

```
// 1-bit full adder
p=1;
q=1;
carry=0;
parity=0;
g=1;
CSWAP(p, parity, g);
CSWAP(q, parity, g);
CSWAP(carry, parity, g);
CSWAP(parity, carry, g);
CSWAP(q, carry, g);
```

Simulation Output



Let’s Build a 1-bit Full Adder

KREQC Program

```
// 1-bit full adder
p=1;
q=0;
carry=?;
parity=0;
g=1;
CSWAP(p, parity, g);
CSWAP(q, parity, g);
CSWAP(carry, parity, g);
CSWAP(parity, carry, g);
CSWAP(q, carry, g);
```

Simulation Output

```
QUBIT                g  parity   carry       q       p
          32 |    64 |     0 |    32 |     0 |    64 |
CSWAP        |       x-------x-------|-------|-------@
          32 |     0 |    64 |    32 |     0 |    64 |
CSWAP        |       x-------x-------|-------@       |
          32 |     0 |    64 |    32 |     0 |    64 |
CSWAP        |       x-------x-------@       |       |
          32 |    32 |    32 |    32 |     0 |    64 |
CSWAP        |       x-------@-------x       |       |
          32 |    32 |    32 |    32 |     0 |    64 |
CSWAP        |       x-------|-------x-------@       |
          32 |    32 |    32 |    32 |     0 |    64 |
             0       1       0       1       0       1

                     g  parity   carry       q       p
   32/64             0       1       0       0       1
   32/64             1       0       1       0       1
```

Let’s Build a 1-bit Full Adder

KREQC Program

```
// 1-bit full adder
p=?;
q=?;
carry=?;
parity=0;
g=1;
CSWAP(p, parity, g);
CSWAP(q, parity, g);
CSWAP(carry, parity, g);
CSWAP(parity, carry, g);
CSWAP(q, carry, g);
```

Simulation Output

```
QUBIT                g  parity   carry        q        p
          32 |    64 |     0 |    32 |    32 |    32 |
CSWAP        |       x-------x-------|-------|-------@
          32 |    32 |    32 |    32 |    32 |    32 |
CSWAP        |       x-------x-------|-------@       |
          32 |    32 |    32 |    32 |    32 |    32 |
CSWAP        |       x-------x-------@       |       |
          32 |    32 |    32 |    32 |    32 |    32 |
CSWAP        |       x-------@-------x       |       |
          32 |    48 |    32 |    16 |    32 |    32 |
CSWAP        |       x-------|-------x-------@       |
          32 |    32 |    32 |    32 |    32 |    32 |
             1       1       0       0       0       0

                     g  parity   carry        q        p
     8/64            0       0       1        1        1
     8/64            0       1       0        0        1
     8/64            0       1       0        1        0
     8/64            0       1       1        1        1
     8/64            1       0       0        0        0
     8/64            1       0       1        0        1
     8/64            1       0       1        1        0
     8/64            1       1       0        0        0
```

What Did I Just See?

- 6 fully-entangled “Q-bits” means 2^6=64 bit AoB
- Each “Q-bit” has an entanglement dimension
- `parity=0;` means |0>
- `p=1;` means |1>
- `p=?; q=?;` means $H^0(0)$, $H^1(0)$
  - $H^0(0)$ = {1, 0, 1, 0, …, 1, 0, 1, 0} or {1,0}*
  - $H^1(0)$ = {1, 1, 0, 0, …, 1, 1, 0, 0} or {1,1,0,0}*
  - $H^k(0)$ = $\{1^p, 0^p\}^*$ where $p=2^k$

What Did I Just See?

- How do we implement `CSWAP(p,parity,g)`?
- Each vector has 64 bits, so:

```
for (i=0; i<64; i=i+1)
 {parity[i],g[i]} =
 (p[i] ? {g[i],parity[i]} :
         {parity[i],g[i]});
```

- Each gate operation on a *E*-way pbit requires 2^E gate operations on bits

KREQC at SC18

Implementing Hadamard Gate

- How do we implement `H(p)`?
- Each vector has 64 bits, so:

```
for (i=0; i<64; i=i+1)
 p[i] = p[i] ^ H(0,0); //dim 0 H(0)
```

- Simply XOR with Hadamard pattern…
  so this operation is its own inverse
- Note that `H(0)` and `H(1)` differ in phase

KREQC16

- **KREQC: Kentucky's Rotationally Emulated Quantum Computer**, SC19
- 16 fully-entangled "Q-bits" driven by laptop C program
- AoB model is still fast enough, but we do better...

Entropy

- What do these bit vectors look like?
  - $0 = \{0\}^*$
  - $1 = \{1\}^*$
  - $H^k(0) = \{1^p, 0^p\}^*$ where $p=2^k$
  - Basic operations don't easily make random patterns, so **entropy tends to be very low**

- **Regular Expression** (**RE**) compression – store/operate on RE that generates bit pattern

Chunks

- Regular expression compression need not be patterns of *bits*, but can be patterns of any *symbols* desired: we use **chunks**
- A **chunk** is a fixed-length subvector, typically between 32 and 65536 bits long; E-way entanglement fits within a single 2^E-bit chunk
- Each ***unique*** **chunk** is assigned a register, so REs treat register numbers as symbols

Chunk Registers (unique IDs)

- Register 0 is $0 = \{0\}^*$
- Register 1 is $1 = \{1\}^*$
- Register $2+k$ is $H^k(0) = \{1^p, 0^p\}^*$ where $p=2^k$
- Every computed result is hashed to see if it is unique:
  - Unique: allocate a new register
  - Seen before: use register it matched
- Bit vectors never leave their registers; higher levels only know chunk (register) numbers

Operations on Chunk IDs

- Symbolic optimizations (runtime compiler tech)
  - Registers 0 and 1 are scalars 0 and 1
  - Register 0 is the only chunk with no 1s
  - 0&x=0, 0|x=x, 0^x=x, 1&x=x, 1|x=1, …
  - x&x=x, x|x=x, x^x=0, …
- Effort is O(1) because these only touch chunk IDs, never touching actual bit vectors

Operations on Chunk IDs

- **Applicative Caching** (**AC**) remembers a tuple for each operation: {op, operands, result}
  - Only works for stateless operations
  - If x op y is requested, first checks cache
  - If {op, x, y, z} is found, return z
  - Else, compute z=x op y, cache {op, x, y, z}
- Effort is ~O(1) because software-implemented cache is simply a hash function on IDs, never touching actual bit vectors

Operations on Chunk IDs

- **Regular Expression** compression is applied to vectors of IDs
  - Can reduce number of evals of IDs by an exponential factor
  - Can recognize when entanglement narrows
  - Worst case, an E-way entangled RE is a list of 2^{E-c} chunk IDs for *c*-way chunks
- With applicative caching, repeating an operation on IDs is cheap anyway…

Operations on REs

- The same optimization done at lower levels can be applied at the RE level
  - REs are identified by unique IDs
  - Symbolic optimizations work here too
  - Applicative caching can also be done here
- A pbit is really just a copy of an RE ID

What Basic Operations?

- In 

, no, not that one…
  the CPE480 Verilog projects from Fall 2020:
  `https://aggregate.org/CPE480/`
- **Qat** (quantum-like accelerator for Tangled)
  - Implemented 256 AoB registers
  - Each register is 2^{16} bits long (a chunk)
  - No control flow
  - Mostly standard quantum operations…

Qat Basic Operations

Table 3: Qat Coprocessor Instructions

Instruction	Description	Functionality
`and @a,@b,@c`	AND	@a=AND(@b,@c)
`ccnot @a,@b,@c`	controlled-controlled NOT (Toffoli gate)	@a=XOR(@a, AND(@b,@c))
`cnot @a,@b`	controlled NOT	@a=XOR(@a,@b)
`cswap @a,@b,@c`	controlled swap (Fredkin gate)	where (@c) swap(@a,@b)
`had @a,imm4`	Hadamard initializer	@a=H(imm4)
`meas $d,@a`	entanglement channel measure	$d=@a[$d]
`next $d,@a`	entanglement channel of next 1	$d=next($d,@a)
`not @a`	NOT (Pauli-X gate)	@a=NOT(@a)
`or @a,@b,@c`	OR	@a=OR(@b,@c)
`one @a`	1 initializer	@a=1
`swap @a,@b`	swap	swap(@a,@b)
`xor @a,@b,@c`	XOR	@a=XOR(@b,@c)
`zero @a`	0 initializer	@a=0

- Standard unitary matrix operators
- Measurement and entanglement channel scan
- Conventional operators
- Initializers

What Basic Operations Now?

- Learned from Tangled+Qat; new ISA in 2021

  `https://ieeexplore.ieee.org/document/9743163`

- The new ISA has
  - Conventional arithmetic/logic
  - Permutations (phase transformations)
  - Entanglement-channel operations
  - Aggregate functions

Arithmetic/Logic Operations

Instruction	Description	LUTs	Delay
`and @a,@b`	`@c=AND(@a, @b)`	1024	1
`or @a,@b`	`@c=OR(@a,@b)`	1024	1
`xor @a,@b`	`@c=XOR(@a,@b)`	1024	1

- Conventional gates simpler than reversible
- Note: **`not @a`** is **`xor @a,@1`**; **cnot** is **`xor`**

Permutation (Phase) Ops

Instruction	Description	LUTs	Delay
`rot @a,b`	`@c=RotateLeft(@a,b)`	5120	4
`flip @a,b`	`@c=Flip(@a,b)`	5120	4

- No such things in previous PBP models (which meant phase was only done via H)
- Flip is a generalized sorting network

Entanglement Channel Ops

Instruction	Description	LUTs	Delay
`tog @a,b`	`@c=Toggle(@a,b)`	576	2
`dom @a,b`	`@c=Domino(@a,b)`	1079	3
`meas @a,b`	`@c=Measure(@a[b])`	273	7

- Alter specific channels with **`tog`** or **`dom`**
- Read a channel with **`meas`**
  (can use a random **`b`**)

Aggregate Function Ops

Instruction	Description	LUTs	Delay
`first @a`	`First b` where `@a[b]==1`	976	5
`ones @a`	count of `Ones` in `@a`	1444	5

- Result is an integer, not a register number
- Can summarize an entangled superposition (related to Grover and QFT operations)
- Can be exponentially faster than Quantum!

This Has Been PBP
Up To REs

- PBP doesn't just do RE operations
  `https://aggregate.org/PBP/`
- Above REs we have:
  - pbit – dynamically managed RE reference
  - pint – variable precision, signed/unsigned
  - pfloat – variable precision (accuracy)