Tangled: A Conventional Processor Integrating A Quantum-Inspired Coprocessor

ICPP DUAC Workshop
August 9, 2021

Henry Dietz
Electrical & Computer Engineering

University of Kentucky

LCPC 2017:

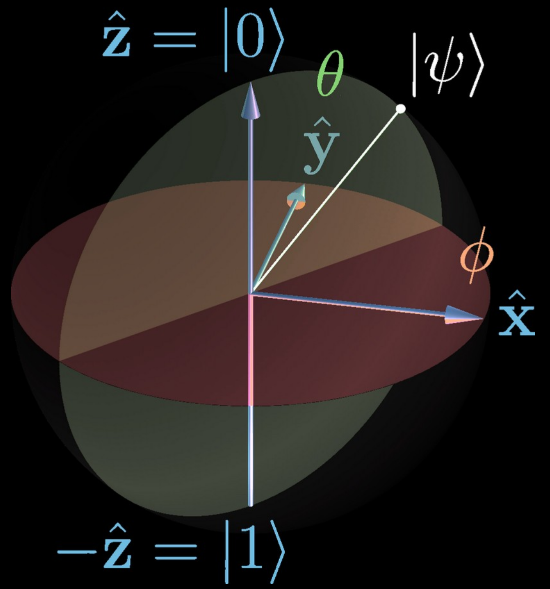
How Low Can You Go?

- Now it's all about power / computation
- Work only on active bits (bit-serial)
- Aggressive gate-level optimization
- Potential exponential benefit from Quantum?

University of Kentucky

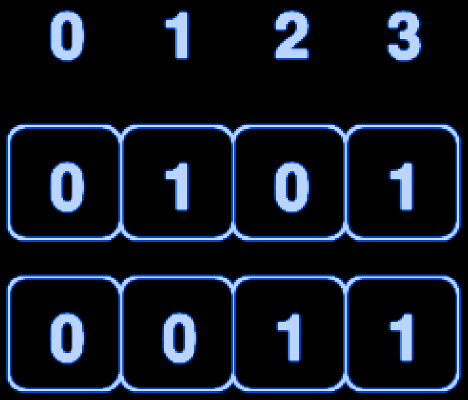
Quantum Computing

- Superposition: 1 qubit, both values
- Entanglement: *e* qubits, 2^e values
  - Exponentially less memory
  - Exponentially fewer gate ops
- Limited coherence, no cloning, only reversible logic gates, ...



University of Kentucky

Encoding *e*-way Entanglement



Array of Bits (AoB): *k* 2^e-bit arrays

- Array indices are **entanglement channels**
- AoB *compressed* as ***Regular Expression (RE) patterns*** of shorter AoB **chunks**

University of Kentucky

Parallel Bit Pattern Computing

- **Operate directly on compressed REs**
  - Up to exponential reduction in storage, gate ops
- Avoids major quantum problems:
  - Forever coherent, error free
  - Cloning: fanout, non-destructive measurement
  - Use any gates, not just reversible logic
  - Parallel AoB chunk ops using SIMD hardware

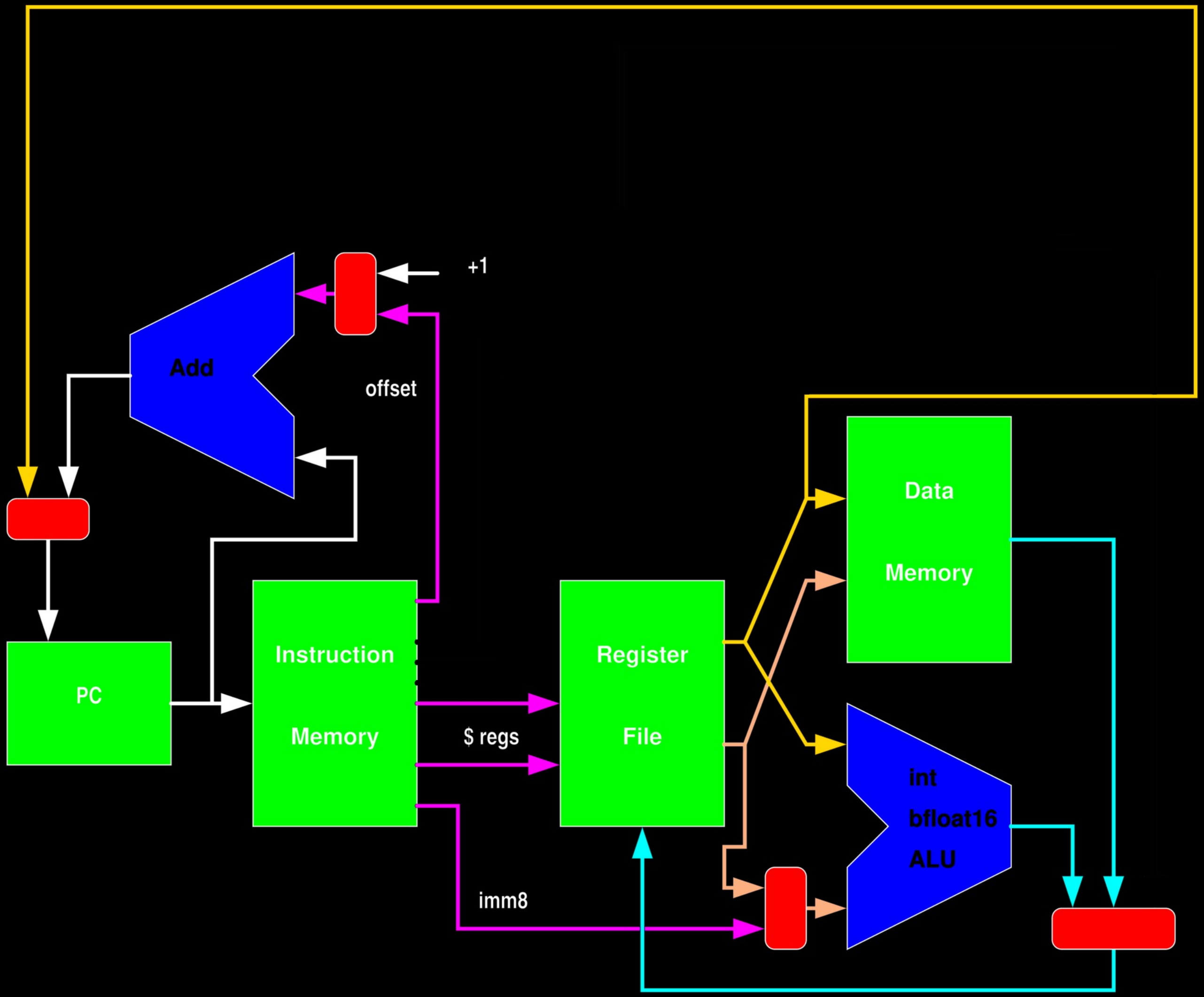
University of Kentucky

Tangled (no, not *that* one)

- Proof-of-concept prototype of PBP Chunk HW
  - **Tangled** processor, **Qat** AoB coprocessor
  - Complete instruction set, toolchain
  - Multiple pipelined Verilog implementations
  - Extensive testing by simulation
- Target architecture for Fall 2020 CPE480

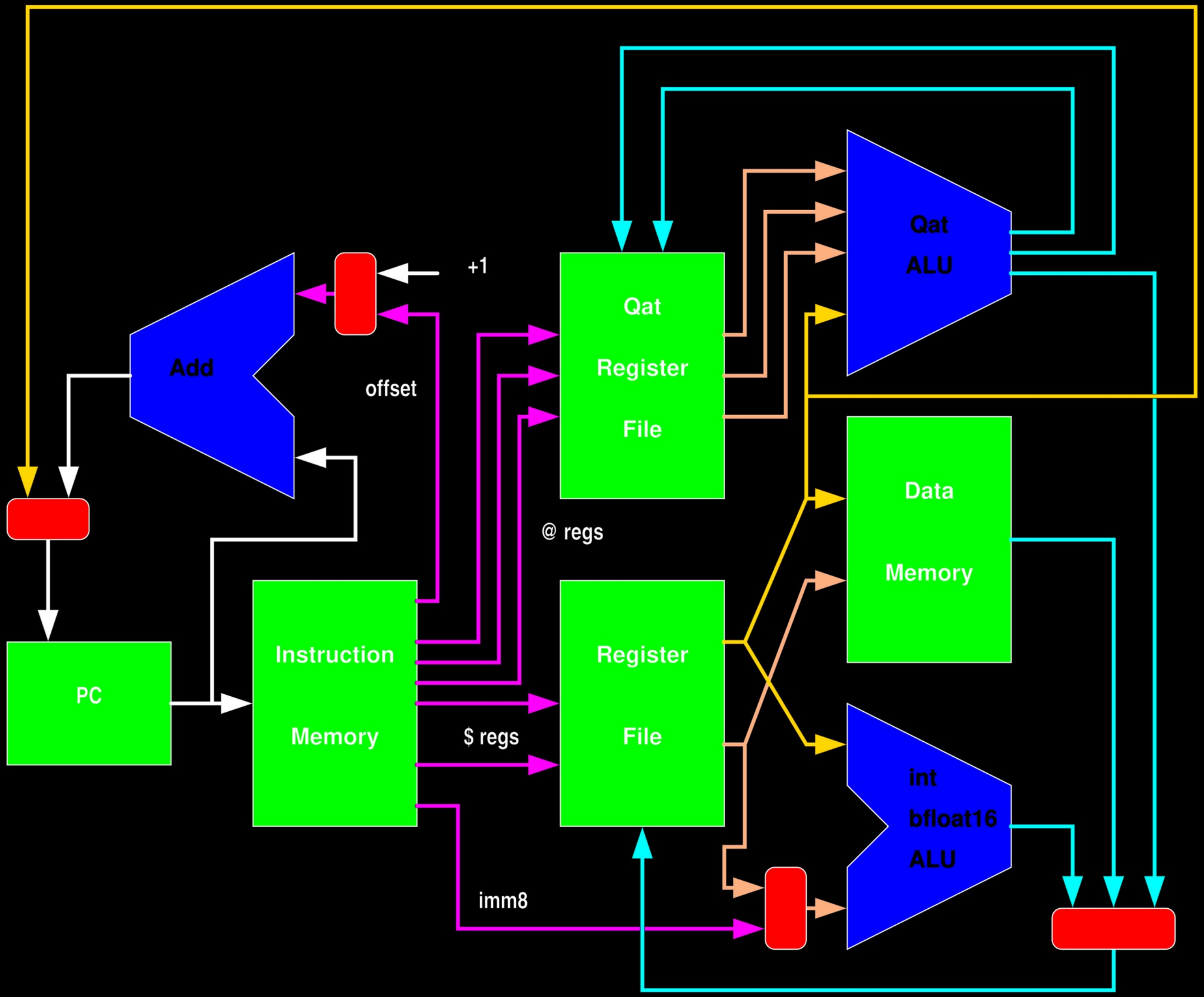
University of Kentucky

Tangled



University of Kentucky

Tangled & Qat



University of Kentucky

Qat Coprocessor Instructions

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

- 256 AoB registers, each up to 2^{16} bits long (16-way entanglement, up to 32-way as REs)
- No control flow nor main memory access

University of Kentucky

Initialization

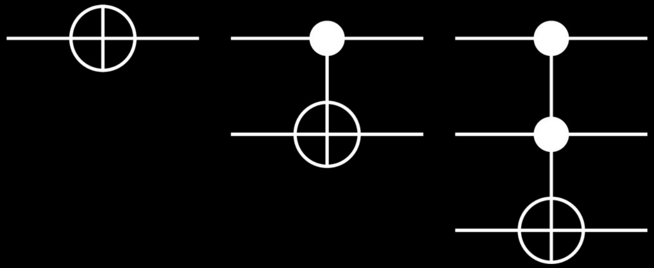
$|0\rangle$ —— $|1\rangle$ —— —[H]—

- Quantum operates in 3 distinct phases: **Initialization**, **Computation**, and **Measurement**
- Quantum initialization is restricted to 0 or 1… but PBP can use **Hadamard** as an initializer
- Qat reversible H(k) is simply XOR with H(k)

➔ Use constant registers rather than instructions

University of Kentucky

Reversible Nots

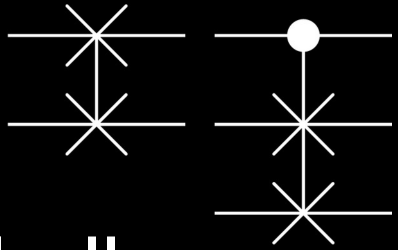


- **Not** (**Pauli-X**), **cnot**, and **ccnot** (**Toffoli**)
- Ccnot requires 3 register reads

➔ `cnot @a,@b` is `@a=XOR(@a,@b)`

➔ `ccnot @a,@b,@c` is `@a=XOR(@a,AND(@b,@c))`

Reversible Swaps



- **Swap** and **cswap** (**Fredkin**; a “billiard-ball conservative” multiplexor, can implement BDDs)
- Cswap requires 3 register reads
- Swap and cswap each require 2 register writes

➔ **`cswap @a,@b,@c`** is expensive to simulate, but PBP infrastructure never used it

University of Kentucky

Measurement

- Quantum measurement **randomly samples** and **collapses** entangled, superposed, value
- Original PBP tried to always read all values

→ **`meas $d,@a`** reads *entanglement channel* `$d` of `@a`
→ Randomly sample using random value for `$d`
→ Does **NOT** collapse entangled superposition

University of Kentucky

Measurement (Interference?)

- Quantum depends on *phase interference* operations to test superposition properties
- We can directly test them…

→ **`next $d,@a`** returns the number of the next *entanglement channel* after `$d` in `@a` that is 1

→ Does **NOT** collapse entangled superposition

→ SIMD **ANY** is measure 0 OR next 0

University of Kentucky

Prime Factoring of 15

```
had @0,3          and @30,@9,@23    and  @60,@58,@59
had @1,5          and @31,@29,@30   or   @61,@49,@60
and @2,@0,@1      xor @32,@15,@16   xor  @62,@43,@45
had @3,4          and @33,@13,@23   and  @63,@61,@62
and @4,@0,@3      and @34,@32,@33   or   @64,@46,@63
had @5,2          xor @35,@29,@30   xor  @65,@61,@62
and @6,@5,@1      and @36,@34,@35   xor  @66,@58,@59
and @7,@4,@6      or  @37,@31,@36   xor  @67,@55,@56
and @8,@5,@3      xor @38,@26,@27   xor  @68,@53,@54
had @9,1          and @39,@37,@38   xor  @69,@32,@33
and @10,@9,@1     or  @40,@28,@39   and  @70,@13,@3
and @11,@8,@10    xor @41,@22,@24   xor  @71,@12,@14
and @12,@9,@3     and @42,@40,@41   and  @72,@70,@71
had @13,0         or  @43,@25,@42   and  @73,@69,@72
and @14,@13,@1    had @44,7         and  @74,@68,@73
and @15,@12,@14   and @45,@0,@44    or   @75,@74,@74
xor @16,@8,@10    and @46,@43,@45   not  @75
and @17,@15,@16   xor @47,@40,@41   or   @76,@67,@75
or  @18,@11,@17   and @48,@5,@44    or   @77,@66,@76
xor @19,@4,@6     and @49,@47,@48   or   @78,@65,@77
and @20,@18,@19   xor @50,@37,@38   or   @79,@64,@78
or  @21,@7,@20    and @51,@9,@44    or   @80,@79,@79
and @22,@2,@21    and @52,@50,@51   not  @80
had @23,6         xor @53,@34,@35   lex  $0,31
and @24,@0,@23    and @54,@13,@44   next $0,@80
and @25,@22,@24   and @55,@53,@54   copy $1,$0
xor @26,@2,@21    xor @56,@50,@51   next $1,@80
and @27,@5,@23    and @57,@55,@56   lex  $2,15
and @28,@26,@27   or  @58,@52,@57   and  $0,$2 ;5
xor @29,@18,@19   xor @59,@47,@48   and  $1,$2 ;3
```

Figure 10: Code prime factoring 15 (3 columns).

```
pint a = pint_mk(4, 15);   // a=15
pint b = pint_h(4, 0x0f); // b=0..15
pint c = pint_h(4, 0xf0); // c=0..15
pint d = pint_mul(b, c);  // d=b*c
pint e = pint_eq(d, a);   // e=(d==a)
pint f = pint_mul(e, b);  // make non-factors 0
pint_measure(f);          // prints 0, 1, 3, 5, 15
```

Figure 9: Word-level prime factoring of 15.

Acknowledgments

- 8 Team projects in Fall 2020 CPE480, pipelined Verilog
- Jordan Caudill, TA

In alphabetical order, the students involved were: Madankrishna Acharya, Malik Twain Allahham, Tristan Aulick Barnes, Travis Luke Bowen, Christopher Jude Butler, Matthew William Castle, Cameron Jacob Charles, Dilan Kiran Dayaram, Andrew Michael Donovan, Blair Emanuel Hall, Cain Aaron Hubbard, Jared Matthew Isler, Euijae Jeon, Collin Lebanik, Benjamin James Luckett, Brock McDaniel, Gerard A Mijares, Isam A Mousharbash, Joshua Aaron Music, Tanner Christian Palin, Nicholas Patrick Santini, Will Cooper Shapiro, Tyler Emerich Sovar, Judah Keith Voth, and Wu Yang.

University of Kentucky

Conclusion

- Parallel Bit Pattern computing HW is viable
  - Multiple **Verilog** pipelined implementations
  - Massively-parallel bit-serial SIMD hardware
  - Disturbingly competitive with quantum
- Lessons learned
  - ➔ PBP only needs a few unusual instructions
  - ➔ Single-cycle chunk ALU is feasible

University of Kentucky