Tangled: A Conventional Processor Integrating A Quantum-Inspired Coprocessor

Henry Dietz
hankd@engr.uky.edu
University of Kentucky
Lexington, Kentucky, USA

ABSTRACT

Quantum computers use quantum physics phenomena to create specialized hardware that can efficiently execute algorithms operating on entangled superposed data. That hardware must be attached to and controlled by a conventional host computer. However, it can be argued that the main benefit thus far has been from reformulating problems to make use of entangled superpositions rather than from use of exotic physics mechanisms to perform the computation – such reformulations often have produced more efficient algorithms for conventional computers. Parallel bit pattern computing does not simulate quantum computing, but provides a way to use non-quantum, bit-level, massively-parallel, SIMD hardware to efficiently execute a broad class of algorithms leveraging superposition and entanglement.

Just as quantum hardware needs a conventional host, so to does parallel bit pattern hardware. Thus, the current work presents Tangled: a simple proof-of-concept conventional processor design incorporating a tightly-coupled interface to an integrated parallel bit pattern co-processor (Qat). The feasibility of this type of interface between conventional and quantum-inspired computation was investigated by construction of an instruction set, building complete Verilog designs for pipelined implementations, and by observing the effectiveness of the interface in executing simple quantum-inspired algorithms involving operations on entangled, superposed, values.

CCS CONCEPTS

• **Computer systems organization** → **Quantum computing**; **Single instruction, multiple data**.

KEYWORDS

quantum computing, SIMD, architecture

ACM Reference Format:
Henry Dietz. 2021. Tangled: A Conventional Processor Integrating A Quantum-Inspired Coprocessor. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3458744.3474044

ICPP Workshops '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-8441-4/21/08…$15.00
https://doi.org/10.1145/3458744.3474044

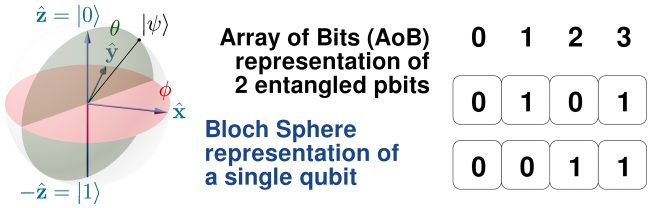

Figure 1: Tangled's quantum-inspired model.

1 INTRODUCTION

Current quantum computers are not particularly practical devices, nor does it help that overenthusiastic claims often are made about what such machines are capable of doing. However, the concepts that underlie quantum computing already have demonstrated considerable value using conventional computer hardware implementations. Various quantum-inspired classical algorithms have obtained significant performance increases over the previously best classical algorithms. Quantum algorithms also have the interesting property that they are optimized at the gate level rather than the word level. As was observed in a 2017 paper at the Languages and Compilers for Parallel Machines workshop[2], extensive application of compiler optimization of programs at the gate level may be able provide orders of magnitude reductions in both the total number of gate actions needed to perform a computation and the total power used.

The basic model for quantum-like computation here is the *parallel bit pattern* model[4], hence referred to as PBP. Without using any exotic quantum phenomena, PBP efficiently supports versions of both *superposition* and *entanglement*. This is done using the concept of *pattern bit*s, or *pbit*s, substituting symbolic computation on a compressed bit vector representation – a bit pattern – for the quantum phenomena. In effect, entangled superposition is transformed into operations on bit vectors, and those operations can be efficiently performed using bit-level SIMD-parallel computer hardware constructed entirely from conventional digital logic.

1.1 The AoB Representation

The value of a qubit is commonly modeled as a real-valued, two-dimensional, probability density function, shown graphically as the Bloch Sphere[10]. Instead of using that model, here an E-way entangled pbit value is represented as an array of 2^E bits (AoB). These two very different representations are shown in Figure 1. The ordering of bits within an AoB vector is significant, determining how those bit values are paired with bit values of other pbits with which it is entangled. In effect, each position within an AoB vector is an *entanglement channel* – a concept not found in quantum computing.

The example shown in Figure 1 gives the AoB vectors for two, two-way entangled, pbits. Either pbit has a 50% probability of 0 or 1, however, entanglement channel 0 pairs {0,0}, 1 pairs {1,0}, 2 pairs {0,1}, and 3 pairs {1,1}. If the top vector is taken to be the least significant pbit of a two-pbit value, the vectors encode the decimal values {0,1,2,3} as four equiprobable values, each having a probability of 1/4. In effect, this is defining a probability density function in which all probabilities are measured in integral parts per 2^E. For example, if the pbit vectors were {0,0,1,0} and {0,0,1,1}, the two-bit values encoded would be {0,0,3,2}, which implies a 50% chance the value is 0, 0% it is 1, 25% it is 2, and 25% it is 3.

1.2 The RE Representation

The complete PBP model does not directly use an AoB representation, but takes advantage of the fact that AoB representations often have very low entropy. By recognizing relatively simple repeating patterns in the AoB vector, the data structure can be compressed and represented as a *regular expression* (*RE*). In the *AoB* example above, {0,1,0,1} can reduce to $(01)^2$ and {0,0,1,1} is $0^2 1^2$ by simple *run-length encoding*. By storing and operating directly on REs, parallel bit pattern computing reduces both storage requirements and computational complexity by as much as an exponential factor... essentially the same goal sought by quantum computing, but achieved using partially symbolic parallel execution on conventional hardware.

Although the RE encoding is clearly beneficial, each symbol within an RE corresponds not to an individual bit, but to a fixed-size sub-vector of the AoB representation. For example, in the PBP software prototype[3], each AoB representation was broken into 4096-bit chunks that became the symbols in the RE representation. Thus, creating an efficient hardware implementation of AoB operations on vectors of relatively modest length not only allows directly executing algorithms that do not exceed the maximum supported entanglement, but also facilitates implementation of RE encodings for constructs having greater entanglement. The hardware implementation described here directly implements 65,536-bit AoB for up to 16-way entanglement, and it is assumed that higher degrees of entanglement would be implemented in software using 65,536-bit chunks as RE symbols. The methodology for that RE processing would be identical to that described in the software-only prototype[3], and thus is not discussed further in the current work.

It is noteworthy that the RE-based PBP model is neither a simulation of quantum computing nor a fully compatible replacement for quantum hardware. It offers new high-level programming and execution models that can efficiently use superposition and entanglement to implement a large class of quantum-inspired algorithms. As is detailed later in this paper, the PBP model is arguably stronger than quantum models because it allows non-destructive measurement and values may be maintained for arbitrarily long without decoherence.

1.3 Tangled and Qat

The current work represents the first attempt to devise and evaluate a conventional hardware implementation of the AoB processing within the PBP model. Like quantum computer models, AoB is not a general-purpose model of computation, nor is it likely to become one: rather, it is a type of specialized attached processor. In this case, the AoB processing is very tightly integrated with the host processor so that it appears as a coprocessor with instructions fetched and decoded by the conventional host processor.

The host processor is called *Tangled*, partly because it manages a coprocessor supporting entangled superposition, but also reflecting the fact that it is designed to operate in a way that is tightly intertwined with that coprocessor. For example, processor pipeline interlocks and forwarding are determined in part by coprocessor operations. The coprocessor itself is called *Qat* (pronounced "k-ha-t"), a TLA standing for Quantum-like Accelerator for Tangled.

Although the author created Tangled and Qat primarily as a research prototype, the research followed a somewhat unusual path because this work was done during the peak of the COVID-19 pandemic. The University of Kentucky campus continued to be open throughout Summer and Fall of 2020, but many precautions were taken against spread of the virus, including restricting use of physical research laboratory spaces to only emergency maintenance for Summer 2020. Thus, to allow students to become involved in the research, the design of Tangled and Qat was crafted so that it also would be appropriate as the basis for student Verilog implementation projects in the CPE480 Computer Engineering undergraduate Computer Architecture course taught in Fall 2020. For example, the incorporation of `bfloat16` arithmetic in Tangled is primarily to better serve the goals of that course; floating-point arithmetic is not necessary for most quantum-inspired algorithms. In that course, four projects involved Tangled and Qat: a project with each student individually determining the instruction set encoding and building an assembler for it using AIK (the Assembler Interpreter from Kentucky)[5] and then a sequence of three team projects, done in groups of 3-4 students, creating synthesizable Verilog implementations of a multi-cycle Tangled, a pipelined Tangled, and finally a pipelined version implementing both Tangled and Qat. The teams were also randomly shuffled so that no two students were on the same team for more than one project, a technique that helps spread experience-based knowledge across teams more effectively.

The remainder of the current work describes the design, implementation, and evaluation of Tangled and Qat.

2 INSTRUCTION SET ARCHITECTURE

Although the Qat coprocessor instructions are fetched and decoded by the Tangled processor, it is useful to consider the Qat instructions as a separate group because they have access to additional resources not present in Tangled.

2.1 Tangled ISA

The basic Tangled instruction set, as described in Table 1, is very straightforward. In some sense, nearly any conventional instruction set would suffice. However, this was designed not only to be a research proof-of-concept implementation of the tight integration of a quantum-like coprocessor, but to also serve pedagogical goals involving having students build Verilog implementations in the course. For example, the 16-bit word size was selected because:

- This instruction word size only has space for a 4-bit fixed opcode field, but there are more than 16 different types of instructions; thus, students needed to be slightly clever about

Table 1: Tangled Base Instruction Set

Instruction	Description	Functionality
`add $d,$s`	int add	$d+=$s
`addf $d,$s`	bfloat16 add	$d+=$s
`and $d,$s`	bitwise AND	$d=AND($d,$s)
`brf $c,lab`	branch false to lab	if (!$c) PC+=offset
`brt $c,lab`	branch true to lab	if ($c) PC+=offset
`copy $d,$s`	copy	$d=$s
`float $d`	int to bfloat16	$d=(bfloat16)$d
`int $d`	bfloat16 to int	$d=(int)$d
`jumpr $a`	jump to register	PC=$a
`lex $d,imm8`	load sign extended	$d={{8{imm8[7]}},imm8}
`lhi $d,imm8`	load high	$d[15:8]=imm8
`load $d,$s`	load	$d=memory[$s]
`mul $d,$s`	int multiply	$d*=$s
`mulf $d,$s`	bfloat16 multiply	$d*=$s
`neg $d`	int negate	$d=(-$d)
`negf $d`	bfloat16 negate	$d=(-$d)
`not $d`	bitwise NOT	$d=NOT($d)
`or $d,$s`	bitwise OR	$d=OR($d,$s)
`recip $d`	bfloat16 reciprocal	$d=1.0/$d
`shift $d,$s`	shift left/right	$d=$d<<$s
`slt $d,$s`	set less than	$d=($d<$s)
`store $d,$s`	store	memory[$s]=$d
`sys`	system call	
`xor $d,$s`	bitwise XOR	$d=XOR($d,$s)

picking an encoding. Students in the course needed to determine how to encode the instructions, and they then used an assembler construction tool to implement the assembler. That tool makes it easy to change the encoding, and students were permitted to change the instruction encoding for each project as they learned more about implementation of the system.

- A 16-bit word size allows for efficient implementation of both integer and floating-point arithmetic. In particular, `bfloat16` has been used in this course for several years, and there are ALU implementations of all the basic floating-point operations that can be treated as single-cycle delay in FPGA renderings of a pipelined processor's Verilog design. Use of `bfloat16` also is convenient in that values can be treated as standard 32-bit float values by simply catenating a 16-bit value of 0.
- Relative to current quantum computers, even 16-bit values are relatively high precision. Thus, the small word size keeps simulations manageable without compromising the quantum-like aspects of the system.

Tangled only has 16 conventional general-purpose registers. Registers 0-10 are for general use. Register 11 is `$at`, reserved for use as an assembler temporary in implementing assembler macros – such as those listed in Table 2. The remaining four registers are used for function/subroutine call handling: return value `$rv`, return address `$ra`, frame pointer `$fp`, and stack pointer `$sp`. In other words, none of the Tangled registers has any special meaning relative to the Qat coprocessor.

Table 2: Tangled Pseudo-Instructions (Macros)

Instruction	Description	Functionality
`br lab`	branch to lab	PC+=offset
`jump lab`	jump to lab	PC=lab
`jumpf $c,lab`	jump false to lab	if (!$c) PC=lab
`jumpt $c,lab`	jump true to lab	if ($c) PC=lab
`load $d,imm16`	load immediate	$d=imm16

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

2.2 Qat ISA

Table 3 lists the Qat coprocessor instructions. These instructions are not very closely related to the instruction sets discussed for quantum computers[12][7]. Some of these instructions obviously echo the basic gate operations used in quantum computers. However, others appear completely classical, and several are essentially neither quantum gates nor classical operations.

While Tangled operates on 16-bit values, all Qat operations act upon 65,536-bit AoB values – each sufficient to hold the 16-way entangled superposed value of a single pbit. This provides a type of symmetry in that any value that Tangled can directly process is only 16 bits long, and all 16-bit values are representable using a set of 16 properly entangled Qat AoB values. As is the norm for quantum computers, Qat does not have any method by which it can access the host system memory. Thus, all AoB values are exclusively held in Qat coprocessor registers, named `@0` through `@255`. The lack of external storage is also why a relatively large number of registers was selected for Qat: 256. The use of 8-bit Qat register numbers does force some Qat instructions to be two 16-bit words long, but only 16 registers could be named within a single-word Qat instruction

encoding, and that would have made it nearly impossible to have a Qat computation produce a 16-bit value result.

It might seem obvious that quantum gate operations could be expressed as assembly language instructions acting on qubits identified by register numbers, but that is not how the Qat ISA is structured. Fundamentally, the semantics of operations on pbits are quite different from those working on qubits. In a quantum computer, there are three distinct phases of operation:

(1) Initialize all qubits to their desired initial state, which for each is strictly 0 or 1 (i.e., it is not a superposition).
(2) Perform the computation. The computation is typically written as a series of thermodynamically reversible gate operations on qubits, but may actually be implemented with modest parallelism in that operations on unrelated qubits need not be ordered if the quantum computer's hardware supports such parallel control. The computation itself creates entangled superpositions, generally by applying a *Hadamard gate* at an early point in the computation. There also are a few interference operations that can be used to sample some aspect of the distribution of possible values within a superposition without collapsing it, and these operations play a critical role in potential applications such as Shor's factoring algorithm[11].
(3) Measure the result of the computation. Unfortunately, this measurement collapses any entangled superposition, so only a single value is returned per qubit. The values returned are often described as "randomly selected" from among the possible values, but it would be more accurate to say that they are not selected at all. The value returned is intended to be consistent with a random sampling, but might be somewhat biased by noise, manufacturing variation, etc.

The parallel bit pattern model, and hence the Qat coprocessor, does not require this type of phased operation. This is primarily due to the fact that **measurement of a pbit is inherently non-destructive**. Superposition and entanglement can be sampled in arbitrary ways without terminating the computation. However, this fundamental difference has many other significant implications. For example, since the value of an entangled superposition can be non-destructively measured, it is also possible to refresh and maintain that value arbitrarily long without degradation: in contrast, all values in quantum computers accumulate noise over time, eventually resulting in errors and/or decoherence. Immune to such errors, **Qat can freely mix initialization, computation, and measurement over arbitrarily long periods**. Given that freedom, there also is no need for pbit operations to be thermodynamically reversible.

What the quantum computing field calls thermodynamically reversible gates is better known in computer engineering circles as adiabatic logic[8]. Unfortunately, the quantum annealing models for quantum computer hardware[9], as used by D-Wave, are also referred to as adiabatic quantum computing, causing some ambiguity with use of that term. By whatever name, adiabatic logic has been well studied, and entire processors have been built using conventional circuitry to implement adiabatic logic that can significantly reduce power consumption per computation[15][13].

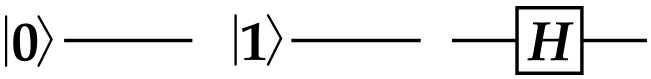

Figure 2: Quantum initialization to 0, 1; Hadamard gate.

Fundamentally, adiabatic logic reduces power consumption by balancing every logic 1 with a logic 0; thus, power is neither created nor absorbed, but merely re-routed. Alternatively, the balancing can be done in the time domain, having a logic circuit that apparently generates a 1 output, in the following cycle, recover the energy from its outputs. In general, a thermodynamically reversible gate cannot destroy information, so overwriting a qubit, as is done in initialization, is not permitted in quantum computers.

2.3 Qat Initialization Instructions

In a quantum computer, the operations associated with initializing a value are symbolically shown in Figure 2. However, initialization of qubits is done as a separate phase, and not as part of the quantum computation per se. Given that, no value can be initialized to a superposed state, and the only true initializers for a quantum computation are 0 and 1. Superposition is created by applying the Hadamard gate during quantum computation to a qubit that was earlier initialized to 0 or 1. It is appropriate to note that the Hadamard gate can also be used in quantum computing as its own inverse, thus preserving the reversibility property required of all quantum gates.

In contrast, Qat does not require reversibility nor a separate initialization phase; initialization of a pbit is an operation that may be performed at any time. In effect, this means that initialization to 0, 1, or a "standard" entangled superposition is essentially a "load immediate" instruction that can be applied to any AoB register at any time. The `zero @a` instruction sets `@a` to 0 and the `one @a` sets `@a` to 1, but the Hadamard initialization in Qat is complicated by the notion of entanglement channels. The default Hadamard pattern for the k^{th} set of entanglement channels would be created by `had,@a,`k. The pattern generated within `@a` is a repeating sequence of 2^k 0 bit values followed by 2^k 1 bit values. Thus, entanglement channel e in `@a` would be the value of bit k within the binary representation of the 16-bit number e. For example, `had @a,0` would make every even-numbered entanglement channel 0 and every odd-numbered channel 1. The AoB value created by `had @a,15` would consist of 32,768 0 bits followed by 32,768 1 bits.

2.4 Reversible Not-based Instructions

The common notation for the standard quantum gates involving logical negation (not) is shown in Figure 3. The way to interpret this notation is that each horizontal line represents the value of a qubit evolving over time. A quantum gate is not really a physical thing, but the imposition of external stimuli at a point in time that interacts with the values of one of more qubits. For the not gate, which is also known as the Pauli-X gate, that interaction simply flips the sense of all bit values in the superposition. The controlled not flips the sense of the superposition values only where the entangled value of the control qubit would be a 1; in other words, the gate passes the control qubit unchanged, but replaces the second qubit

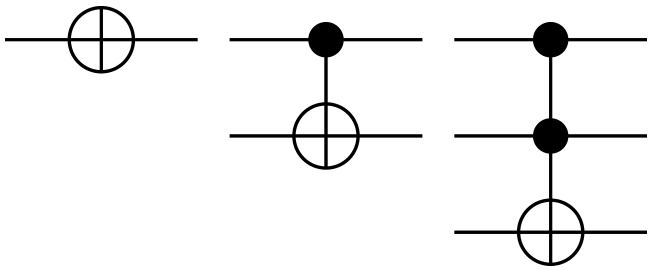

Figure 3: Quantum not, cnot, and ccnot gates.

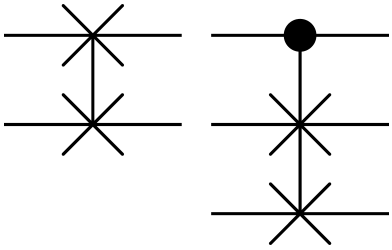

Figure 4: Quantum swap and cswap gates.

with the XOR of the two qubit values. Similarly, the controlled-controlled not, which is also known as the Toffoli gate, passes the two control qubit values unchanged, but replaces the value of the third qubit with the XOR of its value and the AND of the the two control qubit values. These operations are all trivially reversible in that each is it's own inverse; a second application of any of these gates will restore the original qubit values.

In Qat, these three gate operations are modeled as instructions in which the potentially altered value is the first named and any additional registers named are the controls. Thus, `not @a` has no control, `cnot @a,@b` has the single control `@b`, and `ccnot @a,@b,@c` uses `@b` and `@c` as controls. All input values to the gates are examined, which implies that the `ccnot` operation requires three inputs – a poor match for the standard ALU symbol, which shows only two inputs, but all that is needed for an efficient implementation is that the register file provide three read multiplexors. In all three instructions, the control inputs are unchanged; only a single register has a potentially new value written.

2.5 Reversible Swap-based Instructions

Another common class of reversible gates is based on swap: exchange of values. The common quantum notation for the standard quantum gates involving swap is shown in Figure 4. The potential exchange of values is indicated by an "X" across the qubits that would be exchanged. For `cswap`, also known as a Fredkin gate[6], the exchange occurs only for the superposed values where the control signal would be 1. As for the not-based quantum gates, `swap` and `cswap` are trivially reversible because each is its own inverse. In fact, the reversibility of swap is stronger than that of the not-based gates because the number of 0s and 1s is trivially preserved passing through the gate – a property sometimes called "billiard-ball conservancy" which can lead to simpler non-quantum adiabatic logic implementation. It is interesting to note that `cswap` can be viewed

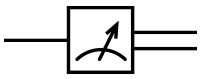

Figure 5: Quantum measurement gate.

as a generalization of a 1-of-2 input multiplexor, which also are used to construct binary decision diagrams (BDDs)[1], and are the universal gates commonly applied in formal verification of logic designs.

In Qat, the swap operations are modeled as instructions in which the potentially swapped values are the first two operands, and `cswap`'s third operand is the control. Thus, the ALU performing `cswap` also needs input from three registers. However, the swap requires that two results be written into registers, thus the ALU should have two outputs and the register file should be capable of three reads and two writes per cycle. While this is feasible, it is not clear that the performance gained by adding this hardware is sufficient to justify its use in Qat.

2.6 Irreversible Logic Instructions

Either `ccnot` or `cswap` alone constitutes a universal logic family, such that no other gate types are needed. However, Qat does not require that instructions be reversible, and it is in many ways more convenient if the usual irreversible logic gate types can be provided. For example, many logic optimization tools directly generate designs using AND, OR, and NOT; similarly, computer arithmetic is often most conveniently expressed using XOR. Thus, Qat also implements the familiar irreversible gates: `and`, `or` and `xor`.

Interestingly, implementation of these instructions is simpler than `ccnot` or `cswap` – only two registers are read and only one result is written. One of the questions this work hoped to answer was: is it worthwhile directly implementing the more-complex reversible gate operations?

2.7 Measurement Instructions

There are only two Qat instructions remaining to be discussed, and both of them relate to the concept of measurement. There was also a potential third sampling operation, which was specified but omitted from the class project versions.

The only measurement mechanism in quantum computers is that shown in Figure 5. The paired lines leaving the measurement gate signify that the result is not a superposed value, but a digital bit that is valued either 0 or 1. It is a fundamental property of quantum computation that measuring a superposed qubit's value collapses it to a simple state of 0 or 1. Further, any qubits entangled with a qubit measured also become locked into their values at that moment. The probability-weighted "random selection" of a measured value from among the set of possible values in an entangled superposition is the only way to obtain output from a quantum computation. Thus, although an entangled superposition at the end of a computation might contain all answers, only one can be examined per run. Further, the inability to deterministically pick which answer is sampled means that there is no number of runs sufficient to guarantee that all values in the entangled superposition have been seen.

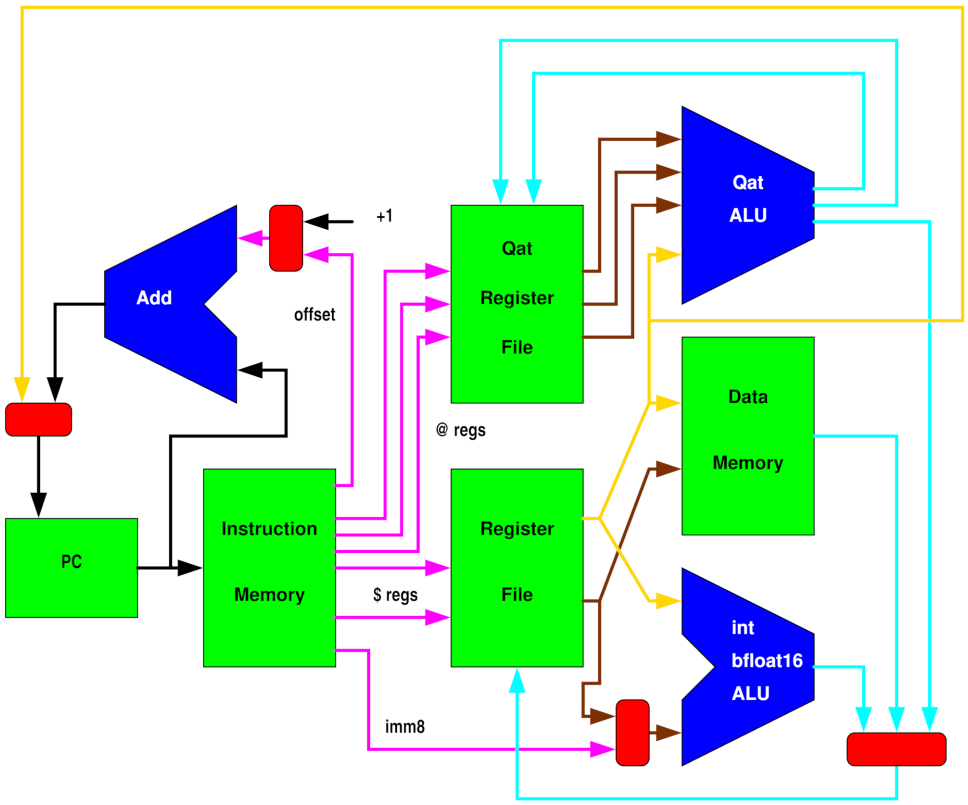

Figure 6: Simplified single-cycle Tangled/Qat.

Measurement in PBP computing does not collapse the superposition – which should give PBP a huge advantage in any computation that may produce more than one result. However, a different concept of measurement is needed.

In an LCPC 2020 paper[3], pbit measurement is defined as an operation that returns *all* values in the entangled superposition. However, there were also operations that could be used to test properties of the entangled superposition:

- ANY reduction, replacing a pbit that has a non-zero probability of being a 1 with the value 1.
- ALL reduction, replacing a pbit that has zero probability of being a 0 with the value 1 and otherwise with the value 0.
- POPulation count, returning an integer count of how many of the 2^E bits in an entangled superposition are 1 – essentially the probability of that pbit being a 1 in parts per 2^E.

In designing Tangled, it immediately became clear that having Qat return all values within an entangled superposition would be a very awkward operation to directly implement in hardware. What was needed was a way to incrementally obtain any sampling of the entangled superposition desired. The key new concept for this was to **explicitly select the entanglement channels to be examined**.

The first new PBP operation in Qat is `meas $d,@a`. It simply returns the value of the bit at entanglement position `$d` in the entangled superposed value in `@a` – essentially `@a[$d]`. This operation is very efficient to implement, yet allows very high-quality random sampling of entangled superpositions by simply using Tangled instructions to place a random number in `$d`. The `meas` instruction even could be used to read-out every element in an entangled superposition by looping through all all 65,536 entanglement channel numbers. The catch is that operations like the ANY, ALL, and POP described in earlier work provide a way to summarize an entangled superposition in as little as $O(1)$ time, whereas `meas` would take $O(2^E)$ time enumerating the values.

For Qat, rather than implementing ANY, ALL, and POP, the new solution was the creation of an instruction that used entanglement channels in a way that could flexibly and efficiently summarize the entangled superposition. The `next $d,@a` instruction is given an initial entanglement channel number in `$d`, and then returns the lowest entanglement channel number in `@a` larger than `$d` where a bit value of 1 is located. If there is no 1 in the remainder of the AoB vector, the value returned is 0. For example, the sequence:

```
had   @123,4
lex   $8,42
next  $8,@123
```

would result in the value 48 being placed in `$8`. This is because `had @123,4` creates a repeating pattern of sixteen 0 followed by sixteen 1, and the first non-0 bit after position 42 in that pattern is in entanglement channel 48.

Consider implementing ANY using `next`. if `next` is used to search for the next 1 after entanglement channel 0 and returns a non-0 value, ANY is true. However, if that returned 0, we would still need to test entanglement channel 0, which can be done using `meas`. Essentially the same logic can be used to test for ALL, except ALL of `@a` would essentially be computed as `not` of the result of applying ANY to `not @a`.

A POP operation was omitted primarily to simplify the class projects, and easily could be added. However, there is again the issue that the number of 1 bits in a 16-way entangled superposition ranges from 0 to 65,536, which is one greater range than fits in a 16-bit Tangled register. The more general solution that was developed, but not explored in the class projects, was to have a `pop` instruction that counted the population of 1 bits after the specified entanglement channel – sharing that logic with the implementation of `next`. The true POP value would thus be the sum of `pop` starting after 0 and `meas` of entanglement channel 0, but the separation into two operations would allow for easy detection of overflow.

3 VERILOG IMPLEMENTATION

The pipelined Verilog implementation of Tangled/Qat proved to be surprisingly straightforward. As a starting point, Figure 6 diagrams a simplified single-cycle design. It is significant that all functions implemented in the Tangled and Qat ALUs can be treated as fully combinatorial for FPGA implementation. The only operation for which purely combinatorial execution might be problematic is `mul` – the Tangled 16-bit integer multiply instruction.

3.1 General Properties of the Pipelined Design

There were eight teams for each of the class projects, so in addition to several versions produced by the author, there are eight independent Verilog implementations of the full pipelined implementation of the Tangled/Qat processor with a simplified memory interface. To speed-up simulation, which was largely done using a WWW-form interface to a server running Icarus Verilog[16] and the Covered[17] test coverage analysis tool, students also were permitted to restrict the AoB values to 256 bits. Although team scores on the multi-cycle design ranged from 57.5% to 100%, the shuffling of team members helped spread and enhance expertise. All eight final team projects were highly functional, with grades ranging from 83.5% to 99%, with an average of 89.7%.

Implementations ranged from 633 to 1006 lines of Verilog code, with an average of 742 lines. Those numbers include approximately 127 lines for the Verilog `bfloat16` floating-point library provided to the students. The floating-point code also required a small VMEM file initializing a lookup table for computing fraction reciprocals, but that is not included in the Verilog line count. Six of the eight pipelines the students implemented used four stages; two used five stages. All implementations were capable of sustaining completion of one instruction every clock cycle, provided there were no pipeline interlocks encountered.

Overall, despite the strangeness of Tangled/Qat (and the pandemic), these Verilog implementations were at least comparably successful to those from the more mundane processor design assignments given in previous offerings of this course. Students seemed to have surprisingly little difficulty creating efficient implementations of Qat. Difficulties tended to center on pipeline handling of conditional control and data dependences. The most common student questions involved the fetch and decode handling of variable-length instructions (recall that some Qat instructions encode as two 16-bit words).

```
module qathad(aob, h);
parameter WAYS=16;
input [WAYS-1:0] h;
output [(1<<WAYS)-1:0] aob;

genvar i;
generate
  for (i=0; i<(1<<WAYS); i=i+1) begin
    assign aob[i] = (i >> h);
  end
endgenerate
endmodule
```

Figure 7: Verilog `had` for WAYS-way entanglement.

The only apparently difficult-to-implement operations are the Qat instructions `had` and `next`. The student teams did not create scalable parametric designs for these operations, but such designs were created by the author and are described in the following subsections.

3.2 Verilog Implementation of Qat `had`

Figure 7 shows a surprisingly straightforward parametric Verilog computation of the Qat `had` instruction AoB result. This implementation directly uses the correspondence between entanglement channel e and the binary value of e, as was described earlier. However, the actual circuit implementation of this would be somewhat more complex than simply constructing a complete pre-computed lookup table of each of the Hadamard initializers. In fact, the student solutions generally used a lookup table expressed as a Verilog combinatorial `always` selecting the appropriate constant pattern using a `case` statement (multiplexor).

Given that there is already a relatively large Qat register file, this suggests a better implementation structure would simply replace some of the registers with the pre-computed constants. The software-only implementation of PBP[3] actually placed the constants 0, 1, H(0), H(1), H(2), etc., in the first pbit descriptors, and it is now clear that making `@0` be 0, `@1` be 1, `@2` be H(0), `@3` be H(1), etc., would be more efficient than having `zero`, `one`, and `had` instructions. In retrospect, those operations were made instructions because the qubit prohibition on copying a value would have required it – but PBP has no such restriction.

3.3 Verilog Implementation of Qat `next`

As shown in Figure 8, the Verilog implementation of the Qat `next` instruction ALU operation is significantly less straightforward than any of the other Qat ALU functions. The underlying logic for the Verilog design can be described as consisting of two steps:

(1) Make a (wire) copy of the AoB value, but with entanglement channels from 0 to the channel specified by `$d` all set to 0.

```
module qatnext(r, aob, s);
parameter WAYS=16;
input [(1<<WAYS)-1:0] aob;
input [WAYS-1:0] s;
output [WAYS-1:0] r;

genvar pow2;
generate
  wire [WAYS-1:0] tr;
  for (pow2=WAYS-1; pow2>=0; pow2=pow2-1) begin:t
    // wires named as t[pow2].v
    wire [(2<<pow2)-1:0] v;
  end
  assign t[WAYS-1].v =
    {((aob[(1<<WAYS)-1:1] >> s) << s), 1'b0};
  for (pow2=WAYS-1; pow2>0; pow2=pow2-1) begin
    assign {tr[pow2], t[pow2-1].v} =
      ((|t[pow2].v[(1<<pow2)-1:0]) ?
       {1'b0, t[pow2].v[(1<<pow2)-1:0]} :
       {1'b1, t[pow2].v[(2<<pow2)-1:(1<<pow2)]});
  end
  assign tr[0] = ~t[0].v[0];
  assign r = ((t[0].v) ? tr : 0);
endgenerate
endmodule
```

Figure 8: Verilog `next` for WAYS-way entanglement.

This can be done using a barrel shifter to right-shift-out the original bits in these positions and then left-shift back in 0s. A barrel shifter generally requires $O(log_2 N)$ gate delays for N bits, or $O(WAYS)$ gate delays for AoB supporting up to $WAYS$-way entanglement.

(2) Implement a combinatorial count-trailing-zeros operation. This operation is very similar to the count-leading-zeros operation used in floating-point adders to determine the shifting needed for normalization. It can be expressed as a recursive decomposition in which each bit of the next 1's entanglement channel number is computed in one step examining 2^k bit positions to determine bit k in the result. The circuit delay really depends on the delay in testing for 2^k bits being non-zero, `|t[pow2].v[(1<<pow2)-1:0]`, as there would be $O(WAYS)$ such steps with varying k.

Thus, this operation might be performed with $O(WAYS)$ gate delays, but could approach $O(WAYS^2)$ gate delays if the hardware implements the OR-reductions of step 2 using a tree of very narrow (e.g., 2-input) OR gates. If OR-reduction is inefficient, the `next` ALU function for 16-way entanglement might more appropriately be split into several pipeline stages rather than executing within a single clock cycle, but that transformation would be easy to implement and would not significantly affect pipeline throughput. It is appropriate to note that the student versions limited $WAYS$ to 8, which is easily viable within a single pipeline stage.

```
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.

4 SAMPLE ALGORITHM

Although the focus of the current work is not use of the PBP model, but efficiently implementing it, the Verilog implementations were subjected to fairly extensive testing. None of the designs were rendered to FPGAs nor ASICs, but all student projects were required to show that their simulation testing constituted 100% line coverage of the Verilog code they wrote. In addition, a simple algorithm demonstrating the utility of the PBP model was given as a test case. That algorithm is presented here as evidence that the PBP model is capable of implementing interesting quantum-inspired computations using the Tangled/Qat implementations.

4.1 Word-level Prime Factoring Algorithm

The word-level algorithm given in Figure 9 is a version of the prime factoring algorithm discussed in the earlier software-only PBP implementation[3], and that software was slightly modified to output the gate-level operations rather than to perform them. In order to fit the problem to the 8-way entanglement supported by the student implementations, the problem was simplified to factoring 15, rather than 221.

The `pint` (pattern integer) algorithm begins by creating the 4-pbit value 15. It then creates two Hadamard-initialized 4-pbit entangled superpositions, called `b` and `c` in the code. Note that the entanglement channels used by `b` and `c` are disjoint; `b` uses H(0) through H(3) and `c` uses H(4) through H(7). Thus, when `b*c` is computed, the result is actually 8-way entangled rather than 4-way. Had `b` and `c` used the same entanglement channels, that multiplication would only have computed 4-way entangled squares. The next step of the computation is to create an 8-way entangled single-pbit value, `e`, which is 1 only where the multiplication produced the value 15. Finally, multiplying `e*b` zeros the values of all non factors. When the non-destructive measurement of `f` is made, the values 0, 1, 3, 5, and 15 are printed: 0 because the non-factors were made 0, 1 and 15 because they are factors however trivial, and the desired results 3 and 5.

4.2 Tangled/Qat Prime Factoring Algorithm

Converting this word-level algorithm into Tangled/Qat code was done by modifying the software-only PBP implementation, but some changes had to be made. The Tangled/Qat non-destructive measurement mechanisms produce a single result at a time, and the 0, 1, and 15 results from the software implementation are effectively irrelevant. Beyond that, Tangled/Qat allows explicit access to the entanglement channels – because channel k would be representing the fact that k%16==`b`, it is thus also unnecessary to perform the

```
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).

multiplication of e*b; the result is really encoded in the 1-valued entanglement channels of e.

Figure 10 gives the complete Tangled/Qat code to place the prime factors of 15 in registers $0 and $1. The operations up to `not @80` were generated by the software-only system, and leave the value of e in @80. To obtain the first prime factor, the remaining (hand written) code simply looks for the first 1-value entanglement channels after the 1 and 15 factors. The last two and operations are implementing the k%16 operation mentioned above.

It is useful to note that the operations used by the software-only implementation use entirely conventional gates, and there is no penalty for doing that in the Tangled/Qat version. Further, the register allocation scheme greedily uses registers so that every intermediate computation's value is still available in a register at the end of the computation; in fact, the system performed extra operations just to preserve intermediate results. For example, the `or @80,@79,@79` operation is simply making a copy of @79 into @80 so that the not will not destroy the value in @79. These inefficiencies were maintained in the interest of more faithfully modeling the software-only implementation's computation, but far fewer registers, and fewer instructions, could have been used to obtain the same results with Tangled/Qat.

5 CONCLUSIONS

The primary conclusion on the current work must be that Tangled/Qat was found to be a highly feasible bit-serial, massively-parallel, SIMD computer architecture for Verilog implementation. The work described also has demonstrated that, using the array of bits (AoB) data structure, none of the parallel bit pattern (PBP) concepts requires excessively complex conventional hardware to provide an interesting, and flexible, implementation of entangled superposition and operations on such values. This was shown by pipelined synthesizable Verilog designs created and tested for 8-way entanglement by eight separate 3-4 person student teams, as well as by 16-way implementations created by the author.

Although the design of the Tangled instruction set was appropriate for this work, it proved to be surprisingly orthogonal to the problems encountered in integrating the Qat PBP coprocessor. Tight coupling of Qat, or a very similar design, should be relatively simple with nearly any base instruction set. Qat's lack of main memory interface, instead focusing on a large set of AoB registers, proved highly effective – and that facilitates separation between the host processor and Qat.

However, it also was observed that Qat's design was needlessly complicated in various ways simply to make it seem more like a quantum computer. Despite being "quantum-inspired" and having many similar capabilities, **Qat is very explicitly NOT a quantum computer nor a simulation of quantum computation**, but implements a very different model that supports entangled superposition. Most notably, the PBP model does not suffer the quantum problems of limited coherence time, "no copying," nor requiring all operations to be reversible. PBP does have the extra complication of explicitly managing entanglement channels, but this did not result in implementation difficulties.

Relative to the Qat instructions summarized in Table 3, the following simplifications now seem justified:

- The swap and cswap instructions are the only instructions requiring two AoB datapaths out of the Qat ALU and a second write port on Qat's register file. However, swap does make some algorithms more efficient because it replaces a three-instruction sequence, and cswap's multiplexor-like functionality also is useful. The "billiard-ball conservancy" of these gates also could simplify reducing Qat's power consumption by using a (conventional) adiabatic logic implementation. Without using adiabatic logic, the performance benefits seem to be outweighed by the hardware complexity, and it would probably make better sense to implement these operations as assembler macros rather than single instructions.
- The ccnot and cswap instructions are the only instructions requiring a third read port on Qat's register file and input datapath to Qat's ALU. This extra hardware does not appear to be justifiable on performance grounds, which argues against including these as instructions. Both easily could be implemented as assembler macros.
- The `cnot @a,@b` operation is actually equivalent to `xor @a,@a,@b`, which makes it also an excellent candidate for implementation as an assembler macro.

- The initialization instructions, `zero` and `one`, were significant in that quantum computers distinguish initialization from quantum execution – but Qat does not. Since copying is easy, it seems most reasonable to simply use two registers to hold these constant values. The obvious choices would be `@0` for 0 and `@1` for 1.
- In quantum computers, the Hadamard gate is used to create superposition that cannot be specified as an initial state – but that is not really a requirement in the PBP model. The Qat instruction set thus treated had as an initialization instruction. While that proved feasible, the gate-level hardware needed to generate a standard entangled superposition (for each of the WAYS-way possible entanglements) is greater than that required to simply reserve constant-initialized registers to hold these values. Thus, for the 16-way entanglement in Qat, it would make sense to simply reserve 16 registers as `@H0`, `@H1`, ..., `@H15`. Note that a quantum-like reversible Hadamard operator can be implemented by XOR with a Hadamard constant register.

In sum, the most important instructions for PBP are the `meas` and `next` instructions. Viewing Qat as a bit-serial, massively-parallel, SIMD, `meas` is recognizable as the relatively common SIMD ability for the control unit (in this case, essentially Tangled) to sample data from a single enabled processor. The `next` instruction is a new concept created as part of the work reported here, but even it is recognizable as closely related to parallel prefix operations. The result is that Tangled/Qat have largely revealed that, by combining the new PBP programming model with aggressive bit-level compiler optimization and mostly conventional-looking, bit-serial, massively-parallel, SIMD hardware, a surprisingly quantum-like system can be constructed without using any exotic physics. It is not difficult to imagine 1980s SIMD supercomputers[14] obtaining reasonable efficiently executing a PBP model.

Of course, although Tangled/Qat works well implementing 16-way entanglement, there is still the question of how large this architecture can scale and how much power savings it will provide. It appears that 16-way entanglement is near the practical scaling limit for AoB representations. However, the PBP model does not suggest representing higher degrees of entangled superposition using AoB, but instead using regular expressions compressing patterns in which AoB representations are treated as individual symbols. It remains to be seen if the manipulation of regular patterns of AoB blocks will effectively scale to very high entanglements while keeping efficiency high and power consumption manageable.

Additional future work is expected to involve rendering an improved version of the Tangled/Qat Verilog design to an FPGA to directly measure qualities of a hardware implementation.

ACKNOWLEDGMENTS

Thanks are due to the students of the Fall 2020 CPE480 Computer Architecture course. It is said that the best way to fully understand something is to teach it. These students not only suffered being taught about Tangled and Qat as an ongoing research project, but also provided valuable insights through their work on the course projects, which included working in teams of 3-4 to construct and test Verilog implementations of the complete system.

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.

It is also appropriate to acknowledge the work of the course teaching assistant, Jordan Caudill, for his help in guiding the students through such a strange set of projects despite the awkwardness of assisting student teams during a pandemic.

REFERENCES

[1] Akers. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* C-27, 6 (1978), 509–516. https://doi.org/10.1109/TC.1978.1675141

[2] Henry Dietz. 2017. How Low Can You Go?. In *Languages and Compilers for Parallel Computing - 30th International Workshop, LCPC 2017, College Station, TX, USA, October 11-13, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Lawrence Rauchwerger (Ed.), Vol. 11403. Springer, 101–108.

[3] Henry Dietz, Aury Shafran, and Gregory Austin Murphy. 2020. A Quantum-Inspired Model For Bit-Serial SIMD-Parallel Computation. In *Languages and Compilers for Parallel Computing - 33rd International Workshop, LCPC 2020, Stony Brook University, NY, USA, October 14-16, 2020.*

[4] Henry G. Dietz. 2019. Parallel Bit Pattern Computing. In *Tenth International Green and Sustainable Computing Conference, IGSC 2019, Alexandria, VA, USA, October 21-24, 2019*. IEEE, 1–5. https://doi.org/10.1109/IGSC48788.2019.8957188

[5] H. G. Dietz and W. R Dieter. 2007. *AIK, the Assembler Interpreter from Kentucky.* Technical Report. University of Kentucky Department of Electrical and Computer Engineering. http://aggregate.org/AIK/aik.pdf

[6] Edward Fredkin and Tommaso Toffoli. 1982. Conservative logic. *International Journal of Theoretical Physics* 21 (1982), 219–253.

[7] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. 2018. cQASM v1.0: Towards a Common Quantum Assembly Language. arXiv:quant-ph/1805.09607

[8] Alan Kramer, John S. Denker, B. Flower, and J. Moroney. 1995. 2nd order adiabatic computation with 2N-2P and 2N-2N2P logic circuits. In *Proceedings of the 1995 International Symposium on Low Power Design 1995, Dana Point, California, USA, April 23-26, 1995*, Massoud Pedram, Robert W. Brodersen, and Kurt Keutzer (Eds.). ACM, 191–196. https://doi.org/10.1145/224081.224115

[9] Catherine C. McGeoch. 2020. Theory versus practice in annealing-based quantum computing. *Theor. Comput. Sci.* 816 (2020), 169–183. https://doi.org/10.1016/j.tcs.2020.01.024

[10] Eleanor Gilbert Rieffel and Wolfgang Polak. 2000. An introduction to quantum computing for non-physicists. *ACM Comput. Surv.* 32, 3 (2000), 300–335. https://doi.org/10.1145/367701.367709

[11] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 124–134. https://doi.org/10.1109/SFCS.1994.365700

[12] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A Practical Quantum Instruction Set Architecture. *CoRR* abs/1608.03355 (2016). arXiv:1608.03355 http://arxiv.org/abs/1608.03355

[13] Michael Kirkedal Thomsen, Holger Bock Axelsen, and Robert Glück. 2011. A Reversible Processor Architecture and Its Reversible Logic Design. In *Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers (Lecture Notes in Computer Science)*, Alexis De Vos and Robert Wille (Eds.), Vol. 7165. Springer, 30–42. https://doi.org/10.1007/978-3-642-29517-1_3

[14] Lewis W. Tucker and George G. Robertson. 1988. Architecture and Applications of the Connection Machine. *Computer* 21, 8 (1988), 26–38. https://doi.org/10.1109/2.74

[15] Carlin Vieri. 1999. *Reversible computer engineering and architecture.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/80144

[16] Stephen Williams. 2020. *Icarus Verilog.* Retrieved April 23, 2021 from http://iverilog.icarus.com/

[17] Trevor Williams. 2011. *Covered - Verilog Code Coverage Analyzer.* Retrieved April 23, 2021 from http://covered.sourceforge.net/