Basic Operations And Structure Of An FPGA Accelerator For Parallel Bit Pattern Computation

Henry Dietz
Electrical and Computer Engineering
University of Kentucky
Lexington, Kentucky USA
hankd@engr.uky.edu

Paul Eberhart
Electrical and Computer Engineering
University of Kentucky
Lexington, Kentucky USA
pseber2@uky.edu

Ashley Rule
Electrical and Computer Engineering
University of Kentucky
Lexington, Kentucky USA
ash.rule@uky.edu

2021 International Conference on Rebooting Computing (ICRC) | 978-1-6654-2332-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICRC53822.2021.00029

***Abstract*—Parallel Bit Pattern computing (PBP) has been proposed as a way to dramatically reduce power consumption per computation by minimizing the total number of gate operations. In part, this reduction is accomplished by employing aggressive compiler optimization technology to gate-level representations of computations at runtime. Massive SIMD parallelism is used to obtain speedups while executing the optimized bit-serial code. However, the PBP model also can potentially exponentially reduce the number of active gates for each such operation by recognizing and operating on symbolically-compressed patterns of bits, rather than on each individual bit within a vector. This not only provides for efficient execution of traditional parallel code, but by using bit vectors to represent entangled superposition, enables quantum-like computation to be efficiently implemented using conventional circuitry.**

Building on lessons learned from various software and Verilog prototypes, this paper proposes a new set of basic operations and interface structure suitable for using inexpensive Xilinx Zynq-7000 boards to implement FGPA-hardware-accelerated PBP computation. Emphasis is on how these operations will implement quantum-like computation, as the first prototype system is currently still under development.

***Index Terms*—Computer Architecture, Parallel Bit Pattern Computing, SIMD, Optimizing Compilers, Logic Optimization**

I. Introduction

The history of high-performance computing has been dominated by the concept of leveraging the exponentially-growing circuitry per chip predicted by Moore in 1965 [1]. Speedup has come primarily by extensive use of parallel execution. The cluster computing model has allowed that parallelism to be scaled across chips and systems to previously unimaginable levels, with single supercomputers now harnessing millions of processors. However, the power consumed by each gate has not been decreasing as fast as our ability to incorporate more gates in a system has grown, resulting in disturbingly huge and increasing amounts of power being consumed by these massive supercomputers. The critical question is thus: how can power per unit of computation be dramatically reduced?

A. Reducing Power Per Computation

Although various techniques, such as reversible computing [2], can obtain modest reductions in power consumption, the new *parallel bit pattern* (*PBP*) [3] model aims to dramatically reduce power per computation in two distinct ways.

The first way in which efficiency can be improved is by aggressive gate-level optimization of programs [4], which is currently accomplished primarily by lazy evaluation using just-in-time compiler optimization technology implementing variable-precision `pint` (pattern integer) operations [3]. Although computers were made easier to use and somewhat faster for sequential execution by treating operations on multi-bit (word) data objects as atomic, that treatment implies many unnecessary gate-level operations will be performed. As a trivial example, using a 32-bit adder to add 4 to a value that is known to never exceed 1000 implies many times more powered gates than if we realize that a 10-bit number will suffice and adding 4 to it requires only as many gates as an 8-bit incrementer. It is easy to find examples where gate-level optimization can reduce active gate count by up to five orders of magnitude over just performing the standard word-level operations. There may be a modest reduction in serial performance by bit-serial execution, but that small constant trivially can be overcome by using SIMD parallelism at the bit level.

The second way echoes the savings associated with *quantum computing* [5]: use of *entangled superposition* to obtain up to exponential reductions in both storage space required and the number of active gate operations. Building effective quantum computers has thus far been impractical, but similar benefits can be obtained without use of quantum phenomena nor simulation of quantum systems. PBP implements entangled superposition using partially symbolic computation: operating on *patterns of bits* rather than individual bits. Operating on compressed bit patterns can reduce both the storage space and active gate count required by up to an exponential factor.

B. Representation Of Data

PBP is fundamentally a variation on bit-serial SIMD processing, and could execute algorithms originally intended for the DAP [6], MPP [7], CM2 [8], GAPP [9], MP-1 [10], etc. However, PBP also can manipulate entangled superposed values analogous to *qubits* in quantum computers. The key idea is that the value of each *pattern bit* (*pbit*) is represented by an *Array of Bits* (*AoB*) [11]. As shown in Figure 1, there are 2^k bits in the AoB representation of a k-way entangled pbit, and the bits in the same *entanglement channel* across multiple

978-1-6654-2332-8/21/$31.00 ©2021 IEEE
DOI 10.1109/ICRC53822.2021.00029

Authorized licensed use limited to: UNIVERSITY OF KENTUCKY. Downloaded on May 29,2025 at 00:14:38 UTC from IEEE Xplore. Restrictions apply.

Entanglement Channels

	7	6	5	4	3	2	1	0
pbit v0	1	0	1	0	1	0	1	0
pbit v1	1	1	0	0	1	1	0	0
pbit v2	0	1	0	1	1	1	0	1
Entangled Superposed Values	3	6	1	4	7	6	1	4

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

Fig. 1. AoB representation of three 3-way-entangled pbits

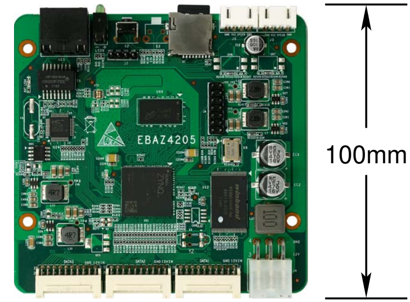

Fig. 2. EBAZ4205 Zynq Z7010 Board

pbits may be treated as entangled together. Pbits {`v2`, `v1`, `v0`} hold an entangled superposition of the 3-pbit value `v`.

Any gate-level operation applied to a pbit is applied in SIMD-parallel fashion to all its entanglement channels. Thus, each simple gate applied to a pbit takes $O(1)$ time, but seems to imply $O(2^k)$ work and power consumption for k-way entangled values. Fortunately, these $O(2^k)$ costs often can be avoided by taking advantage of the fact that entropy of AoB bit patterns tends to be very low: operations can be implemented on a compressed representation.

PBP compression is based on generative *regular expression* (*RE*) pattern representations [3]. For example, Figure 1's pbit `v1` is effectively $(1^2 0^2)^2$. Rather than treating individual bits as RE symbols, each k-way entangled AoB "chunk" is treated as a symbol in an RE describing a $(k + m)$-way entangled AoB. This offers up to $O(2^m)$ reduction in storage space, work, and power consumption, while still providing speedup through parallelism within a chunk. Duplicate k-way chunks also can be recognized, and *applicative caching* [12] used to avoid repeated computations, potentially offering an additional $O(2^k)$ reduction.

The PBP model is thus very promising, and has been shown to work for problems like prime factoring [3], but is still in a very preliminary research stage. Although a number of software simulators and even Verilog designs for simplified hardware (Tangled [13]) have been created, the work described in this paper is the ongoing first attempt to build a hardware accelerator for the PBP model tightly integrating with an existing conventional processor to make a usable prototype hardware and software system. A number of new challenges and insights have been discovered in this process, and these aspects are the primary focus of the current paper.

II. FPGA Target

The primary target for this work is a Xilinx Zynq Z7010 [14] on a EBAZ4205 board, as shown in Figure 2. These boards were used as the controller in the Ebit E9 Plus bitcoin miner, which is no longer profitable, and thus the boards sell as surplus for under $20. The PL (programmable logic) portion of the chip is a 28K logic cell FPGA including 17,600 LUTs, 35200 FFs, and 2.1Mb of Block RAM, and is programmed using Xilinx Vivado 2020.2. The PS (processing system) provides two ARM Cortex-A9 running at 866MHz, and runs PetaLinux, Xilinx's modified OpenEmbedded/Yocto with toolchain integration. The board additionally provides 256MB DDR3 RAM, 128MB SLC NAND flash, Ethernet PHY, 42 programmable I/O pins on three PH-B 2.0 connectors, etc. Everything is powered by 12V that can be provided via a 6-pin PCI-E power connector, and power draw depends on the FPGA programming, but is low enough to be driven by a laptop's 5V USB port via a boost converter.

In Tangled [13], the coprocessor interface allows direct access to host processor registers. That is not true here, so each coprocessor instruction is explicitly constructed in the PS and sent to the PL coprocessor, which responds by sending a return value to the PS host when done.

A. The AoB Register Model

The current version aims to implement 10-way entangled AoB chunk processing in the PL as a simple coprocessor operating on about a thousand automatically-managed 1024-bit AoB registers. Unfortunately, the Block RAM is strictly dual ported, and cannot support two reads and a write to the same memory in one cycle; this results in an additional cycle of delay for most ALU operations. Initially, the PL coprocessor will not access off-chip DRAM, but it would be feasible to invisibly use the Block RAM registers as a cache for a much larger register array in DRAM.

The AoB operands for coprocessor instructions are register numbers, indicated by a `@` prefix in the instruction set description. AoB vectors are never transmitted between the PL and PS. In fact, PL instructions only specify source registers; the PL return value is the coprocessor register that holds the result. A key innovation here is that the coprocessor ensures that **if any AoB values are identical, they will be recognized as such and share the same PL register**. Given this, all PL registers are essentially write-once (aka, single assignment); until a `reset`, the same register number always holds the same value.

Also unlike Tangled, there are no instructions to initialize an AoB register to all 0, all 1, nor any of the Hadamard entanglement channel patterns. Instead, at `reset`, these patterns are created in specific registers. Register 0 is all 0s, register

Authorized licensed use limited to: UNIVERSITY OF KENTUCKY. Downloaded on May 29,2025 at 00:14:38 UTC from IEEE Xplore. Restrictions apply.

1 is all 1s, and H(k) is in register $k+2$. This design has at least three important implications.

Multiple copies of an AoB pattern never exist, resulting in a significant reduction in memory usage. For example, `and @42,@42` logically generates a new copy of the AoB, but in fact simply returns that the result is the same as in register `42`. Similarly, `or @42,@0` would return `42`. Of course, it is easy for the PS host to infer the result of instructions like `and @42,@42` or `or @42,@0` without even asking the PL coprocessor; only operations that defy algebraic simplification need to be issued to the PL coprocessor.

Comparisons for equality are always trivially accomplished without accessing the coprocessor nor any AoB data. Iff two values are identical, then they will be references to the same register number. For example, the SIMD `any` test, checking to see if there exists at least one non-zero value in an AoB register, is simply checking that the register is not `@0` – because all other values have at least one 1 bit. Similarly, the SIMD `all` test would be checking if the register number is `@1` – because all other values have at least one 0 bit.

Using the single-assignment property of PL registers, it is trivial for the PS software to implement the applicative caching [12] mentioned earlier, thus avoiding repeat evaluation of any computation that has been performed before on identical AoB bit patterns. The applicative caching should normalize the operand order for commutative operations before checking the memo function so that, for example, `and @601,@22` will be recognized as equivalent to `and @22,@601`.

B. The PL Coprocessor Instruction Set

At this writing, the PL coprocessor instruction set consists of just 11 instruction types, as summarized in Table I. The LUTs and Delay columns give approximate cost of implementing each instruction in the Zynq 7010 for 1024-bit AoB; all are feasible and together consume about 60% of the FPGA's logic. Encoding the opcode requires just 4 bits. With up to R AoB registers, encoding a register number requires $ceil(log_2(R))$ bits – up to 1024 registers can be encoded in 10 bits. The instructions that take an entanglement channel (bit position number within an AoB) require a k-bit encoding for AoB registers supporting k-way entanglement – coincidentally also 10 bits for the current design. Thus, an instruction as sent to the PL fits in as few as 24 bits. One would expect that the value returned by an instruction is thus also 10 bits long, however, the `first` and `ones` instructions each can result in any of 2^k+1 possible values, so the the value returned by the PL coprocessor must have at least 11 bits. Using a 32-bit word to encode each instruction or result, up to 262,144 10-way entangled AoB registers could be supported using 32MB of DRAM, but performance would depend on PL Block RAM caching to hide DRAM access delay.

The following section describes the function of, and rationale for, each of the operations implemented by these instructions.

TABLE I
PL COPROCESSOR INSTRUCTIONS

Instruction		Description	LUTs	Delay
`reset`		reset the coprocessor		
`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
`rot`	`@a,b`	`@c=RotateLeft(@a,b)`	5120	4
`flip`	`@a,b`	`@c=Flip(@a,b)`	5120	4
`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
`first`	`@a`	`First b` where `@a[b]==1`	976	5
`ones`	`@a`	count of `Ones` in `@a`	1444	5

III. THE NEW BASIC OPERATION SET

Tangled [13] significantly simplified the set of basic operations as compared to earlier software simulators. However, additional lessons were learned in evaluating Tangled. As a result, the basic operation set described here is even simpler, despite incorporating some new, more general-purpose, operations. In part this is because **the current AoB accelerator is designed not only to efficiently handle quantum-like entangled superposition operations, but also to efficiently execute more traditional SIMD code**. In fact, it may offer a comparable level of benefit in executing purely conventional SIMD code by treating the k-way AoB processing hardware as 2^k bit-serial SIMD *processing elements* (*PE*s).

Rather than discussing the instructions individually, it is useful to describe them grouped by the type of functionality provided.

A. Initialization

One of the key insights to come from Tangled is that there was no need to distinguish initialization, quantum computation, and measurement phases [13]. In a quantum computer, there are explicit initialization operations setting qubits to the non-superposed values 0 or 1. Tangled took that a step further, making Hadamard entanglement patterns valid initialization instructions. The PBP notion of entanglement is based on the concept of entanglement channels, and entangling multiple bits requires assignment of a unique set of entanglement channels for each dimension of entanglement. For example, pbit `v0` in Figure 1 is the pattern for H(0), and `v1` is the pattern for H(1). These patterns were generated by the Tangled `had` instruction.

In contrast, the current design has a distinguished initialization phase that can be triggered at any time by a `reset` operation. However, reset implies clearing the hardware and establishing the value of 0 in register 0, 1 in register 1, H(0) in register 2, H(1) in register 3, etc. There are no user variable initializations during a reset, nor are there any initialization instructions that can be executed later. After reset, these entangled-superposition constants simply can be referenced directly by their register numbers. This works because, unlike quantum computers, there is no constraint on fanout/copying nor any limit on the error-free period of coherence.

Authorized licensed use limited to: UNIVERSITY OF KENTUCKY. Downloaded on May 29,2025 at 00:14:38 UTC from IEEE Xplore. Restrictions apply.

B. Arithmetic/Logic Operations

While Tangled implemented a variety of arithmetic and logic operations commonly used in quantum computing, including Pauli, Toffoli, and Fredkin gates [15], it also was observed that some of these operations required an additional register read and/or write port that made the operation more expensive to implement [13]. Perhaps even more significantly, the gate-level optimization software never actually generated any of these types of gates. On that basis, only `AND`, `OR`, and `XOR` (exclusive OR) are implemented here. One might expect that some type of NOT gate also would be needed, but `XOR` with register `@1` implements precisely the same functionality.

C. Permutations

Neither Tangled nor its software-only predecessors implemented any operations permuting AoB vectors. Such operations are of course useful as inter-processor communication primitives for traditional SIMD computing. It might seem strange that one would want to re-arrange entanglement channels, but these rearrangements are analogous to phase operations in quantum systems.

The `RotateLeft` operation clearly constitutes a phase shift, and is fairly easy to understand as such. More complex is `Flip`, which is a parametric type of reversing permutation network. The Flip concept can be traced to the network in Staran [16]. However, Hacker's Delight [17] notes that, in a private communication from Guy Steele, an improved control structure was suggested for a within-a-word Flip instruction, and that is essentially the operation implemented here for AoB vectors. `Flip` with a control value of 2^h is equivalent to swapping neighbors within the H(h) pattern. Other control values allow "mirroring" transformations that, combined with the 2^h patterns, allow efficient implementation of algorithms such as bitonic sort of values within an AoB.

D. Entanglement-Channel Addressing

In quantum computers, it is difficult to create an entangled superposition holding a particular set of values. It also is impossible to read a specifically-selected value – only random sampling is supported. In contrast, PBP allows efficient addressing of an entangled superposition by entanglement channel.

The simplest such operation is `Toggle`, which inverts the value in the selected entanglement channel of a pbit. This allows direct construction of the elements in an entangled superposition; for example, a pbit with only entanglement channel 42 set to 1 could be created by `Toggle` of position 42 in `@0`. However, if a run of entanglement channels must be turned on, this is inefficient; thus, there is also a `Domino` operation. `Domino` toggles not only the specified entanglement channel, but also all those below it – like the chain reaction as one domino falls into another. For example, a pbit with entanglement channels 42 to 601 set could be accomplished by `Domino` of position 41 in an initially all-zero pbit followed by `Domino` of 601.

Measurement of a selected entanglement channel's value is accomplished using the `Measure` operation, which returns either `@0` or `@1`. Quantum-like random measurement simply would specify a randomly-selected entanglement channel. However, unlike quantum systems, measurement does not destroy the entangled superposition. Thus, repeated measurements of different channels can read the complete entangled superposition, whereas a quantum computer would have to repeat the complete computation to measure a second randomly-selected value. This makes PBP at least exponentially more efficient than quantum computing for such operations.

E. Aggregate Operations

As powerful as PBP measurement is, it only can sample one entanglement channel at a time. In networking terms, a message is being sent from one entanglement channel to the host. Aggregate function communication [18] [19] is an alternative model for network communication in which all nodes submit values, the network computes properties of the values aggregated, and these properties may be read from the network. Thus, in a PBP system, aggregate function communications can sample properties of many channels in a single operation. Theoretically, most aggregate function computations take $O((log_2 k)^2)$ time for a k-way entangled system, but in practice the circuit delay is often small enough to behave as $O(1)$ operations relative to the network overhead – or, in this case, the coprocessor interface overhead.

The `First` aggregate function operation returns the entanglement channel number of the first 1 value within an entangled superposition. There is no comparable concept in quantum systems because their entangled superpositions are not ordered. This allows skipping over groups of 0-value entanglement channels in $O(1)$ time, dramatically improving the time it takes to sample all values in an entangled superposition.

The second aggregate function operation is somewhat more familiar: population count, or `Ones`. This operation simply sums (performs a reduce-add over) all the bits in an entangled superposition and returns that count. Since the number of entanglement channels for a k-way system is 2^k, the `Ones` count returns the probability of a 1 in parts per 2^k. Again, it would take many runs of a quantum computer to approximately determine a similar metric.

IV. Conclusions And Future Work

The simplicity and flexibility of the new basic operations for PBP show great promise for exponential reduction in gate operations, and hence power, per computation. However, this is a very new and still evolving model for computation.

After demonstrating good efficiency using a single EBAZ4205, the plan is to use the programmable I/O pins to communicate via an aggregate function network [18] [19] so that computations can be efficiently coordinated across a cluster of EBAZ4205 boards. Both an 8-node portable proof-of-concept system and a larger system with at least 128 nodes are being designed.

Authorized licensed use limited to: UNIVERSITY OF KENTUCKY. Downloaded on May 29,2025 at 00:14:38 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 114–117, 1965.

[2] M. P. Frank, R. W. Brocato, B. D. Tierney, N. A. Missert, and A. H. Hsia, "Reversible computing with fast, fully static, fully adiabatic cmos," in *2020 International Conference on Rebooting Computing (ICRC)*, 2020, pp. 1–8.

[3] H. Dietz, A. Shafran, and G. A. Murphy, "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*, 2020.

[4] H. Dietz, "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*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed., vol. 11403. Springer, 2017, pp. 101–108.

[5] E. G. Rieffel and W. Polak, "An introduction to quantum computing for non-physicists," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 300–335, 2000. [Online]. Available: https://doi.org/10.1145/367701.367709

[6] S. F. Reddaway, "Dap—a distributed array processor," *SIGARCH Comput. Archit. News*, vol. 2, no. 4, p. 61–65, Dec. 1973. [Online]. Available: https://doi.org/10.1145/633642.803971

[7] Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers*, vol. C-29, no. 9, pp. 836–840, 1980.

[8] L. Tucker and G. Robertson, "Architecture and applications of the connection machine," *Computer*, vol. 21, no. 8, pp. 26–38, 1988.

[9] R. Morley and T. Sullivan, "A massively parallel systolic array processor system," in *[1988] Proceedings. International Conference on Systolic Arrays*, 1988, pp. 217–225.

[10] T. Blank, "The maspar mp-1 architecture," in *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, 1990, pp. 20–24.

[11] H. G. Dietz, "Parallel bit pattern computing," in *Tenth International Green and Sustainable Computing Conference, IGSC 2019, Alexandria, VA, USA, October 21-24, 2019*. IEEE, 2019, pp. 1–5. [Online]. Available: https://doi.org/10.1109/IGSC48788.2019.8957188

[12] R. M. Keller and M. R. Sleep, "Applicative caching," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 1, p. 88–108, Jan. 1986. [Online]. Available: https://doi-org.ezproxy.uky.edu/10.1145/5001.5004

[13] H. Dietz, "Tangled: A conventional processor integrating a quantum-inspired coprocessor," in *Proc. 2021 Int. Conf. Parallel Processing*, 2021.

[14] Xilinx, *Zynq-7000 SoC Technical Reference Manual*, 2021.

[15] E. Fredkin and T. Toffoli, "Conservative logic," *International Journal of Theoretical Physics*, vol. 21, pp. 219–253, 1982.

[16] K. E. Batcher, "The flip network in staran," in *Proc. 1976 Int. Conf. Parallel Processing*, 1976, pp. 65–71.

[17] H. S. Warren, *Hacker's delight*. Pearson Education, 2003.

[18] R. Hoare, H. Dietz, T. Mattox, and S. Kim, "Bitwise aggregate networks," in *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*, 1996, pp. 306–313.

[19] H. G. Dietz, T. M. Chung, and T. I. Mattox, "A parallel processing support library based on synchronized aggregate communication," in *Languages and Compilers for Parallel Computing*, C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 254–268.

Authorized licensed use limited to: UNIVERSITY OF KENTUCKY. Downloaded on May 29,2025 at 00:14:38 UTC from IEEE Xplore. Restrictions apply.