Globally Irreversible Locally Reversible

Colin Drewes
University of California, San Diego
colindrewes@gmail.com

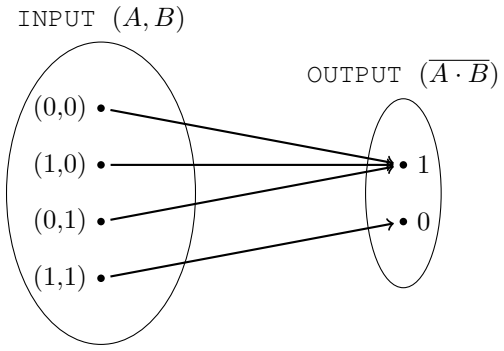
***Abstract*—The 60 year trend of exponentially increasing computational efficiency is coming to a close. This scaling has relied primarily on 1) the exponential decrease in the capacitance of transistors by reduction in gate size, and 2) a reduction in the voltage difference between a logical 0 and logical 1. Regardless of continuing scaling innovations in the next decade, we will reach a theoretical wall in reducing transistor energy in traditional digital logic. This has prompted researchers to consider alternative models of CMOS based computation.**

Their key observation is that there is nothing intrinsic to computation that necessitates the dissipation of heat when processing a logical signal. Heat dissipation in digital logic is primarily due to the destruction of information (i.e flipping a bit). This is an artifact of digital design being primarily *surjective* and *non-injective* (i.e an AND gates collapse 4 possible input configurations into 2 possible output configurations). This has prompted researchers to consider *reversible computation* as an alternative computational paradigm. This involves a transformation of digital logic to be *bijective* at all levels (logical reversibility) while recapturing signal energy after reversing the computation (physical reversibility). Under this construction, there is no known theoretical limit on the amount of energy that can be recovered from a computation.

However, full logical and physical reversibility demand new simulation libraries, HDLs, processor implementations, and compiler/programming languages. In this paper, we will present the idea of *globally irreversible locally reversible* architecture for processor design. This will exploit the energy savings of reversible computation locally (across a single RISC-V instruction), but retain simple programability through global (across successive RISC-V instructions) irreversibility.

I. Introduction

We begin with a description of *irreversible* computation. It is trivial to see that our current model of digital logic is inherently *irreversible*. For example, observe the possible transformations from inputs to outputs for a standard NAND gate. This gate is performing a *non-injective* operation (i.e

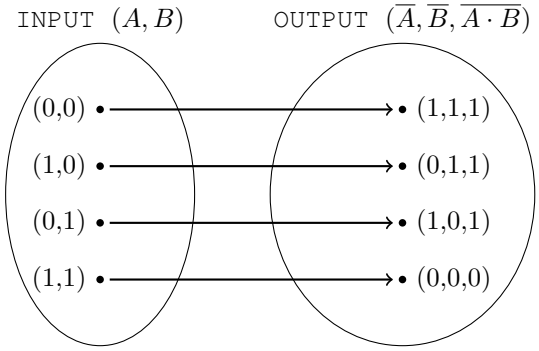


it is not possible to recover the input from the output). OR, NOR, AND all follow a similar structure. This is what we know to be an *irreversible* operation, where either a bit is erased or computational paths are merged. According to Landauer's limit, any *irreversible* operation must generate an amount of energy or heat lower bounded by:

$$E = k_B T \ln 2$$

where k_B is the Boltzman constant [2]. While Landauer's limit is a hard lower bound at 0.0175 eV (at room temperature 20°C) per bit erased, real-world constraints prevent energy expenditure from decreasing bellow 1 keV [3]. This paradigm leaves no path for decreasing gate energy bellow 1 keV, let alone reaching and exceeding the Landauer limit. As a result, there is no possible long-term energy scaling for digital logic without considering novel paradigms.

Consequently, *reversible* computation [1] has been proposed. No limits such as the Landauer's principle are known for *reversible* computation [1] as there can exist—under proper implementation—no bit erasure. For example, consider the following design of the NAND gate [9]:



In contrast to the standard NAND implementation, this gate is *injective*. A similar process can be taken to construct *injective* versions of AND, OR, NOR, etc [9]. These gates are *logically reversible*, and the building blocks for reversible computation.

However, logical reversibility is necessary but *not* sufficient for reversible computation (and thus reducing energy dissipation). The injective gates must be implemented in a *physically reversible* way, so that the charge used in the computation may be recovered as opposed to being dissipated in heat energy. Physical reversibility, or the recovery of energy, relies on being able to *unwind* the computation (i.e perform the computation backwards). This necessitates logical reversibility, as to perform the computation backwards there must be

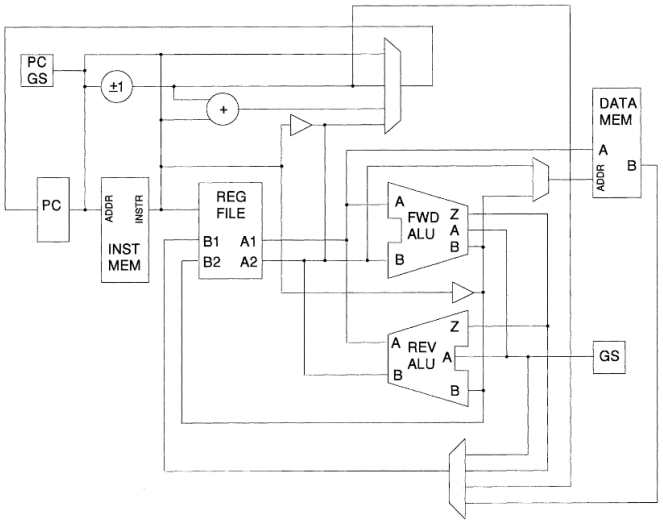


Fig. 1. Carlin Vieri's `Pendulum` processor data path from [8]. All operations are fully reversible.

sufficient products to recompute the inputs. There are multiple ways to implement physical reversibility, but this work will examine S2LAL presented in [4] although this is not the focus of this work.

While the benefits of reversible computing are clear, its adoption requires redesigning the entire computer stack. New EDA tools are needed to implement the physical reversibility, HDLs to describe reversible gates, processor architectures/ISAs, and programming languages/compilers. This appears insurmountable to many.

To ease the transition to reversible computing, this work aims to present at a high level a processor architecture that exploits the benefits of reversible computing in *localized* areas, but will separate these reversible logic blocks with irreversible copy operations, and so, be *globally* irreversible. We claim this will result in the majority of power saving gains that reversible computation will offer, while not requiring new ISAs and programming language paradigms.

II. S2LAL

Static two-level adiabatic logic (S2LAL), pioneered by Michael Frank [7], is one possible approach for physically realizing CMOS based reversible computing. Note that *physically reversible* implementations of computation, such as S2LAL, can only implement *logically reversible* computation. For a circuit to be *physically reversible*, often referred to as adiabatic, it must adhere to the following properties [7]:

1) Do not use diodes due to their inherent voltage drop
2) Don't turn a transistor on when there is a voltage differential between the source and drain
3) Don't turn a transistor off when there is a current differential between the source and drain unless the source and drain are connected along another path

Adhering to these tenets can be used build CMOS circuits that have asymptotically 0 energy dissipation (barring leakage). Again, these tenets can only be used to implement *logically reversible* computation. S2LAL incurs some complexity and speed overhead, but is capable meeting these constraints. Even though implementing circuits in S2LAL is slower than the irreversible alternative, the lower energy dissipation will allow for significantly greater transistor density while remaining within thermodynamic limits [6].

The details of S2LAL are best left to [5], but we will discuss some of the current simulation and implementation results. A slight variant of S2LAL, called 2LAL, has been simulated and physically implemented with results that we can discuss here.

Cadence Spectre simulations are performed on 2LAL on 350nm and 180nm MESA (Sandia's in house fab) operating at 1MHz [7]. At 350nm, the energy dissipated per FET was measured 230 eV. At 180nm, the energy dissipated per FET was measured at 43 eV. Comparable results have been observed from other fabs, including TSMC. This is significantly better than even end of life CMOS energy dissipation of 1 keV that we could achieve in the best-case of the next 30 years using standard circuit construction! Physical implementations of 2LAL have also been created awaiting measurement [7].

We will take adiabatic physical circuits as a given, which is a realistic assumption based on the referred Michael Frank work at Sandia. For the remainder of this work we will focus on the logical reversibility challenges, particularity for processor implementations.

III. Reversible Processors

Computer operations per second have historically scaled linearly with power usage [3]. This scaling will lead power usage for a ZFLOP/s computer to reach 10s of Gigawatts power dissipation. Even if transistor efficiency was operating at the boundary at which computation begins to become indistinguishable from thermal noise, a ZFLOP/s computer would consume more than a Megawatt. Even if leaps were made in CMOS or post-CMOS technologies, computational efficiency is still restricted by Landauer's limit, which prevents any ZFLOP/s computer from operating bellow roughly 500 kilowatts. This is untenable for a number of reasons. First, because the majority of the power consumed is expelled in heat, we cannot to continue to increase FLOP/s without the internal temperature of the chips causing damage to themselves. Second, it becomes prohibitively difficult to provide power to large scale computer systems [3].

As a result, processor architecture has been an attractive place to apply reversible computing which theoretically can break the relationship between computation and power—in ideal circumstances. In reality, parasitic effects and gate leakage retain this linear relationship between power consumption and FLOP/s, but this is something that can be pushed artificially close to zero. Regardless, even with the parasitic/leakage in CMOS technology, we can achieve tens to hundreds times better power efficiency than non-reversible counterparts.

The most complete of processor implementations which leverage reversible techniques is Carlin Vieri's 1999 PhD dissertation [9]. Vieri utilizes a primitive variant of S2LAL/2LAL called Younis Pipelined Split-Charge Logic Recovery [10]—another power clock based scheme. As previously mentioned, adiabatic circuit techniques can only implement *logically reversible* computation (i.e. information cannot be destroyed) [5]. This means the processor architecture must be completely reversible. The datapath of this architecture is presented in Figure 1. This requires re-designing CPU architecture, instruction set architectures, and programming languages. We discuss these constraints in the following sections:

A. CPU Architecture

The execution path of any instruction within the CPU must invertible at a later date. This means that all intermediate products must be stored so that the computation can be reversed to re-derive the inputs from the products (the fundamental notation of logical reversibility). For example, an `ADD` instruction would take two integers A and B, and result in A+B, and A. Even though the result A+B is all that we desired, we must also store A, so that the input B can be re-derived (through subtraction). It is not immediately clear why performing a subtraction operation is needed to improve power performance. To understand this, let's appeal to the first principles of reversible computation.

For a computation to release asymptotically 0 energy, the computation must not destroy information. For an operation like `ADD`, the inputs A and B are not recoverable from merely A+B, thus there must be some information destruction. So, either A or B must be retained (WLOG we assume that A is kept), so that A and B can be derived from A+B and A. A+B, the desired result of the computation, may then be used for whatever purpose it was computed for. If we no longer want to continue to track both A+B and A as separate signals, we can collapse them back into A and B by performing the opposite operation of what was performed originally. In this case it would be a subtraction operation.

The operation of any instruction within the CPU must follow similarly. For an instruction to be executed, sufficient products must be stored so that the instruction may be subsequently undone. Memory operations may not result in the destruction of data, thus necessitating "swap" operations rather than read and write operations. A history of program counter values must always be stored so as to not overwrite the current program counter value non-reversibly. The data path is presented in Figure 1, with a full description of the CPU and its operation in [9]. Due to the fundamental reversibility of the CPU, the instruction set must be built to accommodate this structure.

B. Instruction Set Architecture

The pendulum instruction set of [9] is based on the MIPS RISC architecture. Both future and past instructions must be known at all times to meet the conditions of reversibility. For an instruction to be later reversed, that means every instruction operation must be an injective function with all products saved so that the instruction can be reversed at a later point. We will briefly characterize the modifications made to the RISC instruction set to be reversible:

1) Register to Register Operations: The author of [9] draws a distinction between register to register instructions that require retaining extra information (i.e information not directly used in the computation) called non-expanding, and instructions that do not require saving information, called expanding. The non-expanding register to register instructions take the form of:

$$R_{sd} \leftarrow \mathcal{F}(R_{sd}, R_s)$$

These operations can be performed reversibly, as the function being applied to the operands R_{sd}, R_s will not overwrite another register (as this would be irreversible). So, the result can only be placed in one of the operand registers. In a non-expanding operation such as this, the operation can be reversed purely based on the result and the input. For example, take $\mathcal{F} = \texttt{ADD}$. You could perform a register to register operation such as $R_{sd} \leftarrow \texttt{ADD}(R_{sd}, R_s)$. You do not need to maintain extra information to reverse this instruction as long as R_s is still present, which it must be if we are unrolling instructions in the order which they were executed (necessary for reversible computation)! For example, to reverse this instruction the core must execute $R_{sd} \leftarrow \texttt{ADD}^{-1}(R_{sd}, R_s)$. As we observed, non-expanding register to register operations can simply be reversed.

Expanding operations require storing extra information alongside the result of the instruction and what the inputs to the instruction were. These instructions take the form of:

$$R_{sd} \leftarrow \mathcal{F}(R_{sd}, R_s) \oplus P$$

The value of P must be stored by the CPU so that the instruction can be reversed at a later date. For example if a left shift operation is performed $\mathcal{F} = \texttt{LSHIFT}$ there is not sufficient information to recover the inputs. For example, we could perform $R_{sd} \leftarrow \texttt{LSHIFT}(R_{sd}, R_s)$, where R_{sd} was a register that we want to shift up by R_s bits. This is not sufficient to recover the high-order bits that were pushed out. Similarly if we were to preform $R_s \leftarrow \texttt{LSHIFT}(R_{sd}, R_s)$, we would have the original value to shift but not how much it was shifted by. This means more information needs to be retained and stored, this value called P, and applied in the reverse step. P will take extra memory to store. All register to register instructions fit this structure. This is somewhat limiting in terms of which registers can be accessed. If registers need to be freed to compute on other data, a memory operation needs to be performed, which again must be fully reversible.

2) Memory Access: Memory access, whether it be instruction fetches, the register file, or the data memory, can only perform exchange operations. In this way, memory addresses are never overwritten and can be unraveled at any time, thus retaining reversibility. The full design of memory operation instructions can be found in [9].

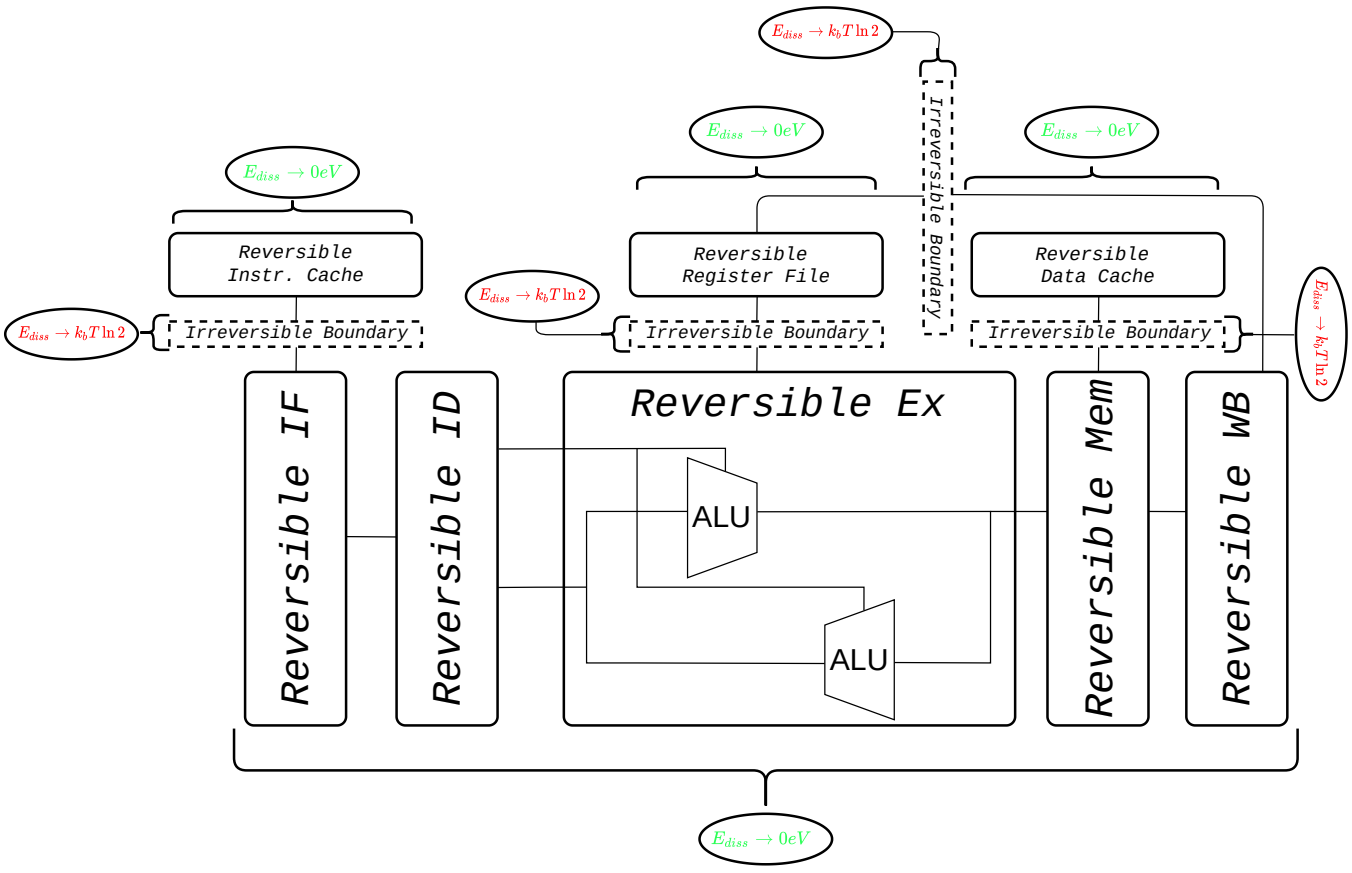


Fig. 2. Proposed alternative *locally* reversible architecture that is *globally* irreversible. This will allow the CPU architecture to execute a standard ISA, incurring minimal penalty compared to a fully reversible architecture.

3) Control Flow: Branch instructions provide the condition to be evaluated as in traditional RISC architecture, the target address, and the value being compared. If a branch is taken the program counter value is exchanged (as you cannot overwrite information) with the target address. The instruction in the destination address must be an identical branch instruction that points back to where the original branch was from. This allows all branches to be taken in reverse. The implementation of this is in [9].

C. Programming Languages

Novel programming languages must be built around the reversible instruction set. These programs must be completely reversible. This requires a different perspective for the programmer, or sophisticated techniques to convert traditional programs to a reversible program. Reversible computers are provably universal (they are computationally equivalent to traditional computers), but to program them requires a significant paradigm shift for programmers and engineers. To alleviate this we propose nesting reversible units within irreversible boundaries. This will provide the majority of the power saving benefits, while not requiring new architectures or programming language paradigms. We refer to this structure as *globally irreversible locally reversible* computation.

IV. Globally Irreversible Locally Reversible

Figure 2 presents a hypothetical architecture of the *globally irreversible locally reversible* processor. The goal here is to retain a simple and unmodified instruction set such as RISC-V, while leveraging the low-power benefits of reversible computing. The design is divided into Reversible Regions cordoned off by Irreversible Boundaries:

A. Irreversible Boundaries

The irreversible boundaries of Figure 2 are used to pass information between reversible regions. In implementation these will just be a set of registers that can be written to that contain data and control signals. These registers will hold n bits of data (1 word of data and control signals). As a result, these operations cannot perform any better than $nk_bT \ln 2$ of energy dissipation as they are irreversible. However, the goal of this architecture is to hoist some of the complexity in implementing reversible operations onto irreversible operations, at a small cost, and then perform the majority of the computation in the reversible hardware.

B. Reversible Regions

Every reversible region will follow the principles of reversible computation (i.e adiabatic circuit implementing non-

destructive computation). These regions can be executed forward to produce some result (and potentially unneeded products to guarantee reversibility). The energy dissipation, E_{diss}, of these regions can asymptotically approach 0 eV if operated properly. The memory and pipeline will be implemented in a reversible manner.

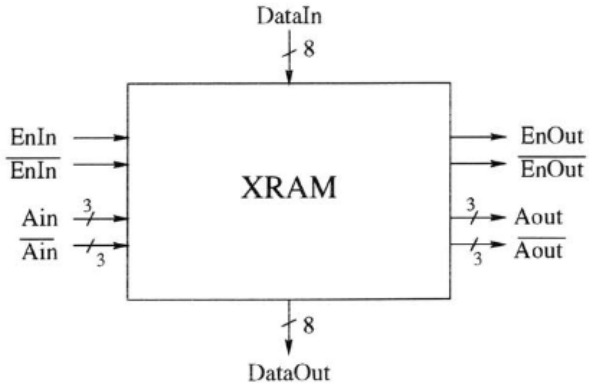


Fig. 3. Vieri's proposed memory design [9] that performs "sawp" operations as to avoid overwriting data which would be an irreversible operation.

1) Memory: As we have seen from [9], it is possible to build fully reversible memories capable of performing "swap" operations. This memory system can then be used for the instruction cache, register file, and data cache. A very high level description of the XRAM flow from Vieri's thesis is presented in Figure 3. The memory block takes some input value (8 bits in Vieri's design) and has an 8 bit output. The control signals allow a "swap" operation be be performed, thus exchanging the `DataIn` with the value at the address `Ain` which is bumped to `DataOut`. The XRAM can then simply be reversed so that the `DataOut` value is written to the value stored at address `Bin`.

In our proposed architecture, the memories blocks will be cordoned off with irreversible boundaries as in Figure 2. If a read needs to be performed the following steps can be taken. The `DataIn` and `DataOut` value is left in whatever its current state is. The `Ain` value and `Bin` will be irreversibly overwritten with the desired address. The reversible memory will then be executed forward to get the desired value on `DataOut`, which is then irreversibly copied. The memory block can then be executed in reverse to restore the value to its original memory location. Internal to the memory block there can be theoretically 0 energy dissipation. The only operations that dissipate energy are the irreversible copies from the irreversible boundary, which constitutes a small percentage of total transistors used in the memory operation. The simple RISC pipeline in Figure 2 will interact in each stage with the memory in this manner.

a) Instruction Fetch: The instruction fetch copies the program counter to the reversible instruction cache. The instruction cache can be executed forward to get the desired instruction, which is then copied irreversibly, and then the instruction cache can be reversed to restore that value back to its correct memory location.

b) Execution: After the control signals from the decode stage have been determined, the registers needed for computation can be gathered from the reversible register file. The register number is used as the `Ain` address. The register file is executed forward to get the requested register value which is irreversibly copied, then the register file is reversed to restore that register value to its original location in memory. The original instruction can then be executed on these values. As in the previous section, no information will be destroyed and all intermediate products are saved, which will be necessary for reversing the operation.

c) Memory: If products need to be written to memory the address can be provided to the reversible data cache and a swap operation can be performed with the new data that needs to be written.

d) Write Back: Finally, if data needs to be written back to the data cache, a value can be written the the irreversible boundary, and a swap operation is performed to set the value in memory.

2) Pipeline: Outside of the memory operations the processor pipeline will closely resemble that of Vieri's design [9]. This means the pipeline itself and actual computation being performed will expel asymptotically 0 eV. Instead of unwinding the instruction at a later data when the program as a whole is being unwound as in Vieri's construction, by decoupling the memory from the pipeline in our design the instructions path of execution could be immediately unwound after its execution. This means that the instruction set implemented by our proposed design in Figure 2 could be a standard RISC implementation like RISC-V. No extra space would be needed to store products for future use in reversing the processor as the pipeline is being unwound immediately after the execution of the instruction. This would also break the dependence across instructions and across branches. For a relatively small cost we can dramatically simplify the control flow and instruction set of a reversible processor.

V. Renewed Demand for Parallel Architecture and Conclusion

Reversible logic and its implementation comes at a significant cost in performance. In particular, every gate needs to be tied to to a set of synchronized power clocks. These clocks provide the power to the system and allow adiabatic transitions to occur. The relationship between the energy dissipation of an adiabatically driven transistor, and clock "period" τ_{tr} is given by [5]:

$$E_a = \zeta_{tr} C_L V_{dd}^2 \frac{RC_L}{\tau_{tr}}$$

In this relation, ζ_{tr} is the "smoothness" of the voltage clock; C_L is the capacitance of the transistor; V_{dd} is the high voltage value; R is the resistance of the charging path; τ_{tr} is the duration of the linear ramp of the power clock [5]. As $\tau_{tr} \to 0$ then $E_a \to 0$. Thus, as the clock period lengthens (τ_{tr}), the energy consumption decreases.

This means that slower clock frequencies will be architecturally desirable. Such a shift will necessitate designs with greater parallelism, larger instruction re-order buffers, and in general more operation performed over a single clock cycle.

REFERENCES

[1] C. H. Bennett, "Logical reversibility of computation," *IBM journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973.

[2] C. H. Bennett, "Notes on landauer's principle, reversible computation, and maxwell's demon," *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, vol. 34, no. 3, pp. 501–510, 2003.

[3] M. P. Frank, "Why reversible computing is the only way forward for general digital computing." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2019.

[4] 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.

[5] 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)*. IEEE, 2020, pp. 1–8.

[6] M. P. Frank *et al.*, "Reversibility for efficient computing," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and ..., 1999.

[7] M. Frank, R. Brocato, T. Conte, A. Hsia, A. Jain, N. Missert, K. Shukla, and B. Tierney, "Special session: Exploring the ultimate limits of adiabatic circuits. in 2020 ieee 38th int'l conf. on computer design (iccd), hartford, connecticut, usa, oct. 18–21, 2020," *IEEE. doi*, vol. 10.

[8] C. J. Vieri, "Pendulum–a reversible computer architecture," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[9] C. J. Vieri, "Reversible computer engineering and architecture," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[10] S. G. Younis and T. F. Knight Jr, "Practical implementation of charge recovering asymptotically zero power cmos," in *Proceedings of the 1993 symposium on Research on integrated systems*, 1993, pp. 234–250.