A Fully Reversible Asymptotically Zero Energy Microprocessor *

Carlin Vieri, M. Josephine Ammer, Michael Frank,
Norman Margolus, Tom Knight
MIT Artificial Intelligence Laboratory, Cambridge MA 02139, USA

May 1, 1998

Abstract

Reversibility is the only way to compute with asymptotically zero power, and is a novel approach to low power, low energy computing. Recent implementations of reversible and adiabatic [15, 7] logic in standard CMOS silicon processes have motivated further research into reversible computing. The application of reversible computing techniques to reduce energy dissipation of current generation CMOS circuits has so far been found to be limited, but the techniques used to design reversible computers are interesting in and of themselves, and other technologies, such as Josephson Junctions and quantum computers as well as future CMOS technologies, may require fully reversible logic. This paper discusses the design of a fully reversible microprocessor architecture.

Computing with reversible logic is the only way to avoid dissipating the energy associated with bit erasure. Low energy techniques such as voltage scaling lower the cost of erasing information. Techniques such as clock gating effectively reduce the number of bits erased. Reversible techniques have already been used to lower the cost of bit erasure for nodes that have a high cost of erasure, but this work is directed at saving every bit, computing fully reversibly. The goal is to convert a conventional RISC processor to completely reversible operation. This investigation indicates where bit erasure happens in a conventional machine and the varying difficulty across datapath modules of computing without erasing bits.

The initial motivation for reversible computing research came from an investigation of fundamental limits of energy dissipation during computation [8]. The link between entropy in the information science sense and entropy in the thermodynamics sense, exhibited by Maxwell's demon [9], requires a minimum energy dissipation of $k_{\mathrm{B}}T\ln 2$, where k_{B} is Boltzmann's constant, when a bit is erased. Erasing a bit is a logically irreversible operation with a physically irreversible effect. A reversible computer avoids bit erasure.

Judicious application of reversibility in adiabatic circuits has already proven its usefulness in reducing energy dissipation [2]. This paper examines the complexity and difficulty in avoiding bit erasure entirely and discusses a set of techniques for designing reversible systems.

1 Introduction

Power and energy dissipation in modern microprocessors is obviously a concern in a great number of applications. Riding the technology curve to deep submicron devices, multi-gigahertz operating

*This work is supported by ARPA contracts DABT63-95-C-0130

frequencies, and low supply voltages provides high performance and some reduction in dissipation if appropriate design styles are used, but for applications with a more strict dissipation budget or technology constraints, more unusual techniques may be necessary in the future.

Adiabatic or energy recovery circuit styles have begun to show promise in this regard. Motivated by the result from thermodynamics that bit erasure is the only computing operation associated with required energy dissipation, various techniques that either try to avoid bit erasure or try to bring the cost of bit erasure closer to the theoretical minimum have been implemented. To truly avoid bit erasure, and therefore perform computation that has no theoretical minimum dissipation, the computing engine must be reversible. Losses not associated with bit erasure are essentially frictional, such as the non-zero resistance of "on" transistors, and may be reduced through circuit design techniques such as optimally sized transistors, silicon processing techniques such as silicided conductors, and by moving charge through the circuit quasistatically such as constant current ramps in adiabatic circuits.

This paper discusses the engineering requirements of building a fully reversible processor. Fully reversible means that both the instruction set and the underlying circuit implementation will be reversible. Such a processor theoretically requires asymptotically zero dissipation, with dissipation falling to zero as the clock period is increased to infinity. This assumes that an appropriate energy-recycling, constant-current power supply could be developed. The ISI "blip circuit" [1], stepwise capacitor charging [10], and MIT's transmission line clock drivers, are steps toward this end. This paper assumes that the clock drivers exist and the datapath currently being constructed uses Younis and Knight's [14] three-phase SCRL logic family in all circuits. Any complete analysis of power dissipation in an adiabatic circuit must include the dissipation in the power supply and control logic. This paper focuses on the architectural and circuit level engineering of a reversible system rather than the actual dissipation of such a system.

2 Why Build a Reversible Processor

A fully reversible processor must implement a reversible instruction set in a reversible circuit implementation. A reversible instruction set is one in which both the previous and next instructions are known for each instruction in a correctly written assembly language program. The dynamic instruction stream may be executed both forwards and backwards. The instruction set described here is a modification of the one designed in Vieri's master's thesis [13].

A reversible circuit implementation is one in which, in the asymptotic limit, charge flows through the circuit in a thermodynamically reversible way at all times. This is only possible if the circuit is performing a logically reversible operation in which no information is lost. A logically reversible operation is one in which values produced as output uniquely determine the inputs used to generate that output. Performing exclusively logically reversible operations is a necessary but insufficient condition for thermodynamically reversible operation. When performed using an adiabatic circuit topology, the operation is thermodynamically reversible.

Performing circuit-level operations in a thermodynamically reversible way allows the energy dissipation to asymptotically fall to zero in the limit of infinitely slow operation. Conventional CMOS circuits have a minimum dissipation associated with each compute operation that changes the

state of the output node, regardless of operation frequency. So-called "adiabatic" techniques, in which the dissipation per compute operation is proportional to the operation frequency, have shown themselves to be useful in practical applications [11, 2].

As mentioned above, adiabatic operation requires that the circuits perform logically reversible operations. If one attempts to implement a conventional instruction set in a reversible logic family, reversibility will be broken at the circuit level when the instruction set specifies an irreversible operation. This break in reversibility translates to a required energy dissipation.

3 The Pendulum Instruction Set

The particular implementation discussed here is known as the Pendulum processor. The Pendulum processor was originally based on the elegantly simple MIPS RISC architecture [6]. The register-to-register operations, fixed instruction length, and simple memory access instructions make it a good starting point for a radically different approach to instruction set design. For ease of implementation, and of course to maintain reversibility, the instruction set has been substantially modified. It retains the general purpose register structure and fixed length instructions, however.

The Pendulum processor supports a number of traditional instructions with additional restrictions to ensure reversibility. The instruction set includes conventional register to register operations such as add and logical AND, shift and rotate operations, operations on immediate values such as add immediate and OR immediate, conditional branches such as branch on equal to zero and branch on less than zero, and a single memory access operation, exchange. The direction of the processor is changed using conditional branch-and-change-direction instructions.

3.1 Register to Register Operations

Conventional general purpose register processors read two operands, stored in two possibly different registers, and perform some operation to produce a result. The result may be stored either in the location of one of the operands, overwriting that operand, or some other location, overwriting whatever value was previously stored there. This produces two difficulties for a reversible processor. First, writing a result over a previously stored value is irreversible since the information stored there is lost. Second, mapping from the information space of two operands to the space of two operands and a result will quickly fill the available memory. However, if the result is stored in the location originally used for one of the operands, the computation takes two operands as input and outputs one operand and a result. This space optimization is not required for reversibility if the processor can always store the result without overwriting some other value, but it is a useful convention for managing resources.

An astutely designed instruction set will inherently avoid producing garbage information while retaining as much flexibility and power for the programmer. This leads to the distinction between expanding and non-expanding operations. Both types of instruction are reversible; the distinction is made only in how much memory these instructions consume when executed.

All non-expanding two operand instructions in the Pendulum instruction set take the form:

$$R_{sd} \leftarrow \mathcal{F}(R_{sd}, R_s) \tag{1}$$

where R_{sd} is the source of one operand and the destination for the result, and R_s is the source of the second operand.

By contrast, logical AND is not reversible if only the result and one operand are retained. Except for the special case of the saved operand having every bit position set to one, the second operand can not be recovered accurately and must be stored separately. Since operations like AND, including logical OR and shift operations require additional memory space after execution, they are termed expanding operations. The problem then arises of how store that extra information. If the two operands continue to be stored in their original location, the result must be stored in a new location. It is still not permissible for the result to overwrite a previously stored value, so the result may either be stored in a location that is known to be clear or combined with the previously stored value in a reversible way. The logical XOR operation is reversible, so the Pendulum processor stores the result by combining it in a logical XOR with the value stored in the destination register, forming ANDX, ORX and so on. Logical ANDX and all other expanding two operand instructions take the form:

$$R_d \leftarrow \mathcal{F}(R_s, R_t) \oplus P \tag{2}$$

where R_s and R_t are the two operand registers, R_d is the destination register, and P is the value originally stored in R_d.

Constructing a datapath capable of executing an instruction stream forwards and backwards is simplified if instructions perform the same operation in both directions. Addition, which appears simple, is complicated by the fact that the ALU performs addition when executing in one direction and subtraction when reversing. The expanding operations are their own inverses, since

$$P = \mathcal{F}(R_s, R_t) \oplus R_d \tag{3}$$

Non-expanding operations could take the same form as the expanding operations, performing `addx` and so on, simplifying the ALU, but programming then becomes fairly difficult. Only expanding operations, which require additional storage space, are implemented to consume that space.

Converting a conventional register to register instruction execution scheme to reversible operation is relatively simple. The restriction on which registers can be operands is minor, and implementation of an SCRL ALU is not particularly difficult. While a conventional processor erases a significant number of bits in these operations, preserving them is not difficult.

3.2 Memory Access

A reversible memory system, named XRAM, has been fabricated in a 0.5 μm CMOS silicon process. The system was intended to be a prototype of the Pendulum register file. Reversible memory system design is discussed in more depth elsewhere [12], and this section draws heavily on previous work by this group.

From a system point of view, the only additional requirement of a reversible memory, beyond a traditional memory system's function, is that it not erase bits when it is read from and written to.

The memory must of course perform as a random access memory, allowing bits to be stored and retrieved. Bit erasure can happen as a fundamental side effect of the operation of the memory or as a function of the particular implementation. For example, one can imagine a memory in which the externally visible values being stored and retrieved are never lost but the implementation of the memory is such that intermediate bits are erased internally.

A traditional SRAM architecture is based on read/write operations. An address is presented to the memory and, based on a read/write and possibly an enable signal, a word of data is read from or written to the memory array. Data may be read from any location an arbitrary number of times, and data written to a location overwrites that location's previously stored data.

Reading a value does not at first seem to be an irreversible operation. Reading from a standard memory creates a copy of the stored value and sends it to another part of the computing system. An arbitrary number of copies may be created this way. If, in a reversible system, the overall system can properly manage these copies, the memory need not be concerned with them. The larger system will, however, probably exhaust its ability to store or recover the bits generated by the production of an arbitrary number of copies. So it is a desirable feature of a reversible memory not to be a limitless source of bits when used in a larger system. It must be emphasized, however, that copy itself is *not* an irreversible operation.

A conventional memory performs explicit bit erasure during writes because the previously stored value is overwritten and lost. A reversible memory must save those bits somehow. The specific mechanism for this may vary. For example, reads may be performed destructively, as in a DRAM, to avoid producing copies of the data. The information is moved out of the memory rather than being copied from it.

During writes, the value which would be overwritten could be pushed off to a separate location, either automatically or explicitly under programmer control. This only postpones the problem until later since any finite capacity storage will be filled eventually.

If the separate location is accessible to the programmer, however, that data may either be useful or it may be possible to recover the space by undoing earlier operations. So if a write is preceded by a destructive read, the old information is moved out of the memory and into the rest of the system, and the new information replaces it in the memory. The old value has been *exchanged* for the new value. This type of eXchange memory architecture, or XRAM, is the memory access technique used in the Pendulum register file and for data and instruction memory access. The instruction set supports a single `exchange` instruction which specifies a register containing the memory address to be exchanged and a register containing the value to be stored to memory and in which the memory value will be placed.

The essential insight of the XRAM is that performing a read and then a write to the same memory location does not lose any information. One data word is moved out of the memory, leaving an empty slot for a new value to be moved in. In general, *moving* data rather than *copying* is a valid technique in reversible computing for avoiding bit erasure on the one hand and avoiding producing large amounts of garbage information on the other.

3.3 Control Flow Operations

If programs consisted solely of register to register and memory access operations, programming and implementation would be relatively simple. Unfortunately, conditional branches are crucial to creating useful programs. The processor must be able to follow arbitrary loops, subroutine calls, and recursion during forward and reverse operation. A great deal of information is lost in conventional processors during branches, and adding structures to retain this information is very difficult.

Any instruction in a conventional machine implicitly or explicitly designates the next instruction in the program. Branch instructions specify if a branch is to be taken, and if so, what the target is. Non-branch instructions implicitly specify the instruction at the next instruction memory address location. To follow a series of sequential instructions backwards is trivial; merely decrement the program counter rather than incrementing it. Following a series of arbitrary jumps and branches backwards in a traditional processor is impossible: the information necessary to follow a jump or branch backwards is lost when the branch is taken. A reversible computer must store enough information, either explicity in the instruction stream or elsewhere, to retrace program execution backwards.

Space does not permit a discussion of possible techniques for performing jumps and branches reversibly, but the literature contains a number of examples that differ from the scheme presented here [4, 13, 5]. The discussion below refers only to the particular scheme used in the current version of the Pendulum processor.

Pendulum branch instructions specify the condition to be evaluated, either equal to zero or less than zero, the register containing the value to be evaluated, and a register containing the target address. The instruction at the target address must be able to point back to the branch address and know if the branch was taken or if the target location was reached through sequential operation. For proper operation, each branch instruction must target an identical copy of itself.

When a branch condition is true, an internal branch bit is toggled. If the branch bit is false and the branch condition is true, the program counter update (PCU) unit exchanges the value of the program counter and the target address. The target address must hold an identical branch instruction which toggles the branch bit and sequential operation resumes. The address of the first branch instruction is stored in the register file so that during reverse operation the branch can be executed properly.

4 Instruction Fetch and Decode

Reading from the instruction memory suffers from the same difficulty as reading from the data memory. Each copy created when an instruction is read must be "uncopied." If instruction fetch operations are performed by moving instructions rather than copying them, they must be returned to the instruction memory when the instruction has finished executing.

After the instruction is read (or moved) from the instruction memory, the opcode is decoded and a number of datapath control signals are generated. Just before the instruction is moved back to

the instruction memory, these control signals must be "ungenerated" by encoding the instruction.

A certain symmetry is therefore enforced with respect to instruction fetch and decode. An instruction is moved from the instruction memory to the instruction decode unit. The resulting datapath control signals direct operation of the execution and memory access units. After execution, the control signals are used to restore the original instruction encoding, and the instruction may then be returned to the instruction memory.

The processor must be able to return the instruction to its original location, so its address must be passed through the datapath and made available at the end of instruction execution. Since the address of the next instruction must also be available at the end of instruction execution, the Pendulum processor has two instruction address paths. One path contains the address of the instruction being executed, and the program counter update unit uses it to compute the address of the next instruction. The second path contains the address of the previous instruction and the PCU uses it to compute the address of the current instruction. The output of the PCU is then the next instruction address and the current instruction address, which are the values required to return the current instruction to the instruction memory and read out the next instruction.

Performing these computations during branching instructions is very difficult, especially when the processor is changing direction. Traditional processors throw away every instruction executed, and ensuring that the instructions are returned to the instruction memory is difficult.

5 Conclusions

This approach to low power computing is clearly impractical in the near-term. All the primary blocks of a traditional RISC processor erase bits, and retaining those bits presents varying levels of difficulty to the designer. These challenges present the opportunity to reexamine conventional RISC architecture in terms of bit erasure during operation. Register to register operations and memory access are relatively easy to convert to reversibility, but control flow and, surprisingly, instruction fetch and decode, are decidedly non-trivial. This knowledge may be used in traditional processor design to target datapath blocks for energy dissipation reduction.

References

[1] W. Athas, L. Svensson, and N. Tzartzanis. A resonant signal driver for two-phase, almost-non-overlapping clocks. In *International Symposium on Circuits and Systems*, 1996.

[2] W. Athas, N. Tzartzanis, L. Svensson, L. Peterson, H. Li, X. Jiang, P. Wang, and W-C. Liu. AC-1: A clock-powered microprocessor. In *International Symposium on Low Power Electronics and Design*, pages 18–20, 1997.

[3] C. S. Calude, J. Casti, and M. J. Dinneen, editors. *Unconventional Models of Computation*. Springer-Verlag, 1998.

[4] Michael P. Frank. Modifications to PISA architecture to support guaranteed reversibility and other fetures. Online draft memo, July 1997. `http://www.ai.mit.edu/~mpf/rc/memos/M07/M07_revarch.html`.

[5] J. Storrs Hall. A reversible instruction set architecture and algorithms. In *Physics and Computation*, pages 128–134, November 1994.

[6] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[7] J. G. Koller and W. C. Athas. Adiabatic switching, low energy computing, and the physics of storing and erasing information. In *Physics of Computation Workshop*, 1992.

[8] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Research and Development*, 5:183–191, 1961.

[9] J. C. Maxwell. *Theory of Heat*. Longmans, Green & Co., London, 4th edition, 1875.

[10] L."J." Svensson and J.G. Koller. Adiabatic charging without inductors. Technical Report ACMOS-TR-3a, USC Information Sciences Institute, February 1994.

[11] Nestoras Tzartzanis and William C. Athas. Energy recovery for the design of high-speed, low power static RAMs. In *International Symposium on Low Power Electronics and Design*, pages 55–60, 1996.

[12] Carlin Vieri, M. Josephine Ammer, Amory Wakefield, Lars "Johnny" Svensson, William Athas, and Thomas F. Knight, Jr. Designing reversible memory. In Calude et al. [3], pages 386–405.

[13] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.

[14] Saed G. Younis and Thomas F.. Knight, Jr. Practical implementation of charge recovering asymptotically zero power CMOS. In *Proceedings of the 1993 Symposium in Integrated Systems*, pages 234–250. MIT Press, 1993.

[15] Saed G. Younis and Thomas F. Knight, Jr. Asymptotically zero energy split-level charge recovery logic. In *International Workshop on Low Power Design*, pages 177–182, 1994.