A Reversible Instruction Set Architecture and Algorithms

J. Storrs Hall
Laboratory for Computer Science Research
Rutgers University
New Brunswick, NJ 08903

Abstract

We describe a reversible Instruction Set Architecture using recently developed reversible logic design techniques. Such an architecture has the dual advantage of being able to run backwards and of being, in theory, implementable so as to dissipate less than log 2 kT joules per bit operation. We analyze several basic control structures and algorithms on the architecture, showing that, for example, a sorting algorithm need only dissipate $O(n \log n)$ bits even though it makes $O(n^2)$ comparisons.

Keywords

reversible computation, entropy, heat dissipation, reversible algorithms, finite automata, computer architecture, retractile cascade, sorting

1 Introduction

In [8] we introduced an "electroid" logic for the design of reversible computer architectures. It was at that time a conjecture, based on an observation of Merkle [10], that such a logic could be used to design a computer architecture that was not radically different from conventional ones at the instruction set level, and yet kept irreversible operations to a minimum.

The purpose of this paper is two-fold. First, to demonstrate an instruction set architecture for a processor in the electroid logic that executes a near-conventional instruction set in a reversible manner, requiring physical bit erasure only where implied by the logical operations; and secondly, to demonstrate reversible programs on such an architecture, logically erasing bits only in the amount implied by the input-output function of the program.

2 Retractile Cascades

Any combinational circuit can be implemented reversibly by the use of a very simple technique: simply remember the inputs.[1] Implementation of such functions in the electroid logic is obviously straightforward.

The only problem with such a technique is that the inputs must remain asserted until after the output clocks have been retracted. A circuit that performs a logic function several layers deep must have a sequence of rising clocks, and then the reverse sequence of falling ones. Such a circuit is called a *retractile cascade.*

A retractile cascade adder is seen in Fig. 1. First the two input numbers A[0-n] and B[0-n] are asserted. Then clock LL1 is asserted; this energizes the first stage of each full adder, consisting of an XOR and an AND. Then clock CC is asserted, energizing the carry chain; finally clock LL2 is asserted, producing the sum.

If the sum is needed more than momentarily, and the adder or its inputs need to be vacated, it will be necessary to save the sum using the clock labelled LATCH. The sum would then presumably be moved using conservative techniques or erased dissipatively. However, it is interesting and important to note that as long as the inputs A and B are available, the sum can be erased non-dissipatively. Fig. 2 shows the clock sequence which accomplishes this. This technique can be used with any retractile cascade to erase as well as to produce the value given the inputs.

Retractile cascades can be used for virtually all the medium-scale components of a standard computer architecture. Hall [8] shows a PLA, a barrel shifter, register set, and so forth, implemented as retractile cascades. The register set (with a correction from the original,) appears in Fig. 3.

[1] This technique appears first to have been used systematically by Drexler [6].

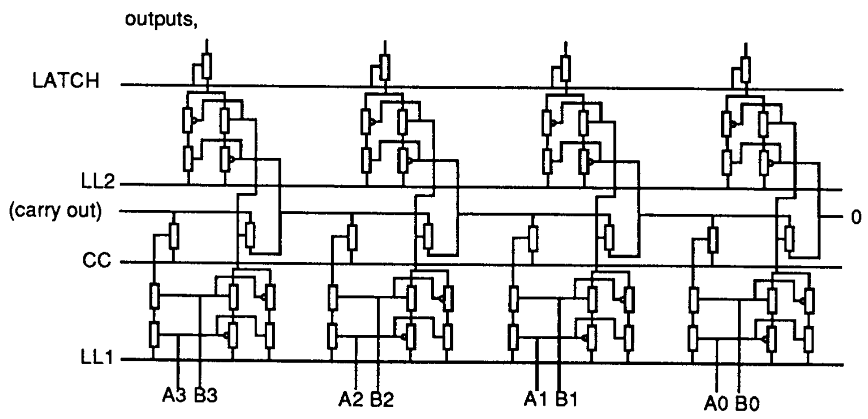

Figure 1. Retractile cascade adder.

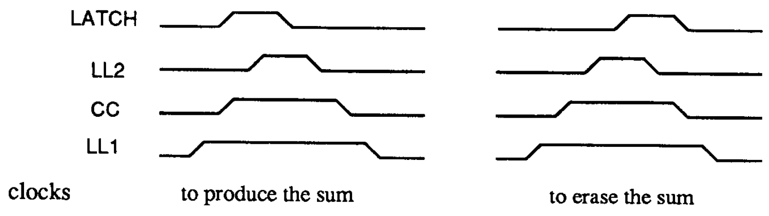

Figure 2. Retractile add and erase clock sequences.

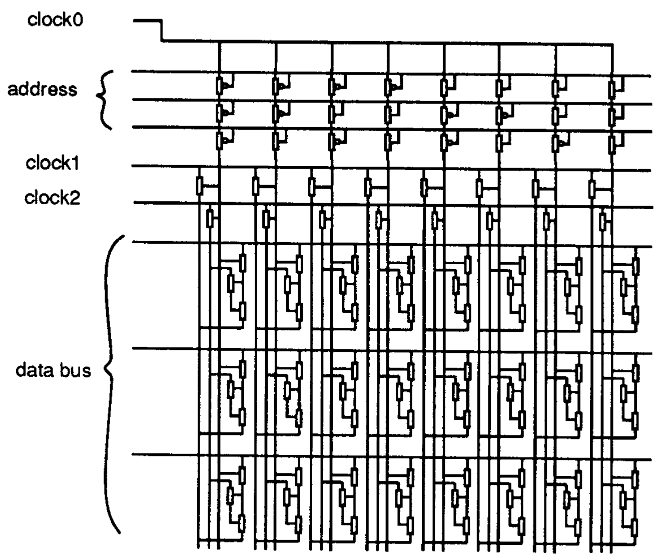

Figure 3: Conservative registers with retractile addressing.

3 Architecture-level Reversibility

In a complete reversible computer architecture, memory could be implemented along the same lines as the register set above. The appropriate method to access memory would be "exchange with register" rather than load and/or store, which of course are irreversible in general. Merkle [10] notes that many register to register operations such as `A=A+B` are reversible. Such instructions can be implemented by making the memory cycle that produces `B` entirely retractile rather than hybrid.

Sources of irreversibility in conventional instruction sets, (as distinguished from actual architectures), are largely of two kinds. First, setting, zeroing, copying over data in registers or memory, implicitly erases the previous contents of the register or memory. Secondly, transfer of control, i.e. branch instructions, cause the implicit erasure of the previous program counter.

In theory, a branch or jump does not cause any information loss if there is no other predecessor instruction to its target. It is only the coalescing of control flow paths that causes logically necessary bit erasure, and then only enough bits to distinguish between the incoming paths. More to the point, if there is sufficient information in the program state to distinguish the paths, no bits have been logically erased at all.

The major burden of reversible instruction set design, then, is allowing the programmer to remove redundant but physically represented information which would otherwise have to be erased dissipatively. This is accomplished by a process we call "uncopying", an instruction- level equivalent to the retraction stage of a retractile logic operation.

For example, consider that register A contains the sum of registers B and C, as if from executing

```
ADD A, B, C
```

We can reset A to 0 (the default "empty" value) by

```
UNADD A, B, C
```

The `UNADD` operation is performed by exactly the same retractile adder which was used for the `ADD`, but using the second, or retraction, timing on the latch clock (Fig. 2).

4 Erasure

If the logical definition of a program requires bit erasure, it cannot be avoided, but the instruction set can separate it and make it explicit. This is accomplished by the `ERASE` instruction. This instruction can be given a number of different interpretations.

For abstract logical reversibility, `ERASE` could push the information onto a stack (or tape). With this proviso, the definition of the rest of the instruction set is such that the program could be run in reverse, inverting its function, reading from the tape or stack at each point an `ERASE` was to be executed in reverse.

For a more practical physical interpretation, we conjecture that computers will eventually operate at such efficiencies that bit erasure is the primary heat-generating mechanism (see Drexler [6]). Such a computer might have a thermostatically controlled clock, and frequency of bit erasure might be one of the more critical bounds on operating speed.

Simply for perspicuity and keeping irrelevant mechanism to a minimum, we will adopt the convention that a physical dissipation means is employed that dissipates only when a 1 bit is being erased (e.g., if 1 is represented by postitve voltage and 0 by ground voltage, and dissipation is by connecting the charge to ground through a resistance.) This convention allows us to execute `ERASE` on word-length data but count as erased only those bits which might have been 1.

5 Style of Instructions

The instruction set has memory/register semantics, similar to the classic PDP-10 instruction set, instead of the register-to-register plus load/store style of a modern RISC architecture. The older style, from an age when most assembly programs were written by programmers instead of optimizing compilers, is more compact and easier to read. More than that, however, the pure RISC style is considerably less amenable to retractile implementation.

For example, in the PDP-10-style effective address calculation, the operand of an `ADD` instruction might be the contents of a memory location whose address is the sum of the contents of a register and an immediate value from the instruction. In a RISC this would be done as several separate instructions, leaving the various intermediate values in registers to be cleaned up explicitly. An architecture with hardware effective address calculation, however, can perform the whole operation as a retractile cascade. This has the practical drawback that the cycle takes twice as long and cannot be pipelined, but it is clearly superior for pedagogical purposes and as a proof of concept.

6 The Isentropic Instruction Set

The instruction set is similar to the PDP-10 instruction set, but does not have byte, halfword, or floating point operations. It has a somewhat extended flexibility of the basic instructions, however, including three-address codes. For example:

```
ADD reg, reg
ADD reg, reg, reg
ADD reg, addr(reg)
ADD reg, reg, addr(reg)
ADD reg, const
ADD reg, reg, const
ADD reg, const(reg)
ADD reg, const, addr(reg)
ADDM reg, reg
ADDM reg, reg, reg
ADDM reg, addr(reg)
ADDM reg, reg, addr(reg)
ADDM reg, const, addr(reg)
```

In the standard form, the destination is the first argument; in the memory (ADDM) form, the last. ADD is a "reversible" instruction, meaning that the two-argument form is legal, and involves adding the second operand destructively to the first. Other reversible instructions are SUB, XOR, and EXCH.

Instructions that are not reversible (bitwise AND and OR) may be used only in three-operand form. All three-operand instructions, reversible or not, require that the destination contain 0 before they are executed. This includes the 3-operand form of "reversible" instructions like ADD.

All three-operand instructions have inverses UNADD, UNOR, and so forth.

It is not allowed to use the same register in the index position and as a destination in a given instruction.

There are separate instructions MOVE, EXCH, and COPY. MOVE and EXCH actually do the same thing (as does MOVEM); use of MOVE is a convention that indicates that one of the items is known to be 0 and the other is being "moved". The destination of COPY must be 0 beforehand; both source and destination contain the same value afterward. UNCOPY is abbreviated UNC. All these instructions exist in the 2-address forms only.

There is a proliferation of jump instructions. The general form is (mod)J(comp) reg, opnd, addr. (mod) is A (annotate), I (increment), D (decrement), or empty. (comp) is GT, GE, EQ, LE, LT, NE, AB, OB,NAB, NOB, or empty. opnd is a register, an indexed address, or a constant.

The destination address of a jump must be the address of a Come-From instruction. The format of a Come-From is the same as a jump with CF instead of J. The function of the Come-From instruction is to uncopy the program counter left over from the Jump. To that end the "destination" of a CF must be the address of the corresponding J.

The conditionals attached to corresponding jump and come-froms are not necessarily, or even usually, the same. A come-from which can be "fallen into" from preceding code must use its condition to distinguish whether this has happened or it was jumped to. This is so the old PC can be uncopied or not, as appropriate.

From the point of view of logical reversibility, this requirement is the same as saying that if the state of the program after the jump is such as to distinguish whether the jump was made or not, no bit has logically been erased. In a more practically oriented interpretation, the come-from has the appropriate conditions to operate as an un-jump when executed in reverse.

If control flow confluence does logically destroy bits, it is necessary to handle them explicitly by way of an annotated jump. For example, each jump to a come-from could have an entry in a table, and should indicate which one it was:

```
j3:   aj w,2,conf
      ...
conf: cf @tab(w)
      ...
tab:  j1   ;item 0
      j2   ;item 1
      j3   ;item 2
      ...
```

The aj copies the 2 into register w and jumps to conf. After the cf, of course, the value in w remains, to be ERASEd or stored or whatever else the programmer wishes; but it has fewer bits than the raw PC would have had.

In this case an explicit move of 2 to w would have worked as well. aj becomes useful when a test occurs, and the annotation is made only when the jump is taken.

The effective address of cf is indirect. A recursive indirection (as on the PDP-10) would require an arbitrarily large hardware stack; only single-level indirections are allowed.

And finally, the ERASE instruction, taking either a register or effective memory address as its single operand, has in the instruction set universe of discourse the effect of setting the operand location to 0, but with some meta-ISA-level interpretation as discussed above.

6.1 Basic algorithmic techniques

A loop (that terminates) is a reversible construct and can be implemented by a matched jump/comefrom pair. The jump, just as in conventional programming, is conditional on the loop invariant, allowing control to fall out when the invariant ceases to be true. The initial comefrom is conditional to distinguish the initial entry from subsequent iterations. For example,

```
 ; for (i=0; i<10, i++)
top: cfgt I, 0, bot
      ...
bot: ijlt I, 10, top
```

(Remember that in normal forward execution, falling into a comefrom in sequence, the comefrom is a no-op.)

6.2 If statements

A conditional branch by itself is reversible since it simply copies a single (generally implicit) bit from the program's data state to the control state. As conventionally implemented, the confluence of control at the end of the `if` is its irreversible feature.

As long as the difference in data state which is reflected in the branch has not been obliterated by the intervening statements, the confluence can be accomplished using the same condition, or to be precise, its inverse.

For example, a reversible `if` can be implemented

```
 ; if (a) b; else c;
top: jeq a,0,mid
     ;code for b
t2:  j bot
mid: cf top
     ; code for c
bot: cfne a,0,t2
     ...
```

with the restriction that neither `b` nor `c` changes `a`. However, a statement such as

```
if (a > 30) a = a-10;
```

cannot be implemented reversibly; if `a==25` after the statement, for example, there is no way to know whether it was 25 or 35 before. The logic of such a program produces a confluence of state, which cannot be sidestepped by programming technique.

Such a statement would have to be implemented with an explicit annotation of state; consider the following, which might be a compare/exchange operation in the inner loop of a sort. After `bot`, register `t` (initially 0) would tell whether the exchange had been skipped or not.

```
top: ajle t,1,a,x(i),bot
     exch a,x(i)
bot: cfeq t,1,top
     ERASE t
     ...
```

Clearly a sort implemented this way would destroy one bit ber comparison/exchange operation.

7 Sorting

Sorting a list of numbers in place destroys, by definition, log(n!) bits, since there is one output for each of the n! different permutations of each given set of numbers. In an efficient implementation, there are only n log n comparisons (which approximates log(n!)), and one bit may be destroyed at each comparison. However, we can show that only n log n bits need be erased even if the algorithm is one of the simpler, if less asymptotically efficient, $O(n^2)$ ones, such as insertion sort.

See Fig. 4. The insertion sort is a nested loop. The outer loop, `bol` to `eol`, successively picks elements from the unsorted portion of the array, and inserts them into the sorted part. The sorted part is initially just the last element.

The element being inserted is held in register `X`, and compared, in the inner loop, with successive elements of the sorted part, which are moved down until the comparison succeeds. The algorithm does $O(n^2)$ comparisons, actually $n(n-1)/2$ if the list is in reverse order initially. However, we can make the inner loop reversible by *counting* comparisons, a number with a maximum of log n bits. Erasing that number once each time around the outer loop gives a total of n log n erased bits.

A copying sort which leaves its input unaltered need erase no bits at all; see Fig. 5. The `ERASE` is replaced by a loop that counts index `J` back down to `I`. This could not be done in the original since the value of the inserted element must remain alive throughout the dountdown loop (`brl` to `erl`) for use by the comefrom. It can then be uncopied from the original array.

```
  ; on entry, the numbers are in A (memory area)
  ; the number of numbers is N (immediate value)
  ; inf is defined as the highest word value
  ; all other registers are 0
       copym inf,A+N      ; set sentinel value at end of list
       copy I,N-1         ; I points to bottom of sorted part
  bol: cflt I,N-1,eol       ; in the outer loop, I decreases
       copy J,I           ; J moves back up thru sorted part
       move X,A(J)        ; element being inserted
  bil: cfgt J,I,eil       ; J increases in inner loop
  bx:  jle X,A+1(J),ex    ; compare insertee to list elements
       move Y,A+1(J)      ; not yet, move elements down
       movem Y,A(J)
  eil: ij J,bil           ; end of inner loop
  ex:  cf bx              ; only get here from bx jump
       movem X,A(J)       ; insert element
       sub J,I            ; an optimization: decrease no. bits in J
       ERASE J            ; < log N bits, happens N times
  eol: djgt I,0,bol       ; end of outer loop
       uncm inf,A+N       ; clean up sentinel (I and J are 0)
```

Figure 4. Dissipation-limited insertion sort.

```
   ;Same, but copy-sort A into B
       copym inf,B+N      ; set sentinel value at end of list
       copy I,N-1         ; I points to bottom of sorted part
       copy J,I           ; J moves back up thru sorted part
  bol: cflt I,N-1,eol     ; in the outer loop, I decreases
       move X,A(J)        ; element being inserted
  bil: cfgt J,I,eil       ; J increases in inner loop
  bx:  jle X,B+1(J),ex    ; compare insertee to list elements
       move Y,B+1(J)      ; not yet, move elements down
       movem Y,B(J)
  eil: ij J,bil           ; end of inner loop
  ex:  cf bx              ; only get here from bx jump
       copym X,B(J)       ; insert element
  brl: cfne X,B(J),erl    ; count J down from B(J)=X
  erl: djne J,I,brl       ;  to J=I
       unc X,A(J)         ; only then can we clear X
  eol: djgt I,0,bol       ; end of outer loop
       uncm inf,B(N)      ; clean up sentinel
```

Figure 5. Reversible copying insertion sort.

8 Conclusion

Our previous paper showed that one could have a logic-level design regime that could either elimimate or precisely control irreversible bit destruction, and yet retain much of the character, accumulated techniques, and knowledge of conventional electronics logic design. The present paper extends that result to instruction set architecture, assembly language programming, and algorithms.

The number of bits a program must destroy depends on its i/o behavior, not necessarily its algorithm. Though a comparison/exchange operation taken alone destroys one bit, a sorting algorithm performing n^2 of them need only destroy $n \log n$ bits. Any destructive (in-place) sort must destroy (or record) this many bits.

References

[1] Athas, W. C. *et al*, "A Framework for Practical Low-Power CMOS Systems Using Adiabatic Switching Principles", ***1994 International Workshop on Low Power Design***, Napa, CA, 1994

[2] Bennett, Charles H., "Time/Space Tradeoffs for Reversible Computation", *SIAM J. Comput.* V. 18 No. 4, 766-776, August 1989

[3] Bennett, C.H. and Rolf Landauer, "The Fundamental Physical Limits of Computation", *Scientific American* 253 pp 48-56, July 1985

[4] Bennett, Charles H., "Logical Reversibility of Computation", *IBM J. Res. Devel.* 17, pp525-532, 1973

[5] Denker, J. S. *et al*, "Adiabatic Computing with the 2N-2N2D Logic Family", ***1994 International Workshop on Low Power Design***, Napa, CA, 1994

[6] Drexler, K. Eric, ***Nanosystems: Molecular Machinery, Manufacturing, and Computation***, John Wiley and Sons, New York, 1992

[7] Fredkin, E., Toffoli, Tommaso, "Conservative Logic", *MIT/LCS/TM-197* Cambridge, MA May 1981

[8] Hall, J. Storrs, "An Electroid Switching Model for Reversible Computer Architectures" *PhysComp 92*, IEEE Press, Los Alamitos, CA, 1993

[9] Landauer, Rolf, "Dissipation and Noise Immunity in Computation and Communication", *Nature*, V. 335 pp 779-784, 27 Oct 1988

[10] Merkle, Ralph C., "Reversible Electronic Logic Using Switches", *Nanotechnology 4* (1993) pp 21-40.

[11] Ressler, Andrew L., "The Design of a Conservative Logic Computer and a Graphical Editor Simulator" *M.S. thesis*, M.I.T., Jan. 1981

[12] Seitz, Charles L. *et al*, "Hot-Clock nMOS", *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, pp.1-17

[13] Younis, S. G., and T. F. Knight, Jr., "Asymptotically Zero Energy Split-Level Charge Recovery Logic", *1994 International Workshop on Low Power Design*, Napa, CA, 1994