Conservative Logic

Bill Silver
19 May 1978

Abstract

This paper is an attempt to organize and present the model of computation known as *Conservative Logic*, originally conceived and developed by Edward Fredkin. Conservative Logic is an attempt to structure a computation by means of certain global laws, similar to the conservation laws of physics, the most important of which states that a computation must conserve information. While it is a commonly held belief that useful computation in general must destroy information, Conservative Logic is shown to be both rich in structure and every bit as powerful as more conventional computer logic. In addition, the model treats in a precise way concepts which are usually ignored in conventional logic, such as fanout and the role of the wire in communicating information.

The basic definitions and principals are given, the conservative logic gates and basic circuits are developed, and some theoretical results are presented. Selector notation, an algebra for writing and transforming Boolean functions, is presented. Selector notation is convenient for working with conservative logic circuits, and may have applications in other areas as well. Finally, several open questions in conservative logic are discussed.

1. Introduction

Conservative Logic (CL) is a model of computation which arises out of the belief that there is a common ground between the study of computation and the study of physics. This common ground is a two-way street; physical law may have something fundamental to contribute to the study of computation, and computer science may provide facinating alternatives to the more traditional models of basic physical phenomina. While a complete treatment of these possibilities is beyond the scope of this paper, they will be used to motivate the development of CL. It is not necessary for the reader to accept our philosophy in order to appreciate the results that follow, however it should be useful to know why we were led in the direction that we took.

Much of the descriptive power of physics comes from the existence of global laws, such as the law of conservation of energy. These laws apply to systems as a whole, independent of microscopic entities involved or of the ways in which those entities interact. In fact, the laws governing the interaction of a collection of fundamental particles may be totally unknown, and yet those particles when taken as a whole are seen to obey the conservation laws. Thus we can analyze and predict the behaviour of large and complex systems without knowing the details of the system at the microscopic level.

Quite the opposite is true in the study of computation. We build large and complex systems consisting of tens or hundreds of thousands of basic elements (gates and flip-flops for hardware systems, instructions for software) and yet to analyze them requires understanding the role of each and every one of them. While the situation is greatly simplified by breaking such systems up into modules, there are still no global laws or principals governing such systems as a whole. The value of such laws is clear; a physicist, for example, can tell you that no engine can be built that is more efficient than the Carnot engine, independent of its internal construction, by appealing to the laws of thermodynamics. It is very difficult, on the other hand, to characterize the efficiency of computer circuits or programs. While this can be done for many particular algorithms, such as a sorting procedure, we have yet to find laws that apply to all algorithms. Such laws might be used to answer some very general questions about computation, such as the relationship between the amount of memory and the amount of time needed by a computation.

The fact that conservation laws have been valuable in physics does not mean that we can just require that computer circuits obey such a law and expect wonderful results to follow. Rather, our motovation for considering computer logic circuits that obey a global conservation law comes from the desire to model our universe as a descrete, digital computation. It may very well turn out, however, that this approach will contribute a new understanding to the theory of computation.

We begin by stating what we believe is one of the most fundamental principals of physical theory:

The laws of physics are invariant with respect to the reversal of time.

In particular, we believe that physics is *exactly* reversible; replace t with -t and a system will evolve backwards, eventually reaching any previous state. This statement is easily shown to be true for classical mechanics and electromagnetism, and while it is open to question in quantum theory, we accept this principal in the discusson that follows. It is important to realize that this does not mean that a previous state can always be attained *forward in time.* If you drop an egg on the floor there may be no process which will restore it to its original state, but if you could somehow set the world in reverse it would in fact attain its previous state as a whole egg.

If a process is to be reversible, its state must at each moment contain enough information to return to any previous state. While this information may change in form as the system evolves, it can never be lost. Thus we arrive at a new law for physics, a law which we will require our circuits to obey, the law of *Conservation of Information.*

If physics is to be modelled as a digital computation, then that computation must be reversible, it must conserve information. Such an idea, however, is completely at odds with the way computer circuits and programs are designed today. In fact, many computer scientists will tell you, if they've thought about it at all, that useful computation in general must create and destroy information. If we are to claim that computation can model physics, we must show that computation can be made reversible without sacrificing any of its power or convenience. Conservative Logic will be seen to satisfy these requirements.

Another principal which we will adopt in our development of CL is the *equivalence of memory and communication.* It may be seen that what we commonly call memory and what we call communication are just conceptual manifestatons of the same physical process: the transmission of information form one point in space-time to another point in space-time. What would look like memory in one reference frame would look like communication in a frame which was moving with respect to the first. Similarly, what one observer might consider communication would look like memory to an observer at rest with respect to the information being communicated.

2. Basic Principals for Conservative Logic Circuits

Conservative Logic is a model of digital, binary hardware systems, such as might be found in a general purpose computer. It is similar to conventional digital logic in that a basic atomic processing element, the *gate*, is defined which may be interconnected to form circuits to compute any desired function. The interconnections are made by means an atomic memory/communication element, the *wire*, according to the rules specified below. In this

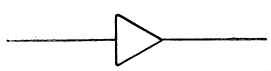
regard, CL attempts to include the cost of wires and fanout, a physical fact that is usually overlooked in conventional logic. (In this section we consider only the general properties that CL gates must satisfy. In the next section we will look for particular gates that meet these conditions, but it should be remembered that there are many gates that satisfy the requirements of CL.)

The *gate* is an atomic logic element with some number of binary inputs and outputs, where each output is some combinational (the gate has no memory) Boolean function of the inputs, subject to the following constraints:

1) The gate must be a universal computing element.
2) The gate must conserve information.

The requirement that the gate be universal means that by appropriate interconnection of gates it is possible to generate *any* Boolean function of any number of variables on some output. This requirement is satisfied by very simple conventional gates, such as the NAND gate. The requirement that the gate conserve information means that the gate must be reversible; the inputs must be uniquely determined by the outputs. Clearly, if a gate is to conserve information it must have at least as many outputs as inputs, and for simplicity we only consider those gates where the number of inputs and outputs are equal. In this case, there must be a one-to-one mapping between input and output states, i.e. the set of output states must be a permutation of the set of input states (remember that a gate with n inputs/outputs has 2^n input/output states). Finally, a gate is assumed to operate in zero time.

The *wire* is an atomic memory/communication element whose role is to move one bit of information from one point in space-time to another, such that the two points are separated by one unit of time. Thus the wire has one binary input and one binary output, where the output is what the input was one time unit ago. For simplicity we assume that all wires operate totally synchronously, somewhat like D-flip/flops of conventional logic, all connected to a master clock. The logic symbol for the wire is:



In order to guarantee that reversible circuits will be constructed out of the reversible gates, we must impose the following interconnection rules:

In any closed conservative logic circuit:
1) To each input of a gate must be connected the output of one wire.
2) To each output of a gate must be connected the input of one wire.
3) To the input of each wire must be connected the output of one gate or wire.

4) To the output of each wire must be connected the input of one gate or wire.

Thus CL circuits may not have any "loose ends", and wires may not be used to generate fanout. Wires may be connected in chains to make delays or serial memories, however.

It will often be the case that we want to design a module that is intended to be connected to other modules, and consequently will need inputs and outputs. Thus it is not a violation of the interconnection rules if one end of a wire is left unconnected if it is intended that that wire serve as an input or output to the module. It is understood, however, that in a complete system all such wires must be connected to something, in accordance with the rules. This restriction applies even if the input or output is just a constant (i.e. 0 or 1). Inputs and outputs are symbolized with half a wire, as follows:

INPUT OUTPUT

Finally, some definitions before we look at specific gates.

A CL circuit is called *combinational* if, after its inputs have remained constant for sufficient time, its outputs are uniquely determined by only the present values of the inputs. The circuit has no memory other than the input to output delay due to its wires. This definition corresponds to the one for conventional logic.

A CL circuit is called *sequential* if it is not combinational.

A CL circuit is called *loop-free* if it contains no closed loops, i.e. no feedback. While a loop-free circuit is clearly combinational, the converse is not true if the signal on the feedback part of the loop is constant over all inputs and all time. Combinational non-loop-free circuits will be seen to be very important in later sections.

3. The Conservative Logic Gates

We will now look for a gate which satisfies the constraints imposed in the preceeding section. We are looking for n Boolean functions of n inputs that when taken together are both universal and reversible. Presumably we would like to use the smallest n for which this is possible. We can immediatly rule out n=1, for none of the four 1-input Boolean functions (F(a) = 0, 1, a, or ¬a) are universal.

There are 16 Boolean functions of two variables, allowing $16^2 = 256$ possible gates for n=2. A little thought shows that only the following functions may be used if the gate is to be reversible:

F(a, b) = a, ¬a, b, ¬b, a⊕b, a≡b (⊕ is XOR, ≡ is EQV)

XOR and EQV by themselves are known not to be universal, and in fact combining them with NOT does not help, so we must conclude that n=2 will not work.

The so far uninteresting situation changes drastically for n=3. In fact, there are so many universal, reversible 3-input/3-output gates that we will introduce a new conservation law to narrow the search: gates will be required to conserve the number of one's and zero's presented to the inputs. Gates obeying this law will be refered to as bit-conserving gates. It is important to realize that bit-conserving does not imply reversible, and reversible does not imply bit-conserving. In fact, most bit-conserving gates are not reversible, and vica versa. Also, note that since wires conserve bits, our circuits will be bit-conserving too.

One way of looking at a gate that conservs bits is to see that its function is to permute its input signals in some way, where the permutation is some function of the inputs. (Note that this is quite different from saying that the input *states* are permutted to get the output states, a condition required for reversibility.) From this perspective we are led to consider a very simple bit-conserving gate, defined as follows (the logic notation is conventional Boolean algebra):

inputs	outputs	logic notation
A	A	(A)
B	if A=1 then B else C	(A∧B ∨ ¬A∧C)
C	if A=1 then C else B	(A∧C ∨ ¬A∧B)

Thus it may be seen that the inputs B and C are either swapped or not, depending on A. The gate is clearly reversible, since we can determine whether or not B and C have been swapped by looking at the first output, which is just A.

The gate we have just defined will be called the *selector* gate, and its logic symbol and a graphical description of its operaton are given in Figure 1. The selector gate has formed the basis for most of the work that has been done in Conservative Logic. Our next step is to show that it is universal.

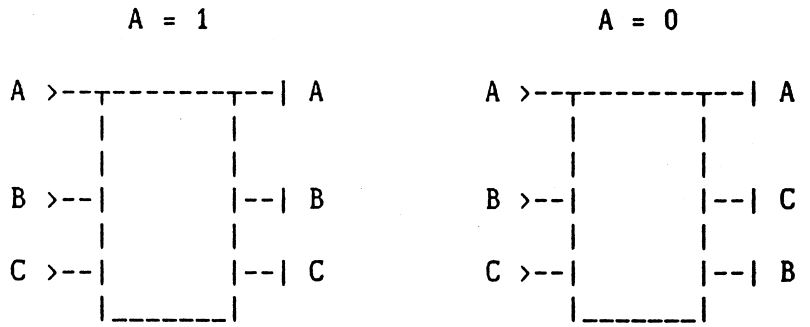


Figure 1 - Selector gate logic symbol and operation.

Normally, a logic family may be shown universal by showing that a set of Boolean functions previously known to be universal can be generated. For example, the sets {AND, NOT} and {NOR} are universal, among many others. Such a proof, however, assumes that many inputs may be connected to one output as needed, which is not allowed by our interconnection rules. Thus we must also show that out gates are capable of generating fanout themselves, without help from wires. Figure 2 shows that the selector gate is capable of fanout and can generate the set {AND, OR, NOT}, and is therefore universal.

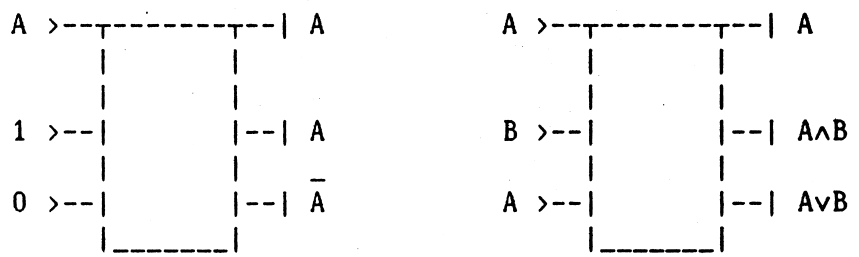


Figure 2 - Proof that the selector gate is universal.

The selector gate is not the only way we can dream up to permute three signals as a function of those signals. Suppose we say that the signals will be rotated either up or down, as shown in Figure 3. We can think of many conditions to use to control the direction of rotation, such as rotate down if there are an odd number of 1's, or rotate down if the majority of the inputs are 1. Interestingly enough, the two gates just mentioned are identical, as well as being reversible and universal, as the reader may verify. Thus we have another gate that can serve as the basis for Conservative Logic, which will be called the *symmetric*

majority/parity (SMP) gate.

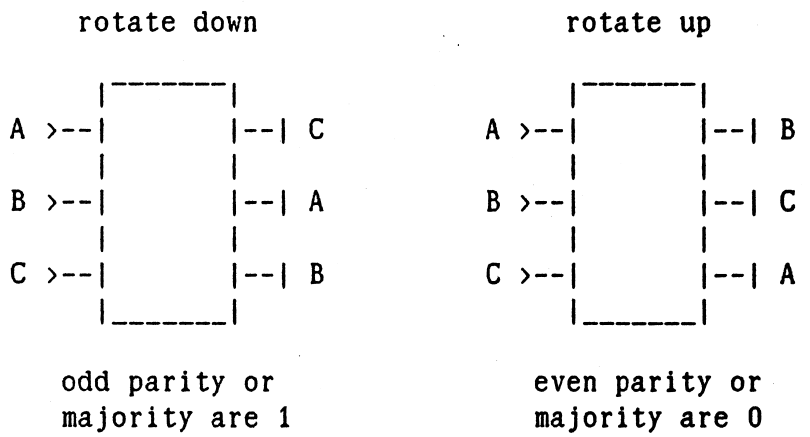


Figure 3 - The symmetric majority/parity (SMP) gate.

We could write the output functions for the SMP gate in Boolean algebra, as we did for the selector gate, and in fact we could use Boolean algebra to describe the behaviour of any combinational CL circuit. We have found it much more convenient, however, to develop a new algebra for the description of CL circuits. This algebra is called selector notation, and will be presented in the next section.

4. Selector Notation

Selector notation is a way of writing Boolean functions of any number of variables that is convenient for use in describing and analyzing CL circuits. The notation itself is essentially identical to one developed by John McCarthy for describing LISP conditionals. There is one syntactic construct, the selector expression, and many rules for transforming and simplifying such expressions.

A selector expression is either a binary constant (0 or 1), a binary variable or its complement, or a string of the form A→B,C (reads A selects B,C) where A, B, and C are selector expressions. The value of the expression A→B,C is B if the value of A is 1, and C if the value of A is 0. Since a selector expression is not easily confused with an expression written in conventional Boolean Algebra, we will often give an expression in both forms to aid in becomming familiar with selector notation. Thus we would write

$$A \rightarrow B, C = A \wedge B \vee \neg A \wedge C$$

where the left side is written in selector notation and the right side is its equivalent in Boolean Algebra.

The following table summarizes the basic transformations that can be made to selector expressions. The symbols A B C D E and F represent any selector expression.

$1{\to}A, B = A$	definition
$0{\to}A, B = B$	"
$A{\to}B, B = B$	
$A{\to}1, 0 = A$	fanout
$A{\to}0, 1 = \neg A$	invert
$\neg A{\to}B, C = A{\to}C, B$	symmetry
$A{\to}1, B = B{\to}1, A = A \vee B$	OR rule
$A{\to}B, 0 = B{\to}A, 0 = A \wedge B$	AND rule
$A{\to}B, \neg B = B{\to}A, \neg A = A \equiv B$	EQV rule
$A{\to}\neg B, B = B{\to}\neg A, A = A \oplus B$	XOR rule
$A{\to}(B{\to}C, D), (B{\to}E, F) = B{\to}(A{\to}C, E), (A{\to}D, F)$	level swap

If E is any selector expression and A is any variable, let $E_{A\leftrightarrow 1}$ be the result of replacing any desired A's with 1's and vica versa in E, and let $E_{A\leftrightarrow 0}$ be defined similarly for 0's. Then we can always rewrite E as follows:

$$E = A \to E_{A\leftrightarrow 1}, E_{A\leftrightarrow 0}$$

This transformation as a very powerful simplification aid, as may be seen in the following example:

$$(A{\to}B, C){\to}(A{\to}C, B), A$$

$$= A{\to}((1{\to}B, C){\to}(1{\to}C, B), 1), ((0{\to}B, C){\to}(0{\to}C, B), 0)$$

$= A \rightarrow (B \rightarrow C, 1), (C \rightarrow B, 0)$ by definition

$= A \rightarrow (B \rightarrow C, 1), (B \rightarrow C, 0)$ AND rule

$= A \rightarrow (B \rightarrow C, A), (B \rightarrow C, A)$

$= B \rightarrow C, A$

We can now give the expressions for the outputs of our two CL gates in selector notation:

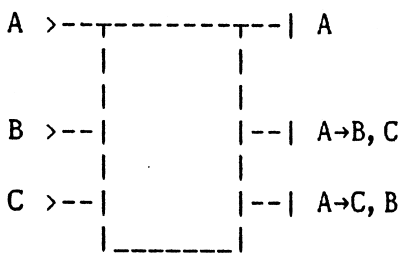
inputs	selector	SMP
A	A	$A \rightarrow B, C$
B	$A \rightarrow B, C$	$B \rightarrow C, A$
C	$A \rightarrow C, B$	$C \rightarrow A, B$

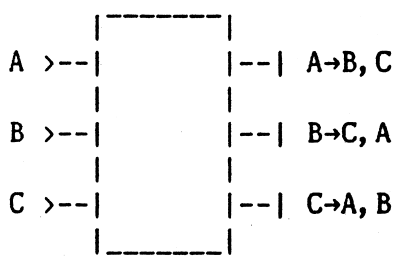
The expressions for the selector gate should be obvious. Those for the SMP gate are not at all obvious from the definition given in Section 3, and it should be instructive for the reader to verify them.

5. Basic Conservative Logic Circuits

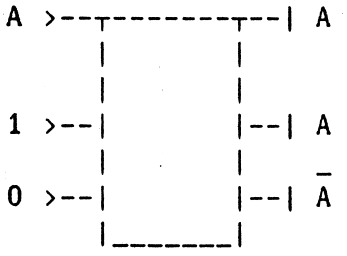
We now show that we can easily make all of the basic functions that we are familiar with from conventional logic.

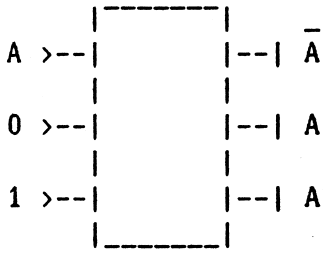
In general:



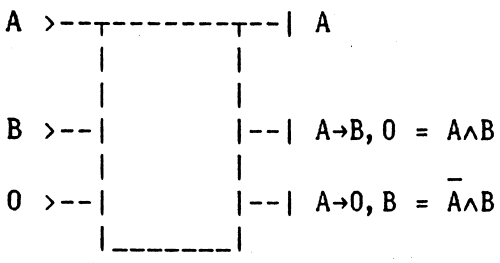


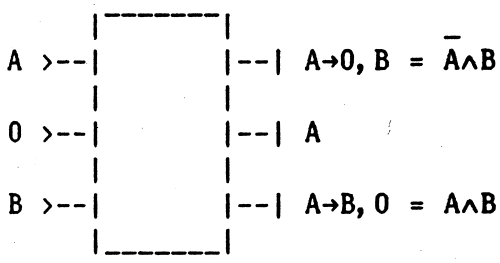
fanout and invert (also known as the SPY circuit):



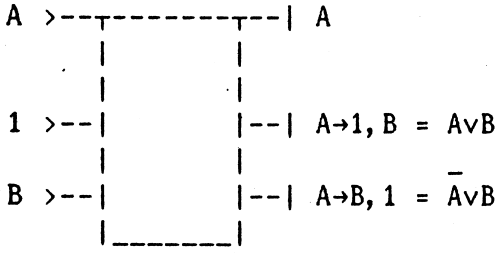


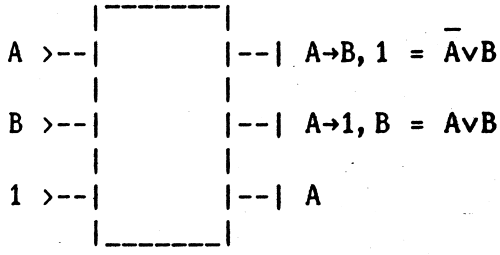
AND:



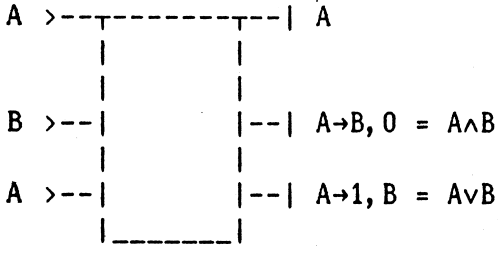


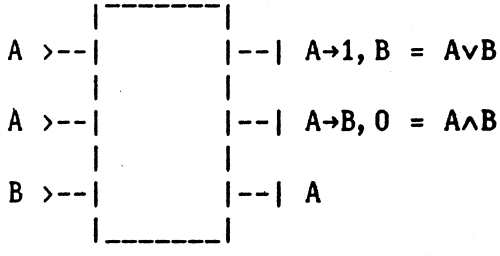
OR:



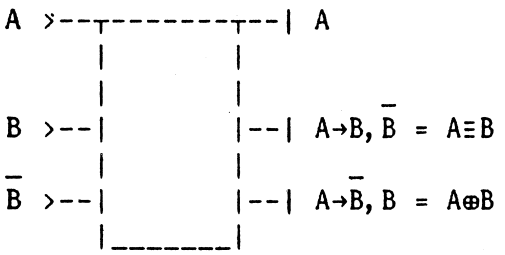


AND/OR:





XOR/EQV:



There is no corresponding SMP circuit, although XOR and EQV can be made seperately.

Figure 4 gives an example of a more advanced combinational CL circuit, a 1-line to 4-line demultiplexor/decoder. When used as a demultiplexor, the data input signal will appear on the output line selected by the control inputs, and the other outputs will be 0. If the data input is made a constant 1, the circuit will function as a 2-to-4 decoder.

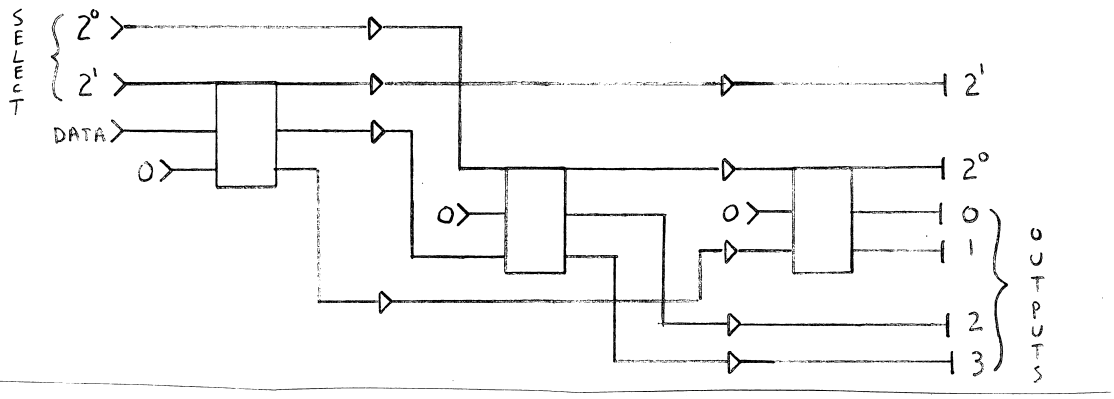


Figure 4 - 1-to-4 Demultiplexor

So far we have only looked at combinational circuits. We can, of course, take almost any combinational circuit and make it sequential by adding appropriate feedback loops. This approach to sequential circuit design tends to produce complex and difficult to understand circuits. What is usually done instead in conventional logic is to use basic sequential building blocks (flip/flops) and connect them with combinational circuits. Thus all of the state information is held in the filp/flops and not in random feedback loops. The same approach can be taken with CL circuits.

Figure 5 shows a minimal J/¬K flip/flop. Note that it has no clock input; the absolute synchronization of all wires serves as our master clock. A D flip/flop can be made by connecting the J and ¬K inputs together (by means of a fanout gate). We can, of course, make more elaborite flip/flops, with an actual clock input and set and clear direct inputs.

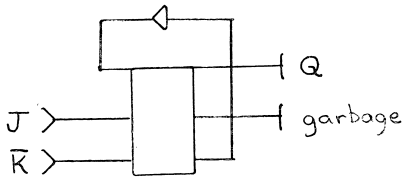


Figure 5 - Minimal J/¬K flip/flop

Figure 6 shows a circuit which uses the J/K flip/flop, a two-input binary serial adder. At each time step, the next higher order bits of the numbers to be added are presented to the circuit, which adds them to the carry generated by the previous input. While the total delay thru the circuit is 9 time steps, a new input can be accepted each time step because of the highly pipelined nature of CL circuits. Also, notice how the one fairly useless output generated by the flip/flop is used to help get back the original inputs. This avoids the need to have to consider that "garbage" signal as an output of the circuit (more about garbage in Section 6).

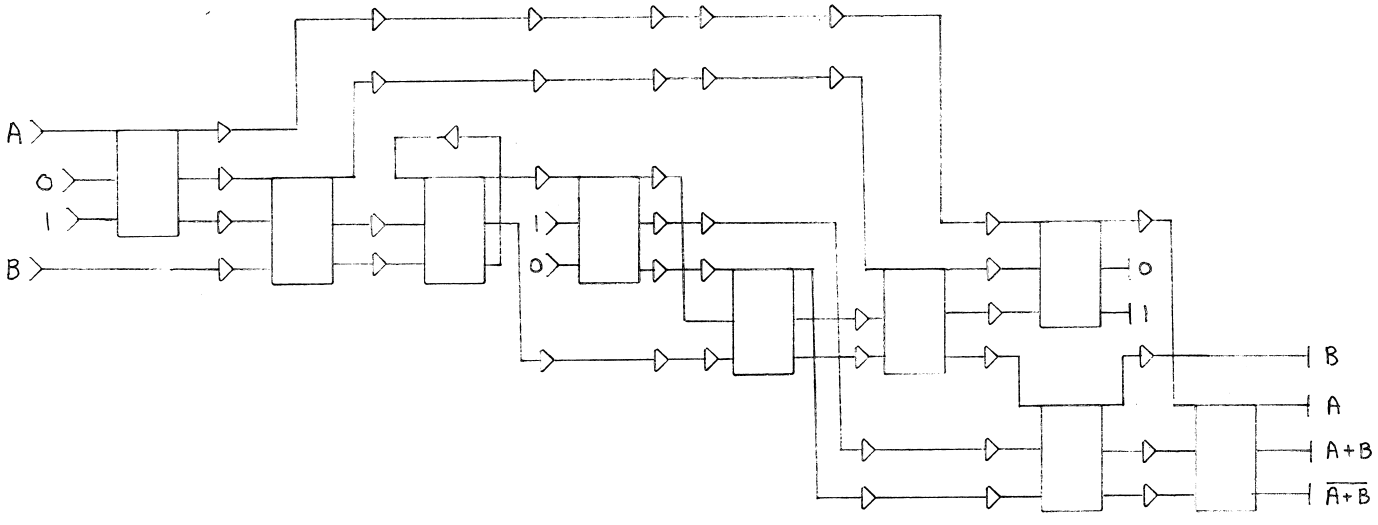


Figure 6 - Two Input Serial Adder

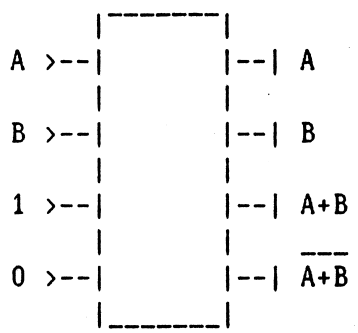


Figure 6 - Two input serial adder

6. Theoretical Results

We have seen two possible bases for Conservative Logic, the selector gate and the SMP gate, and we have seen how to construct many of the basic functions that we are familiar with from conventional logic. There are, however, a number of important questions that must be answered about the system that we have defined. These questions concern the nature of CL circuits that can be built within the restrictions imposed by our interconnection rules.

The gates that we have defined are universal, but that only means that any function can be generated on *some* output wire. Many other signals may be generated in the process of computing a desired function, and our interconnection rules require that they all be used somewhere, that they cannot be ignored. If an output is generated in the process of computing a desired function that is truely useless for the purposes of our design, we refer to that signal as *garbage*. Since all garbage must be used, we are naturally very interested in functions that can be computed without generating any. Clearly, if a function is not reversible it cannot be computed without garbage, for there must be extra outputs to contain the input information not contained in the function. Thus, the question that we would like to answer is "For what reversible functions can a CL circuit be contructed that computes that function without any extra inputs or outputs?" We will restrict our discussion in this section to combinational circuits.

Another interesting (although somewhat less important) question we would like to answer is "What other gates could we have chosen as the basis for Conservative Logic?" As we will see, there are 16,777,216 ways of selecting 3 Boolean functions of 3 variables; we would like to know if the two that we have chosen are the simplest, or the most interesting, or the most powerful of the ones we could have picked. In fact, we will show that they are the *only* ones (that conserve bits)! We will do this before returning to our main question.

A gate with 3 binary input wires and 3 binary output wires has $2^3 = 8$ possible input and output states. The purpose of a gate is to map each input state into some output state. In general, for each of the eight inputs states there are eight possible output states, giving $8^8 = 16{,}777{,}216$ possible gates. Most of these gates are not reversible, however. To be reversible, each input must map into a different output. There are 8! = 40320 ways to sepecify such a mapping, the number of permutations of eight things.

If a gate is going to conserve bits, input states may only map into output states with the same number of 1's and 0's, like this:

$$\begin{aligned} \{000\} &\rightarrow \{000\} \\ \{001, 010, 100\} &\rightarrow \{001, 010, 100\} \\ \{011, 101, 110\} &\rightarrow \{011, 101, 110\} \\ \{111\} &\rightarrow \{111\} \end{aligned}$$

There are $1 \cdot 3^2 \cdot 3^2 \cdot 1 = 729$ such mappings. Finally, if a gate is to conserve both bits and information, each of the mappings shown above must be 1 to 1, giving 3!3! = 36 possibilities. These observations are summarized in the following table:

		conserve information: no	conserve information: yes
conserve bits	no	16,777,216	40320
	yes	729	36

So we see that there are 36 3-input/3-output bit and information conserving gates, although since we have counted any fixed rearrangement of the input or output wires as a seperate gate, many of these might be equivalent. One of them certainly is the identity gate (i.e the outputs just copy the inputs), for this function clearly conserves bits and is reversible. In fact we have counted 6 of these, corresponding to the 6 possible permutations of the 3 output wires. The identity gate is not universal, however, so it cannot serve as the basis for CL. The selector gate must surely be included, since we know that it conserves both quantities. We have counted 18 of these, corresponding to 6 permutations of the outputs for each of 3 choices of the control input. Similarly, we have counted 12 SMP gates, corresponding to 6 permutations of the outputs for each of 2 choices of the controlling condition (rotate down on even or odd parity, for example). Since we have covered each of the 36 possibilities, we have shown the following theorem:

Thoerem 1. There are exactly two bit-conserving, information conserving, universal 3-input/3-output gates. These are the selector gate and the symmetric majority/parity gate.

There are in fact many reversible gates that do not conserve bits, and we could in principal use one of those as a basis for CL. None of them are as simple as the bit-conserving ones, however, so we have not investigated any of them in any detail.

Returning to our main question, we first notice that except for time delays, we can make an SMP gate out of selector gates without any garbage, as shown in Figure 7. Similarly, we can make a selector gate out of SMP gates without garbage, as shown in Figure 8. Notice the constant feedback that was used in making the selector gate. The constant 1 was needed to generate the desired function, but since the circuit generates a 1 itself we can connect these two points and avoid the need to regard them as inputs and outputs. In fact, it is easy to show that no loop-free SMP circuit can generate the selector gate without garbage.

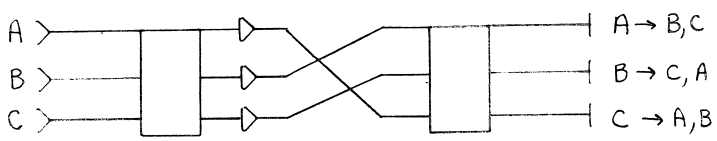


Figure 7 - SMP gate made out of selector gates

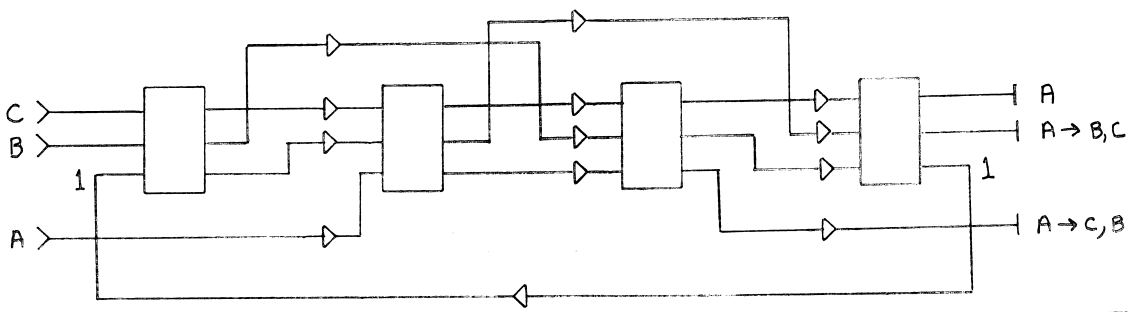


Figure 8 - Selector gate made out of SMP gates.

An interesting historical note is that while the circuit of Figure 7 was found easily, for a long time it was not known whether a circuit like Figure 8 could be made at all. Considerable effort was expended both trying to prove that it couldn't be done and trying to

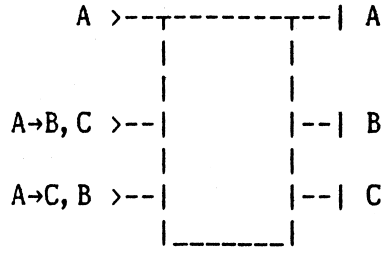
find one. The question was finally settled with a circuit that used nine gates. The circuit given above was found and shown minimal in number of gates by exhaustive search with a computer program written by Guy Steele.

We know that all CL circuits are reversible, but as pointed out in Section 1 this does not necessarily imply that the inverse of a function can be computed forward in time. We will now show that this can be done for any loop-free circuit made out of selector gates, by constructing a "mirror-image" circuit (for the rest of this secton we assume that the circuits mentioned are made out of selector gates only).

Lemma 1. If a loop-free circuit can be constructed that produces outputs F_i from inputs X_i, then a circuit can be constructed that produces X_i from F_i (with no extra inputs or outputs).

Proof: The proof is by induction on the number n of gates in the circuit that generates the F_i's from the X_i's.

Basis: The only one gate circuit is the gate itself, which happens to be its own inverse:



Induction: Assume the theorem for n = k. Consider any loop-free circuit C with k+1 gates. Since the circuit is loop-free and contains a finite number a gates, there must be at least one gate whose three outputs are all outputs of the circuit. Select any such gate, call it G, and consider the circuit C' with G removed. Let the outputs of C' be F_i'. Now C' is a k-gate circuit that produces F_i' from X_i, and G is a one gate circuit that produces F_i from F_i'. By the induction hypothesis we can make X_i from F_i', and we can certainly make F_i' from F_i using G itself. Therefore, we can make X_i from F_i and since we introduced no extra inputs or outputs in the construction, we have the theorem.

We can now extend this lemma to any combinational circuit.

Theorem 2. If F_i can be produced from X_i by any combinational circuit, then there exists a circuit to produce X_i from F_i.

Proof: A combinational circuit C in general may have some constant feedback signals K_i. Consider the equivalent loop-free circuit made by breaking all of the feedback loops. This new circuit C' will have inputs K_i in addition to the X_i's, and produce the K_i's as

output in addition to the F_i's. By Lemma 1 we can construct a circuit to produce X_i and K_i from F_i and K_i. We can now reconnect the input and output K_i's and we are done.

We now show a very general way to use garbage generated in a circuit, namely to turn that garbage back into the original inputs (see Figure 9).

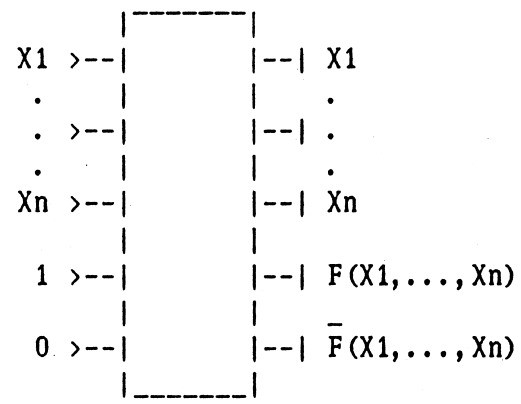
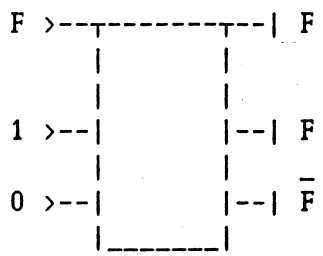


Figure 9 - General function box that can be made with no garbage

Theorem 3. The circuit of Figure 9 can be consrtucted for any Boolean function F.

Proof: Since the gate is universal, a circiut can be constructed that takes as input the X_i's, and possibly some constants K_i, and produces outputs F_i one of which is the desired function F. We then use our spy circuit as follows:



Using an extra 1 and 0, we have copied F and produced an ¬F. By Theorem 2, we can use the original F together with the rest of the F_i's to get back the X_i's and the K_i's. Finally, we connect the output K_i's to where they are needed as input, and we are done.

Before we can show our main result we need to show that we can build a circuit that

acts like a general permutation box.

Lemma 2. Given any set of signals X_i, a circuit can be constructed that takes those signals and some number of control signals C_i as input and produces as output the C_i's and any permutation P_i of the X_i's by appropriate settings of the C_i's.

Proof: The proof is by induction on the number n of signals in the set X_i (n>1).

Basis: For n = 2, we observe that the gate can generate both permutations of its 2nd and 3rd inputs by appropriatly setting the first.

Induction: Assume the lemma for n = k. Figure 10 shows how to use a k-signal permutter to make a k+1-signal permutter (this construction is not very efficient, but existance is all that we are trying to show). The circuit works by building a network that allows the k+1st signal to be inserted anywhere among the outputs of the k-signal permutter. All of the control signals needed as input are produced as output, and no extra inputs or outputs are generated.

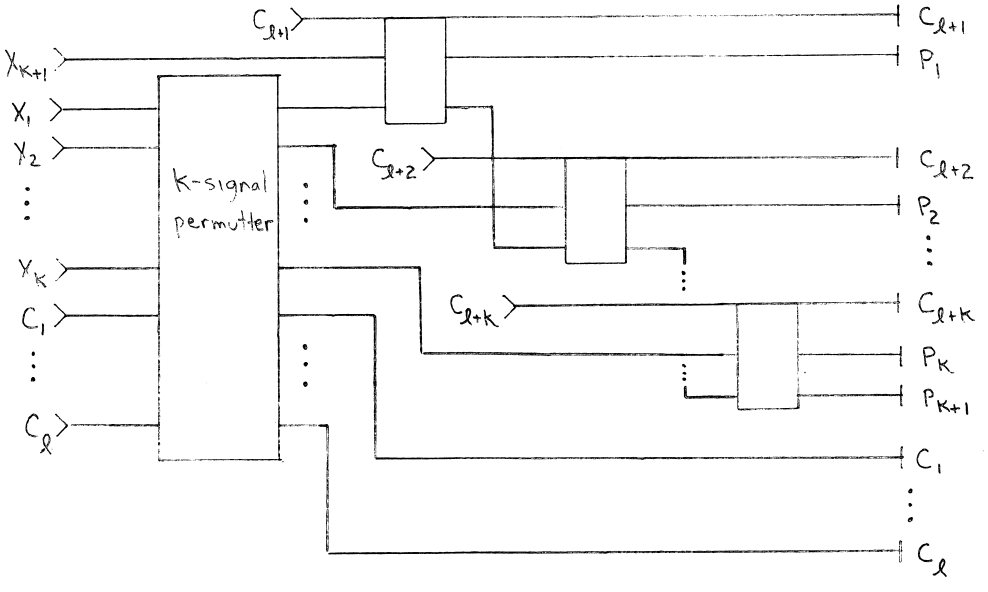


Figure 10 - Construction for the permutation box

Now for the main result and its consequences.

Theorem 4. If F_i is any set of functions of the set X_i that conserve bits and information, a circuit can be constructed that generates the F_i's from the X_i's.

Proof: Since F_i is a bit conserving function of X_i, any particular value of F can be viewed as just a permutation of the corresponding input bits of X. The permutation may be different for each of the possible input values, but it is in fact uniquely determined by the input. Thus, by using the permutation box of Lemma 2 we can build a circuit that produces F_i from X_i and an appropriate set of control signals C_i, which are themselves some function of X_i. Since F_i is an information conserving function, X_i is a function of F_i and so C_i must also be a function of F_i. Therefore, by Theorem 3 we can generate C and ¬C from either X or F and n 1's and n 0's, where n is the number of signals in C. By Theroem 2 we can then generate F and the constants from F, C, and ¬C. Putting this all together as shown in Figure 11, we see that we have in fact generated F_i from X_i without any extra inputs or outputs.

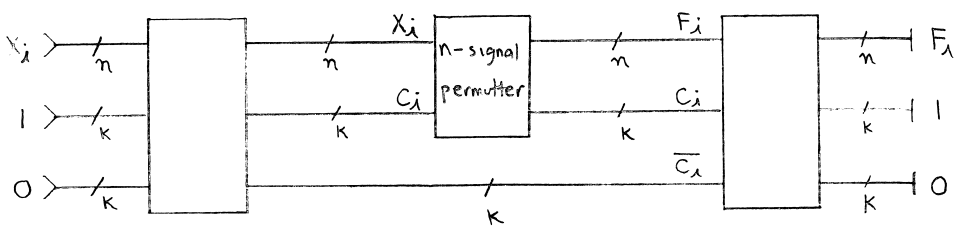
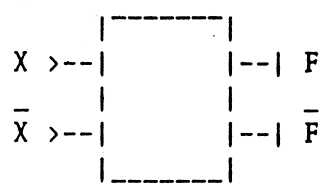


Figure 11 - Block diagram for proof of Theorem 4

Corollary 4.1. Any bit and information conserving function can be generated by a circuit of SMP gates with no extra inputs or outputs.

Corollary 4.2. If F is any information conserving set of functions of X, the following circuit can be made without garbage:



We have shown the strongest result possible, that a function can be generated from

either of our two gates if and only if that function is reversible and bit conserving. If our interconnection rules had been too restrictive, or our gates not powerful enough, this might not have been possible.