

A Framework for Conservative and Delay-insensitive Computing

Priyadarsan Patra

Donald S. Fussell

Department of Computer Sciences

The University of Texas at Austin

Austin, TX 78712-1188, USA

Asynchronous circuit elements are quiescent whenever they are not actually performing a computation, and thus they potentially waste less power than synchronous circuits. However, previous research on asymptotically non-dissipative computation has concentrated exclusively on synchronous computing models, while researchers on asynchronous circuits have ignored the issues of conservative, reversible computing inherent in ultra low power systems. We show that delay insensitive asynchronous systems can be made asymptotically non-dissipating. Our construction achieves this without the need for explicit uncomputation of results that has characterized previous synchronous approaches.

1 Introduction

Computation is a physical process and hence subject to physical laws. One needs to understand the fundamental limits imposed by these laws in order to develop asymptotically non-dissipative computers. Some of the basic work on the limits of computation has been addressed in the pioneering work of Bennet, Landauer, Fredkin, et al. (see, e.g., [2, 3, 8]). Landauer showed that erasure of information leads to a thermodynamic minimum heat dissipation of $kT \ln 2$ joules per bit.¹ Bennet [2] proved that irreversible logic gates and information erasure is not fundamental to computation. Since then various computational techniques have been proposed to avoid dissipation by “unwriting” information [1, 8, 9, 13, 15, 25] instead of erasing it.

Some of the important facts about physical processes relevant to asymptotically non-dissipative computation may be summarized as follows.

- The physical state of a closed system is a subset of all allowable physical configurations of the system, i.e., a subvolume of its *phase space*. Physical states are mapped and abstracted as logical states relevant to computation.
- Information erasure in a system leads to increase in entropy, hence to energy dissipation.
- Duplication of information (without erasure) implies a smaller phase volume and reduction in entropy.
- Abrupt change of physical state involves thermalization. (For example, slower charging of a physical capacitor and slower transportation of charge packets in a wire lead to lower dissipation.)

Conventional circuit design paradigm runs counter to both of the last two principles. For example, consider the computation performed by a *Nand* gate in isolation. The output of a *Nand* gate contains 0.811 bits of information (H) when supplied with two bits of information as input. That is, given that the two logic values assumed by an input are equally likely, the probability of the output going Low (p_0) is $1/4$ while that of going High (p_1) is $3/4$. By Shannon’s definition of information,

$$H = - \sum_{i=0,1} p_i \log p_i = -\frac{3}{4} \log \frac{3}{4} - \frac{1}{4} \log \frac{1}{4} \approx 0.811,$$

and therefore Landauer’s result indicates that $1.189 \times kT \ln 2$ joules must be dissipated by the *Nand* gate. Moreover, in current CMOS circuits, node voltages are abruptly switched to V_{ss} or V_{dd} , leading to high dissipation through thermalization in the parasitic circuit resistances.

¹Present-day computers dissipate upwards of $10^8 kT$ joules per bit operation – far from the physical limits [23].

Two related approaches to designing asymptotically non-dissipative systems have addressed these issues.

1. Information is treated as “objects” – e.g., ones and zeros in a computer – that are conserved during transformations as the computational process evolves. This type of *conservative* logic is exemplified by the Fredkin gate and the billiard-ball computer in [8].
2. Energy from power sources (usually the clock sources) is used in processing and transferring information from input ports to the output ports *adiabatically* [1, 13, 25]. At suitable times, explicitly determined by synchronizing clocks, this supplied energy is recovered or recycled back to the power source. During a computation, any intermediate, redundant values generated are eventually “unwritten.”

All of these approaches are based on synchronous computation – they assume lockstep, globally synchronized control. In this paper we present a model of asynchronous conservative logic. Our model is based on *delay insensitive* (DI) circuits, which use local handshaking instead of a global clock for synchronization. DI circuits are not based on boolean logic gates, which are as unsuited for DI computation as they are for non-dissipative computation. Our circuits are based on an entirely distinct set of primitive elements, which are more readily modified to support conservative computing than are boolean circuit elements. An interesting feature of our *conservative delay insensitive* (CDI) logic is that it does not require explicit uncomputation or unwriting of results as do existing synchronous approaches to asymptotically zero-power computation.

In the next section, we briefly review an prominent example of an existing synchronous approach to conservative logic. We then explain our notation and DI circuit primitives. Finally, we show how to modify the DI primitives to obtain CDI primitives, and we show how to build conservative state machines from these primitives.

2 Synchronous Conservative Systems

The key characteristics of proposed synchronous conservative systems are illustrated by the “Garbageless” circuit of Fredkin and Toffoli [8]. Fig. 1 shows the scheme used for designing the “combinational” part of a *synchronous* sequential machine. X_1 through X_m are the set of inputs, C_1 through C_k are the set of constants, and Y_1 through Y_n are the set of desired outputs where $Y = f(X)$. F is an extension of the combinational function f such that $F(X, C)$ when suitably restricted represents the output Y . The intent is that no information be erased and no “garbage” bits produced during evaluation of f . A bit is garbage if it is neither an output nor a copy of an input.

The circuit is an interconnection of 3 blocks: F , F^{-1} and “*Spies*”. F takes in C and X and produces G and Y . The “*Spies*” box transforms Y and “reconfigures” a set of constant objects to Y and \bar{Y} while making another copy of Y as well. F^{-1} takes in this copy of Y , and the output G from F to produce back the inputs: X and C . G represents the garbage or intermediate bits/objects that are “uncomputed” by F^{-1} . Each of these three blocks can be built from the so-called Fredkin Gates (FG) and hence, each is conservative and invertible. The transformation implied by a FG is indicated in the dotted box of Fig. 1.

For our purposes, a function $Y = fn(X)$ with $X, Y \in \{0, 1\}^n$ is *conservative* if $\forall X \wedge Y$ such that $Y = fn(X)$, the number of 1 elements of Y equals the number of 1 elements of X . Function fn is *invertible* if $\forall X \wedge Y$ such that $Y = fn(X), X = fn^{-1}(Y)$. It can be shown that for any function f , there exist functions F, F^{-1} and “*Spies*” which are conservative (so that no objects are destroyed or duplicated within the circuit), and invertible as well (allowing to avoid output of garbage bits).

You may note the following characteristics of this scheme.

- Explicit uncomputation of “garbage” G through F^{-1} is done to restore the internally used “scratch register” C and input “register” X .
- Input X is output again even if it is not useful as part of the output.
- A clean output “register” of n 0’s and n 1’s is needed for a subsequent evaluation of f as this clean register is rearranged by “*Spies*” to contain the desired output Y and \bar{Y} at the end of an evaluation.

3 Asynchronous Systems

The design of synchronous digital circuitry is based on the discretization of time. A synchronous system changes from one state to the next at transitions of a system clock. The state is held in a set of registers and the *next state*

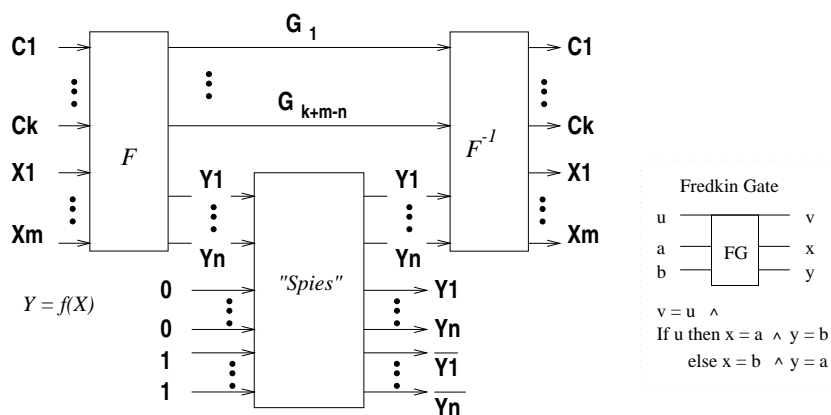


Figure 1: “Garbageless” circuit of Fredkin and Toffoli

and outputs are derived from boolean logic acting on the old state and present inputs. The next state is copied through the registers on every rising or falling edge of a global clock signal. The system exhibits deterministic behavior as specified by a global *state table* as long as certain timing constraints on the inputs are met.

Asynchronous design does not follow this methodology; in general there is no global clock to govern the timing of state changes. Subsystems exchange information at mutually negotiated times with no external timing regime. As increasing clock frequencies and circuit densities result in levels of power dissipation that threaten to disrupt the operation of a chip, asynchronous circuits are the subject of renewed attention due to their potential to achieve high levels of performance with lower power consumption than synchronous design. Some of the common claimed potential advantages of asynchronous circuits over synchronous ones are listed below.

- Clock distribution and skew problems disappear with the (global) clock.
- Circuit parts are always quiescent when they are not in use, thus minimizing their consumption of energy. Unplanned and unnecessary voltage transitions are prevented by design.
- Functional correctness of a circuit is independent of variations in element delays caused by the physical environment or by decreases in feature size (which increases diffusion delay.) As such, layout and routing become much less daunting. Modular composition allowing interchangeability of functionally equivalent subsystems is much easier in such conditions.
- Hardware runs as fast as the computational dependencies, input rate, and basic device switching times allow, rather than at the speed of the slowest path through the system.
- Logic hazards, races, and certain synchronization failures due to metastability become non-problems.
- Asynchronous systems seem particularly amenable to certain mathematical techniques for proving circuit correctness.

We are concerned with the energy efficiency *delay-insensitive* asynchronous circuits. A *delay-insensitive (DI)* system is one whose specified functional behavior is independent of any finite delays in the subsystems or in the wires interconnecting the subsystems’ terminals. There are a number of other models of asynchronous circuits. DI circuits are in a sense the strongest class of asynchronous circuits in that they make weakest assumptions about the timing behavior of circuit elements.

4 Basic terminology and concepts

A module or system is a hardware process with a well-defined set of external *behaviors* and with a set of input and output *ports*. An external behavior is a sequence of input and output *events*. An event is an identifiable physical condition or change at a port such as a voltage level or a pulse. A subsystem is a constituent part of

a system. Input terminals are connected to output terminals via directed “wires” (or channels). A well-formed system has no input terminal unconnected (“dangling”) and has no two output terminals connected together.

The first step in the functional design of a module is to obtain, from the problem requirements, a formal specification. A specification consists of the set of all *admissible* and *required* interface *behaviors* of a system and its external interface. We use *Trace Theory* ([21, 24, 10]) for such specifications. A *trace structure* TS is a triple $\langle I, O, T \rangle$ where I and O denote disjoint input and output alphabets, respectively, of the system, and $T \subseteq (I \cup O)^*$ is the *trace set*. An alphabet is a set of symbols; a trace is a sequence of symbols; a trace set is a set of traces. In a mechanistic interpretation of TS , the symbols in input/output alphabets correspond to the circuit’s input/output ports. Each symbol in a trace is an occurrence of an event at the port corresponding to the symbol. T corresponds to all the allowable sequences of external events. In other words, a trace is a (possibly partial) history of a circuit’s computation.

4.1 A formal characterization of DI circuits

A circuit C is specified as a triple (I, O, Q) , where I, O model the (finite, disjoint) input and output port sets, and Q is the non-empty set of *quiescent* traces of C . A trace t , which represents a particular history of circuit behavior, of C is quiescent iff t is not guaranteed to be extended (immediately followed) by an output event. Quiescent traces are meant to capture *liveness* properties of a circuit.

A circuit may be viewed as a specification or as an implementation under different circumstances. Consider two circuits $C1 = (I1, O1, Q1)$ and $C2 = (I2, O2, Q2)$.

Definition: $C1$ is a *tentative implementation* of $C2$ (and vice versa) iff $I1 = I2$ and $O1 = O2$.

Let $T1 = \mathbf{pref}(Q1)$ and $T2 = \mathbf{pref}(Q2)$. Note that $\langle I1, O1, T1 \rangle$ and $\langle I2, O2, T2 \rangle$ define trace structures.

Definition: A tentative implementation $C1$ is *safe* with respect to specification $C2$ iff

$$\begin{aligned} & (\forall t \in T1, a \in O : t \in T2 : ta \in T1 \implies ta \in T2) \wedge \\ & (\forall t \in T2, b \in I : t \in T1 : tb \in T2 \implies tb \in T1) \end{aligned}$$

Definition: Tentative implementation $C1$ is *live* with respect to $C2$ iff $Q1 \subseteq Q2$.

Definition: A circuit $C1$ is an *implementation* of specification $C2$, sometimes written $C2 \preceq C1$, iff $C1$ is safe and live with respect to $C2$.

Several authors [16, 22, 6, 5, 11] give equivalent definitions of DI specifications of systems. As in [22], we define a trace structure $TS = \langle I, O, T \rangle$ to be *delay-insensitive* iff the following five conditions are satisfied:

Prefix-closure and Non-emptiness

$$\epsilon \in T \quad \wedge \quad (\forall s, a : s \in T \wedge a \in I \cup O : sa \in T \implies s \in T)$$

An empty trace set represents a circuit with no possible history, and hence it is ruled out. Prefix-closure legalizes all prefixes of a trace, reflecting the “real-world” fact that, for a sequence of events to occur, all its prefixes must occur as well.

Independence From Relative Wire-Delays

$$\begin{aligned} & (\forall s, t, a, b : s, t \in T \wedge a, b \in I \vee a, b \in O \\ & : sabt \in T \equiv sbat \in T) \end{aligned}$$

This captures the fact that if two consecutive inputs arrive at a system in some order *without an intervening output*, the system cannot tell their time precedence (nor act differently). Note that this and all the other conditions are symmetric w.r.t. the circuit and its environment.

Absence of Computation Interference

$$\begin{aligned} & (\forall s, t, a, b, c : s, t \in T \wedge ((a, b \in I \wedge c \in O) \vee \\ & (a, b \in O \wedge c \in I)) \\ & : sacb \in T \wedge scat \in T \implies scab \in T) \end{aligned}$$

This specifies that the circuit cannot produce an output for the environment as long as the latter is not in a state to accept it. Usually the input transitions occur if certain output transitions have occurred and vice versa in the so-called *input-output mode of operation*. For example, a circuit output could serve as an acknowledgment of some previous input transition and/or as a “go-ahead signal” to the environment to produce

the next input. The formula above roughly implies that if we assume no *causal* relationship between input event a and output event c then any valid extension of computation sac must also be a valid extension of computation sca .

Absence of Transmission Interference

$$(\forall s, a : s \in T \wedge a \in I \cup O : s \in T \Rightarrow \neg (saa \in T))$$

Informally, this property rules out the situation where two consecutive transitions on an input line occur without any intervening output transitions. This relates to a practical concern, because two transitions in rapid succession on a wire may interfere with each other electrically and lead to “misreading” by the receiving component. Two such transitions may interact undesirably – e.g., they may cancel each other without the sender’s knowledge, or they may produce a “runt” pulse that is interpreted differently by different parts of the receiving circuit.

Proper Arbitration

$$\begin{aligned} (\forall s, a, b : s \in T \wedge ((a \in I \wedge b \in O) \vee \\ (a \in O \wedge b \in I)) \\ : sa \in T \wedge sb \in T \Rightarrow sab \in T) \end{aligned}$$

Proper arbitration implies that a DI system cannot choose between an impending input event and an impending output event. This type of arbitration is not allowed because for safety both the circuit and its environment must agree on the decision independently of each other while communication between them is not necessarily instantaneous!

Definition: A circuit $C = (I, O, Q)$ is delay-insensitive if $\langle I, O, \mathbf{pref}(T) \rangle$ is a DI trace structure.

4.2 A specification language

As noted above, symbols represent events and traces represent behaviors. Lower-case letters (subscripted or not) serve as symbolic names for communication *events* at similarly named communication ports. An event is an occurrence of the corresponding *action*. In circuit physics, signal (voltage) transitions are arguably the simplest and the most natural events of communication. There is a one-to-one mapping between actions and ports of a module. Sets of input and output actions – equivalently, their symbol sets – in a specification are implicit and disjoint: we append ‘?’ or ‘!’ to a symbol name to denote that the name stands for an input or output action, respectively. We may leave these suffixes out for internal signals or when no confusion is to be expected.

Symbol names in a trace structure can also be viewed as atomic *trace-structures* representing simple specifications. A more complex specification is built recursively from the primitive specifications by applying following operations: ‘**pref**’ is prefix-closure, ‘;’ is sequential composition,² ‘||’ is parallel composition, ‘|’ is non-deterministic choice, and ‘*’ is the Kleene-closure or *repetition*. A symbol s raised to a (natural) N indicates sequential composition of N such symbols.

The 1-place **pref** operation, when applied to a set of behaviors, represents all prefixes of those behaviors. All module behaviors are prefix-closed; all partial interface behaviors (i.e., communications) that lead to an admissible behavior are themselves admissible. The sequential composition of u and v , i.e., uv , denotes that behavior v necessarily follows behavior u . The 1-place ‘*’ operation denotes all finite concatenations (i.e., sequential compositions) of the behaviors in its argument set. The 2-place operation ‘||’ denotes concurrency³ between the two argument sets. The parallel composition of two argument sets of behaviors is the set of *all* behaviors satisfying the following:

1. It has only those symbols that appear in the implicit input or output sets of either argument.
2. When it is *restricted* (or projected) to the set of implicit input *and* output symbols of either argument behavior set, the result must be a legal behavior in that argument set.

²Very often we will use mere juxtaposition to denote sequential composition in order to avoid clutter. For a good introduction to trace theory for specifying circuits, see [7].

³Concurrency is used to capture the effect of arbitrary delays allowed in a DI model of communication wires carrying possibly simultaneous events. Therefore, all causally unrelated events are concurrent, and there is no notion of simultaneity unlike in many synchronous systems.

The 2-place ‘|’ operation denotes the union of the two argument sets of behaviors: e.g., $S = a(b | c)$, where all the symbols are either inputs or outputs, specifies that after a , either b or c , but not both, is allowed; thus, the complete set of valid traces is $\{ a b, a c \}$.⁴

5 Universal DI Primitives

In 1974, Keller [12] gave one of the first and best attempts to characterize the class of delay-insensitive (DI) circuits, and also provided a **universal** set of circuit primitives such that any circuit in this class is *realizable as a delay-insensitive network* of the primitives, i.e., a **DI decomposition** into primitives exists. It is well known that the class of DI circuits that can be implemented using only C-elements and boolean logic gates is quite small [14]. However, this need not be true for circuits constructed from a more robust set of DI primitives, indeed *Keller’s class* of DI circuits is essentially equivalent to the class of finite state machines realizable as synchronous circuits.

We have introduced a new set of primitive modules for delay-insensitive circuit design which are in a number of ways more efficient than Keller’s primitives [20]. These are shown in Table 1. Each primitive’s specification is shown next to it in the form of a trace-theoretic specification as described above.

A $m \times n$ -*Join* is operationally described as follows: It has m row inputs, n column inputs, and a matrix of $m \times n$ outputs—one for each pair of row and column inputs. The device and its environment repeat the following behavior: The device waits to receive exactly one row-input and exactly one column-input; upon receiving the two inputs it makes a transition on the output corresponding to the input pair. (*Joins* are equivalent under swapping of the row and the column inputs.) *Join11* is also called a *C-element*.

Example: We illustrate our trace-theoretic specifications using, as an example, a 1×2 -*Join* (equivalently 2×1 -*Join*), whose set of traces is given by: $\mathbf{pref}(((a?||b0?)c0!) | ((a?||b1?)c1!))^*$. The symbols with the suffix ‘?’ represent transition events at the three input ports of this module, namely, a , $b0$, and $b1$. Similarly, output symbols $c0$ and $c1$ represent two output actions (and their occurrences). Hence, the input set is $\langle a, b0, b1 \rangle$ and the output set is $\langle c0, c1 \rangle$. Examples of valid partial behaviors are: a , $a b0$, $a b0 c0$, $a b0 c0 b0$, $b0 a c0$, $b1 a c1 a b1 c1$, $a b1 c1 b0 a c0$.

Some invalid traces are: (1) $a b0 b1$, (2) $a a$, (3) $b0 c0$, (4) $a b0 c1$. The first two traces represent errors in the environment, while the last two in the module (refer to the operational description of a *Join*): (1) The environment cannot send both column inputs $b1$ and $b0$ without an intermediate output from the *Join*. (2) a cannot immediately follow itself for the same reason as above. (3) output $c0$ is produced too early. (4) $c1$ is the wrong output, $c0$ should be produced in stead. \square

A *Tria* outputs an event on a vertex when it has received events on the two edges adjacent to that vertex. A *Fork*, represented by branched lines, repeatedly copies each input event to both of its output ports. A *Merge*, usually implemented as an *Xor* gate in CMOS, repeats the following: it can receive exactly one event on either of the input ports and, upon reception, copies it to the output port. A *Toggle* device repeatedly copies an input event to an output before getting ready to receive the next input, but the output events are distributed between the two output ports alternately. Note that the *Toggle* can be implemented as a special case of 1×2 -*Join*. The *Mutex* and *Mem* elements provide mutual exclusion and delay insensitive memory, respectively.

A ‘bubble’ at an input terminal of a *Join* device implies an initialization that corresponds to a state where a transition at that terminal is assumed to have been received initially. We can alternatively imagine a *Merge* gate at the bubbled input that has an input port connected to a ‘START’ signal. For instance, the behavior of a C-element with a bubble at the a -input is a device with the behavior of a 1×1 -*Join* after receiving a transition on a first, i.e., $\mathbf{pref}(b? c!((a?||b?)c!))^*$. A heavy dot near a terminal of a device denotes an initialization where only the thus indicated terminal may produce the first output of the device. We occasionally use a circle labeled with ‘P’ as a short-hand for a tree of *Merges*, in our figures.

We have shown [20] that any module in Keller’s class is realizable as a DI network of *Mem* or alternatively as a network of *Fork*, *Merge*, *Mutex*, and *Tria* as primitive modules. That is, the sets $\{Fork, Merge, Mutex, 2 \times 1\text{-}Join, Mem\}$ and $\{Fork, Merge, Mutex, Tria\}$ are *universal* for Keller’s class of DI modules. We have also

⁴This trace-structure is not *delay-insensitive* [21], because it does not contain the trace ba , although ab is a valid trace, and both the symbols are inputs or outputs (hence, not *causally* related). The prefix-closure of S is $\{\epsilon, a, a b, a c\}$, so S also fails to meet the prefix closure requirement of DI trace structures.

shown that both these universal sets of primitives are *minimal* in that no proper subset of either set is itself universal.

The intuition behind our choice of primitives is to capture the following orthogonal aspects of functional (timing, area complexity, etc. are ignored) computation:

- Initiation of parallel processes
- Combining of sequential events (‘Or-causality’)
- Arbitration or non-deterministic choice between events
- Deterministic choice and synchronization
- Storage of values

Note that the aspect of computation that is sequential is inherent to the ‘inputs causing outputs’ nature of primitives and the structural connections among those primitives in a network. Informally, the first three primitives in our set match the first three notions in the list above. By appropriately hiding a pair of input and output ports, 2×1 -*Join* or *Tria* can be made to achieve synchronization among events. A $M \times 1$ -*Join* can be DI decomposed into *Merges*, *Forks*, and *Trias* or 2×1 -*Joins*, and hence can essentially support deterministic choice among a set of input events. We found *Mem* to be the simplest module that can support storage of a boolean value delay-insensitively.

6 Conservative and invertible DI operators

Consider a model of computation that treats each active signal event or transition or pulse in a circuit as an object distinguishable only by the associated I/O channel. A “gate” (or circuit operator) functionally transposes/maps a configuration of input objects into a configuration of output objects, according to its specification, when activated. A gate is activated when an acceptable set of input channels each receives an object.

During a computation step a standard boolean gate (e.g., *Nand*, *Or*, *Not*) may be thought to map input logic values to output logic values. Additionally, if we interpret a logic high (low) value as a presence (absence) of an object, then it is easily shown that all such gates, excepting the *Not*, are neither conservative nor invertible. In contrast, we show next that the DI primitives described above satisfy many of the requirements of conservatism and invertibility. Subsequently we formulate marginal changes to the original DI primitives that enable design of conservative DI circuits.

The *Merge* and *Mutex* are conservative but not invertible — i.e., the output does not uniquely determine the corresponding input transition(s). These are conservative in the sense that, for each input event consumed, *eventually* exactly one corresponding output event is produced.⁵ The non-deterministic operator *Mutex* consumes exactly one input event for every single output event it produces, but for now, we will ignore the fact that it may be hard to design a physically reversible operation corresponding to a *Mutex*.

The *Toggle* is clearly conservative, as it puts out exactly one event for each input event it consumes.

The various *Join* elements, the *Fork* and the *Tria* are all invertible operators but not conservative. A *Join* or a *Tria* consumes exactly two input objects (or transitions) at a pair of input ports to produce exactly one output object at the *specified* output port. Therefore, it is invertible. On the other hand, a *Fork* consumes one transition and generates two output transitions. Note that the DI-property ensures that a signal wire/terminal does not carry more than one transition at the same time through causal relations, i.e., by a feedback-acknowledgement mechanism.

7 Building-blocks for conservative DI Circuits

In this section we will define a set of DI primitives and other building-blocks that are later shown sufficient to realize conservative versions of circuits in Keller’s class.

⁵We note that DI theory mandates that a *Merge* receive at most one input event at a time. Therefore, it is perhaps a slight abuse to represent *Merge* pictorially by the common “exclusive-or” gate. The standard combinational operator exclusive-or is not conservative in the sense that it produces a *zero* output when its input signals are *one* each.

7.1 Conservative Joins

Earlier we saw that *Joins* are invertible but not conservative. The conservative version of a *Join* is called a *Cjoin* (or *conservative Join*), and we define it to have a matched pair of output terminals for each output terminal of the original (non-conservative) *Join*. We simply modify the specification of the *Cjoin* to have two concurrent events on the corresponding (matched) pair of output terminals if and when the non-conservative version has a single output event. This ensures that a *Cjoin* produces two output events for each appropriate pair of input events. In our pictorial notation, the symbol for a *Cjoin* differs from that of a *Join* in that each of the heavy dots representing an output terminal is additionally circumscribed by a dotted circle. Each such terminal is called a “doubled terminal”. Interconnection to such a doubled terminal is shown by using two lines — for instance, see Fig. 2. Whenever two signal wires emanate from such a terminal, they will be understood to carry the two concurrent transitions when that output terminal is activated — in some sense, this doubling acts like an internalized *Fork*.

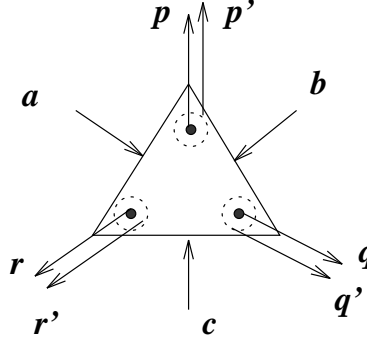


Figure 2: A symbol for *Ctria*

For an example of specification, consider the *Ctria* in Fig. 2. This primitive has 3 inputs a, b, c and 3 pairs of outputs (p, p') , (q, q') and (r, r') . The trace-theoretic specification is given as

$$\mathbf{pref} \left(((a? \| b?) (p! \| p'!)) \mid ((b? \| c?) (q! \| q'!)) \mid ((a? \| c?) (r! \| r'!)) \right)^*.$$

A 2×1 -*Cjoin* — with column input a , row inputs b_0 and b_1 , and output port pairs c_{00}, c_{01} and c_{10}, c_{11} — may be specified as

$$\mathbf{pref} \left(((a? \| b_0?) (c_{00}! \| c_{01}!)) \mid ((a? \| b_1?) (c_{10}! \| c_{11}!)) \right)^*.$$

Merge devices are conservative, but *Forks* are not. Hence, as a strategy to design conservative DI (CDI) circuits, explicit *Forks* should be avoided and the “internalized” forks of *Cjoins* should be used instead.

7.2 Conservative Sequencers

In Section 8 we will construct an arbitrary finite state machine using conservative DI devices. This forms the basis for the design of circuits that do not destroy events internally. Hence, there is no irreducible energy loss, in principle, under the model of physical processes we discussed earlier in this paper. One of the key modules used there is a multi-way *CSequencer*— a conservative *Sequencer*. In the following we develop specifications and constructions of such conservative arbitration components.

A realization of the (2-way) *Sequencer* shown in Fig. 3(b) is specified as $\mathbf{pref} \left((r_0? g_0!)^* \parallel (r_1? g_1!)^* \parallel (c? (g_0! \| g_1!))^* \right)$. A multi-way *Sequencer* may be constructed as suggested in Fig. 3(d). The *Sequencer* can be thought of as serving two clients capable of generating simultaneous service-requests. (1) After the “clock” event ($c?$) arrives, the device non-deterministically grants exactly one from any pending request(s) by outputting either $g_0!$ or $g_1!$. (2) Subsequently, when $c?$ arrives again, presumably indicating that the previously granted requester/client has relinquished a shared resource, it repeats its granting process from the first step.

Fig. 4 shows a *CSequencer* implementation which, unlike a *Sequencer*, emits the extra event out through the added *Merge* ($c'!$). The specification for *CSequencer* is thus:

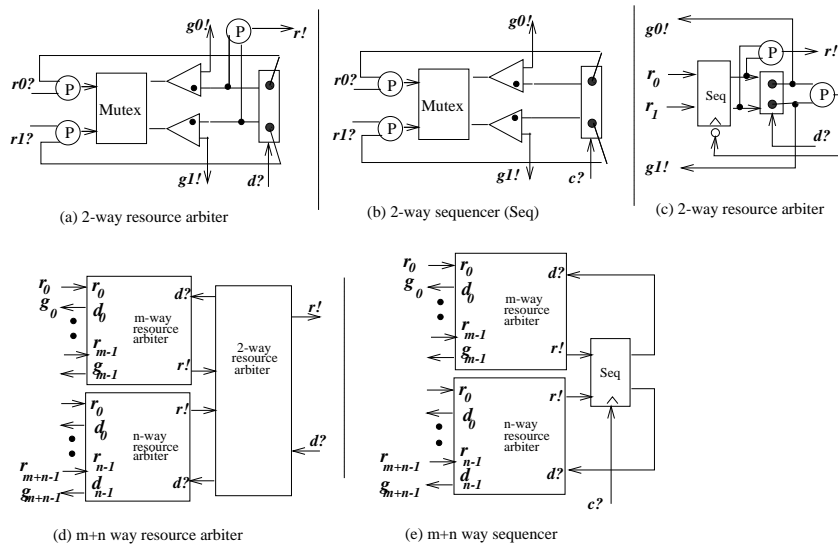


Figure 3: Realization of larger arbiters and sequencers

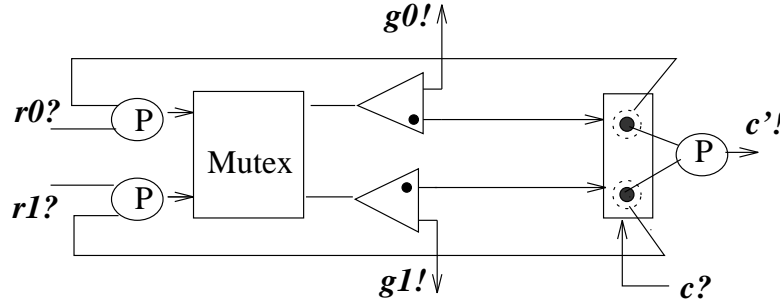


Figure 4: A 2-way CSequencer

$$\text{pref } ((r0?g0!)^* \parallel (r1?g1!)^* \parallel (c?c')^* \parallel (c?(g0!g1!))^*)$$

7.3 Conservative Resource arbiters

A multi-way *Sequencer* can be constructed through the use of the multi-way *Resource arbiter* drawn in Fig. 3. Fig. 3(a) depicts a delay-insensitive decomposition of the 2-way *resource arbiter* specified as

$$\text{pref } ((r0? a p g0!)^* \parallel (r1? b q g1!)^* \parallel ((a | b) r! d?)^* \parallel (d? (p | q))^*).$$

The *Resource arbiter* can be thought of as serving two clients capable of generating simultaneous service-requests. It repeatedly performs the following. (1) Non-deterministically pick up and service exactly one pending request at a time. (2) “Invoke” the resource ($r?$). (3) When the resource sends back a “done” signal ($d?$), the arbiter lets the correct requester know of completed service by emitting either $g0!$ or $g1!$.

The *Resource arbiter* is conservative at its external interface, but it internally uses 2×1 -*Join* and *Forks*, which are active one at a time and are not conservative. Yet on each “cycle” the 2×1 -*Join* consumes one transition extra while a *Fork* outputs one extra; there is perhaps hope! The scheme for recursive composition of a multiway *Resource arbiter* is shown in Fig. 3(d). The same scheme may be applied to build a multiway *CResource arbiter* based on the 2-way *CResource arbiters* discussed later.

Now, by using *Cjoins* in place of ordinary *Joins* and by using the trick of internalizing *Forks*, one can design a *Resource arbiter* that is internally conservative as well. This new decomposition for a *CResource arbiter* is shown in Fig. 5.

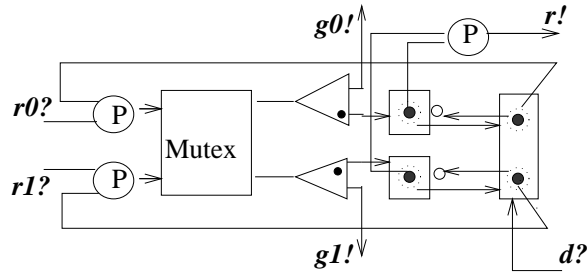


Figure 5: A conservative 2-way *Resource arbiter*

Fig. 3(d) shows how to build a multi-way *Sequencer* delay-insensitively from a 2-way *Sequencer* and a multi-way *Resource arbiters*. Making them conservative is as simple as replacing the parts by their conservative equivalents.

7.4 Decompositions of larger *Cjoins*

A 2×2 -*Join* can be decomposed into *Trias*, *Merges* and *Forks*, as shown in Fig. 6. However, we will often use the 2×2 -*Join* as a basic primitive in constructions for convenience. We will later also see that the 2×2 -*Cjoin* has a decomposition in the set $\{Merge, Ctria\}$ (see Fig. 11).

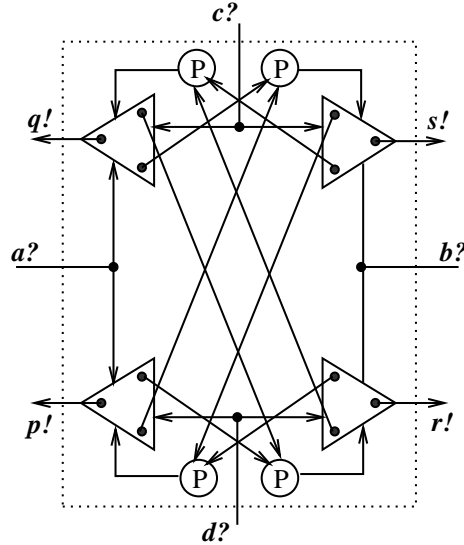


Figure 6: Decomposition of 2×2 -*Join* into *Trias*

In order to see how to decompose large $i \times n$ -*Cjoin*s to basic conservative modules, let us first consider how one might decompose an ordinary $M \times N$ -*Join* into a set of *Fork*, *Merge*, 2×2 -*Join*, 2×1 -*Join*, and 1×1 -*Join* primitives. Consider Fig. 7 which is a decomposition of $M \times N$ -*Join* into four ‘balanced binary decoders’ (*BBD*) and a $M \times N$ -*Tjoin* (dotted box) module, described below.

Column and row inputs of the $M \times N$ -*Join*, to be decomposed into the primitives, are each divided into nearly equal halves, and the resulting four halves are fed into four *BBD*s. A *BBD* is a conventional binary parity tree of *Merges* but, all the intermediate as well as input nodes of the parity tree are also made visible as output ports – it decodes an input signal in a special way. So, a *BBD* with K inputs has $2K - 1$ outputs.

A $M \times N$ -*Tjoin* module assimilates exactly $2N - 2$ column inputs and $2M - 2$ row inputs concurrently from the *BBD*s to generate the output that a $M \times N$ -*Join* is supposed to produce. The *Tjoin* repeats this behavior to simulate the $M \times N$ -*Join*. A simple strategy is used to determine the ‘quadrant’ of the $M \times N$ output matrix of a $M \times N$ -*Tjoin* to which the output belongs – corresponding to the pair (row and column) of inputs of the simulated $M \times N$ -*Join*. The structure of the *Tjoin* is described now.

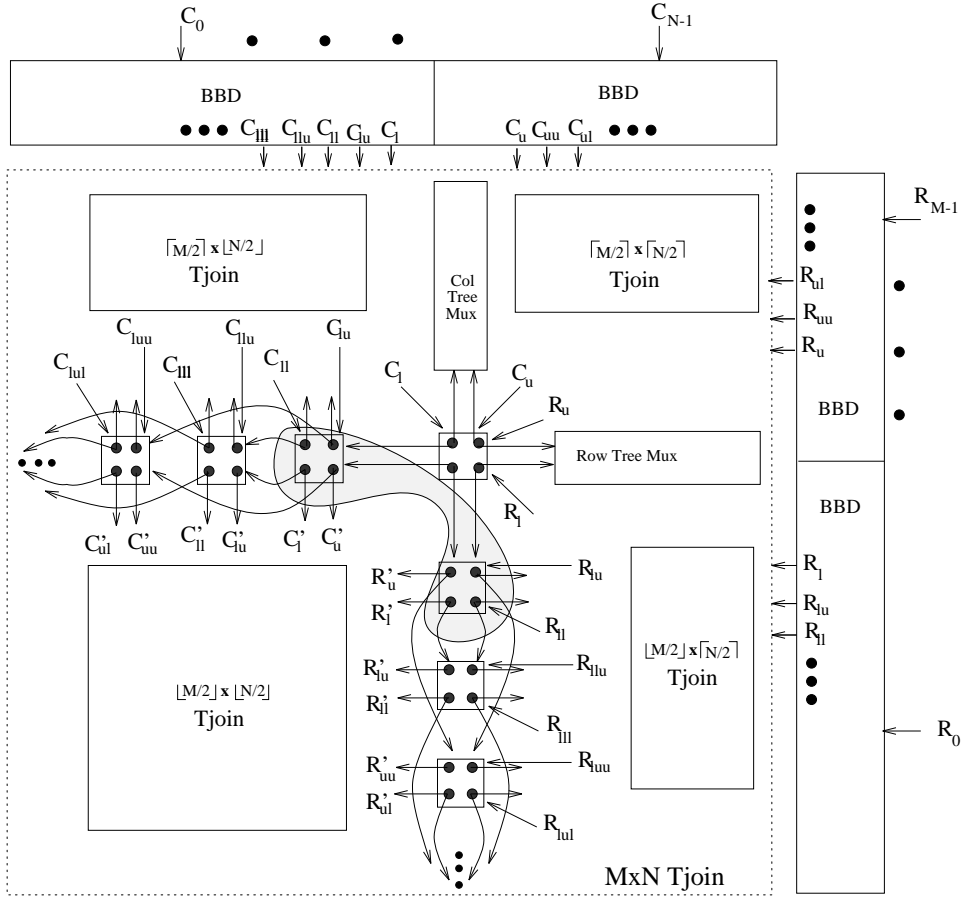


Figure 7: Optimal decomposition of $M \times N$ -Join module

Nominally, the $M \times N$ -Tjoin consists of a ‘central’ Join, a $2 \times \lfloor N/2 \rfloor$ -Tjoin, a $\lfloor M/2 \rfloor \times 2$ -Tjoin, a $2 \times \lceil N/2 \rceil$ -Tjoin, a $\lceil M/2 \rceil \times 2$ -Tjoin, two row and two column *Tree-Muxes*. The four most significant outputs – C_l, C_u, R_l, R_u – from the four *BBDs* are fed to the *central Join* which is usually a 2×2 -Join. (If there is only one column or one row input, then the corresponding central Join is a 2×1 -Join, a 1×2 -Join, or a 1×1 -Join.) This central Join steers the decoder outputs to the appropriate quadrant with the help of the *Tree-Muxes*. The lower column (row) *Tree-Mux* steers the decoded signals, C_{ll}, C_{lu}, \dots (R_{ll}, R_{lu}, \dots), to left or right (up or down) quadrant under the ‘control’ of the central Join. A *Tree-Mux* is a binary (preferably balanced) tree of 2×2 -Joins. See the Fig. 7 for a clearer picture.

Intuitively, you may think of the transitions in a Tjoin to be progressing like a wave. By the time the central Join generates its output, the 2×2 -Joins on its left row *Tree-Mux* and the column *Tree-Mux* below should have their column and row input signal, respectively. The output from the central Join is forked to the appropriate pair of *Tree-Muxes* whose partial outputs activate the $\lfloor N/2 \rfloor \times \lfloor M/2 \rfloor$ -Tjoin. Thereafter, the Tjoin and the *Tree-Muxes* synchronize as suggested and compute concurrently. In Fig. 7, we have shown a shaded region to indicate the Joins that are producing outputs after the central Join– assuming the inputs to the simulated $M \times N$ -Join occur in the lower halves.

Each Tjoin quadrant is decomposed recursively, as its parent is. The recursive decomposition ‘terminates’ when the present Tjoin quadrant to be decomposed has only one or two ports in each dimension (row or column). In this case the Tjoin is just the central Join of appropriate type.

We have $\log \max(M, N)$ sequentially ordered levels of signal flow before an output is produced. The parallelism between computations of a *Tree-Mux* and a Tjoin quadrant made possible due to the decoder signals in our method helps achieve a response-time complexity of $\Theta(\log \max(M, N))$, which is optimal. We assume that the

response-time of each constant-sized device is a constant.⁶

The (switching) energy expended in a module for a given computation is roughly proportional to the number of transitions made, during that computation, at the input and output ports of all the primitives constituting the module. Therefore, energy used for mapping (producing) an output corresponding to a pair of inputs is $\mathcal{O}((\log N)^2)$ for the $N \times N$ -Join, assuming each transition at a primitive's port consumes constant energy. We conjecture that this is also asymptotically optimal.

Now consider the special case of the 2×8 -Join module. Our asymptotically time-optimal design for this case is given in Fig. 8. Note the use of two ‘balanced binary decoders’ or *BBDs* for the column inputs. The unlabelled arrows with no destination are the output wires of the 2×8 -Join. Signal lines with same names are assumed to be logically connected. The tree-like topological structure at the bottom is a ‘Row Mux’, the other similar Row Mux on the left is indicated by a labelled box. The subscripts on the signal names indicate pattern – ‘l’ for lower and ‘u’ for upper half when the left half of the *BBD* inputs are considered lower, etc. – e.g., C_{ll} is for signals on lower half of the lower half of column inputs of the 2×8 -Join.

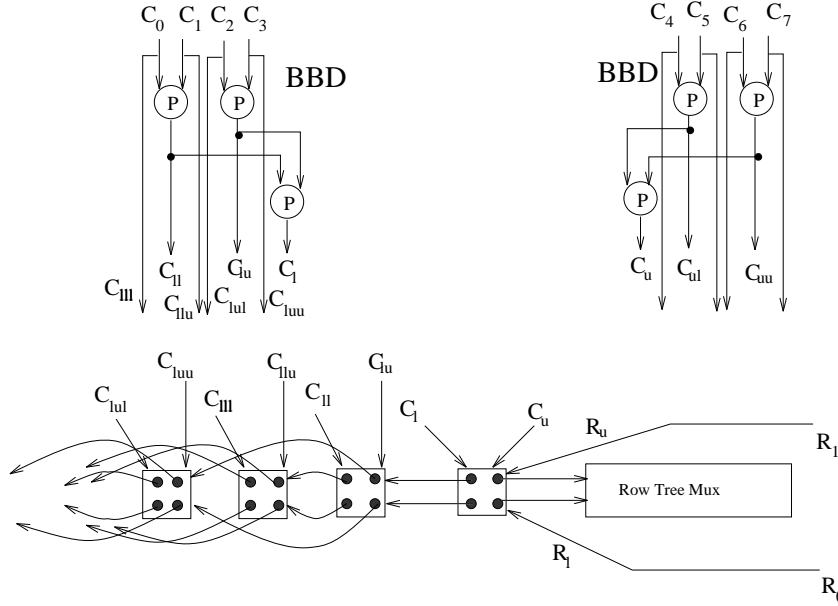


Figure 8: 2×8 -Join as part of Row Tree Mux

Fig. 9 shows the conservative version, $2 \times N$ -Cjoin. Here, we have replaced the *Joins* by *Cjoins*, and we have replaced *Forks* by 1×1 -*Cjoins*. Furthermore, we use arrows with triangle heads to indicate ‘doubled’ signals, i.e., two concurrent events on two separate signal wires. Note that some 2×2 -*Cjoins* receive a doubled row input and a non-doubled column input. This somewhat unusual 2×2 -*Cjoin* generates a corresponding doubled as well as a non-doubled output — in other words, three concurrent signal transitions (see below). Pairs of these non-doubled outputs feed the initialized port of an appropriate 1×1 -*Cjoin* through a *Merge*. Neither the *Merges* nor the initialization of these 1×1 -*Cjoins* are shown, to avoid clutter. For example, the two wires labelled C'_{ll} feed the 1×1 -*Cjoin* from which C_{ll} emerges.

We next show a simple way to interpret these triangle-head signal arrows in terms of an implementation within our basic repertoire of primitives — in Fig. 10.

Finally, we are ready to show the decomposition scheme for an arbitrary sized $M \times N$ -Join module. Consider Fig. 7, which is an asymptotically optimal decomposition of $M \times N$ -Join into four *BBDs* and an $M \times N$ -*Tjoin* (dotted box) module, each decomposable to our basic primitives. It is fairly easy to convert this design to a conservative equivalent based on ideas described above. All corresponding conservative versions of primitives are used. The ‘leaves’ of the Row Mux and Col Mux trees feed the appropriate 1×1 -*Joins* that have replaced *Forks* in the *BBD* subcircuits. The leaves of a tree at the outermost level of decomposition feed the 1×1 -*Joins* at the

⁶We have ignored delays in interconnects between primitives since delay-insensitivity places no functional requirement on them. Moreover, this somewhat simplifies our first-order comparison and analysis.

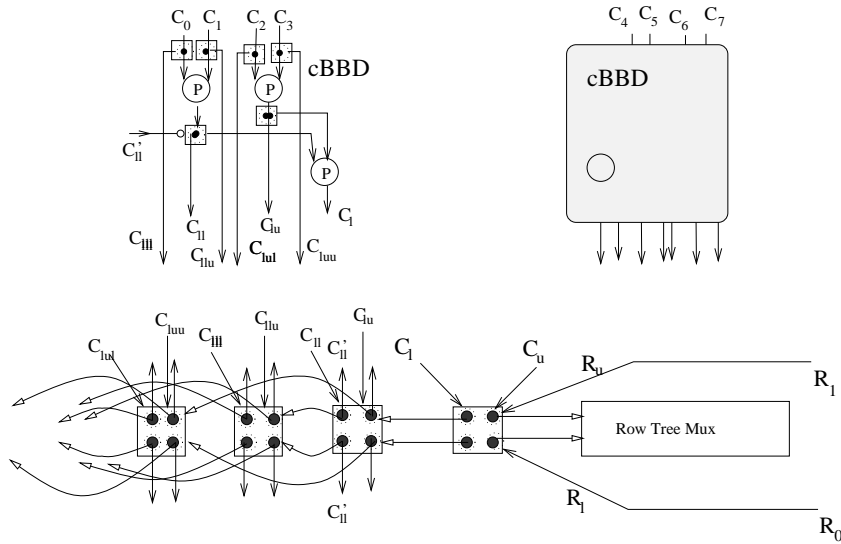


Figure 9: Decomposition of a 2×8 -*Cjoin*

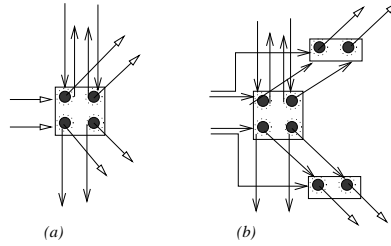


Figure 10: 2×2 -*Cjoin* with “doubled” inputs

outermost levels of the *BBDs*, and so on. Note that, for $M = N$, there is no need for using “doubled” 2×2 -*Cjoins*.

With this arrangement, we get two output events (from the 2×2 -*Joins* present at the “leaf-level” decomposition) for each pair of column and row input events to the $M \times N$ -*Cjoin*.

We are now ready to show below a general approach to implementing DI specifications as conservative circuits.

8 Construction of CDI circuits

We have shown that the set $\{Fork, Merge, Tria, Mutex\}$ is *universal* (i.e., functionally complete) for the large class of DI circuits called Keller’s class. Here we will consider construction of conservative circuits for specifications in Keller’s DI class.

Definition: A DI specification is *conservative* if, for every allowable *trace*, the number of input events differs from the number of output events by no more than a fixed constant.

The specifications $\mathbf{pref}((a?|b?)c!)^*$, $\mathbf{pref}(a!;a?)^*$ are conservative, as is the specification of a DI multiplier with two n -bit unsigned input operands producing a $2n$ -bit result. Specifications of the *Joins* are not conservative.

Definition: A DI circuit primitive is called CDI if it is conservative.

Definition: A circuit *implementation* of a DI specification is *conservative DI* (CDI) if it is a DI composition of CDI primitives only.

It follows easily that 1×1 -*Cjoin* and 1×2 -*Cjoin* are specializations of *Ctria* by way of hiding appropriate inputs and outputs. Fig. 11(a) shows a DI decomposition of 2×2 -*Join* (a variation from the design in Fig. 6). In this figure, signal lines numbered the same are logically connected — we have not drawn the connections to avoid clutter. A design for the conservative 2×2 -*Cjoin* is diagramed in Fig. 11(b) where several *initialized* 1×1 -*Cjoins*

are used in place of *Forks* in the non-conservative variation. We denote the “doubled” outputs by triangle-head arrows as usual. Note the use of conservative primitives only in this construction. Whenever we use a forked line emerging from a “doubled” terminal of a conservative *Join* device, we mean the two branches carry the two concurrent signals separately — an artifact to avoid clutter. The correctness of this decomposition can easily be verified.

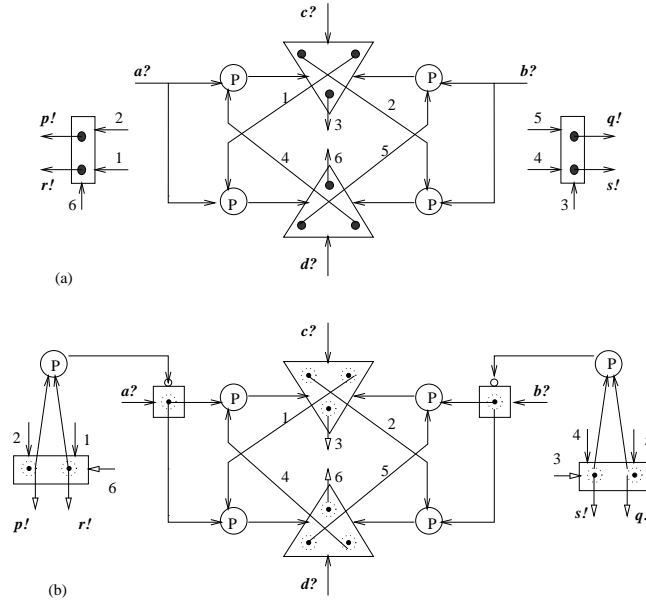


Figure 11: Realization of 2×2 -*Cjoin* from *Ctria* and *Merge*

The following theorem follows from the fact that any *Cjoin* device can be decomposed into 2×2 -*Cjoins* and *Merges*, and the DI decompositions of multi-way sequencers and the 2×2 -*Cjoin* shown above. of *CSequencer* above.

Theorem: The set $\{ Merge, Ctria, Mutex \}$ is universal for the class of CDI circuits. \square

Recall that in our model each primitive DI element consumes some number of input events and produces some number of output events during operation. Likewise, a large DI circuit also consumes input events provided by its external environment and provides output events to the environment. To avoid confusion in the context of large DI circuits, we may call the input and output events seen by a primitive circuit element *local* events, and we will call events seen by the DI circuit as a whole with respect to the external environment *external* events. Note that external events are also local events for some circuit primitives.

Definition: A DI circuit is *marginally conservative* if for every execution of the circuit, the sum of the local output events produced by all circuit primitives exceeds the sum of the local input events consumed by all circuit primitives by no more than the number of external output events of the circuit.

Theorem: Any DI specification in Keller’s class can be implemented as a marginally conservative DI circuit.

\square

Proof:

Control and environment inputs are allowed to occur concurrently, whereas the environment inputs are still sequenced serially by means of the multi-way *Sequencer*. The $M \times N$ -*Join* module simulates a ‘transition relation’ where the columns of the $M \times N$ -*Join* correspond to “internal-states” while the rows serve to provide inputs to the monolithic state-machine that takes one step at a time – assimilates a pair of events comprising a present-state input and an input from the environment – to produce any appropriate output signals, to change internal-state, and to “prime” the input *Sequencer* for the next external input. We simply extend this idea below and use conservative primitives.

The column of the *Cjoin* module with a bubble in Fig. 12 represents the initial state of the circuit. The top $1 \times N$ -*Cjoin* generates a ‘clock’ signal $c?$ for the M -way *CSequencer*. We have already demonstrated that

$M \times N$ -Cjoin and M -way CSequencer modules can be decomposed into a conservative network of primitives in $\{Merge, Mutex, Ctria\}$.

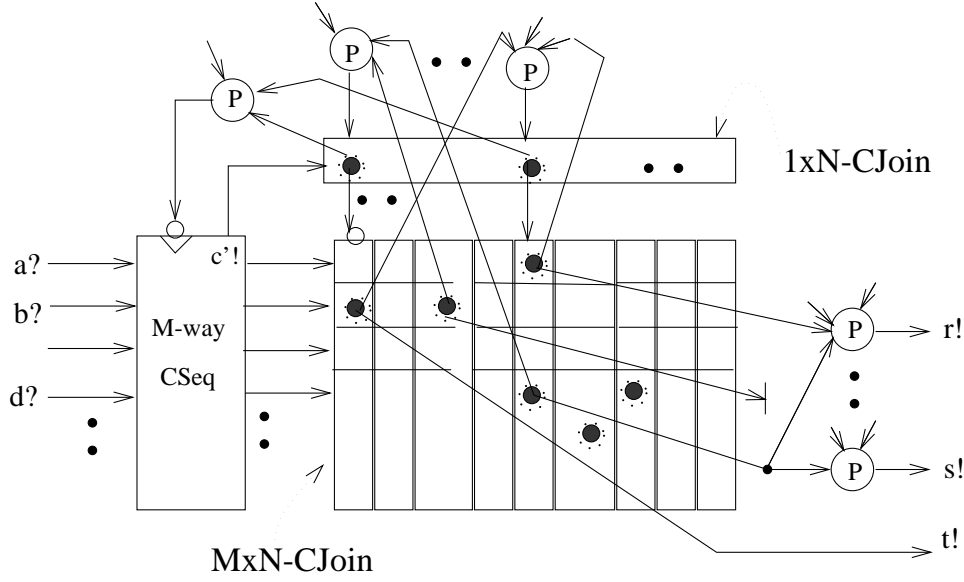


Figure 12: Marginally Conservative State Machine

One of the two resulting output events from the $M \times N$ -Cjoin goes on to activate the correct “next state” of the Conservative Finite State Machine (CFSM) and the other is steered to the appropriate output channel. At this level of abstraction, if no output is defined, the signal is “destroyed”, and if more than one output event is specified, then a number of extra events are generated and steered to the specified output channels. The CFSM is then ready to consume any input at the newly arrived internal state, repeating its previous actions.

All the sources of event destruction and creation are indicated on the right of the $M \times N$ -Cjoin. The *Sinks* depicted as short vertical bars indicate sites for destruction of an event while *Forks* indicate generation of events. Note that events are generated by *Forks* only when two or more concurrent external events (e.g., $r!$ and $s!$ in figure) are output per step of the machine. Events are not generated for any other internal computation, such as the determination of next state. \square

As stated before, *Sinks* in Fig. 12 indicate destruction of an event in the system. This happens when an input signal is consumed by the state machine without producing an useful output event. We can minimize this kind of event destruction, by storing the event internally for later emission as an output event.

Theorem: A conservative DI specification can be implemented as a conservative DI circuit. \square

Proof:

It suffices to show how events may be stored for latter emission without ever generating fresh ones or destroying them. Fig. 13 depicts a “conservative” architecture based on the design in Fig. 12. A “Storage Stack” module is employed to store and retrieve events during a circuit’s operation. The stack provides several channels for both “push” and “pop” operations. A push operation involves reception of two events and then transmission of an acknowledgement. A pop operation similarly involves one input event and two output events. The push and pop operations are strictly serial. Signals with the same labels are assumed to be connected in the circuit. An event is pushed into the stack if otherwise it would be destroyed. An event is popped if an extra external output event is to be generated consistent with the specification. In Fig. 13 we have shown two illustrative situations in relation to Fig. 12. In one case, the state machine “pushes an event” which gets acknowledged by an event on z . In the other case, it pops the stack to generate one additional, needed external output event – receives a pair of events x and y in response to a pop request (event).

Let the maximum difference between the number of input and output events in any trace be d . Then the capacity of the stack need not be more than $2d$. If the number of output events can exceed the input events in a trace by n , then the stack is initialized with $n \leq d$ stored events. The stack has a channel for each distinct push and pop signal.

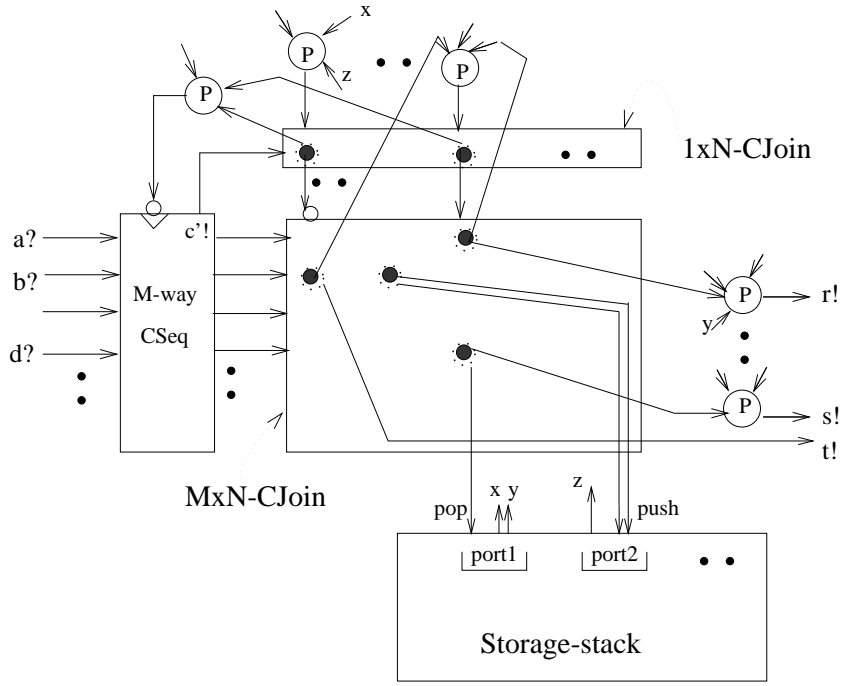


Figure 13: Conservative State Machine

We show below that a DI decomposition of such a stack exists. Consider the DI design of a storage stack of depth 4 in Fig. 14. It has two “push channels” and two “pop channels”. The stack is initialized with one “item” (i.e., three pushes can initially occur before a pop.) The inputs “store” and “push” always occur concurrently as a pair. The output of a pop is an (x, y) pair. (Signals are named to be corresponding symbols we have used in Fig. 13.) Forked signals, as before, are short hand for two concurrent output signals from a doubled terminal. z signals are acknowledgements to their corresponding push requests. The 2×5 -Cjoin serves as the “stack pointer”. The four 1×1 -Cjoins serve to store an event until it is popped. The “shadow register” simply shadows the stack pointer during a push. We have omitted *Merges* at the column inputs to the stack-pointer – whenever more than one input signal feed a single row/column of a *Join* device, we implicitly assume a *Merge* of those inputs. The stack-pointer (the active column input) changes its position in response to push and pop requests. The left most position corresponds to the stack-empty condition.

It is straightforward to extend this scheme, by an induction argument, to a stack of arbitrary depth k with any fixed finite number of push and pop channels. We need a $2d$ -depth stack in general because we are allowing output events to overtake input events by no more than d and vice versa.

This completes the proof of the theorem. \square

Corollary: Any CDI specification can be implemented as a DI network of *Mutex*, *Ctria* and *Merge* primitives. \square

9 Discussion and conclusion

It is interesting to note that it is possible to realize an invertible conservative, delay-insensitive sequential function even when using clearly non-invertible components such as *Merge*. This is possible because enough information is kept around for the function’s implicit inversion despite the local use of *Merges*. This is in contrast to garbageless circuits [8] built using only Fredkin gates which are invertible as well as conservative.

Note that our model requires little extra hardware or sequential time to make a delay-insensitive circuit conservative as well, for it does not need *explicit* phases or logic blocks for uncomputing – unlike all the existing approaches to reversible computing (see, e.g., [8, 1]). Moreover, our circuits can reduce the “garbage bits” that the existing approaches [3, 8] have to contend with — by transforming some or all input objects into output

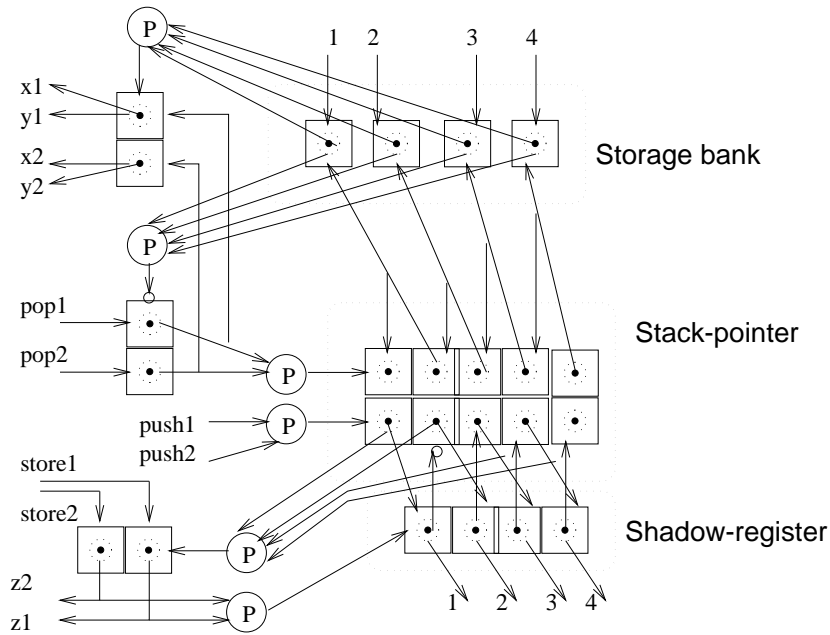


Figure 14: Storage stack of depth 4

objects. Much if not all of the “overhead of reversibility” in clocked systems is avoided by our DI approach to ultra low power circuits. While DI circuits admittedly carry some overhead for the necessary extensive local synchronizations compared to conventional, non-conservative synchronous circuits, it appears that this overhead also provides the much of the capability for nondissipative computing and that it may be much less overhead than that required for synchronous circuits to achieve similar power savings.

In the simulation of the state machine of previous section, we have used *Sequencers* which are inherently non-deterministic and may be difficult to physically realize in some technologies. However, non-deterministic behavior appears natural for quantum devices. It is possible, however, to design a delay-insensitive, conservative circuit without sequencers if the circuit’s DI specification does not imply true (output) arbitration for the circuit. Concurrency and input non-determinism are still allowed in such specifications. Note also that, although we have used monolithic state machine simulation for the proofs, these designs can be greatly improved (optimized) for specific circuits.

References

- [1] W. C. Athas, L. J. Svensson, J. G. Koller, N. Tzartzanis, and E. Chou. Low-power digital systems based on adiabatic-switching principles. *IEEE Transactions on VLSI Systems*, 2(4):398–407, 1994.
- [2] C. H. Bennet. Logical reversibility of computation. *IBM J. of Res. Dev.*, 17:525–32, 1973.
- [3] C. H. Bennet and R. Landauer. The fundamental physical limits of computation. *Scientific American*, pages 38–46, July 1985.
- [4] Janusz A. Brzozowski and Jo C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computers*, 41(11):1349–1360, November 1992.
- [5] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.
- [6] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

- [7] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.
- [8] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3/4):219–53, 1982.
- [9] J. S. Hall. An electroid switching model for reversible computer architectures. In *Proc. of the Workshop on Physics and Computation*. IEEE Press, 1993. PhysComp '93.
- [10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [11] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.
- [12] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.
- [13] J. G. Koller and W. C. Athas. Adiabatic switching, low energy computing, and the physics of storing and erasing information. In *Proc. of the Workshop on Physics and Computation*. IEEE Press, 1992. PhysComp '92.
- [14] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.
- [15] Ralph C. Merkle. Reversible electronic logic using switches. In *Nanotechnology*, volume 4, pages 21–40. (incomplete), 1993.
- [16] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.
- [17] Priyadarsan Patra and Donald S. Fussell. Building-blocks for designing DI circuits. Technical report tr93-23, Dept. of Computer Sciences, The Univ of Texas at Austin, November 1993.
- [18] Priyadarsan Patra and Donald S. Fussell. Efficient building blocks for delay insensitive circuits. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994.
- [19] Priyadarsan Patra. Asymptotically zero power in reversible sequential machines. Technical report, Dept. of Computer Sciences, Univ of Texas at Austin, 1995. CS Tech Report CS-TR-95-14.
- [20] Priyadarsan Patra. Approaches to Design of Circuits for Low-Power Computation. PhD Dissertation, Department of Computer Sciences, The University of Texas at Austin, December, 1995.
- [21] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.
- [22] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.
- [23] Jan L. A. van de Snepscheut. Reversible Computations. *What computing is all about*. Springer-Verlag, 1993.
- [24] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [25] S. G. Younis and T. F. Knight. Practical implementation of charge recovering asymptotically zero power cmos. In *Proc. of the 1993 symp. on Integrated Systems*. MIT Press, 1993.

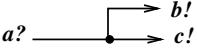
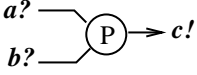
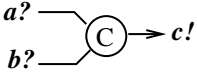
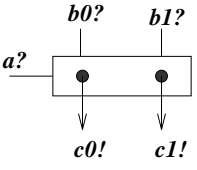
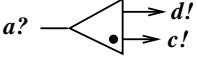
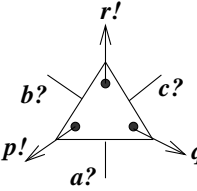
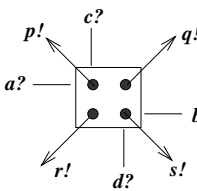
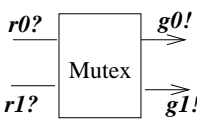
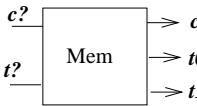
Name	Symbol	Specification
Fork		$\mathbf{pref}(a? (b! c!))^*$
Merge		$\mathbf{pref}((a? b?) c!))^*$
1x1-Join		$\mathbf{pref}((a? b?) c!))^*$
1x2-Join		$\mathbf{pref}(((a? b0?) c0!) ((a? b1?) c1!))^*$
Toggle		$\mathbf{pref}(a? c! a? d!))^*$
Tria		$\mathbf{pref}(((a? b?) p!) ((a? c?) q!) ((b? c?) r!))^*$
2x2-Join		$\mathbf{pref}(((a? c?) p!) ((a? d?) q!) ((b? c?) r!) ((b? d?) s!))^*$
Mutex		$\mathbf{pref}(r0? g0! r0? g0!)^* (r1? g1! r1? g1!)^* ((g0! g0!) (g1! g1!))^*$
Mem		$\mathbf{pref}((t? t0!)^* c? c!) (t? t1!)^* c? c!))^*$

Table 1: A set of representative DI Primitives