

Efficient Building Blocks for Delay Insensitive Circuits

Priyadarsan Patra and Donald S. Fussell

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA

Abstract

We introduce a new set of primitive elements for delay-insensitive (DI) circuit design. This set is shown to be universal and minimal, that is, any DI circuit can be constructed using only these primitives, and no proper subset of them is sufficient for constructing all such circuits. We give area efficient, fast, and robust switch-level implementations of key primitives and show how to use them to construct other DI circuit elements commonly found in the literature.

1 Introduction and background

In recent years, interest in asynchronous circuits has been on the rise, in spite of the fact that correct asynchronous systems have long been considered much harder to build than synchronous systems in the absence of rigorous approaches to design. This interest is due to the potential advantages of asynchronous circuits over synchronous circuits in a number of areas ([1, 2, 3]). These advantages include the robustness of fully asynchronous circuits to scaling of circuit feature sizes, metastability, and to large variations in operating conditions such as temperature and their potentially higher power-efficiency and speed. For example, a great deal of power can be saved by eliminating the global clock, which causes transitions at every storage element in a circuit every time a clock transition occurs. Similarly, since only those portions of an asynchronous circuit actually used in performing a particular operation contribute to the delay or dynamic power consumption required to complete an operation, additional energy savings can be expected, and the average speed required to complete a complex computation need not be limited to the speed of the slowest path in the circuit as with synchronous systems.

On the other hand, it seems likely that asynchronous circuits generally require more area to implement a given function than do synchronous circuits, since the former commonly make use of extensive handshaking and parallel signal paths to maintain the synchronization of computations and data. However, without satisfactory theoretical or practical demonstrations, it remains debatable whether or not the overhead of asynchronous circuit designs counterbalances their potential advantages. Many practitioners and even researchers on asynchronous circuit design continue to believe that asynchronous circuits entail a prohibitively high area penalty when compared to synchronous circuits and that they are therefore impractical to use in most cases.

Perhaps surprisingly, there is little existing work on systematic methods for optimizing the area, speed and energy requirements of asynchronous circuits. For instance, Keller[4] 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. However, no even moderately efficient implementation strategy or methodology was provided for accomplishing this. More recently, [5] formalized classes of delay-insensitive circuits in Trace theory by defining closure properties of Trace structures representing *delay-insensitive(DI) computations*, but did not deal with the synthesis of efficient circuits using these elements. [6] and [7] have developed composition operators and algebras to model speed-independence and to verify equivalence of DI specifications, respectively. [8] has developed grammars (not provably complete, see [9]) to specify DI circuits that induce a syntax-directed translation into a basic set of primitives. [10] uses most of Keller's primitives and some more complex primitives to compile process algebras into DI circuits. [11, 12, 13, 14] have devised practical syn-

thesis techniques, yet they impose several restrictions on the specification unrelated to delay-insensitivity or speed-independence and provide very limited means for *composing* and *decomposing* DI modules. An automatic compiler in [3] applies some area optimization techniques to DI ‘control’ modules built from a larger and more complex set of primitives than we propose. However, in each case, these methods are primarily concerned with correctness issues, and often ignore the cost and optimization of the resulting designs.

We believe that *delay-insensitive* (DI) circuits are potentially the most desirable asynchronous circuits. They make the weakest assumptions about the timing relationships of elements in the system and thus are most robust to conditions which may change timing relationships such as temperature variations. They also most effectively eliminate the need to explicitly reason about time for designing circuits and thus make the problem of correct design simpler than for other types of asynchronous circuits. Thus, they are well-suited for use with formal specification techniques for designing asynchronous circuits, which we believe are essential for making it possible to design a large class of correct and efficient asynchronous circuits.

However, there is reason to be concerned that DI circuits require the greatest overhead of all asynchronous designs, enough overhead, perhaps, to prohibit their use in practice. Our aim is to mitigate this concern by demonstrating that the overhead can be kept quite small. In this paper, we take a significant basic step toward this goal. First, we define a small set of basic DI circuit primitives which are efficient in the sense that their area and power requirements are small with no sacrifice in speed. We show that this set of primitives is *universal* in that any possible DI finite-state machine can be constructed of such primitives. We also show that the set is *minimal*, i.e. that no subset of these primitives is universal. Moreover, our primitive set is shown to be *optimal* in its *I/O modularity*, a measure of DI element complexity introduced by Keller [4] in 1974. This is the first set of DI primitives we know of that is optimal in this sense. Finally, we demonstrate how to construct some larger DI circuit elements, commonly used in the literature, from our basic set of primitives. [15] was pointed out to us which is another attempt to build purely DI-circuits, but they use much more complex primitives (‘Demultiplexors’) that embody arbitration as well as decision-wait functions. They do not show minimality of primitives or DI-decomposability of arbitrary-sized *Join* modules into a small-set of basic ones, as we do.

2 A formal model and a language of specification

The first step in functional design of a circuit is to obtain, from the problem requirements, a formal specification. A specification is the set of all *admissible* and *required* interface **behaviors** of a system. A behavior is an interleaving of **input** and **output** events. The specification models a circuit module by help of a theory: a formal language (or syntax) and a formal interpretation (or semantics). We chose **Trace Theory** ([5, 16, 17]) for exposition of our methods and for describing our primitive modules.¹

In trace theory, 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 those suffixes out for internal signals or when no confusion may arise.

Symbol names 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 sequential² composition, ‘||’ is parallel composition, ‘|’ is non-deterministic choice, and ‘ \star ’ 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 – this formally legitimizes the common-sense observation that 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 ‘ \star ’ operation denotes all finite concatenations (i.e. sequential compositions) of

¹Various other languages based on **Temporal Logics** exist that capture **safety**, **progress** as well as **fairness** properties of general systems.

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 [18].

the behaviors in its argument set. The 2-place operation ‘ \parallel ’ is more complex and 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 behavior in that argument set.

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

2.1 Primitive modules and notation

We need a repertoire of basic devices or **primitives** to build general circuits and our choice is given in Table 1. Each primitive’s specification is shown next to it in the form of a specification.

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. this paper.)

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.

⁴This trace-structure is not *delay-insensitive* [5], 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). Prefix-closure, a requirement for DI trace-structures, of S is $\{\epsilon, a, a b, a c\}$.

Example:

We illustrate trace-theory notation using, as an example, a 1×2 -*Join*, whose set of traces is given by: $\text{pref}(((a?||b0?)c0!) \mid ((a?||b1?)c1!))^*$. 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.

A *Fork*, represented by branched lines, accepts one input and then produces two outputs, before repeating self. A *Merge*, also known as a 2-input *Xor*, can accept exactly one input which is followed by an output transition, before it repeats itself. A *Toggle* device distributes an input transition between the two outputs alternately.

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. $\text{pref}(b? c!((a?||b?)c!))^*$. 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.

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. $\text{pref}(b? c!((a?||b?)c!))^*$.

. 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.

2.2 A minimal, universal set of primitives

Keller ([4]) characterizes the class of delay-insensitive modules that are finite DI networks of finite primitive modules. We show here that any module in Keller’s class is realizable as a DI network of *Fork*, *Merge*, *Mutex*, 2×1 -*Join*, and *Mem* as primitive modules. We believe that not only do these primitives have few I/O ports, and have simple, efficient physical implementations under the robust *transition* signalling convention, but also they facilitate more efficient decompositions of a very large class of DI modules. Furthermore, finding such sets has deeper significance in parallel language design for asynchronous communicating processes and dataflow networks in regard to the necessity and sufficiency of language operators for classes of computations. The initial intuition behind any choice is to capture the following orthogonal aspects of 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* can be made to achieve synchronization among events. A $M \times 1$ -*Join* can be DI decomposed into 2×1 -*Joins*, *Merges*, and *Forks* 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. *Mem* turns out to be identical to the “G module” described in [4].

Choosing a minimal set is more of theoretical significance than practical, although it helps as a starting point for building a library of VLSI delay-insensitive standard-cells.

We have found certain other primitives to be very handy in many design situations and also while proving equivalences between sets. Therefore, our strategy is to show transformations between sets of these primitives. Specifically, we exhibit DI realizations of the modules in Keller’s universal sets as networks of our primitives. First, we show some useful constructions before we give the necessary decompositions. We give the decompositions which are easy to check, but we provide no formal verification.

2.3 Decomposing Larger Joins

We show here that an arbitrary $M \times N$ -*Join* can be decomposed, with asymptotic optimality, into a set of *Fork*, *Merge*, 2×2 -*Join*, 2×1 -*Join*, and 1×1 -*Join* primitives. Consider Figure 1 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 *BBDs*. 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 *BBDs* 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:

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

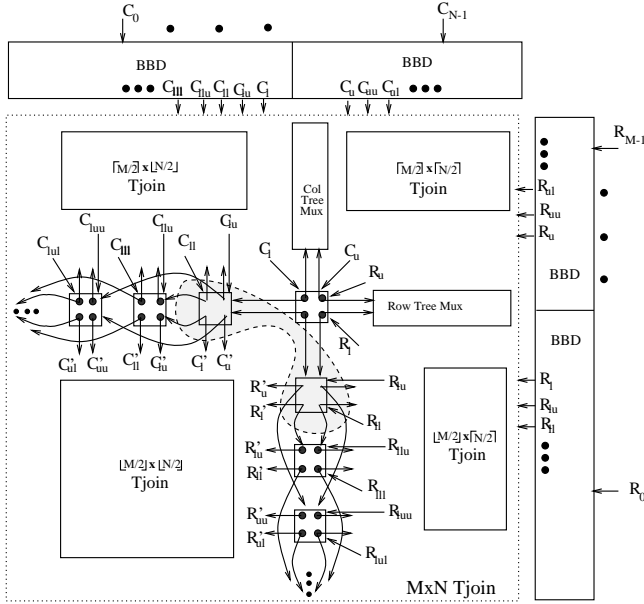


Figure 1: Optimal decomposition of $M \times N$ -Join module

balanced) tree of 2×2 -Joins. See the Figure 1 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 $Tree-Mux$ es whose partial outputs activate the $[N/2] \times [M/2]$ - $Tjoin$. Thereafter, the $Tjoin$ and the $Tree-Mux$ es synchronize as suggested and compute concurrently. In Fig 1, 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.

[19] has a decomposition where in place of our BBD , $Tree-Mux$, $Tjoin$ quadrant, a simple Parity Tree, a $p \times 2$ -Join or $2 \times p$ -Join, and a $Join$ quadrant are used respectively.

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

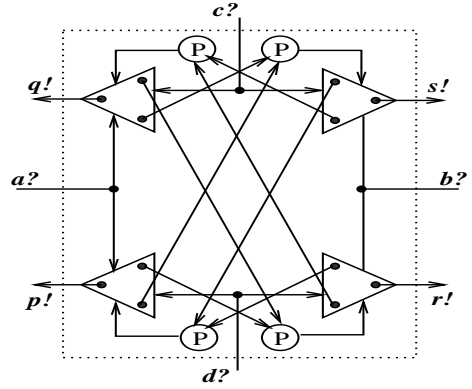


Figure 2: Decomposition of 2×2 -Join into *Trias*

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.⁵ (The response time of a $p \times 2$ -Join can be seen as $\Omega(\log p)$. At each level i , a $(N/2^i) \times 2$ -Join responds in $\mathcal{O}(\log N - i)$ units of time, making the time complexity of the method of [19], as described to us, equal to $\Omega((\log \max(M, N))^2)$.)

Moreover, since neither a $Tree-Mux$ nor a $TJoin$ uses *Merges* in their respective decompositions, our method has slightly better area complexity than in [19].

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.

2.3.1 Universality

Figure 2 shows a realization of 2×2 -Join as a network of *Forks*, *Merges*, and *Trias*.

Note that by hiding appropriate inputs and outputs of a 2×2 -Join, we can obtain 1×1 -Join (Muller’s C-element, shown as a circle around ‘C’) as well as a 2×1 -Join. Now, we are ready to show the decompositions of Keller’s *Select* and *ATS* modules in Figure 3.

⁵We 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.

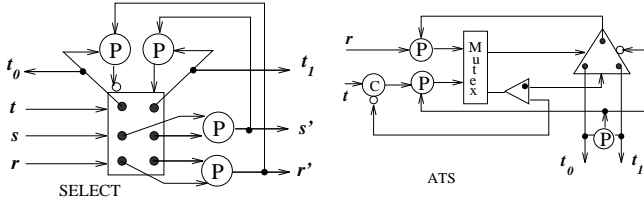


Figure 3: Decomposing *Select* and *ATS* modules

Their specifications are given below in a slightly different, but very concise, form. H and L in the following are *states* of the primitives. Observe this interpretation: wherever a state is mentioned (in a trace) the specification shown equivalent to that state is substituted for it, recursively as needed. Assume L and H to be the initial states for *Select* and *ATS*, respectively.

Specification for *Select*:

$$H \equiv (ss' H \mid rr' L \mid tt_1 H)$$

$$L \equiv (ss' H \mid rr' L \mid tt_0 L)$$

Specification for *ATS*:

$$H \equiv (tt_1 H) \parallel (r L)$$

$$L \equiv (tt_0 H)$$

Very often the number of I/O ports is proportional to the complexity of interaction between a module and its environment and hence, inversely proportional to ease of use. For proofs of correctness concerning such practical issues as testability, it is far easier to deal with a smaller set of primitives.

Definition 1: I/O-modularity of a set of modules is the maximum number of I/O lines on any module in the set.

Definition 2: A module is called *serial* if its specification prohibits all concurrency in inputs and outputs, i.e. each output event has exactly one unique input event that *causes* it and vice versa. A module is called *parallel* if its specification allows concurrent inputs and concurrent outputs.

It is important to be able to design circuits that do not have signals racing continuously, for the sake of energy-efficiency and speed. For example, a *Sequencer*, classically specified as $\text{pref}((r0?g0!)* \parallel (r1?g1!)* \parallel (c?(g0!g1!)*))$, can be implemented using two *ATS* modules as in Figure 4. A signal races around continuously after $c?$ is received ‘polling’ the inputs until an arbitration is made.

Definition 3: A module’s decomposition is *without busy-waiting* if the internal wires and component modules have no transitions imminent, when no in-

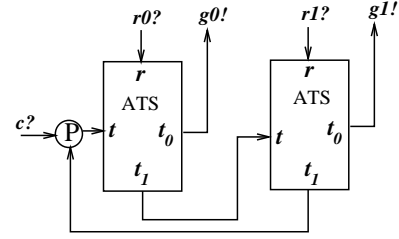


Figure 4: Busy-waiting Sequencer

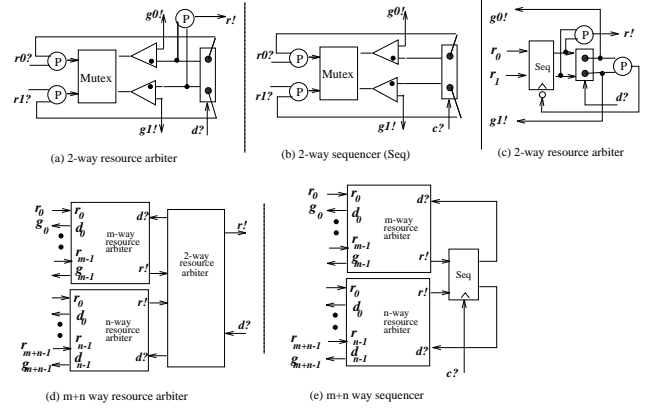


Figure 5: Realization of larger arbiters and sequencers

puts are supplied to the module for a sufficiently long time. A set of modules is **wbw-universal** for a class if any module in the class is decomposable without busy-waiting.

Figure 5(a) depicts a decomposition of the *2-way resource arbiter* specified as

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

A realization of the *2-way Sequencer* is shown in Figure 5(b). Recursive realizations of larger resource-arbiters and sequencers, without busy waiting, are indicated in Figures 5(d & e). For sake of completeness, we observe that an n -input *Merge* can be realized from 2-input *Merges* as a ‘parity tree’, which we often pictorially denote as a circle with a ‘P’ on it.

Theorem 1: The set $\{\text{Fork}, \text{Merge}, \text{Tria}, \text{Mutex}\}$ is of I/O-modularity 6, cardinality 4, and is *wbw-universal* for Keller’s class of DI modules. \square

[4] shows that any parallel module M may be realized by serializing the possibly concurrent inputs, and then feeding to an appropriate serial module M' , the details of which are not important for the following discussion and for which the reader is directed to that paper. To avoid busy-waiting in the input serializer, Keller used a *Arbitrating Call (AC)* module which is exactly what we call a *2-way resource ar-*

biter. But, we have demonstrated constructions of multi-way *Sequencers* and *resource arbiters* in Figure 5. Moreover, note that a *Sequencer* implements a 2×1 -Join directly, if we interpret the column input of 2×1 -Join as input $c?$ of the *Sequencer*.⁶ A *Toggle* is a specialized 2×1 -Join. Therefore, multi-way *Sequencers* can be delay-insensitively composed from 2-way *Sequencers*, *Merges* and *Forks*. Hence,

Theorem 2: The set $\{Fork, Merge, Select, Sequencer\}$ is *wbw-universal* for Keller’s class of DI modules. \square

To the extent that low I/O-modularity and low cardinality of a universal set signify simplicity, we have partially answered some pertinent philosophical questions raised in [4], by discovering *wbw-universal* sets which have a lower I/O-modularity in comparison to Keller’s *wbw-universal* sets with same cardinality.

Theorem 3: The set $\{Fork, Merge, Sequencer, Mem\}$ is *wbw-universal* for Keller’s class of DI modules. \square

Proof: A *Sequencer* implements a 2×1 -Join which in turn implements a *Toggle* upon proper configuration, as noted before. A decomposition of a 2×2 -Join is demonstrated in Figure 6. This theorem follows from our previous discussion of transformations to Keller’s set. \square

Theorem 4: The set $\{Fork, Merge, Sequencer, Mem\}$ is *wbw-universal* for Keller’s class of DI modules, and is of IO-modularity 5 and cardinality 4. \square

Proof: A *Sequencer* implements a 2×1 -Join which in turn implements a *Toggle* upon proper initialization, as noted before. A decomposition of a 2×2 -Join is demonstrated in Figure 6. Theorem follows from our previous discussion of transformations to Keller’s set \square

Theorem 5: The set $\{Fork, Merge, Mutex, 2 \times 1$ -Join, *Mem* $\}$ is *wbw-universal* for Keller’s class of DI modules. \square

Proof: This theorem follows from the previous theorem, and the previously shown decomposition of a *Sequencer* into $\{Fork, Merge, Mutex, 2 \times 1$ -Join $\}$. \square

Finally we observe that a 2×2 -Join can be decomposed, albeit inefficiently, into $\{Select, ATS, Merge and Fork\}$.

⁶This points out that an implementation could be ‘better,’ by being able to ‘tolerate’ a worse environment than the specification requires. But, bear in mind that the constructions in this section are not necessarily what we will use in practice. with efficient transistor-implementations is currently being investigated.

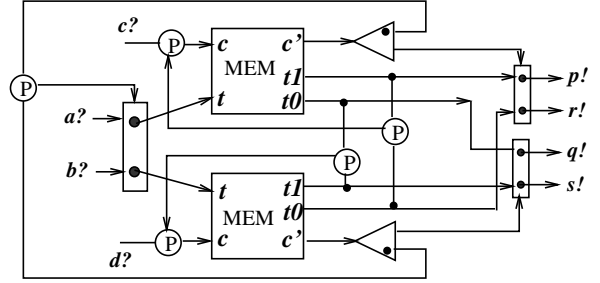


Figure 6: A decomposition of 2×2 -Join using *Mem*

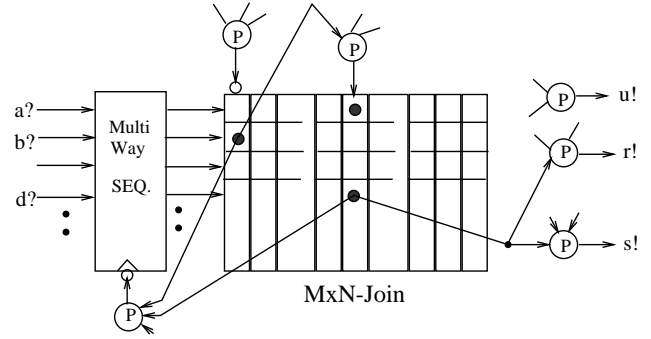


Figure 7: State Machine Simulation

2.3.2 State Machine Simulation

Here we indicate a direct way to show universality by simulating a finite state machine that represents an arbitrary DI specification. This construction exploits parallelism and synchronization properties of *Trias* (alternatively, 2×2 -Joins). In Figure 7 control and environment inputs are allowed to occur concurrently whereas the environment inputs are still sequenced serially. The columns of the $M \times N$ -Join simulate flow of control while the rows serve as inputs to the monolithic state-machine that takes one step at a time (assimilates a single control signal and a single environment input) to produce any appropriate output, to change control, and to ‘clock’ the input serializer to obtain the next input. The dotted boxes indicate banks of similar components, representatives of which are pictured in each bank. The parity trees, the multi-way *Sequencer* and the $M \times N$ -Join have been demonstrated before to be DI decomposable into $\{Fork, Merge, Mutex, 2 \times 1$ -Join, *Mem* $\}$. The $M \times N$ Join can be made sparse, for efficiency, by excluding output ports for all those control and input combinations which are disallowed by the machine’s specification.

2.3.3 Minimality

Definition 4: A set of modules is *minimal* if none of them can be realized by the rest. In other words, a set S is minimal if no proper subset of S is universal with respect to S .

Lemma 1: $\{Fork, Merge, Mutex, 2 \times 1\text{-Join}, Mem\}$ is minimal.

Proof: $2 \times 1\text{-Join}$ is the only component that assimilates more events than it produces — the number of input events exceeds the number of output events unboundedly with increasing length of a valid trace. So no finite composition of only the rest of the primitives in the given set can correctly extend a trace to produce this property. Therefore, $2 \times 1\text{-Join}$ is irredundant. (The above comment about a $2 \times 1\text{-Join}$ applies to $Tria$ also.)

None in the set other than a *Fork* produces more events than it assimilates. Hence, *Fork* is irredundant.

Merge is the only primitive such that an event at its output does not identify the actions directly causing (i.e., the immediate predecessors of) this event. Identification through events means separate output channels, because an event itself does not carry information. Hence, no finite (DI) network of the other primitives in the set can realize a *Merge*.

The *Mutex* clearly is the only primitive for mutual-exclusion between two concurrent events.

Mem is the only module capable of producing one of two events (viz, t_0 and t_1) each having exactly one and the same immediate predecessor event (viz, t). In other words, it exhibits memory. \square

A weaker version of the following was cited as open in [4].

Theorem 6: No finite set of primitives with I/O-modularity less than 5 is universal with respect to the class of serial modules. \square

Proof: A single boolean value cannot be distributed on more than one atomic DI module (i.e., the bit of information is ‘indivisible’). Transfer of information, if any, between modules takes place in bits.

Consider implementing the serial module *Mem*. Any finite network of DI primitives realizing *Mem* needs to have a primitive capable of storing and reading a boolean value. For modules communicating delay insensitively, an event (transitions or pulses, whichever the system is based on) by itself carries no information. It is the association of an event with a ‘channel’ that carries this information. In other words, to communicate 1 bit of information, there needs to be at least two channels, each one representing one of

the two different states of a bit. Hence, at least one ‘write’ input, one ‘read’ input, and two ‘value-indicating’ outputs are necessary in a primitive implementing a bit of read/write storage. In addition, we need a separate event to acknowledge delay-insensitive completion of a write operation. This means at least 5 ports unless the ‘write acknowledge’ is multiplexed (merged) with one of the two value outputs. If this multiplexing is done, then it needs to be demultiplexed elsewhere to separate the acknowledge and the value signals in order to guarantee *Mem*’s specified behavior. But, this demultiplexing module will nominally need 5 I/O ports: a pair to set the ‘control state’ and to acknowledge it, the input signal to be demultiplexed, and the two demultiplexed outputs. If one tries to multiplex the two outputs with the acknowledge, it leads to a circularity. Hence, due to the finiteness of the network, the theorem holds by contradiction. \square

It, therefore, follows that the universal set $\{Fork, Merge, Mutex, 2 \times 1\text{-Join}, Mem\}$ is strongly minimal in that it is irredundant, it has the lowest I/O-modularity. The primitives are orthogonal in the sense that each one addresses a unique issue given in 2.2.

(For a proof why *Tria* is not redundant in $\{Fork, Merge, Mutex, Tria\}$, see [20].)

3 Switch-level design

[4, 8] show delay-sensitive implementations for a $2 \times 2\text{-Join}$ that are too inefficient in transistor counts and/or require insertion of several delay-elements for a safe operation. Here we show significantly area and time efficient and, yet, robust transistor-design schemes by examples. Figure 8(c) shows an implementation for one of the three symmetric cells (outputs) of a *Tria*. This can be adapted for other *Join* primitives ([20]). Figure 9 shows a novel implementation for the *Mem* module using combinational Muxes. We omit the implementations of initialization (reset) logic as well as of other primitives.

4 Summary and Conclusions

We have investigated a set of primitives and constructions to design purely delay-insensitive (DI) modules in a large class based on a fairly general model of delay-insensitive hardware. We have chosen our universal set of DI primitives such that their VLSI implementations are simple and their number is small.

In practice, we use several other building blocks directly implemented, or built from the primitive

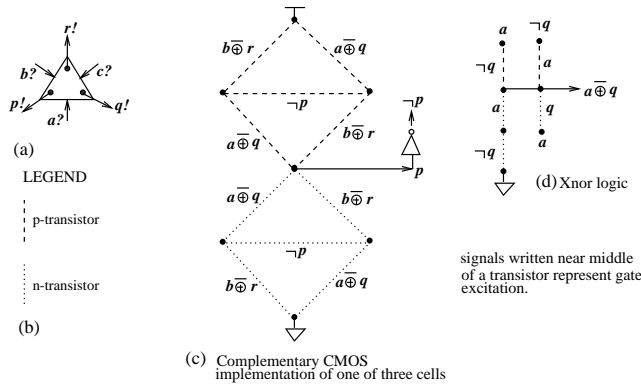


Figure 8: Implementation of a *Tria*

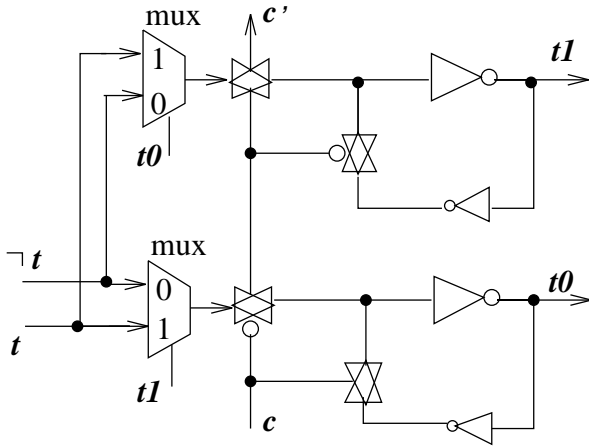


Figure 9: Implementation of a *Mem*

set if they facilitate the design in certain situations. Take for example the C-element, the *Toggle* and the 1×2 -*Join* which can all be realized from a *Tria* by appropriately hiding or reconnecting the module ports. But, we prefer to build them separately for efficiency. Note that a *Sequencer* is a sort of arbitrating 2×1 -*Join* and is too complex and slow to be considered a fundamental block. However, we find the *Sequencer* useful in some designs adopting 2-phase transition signalling. Similarly, 2×2 -*Join* serves well as a block for data processing circuits.

The fundamental issues of minimality and universality of DI primitives to implement DI specifications are addressed which answer several questions left open in [4]. Our primitives allow concurrency as well as avoid arbitration when only a deterministic choice is called for – unlike Keller’s.

Several new modules and constructions are introduced that are helpful in implementing larger modules. An interesting and energy efficient decompo-

sition of an arbitrarily large (two dimensional) *Join* module is shown. Moreover, realistic and efficient transistor implementations of some DI primitives are also given.

Acknowledgements

We thank the anonymous referees for their helpful comments.

References

- [1] I. E. Sutherland, “Micropipelines,” *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.
- [2] S. M. Burns and A. J. Martin, “Performance analysis and optimization of asynchronous circuits,” in *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference* (C. H. Séquin, ed.), pp. 71–86, MIT Press, 1991.
- [3] E. Brunvand, “A cell set for self-timed design using Actel FPGAs,” Tech. Rep. UUCS-91-013, Dept. of Comp. Science, Univ. of Utah, Salt Lake City, Aug. 1991.
- [4] R. M. Keller, “Towards a theory of universal speed-independent modules,” *IEEE Transactions on Computers*, vol. C-23, pp. 21–33, Jan. 1974.
- [5] J. T. Udding, *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.
- [6] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations, MIT Press, 1989.
- [7] M. B. Josephs and J. T. Udding, “An algebra for delay-insensitive circuits,” in *Proc. International Workshop on Computer Aided Verification* (R. P. Kurshan and E. M. Clarke, eds.), vol. 531 of *Lecture Notes in Computer Science*, pp. 343–352, Springer-Verlag, 1990.
- [8] J. C. Ebergen, *Translating Programs into Delay-Insensitive Circuits*, vol. 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.
- [9] S. Hauck, “Asynchronous design methodologies: An overview,” Tech. Rep. TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.
- [10] G. M. Brown, “Towards truly delay-insensitive circuit realizations of process algebras,” in *Proceedings of the Workshop on Designing Correct Circuits* (G. Jones and M. Sheeran, eds.), pp. 120–131, Springer-Verlag, 1990.
- [11] T.-A. Chu, *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

- [12] T. H.-Y. Meng, *Asynchronous Design for Digital Signal Processing Architectures*. PhD thesis, UC Berkely, 1988.
- [13] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli, "Solving the state assignment problem for signal transition graphs," in *Proc. ACM/IEEE Design Automation Conference*, pp. 568–572, IEEE Computer Society Press, June 1992.
- [14] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz, "The design of a high-performance cache controller: a case study in asynchronous synthesis," in *Proc. Hawaii International Conf. System Sciences*, vol. I, pp. 419–427, IEEE Computer Society Press, Jan. 1993.
- [15] C. G. Huang, C. G. Jesshope, and I. M. Nedelchev, "Systematic method for synthesising purely delay-insensitive circuits," *IEE Proceedings, Part E, Computers and Digital Techniques*, vol. 140, pp. 269–276, Sept. 1993.
- [16] J. L. A. van de Snepscheut, *Trace Theory and VLSI Design*, vol. 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [17] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [18] J. C. Ebergen, "A formal approach to designing delay-insensitive circuits," *Distributed Computing*, vol. 5, no. 3, pp. 107–119, 1991.
- [19] M. Josephs. Private communication, 1992.
- [20] P. Patra and D. Fussell, "Building-blocks for designing DI circuits," technical report tr93-23, Dept. of Computer Sciences, The Univ of Texas at Austin, Nov. 1993.

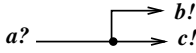
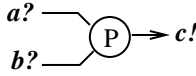
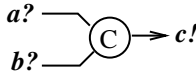
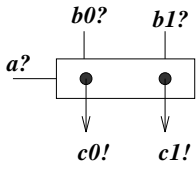
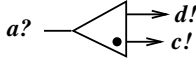
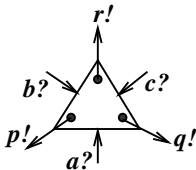
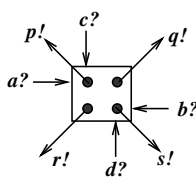
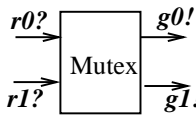
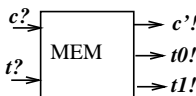
Name	Symbol	Specification
<i>Fork</i>		$\text{pref}(a?(b! c!))^*$
<i>Merge</i>		$\text{pref}((a? b?)c!)^*$
1×1 -Join		$\text{pref}((a? b?)c!)^*$
1×2 -Join		$\text{pref}(((a? b0?)c0!) ((a? b1?)c1!))^*$
<i>Toggle</i>		$\text{pref}(a?c!a?d!)^*$
<i>Tria</i>		$\text{pref}(((a? b?)p!) ((a? c?)q!) ((b? c?)r!))^*$
2×2 -Join		$\text{pref}(((a? c?)p!) ((a? d?)q!) ((b? c?)r!) ((b? d?)s!))^*$
<i>Mutex</i>		$\text{pref}(r0?g0!r0?g0!)^* (r1?g1!r1?g1!)^* ((g0!g0!) (g1!g1!))^*$
<i>Mem</i>		$\text{pref}((t?t0!)^*c?c'! (t?t1!)^*c?c'!)$

Table 1: A set of representative DI Primitives