

Optimization of Delay-Insensitive Circuits – a Case Study

Priyadarsan Patra
Donald Fussell

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
Email: {darshan, fussell}@cs.utexas.edu

abstract

We explore ways of constructing efficient delay-insensitive (DI) networks from a set of primitive DI elements by means of a case study. Several approaches that improve on recent designs of a modulo- N counter in the literature are illustrated. We obtain low constant latency, low constant response time, constant power consumption and optimal area-complexity designs for this circuit. For moderately large N , the area complexity compares well even with standard designs under synchronous (clocked) discipline. Many of these efficiencies derive from the exploitation of the powerful property of timing-independent composability of DI circuits.

1 Introduction

The past few decades have seen periods of waxing and waning interest in the design of **asynchronous circuits**, which have no global clock synchronizing the operation of various portions of the circuit. In the absence of the temporal simplicity obtained from discretization of time by a global clock, it can be much more difficult to deal with the correct synchronization of events in the circuit. Designing correct asynchronous circuits thus requires a much more rigorous approach to timing issues than has been the case for synchronous systems. As a result, asynchronous designs have rarely been used in practice.

However, interest in asynchronous circuits has been maintained in spite of these difficulties because of the potential advantages of such circuits over synchronous circuits ([Sut89, BM91, Bru91]). These advantages include potentially higher power efficiency, higher speeds, and greater resilience to metastability, temperature variations, and technological changes such as scaling of integrated circuit feature sizes. It is clear that a great deal of power can be saved by eliminating a global clock which causes transitions at every storage element in a circuit every time a clock transition occurs. However, without satisfactory theoretical or practical demonstrations of alternative asynchronous circuits, it remains debatable whether or not the area overhead of asynchronous circuit designs counterbalances this potential power savings. In fact, some researchers believe that asynchronous circuits consume higher energy on the average because asynchronous datapaths often use dual-rail and 4-phase handshaking techniques.

Moreover, there is still much contention about the relative area efficiency of synchronous and asynchronous circuits. In [Mar92] a “delay-insensitive” parallel adder design using a 4-phase protocol is described which is claimed to compete well with its synchronous cousins in area efficiency.

However, 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.

[Kel74, Udd84] have formalized classes of delay-insensitive circuits using trace theory by defining closure properties of trace structures representing *delay-insensitive (DI) computations*, but their work did not deal with the synthesis of efficient circuits using these elements. [Dil89] and [JU90] have developed composition operators and algebras to model speed independence and to verify equivalence of DI specifications, respectively. Again, however, performance and implementation issues of the circuit modules have not been addressed.

[Ebe89] has developed grammars to specify many DI circuits that induce a syntax-directed translation into an abstract set of DI primitives. [Bro90] compiles process algebras into primitives that have a straightforward mapping to software constructs. These methods assume a one-to-one correspondence of language constructs and hardware modules, and essentially serve to specify the interconnection of predefined modules to achieve a function described in software. As yet, they provide no support for dealing with efficiency or optimization issues, however. Several other synthesis techniques [Chu87, Men88, LMBSV92, NDDH93] have been devised to be more easily realized in practice by imposing several restrictions on the types of specifications allowed. These restrictions are unrelated to requirements for achieving delay-insensitivity or speed-independence, and the methods proposed provide little or no general capability for *composing* and *decomposing* DI modules, much less for achieving efficient compositions or decompositions.

While an automatic compiler described in [Bru91] applies area optimization techniques to circuits based on the ‘bundled-data’ scheme, to date there has been little or no work in identifying general principles for improving speed, area and energy performance of a circuit at the module interconnection level or for increasing efficiency at the circuit primitive level, beyond some ‘peephole’ area optimization techniques.

We believe that *delay-insensitive (DI)* circuits are potentially the most desirable asynchronous circuits, given the increased availability of transistors on chip, in spite of the likelihood that they require more area to implement than synchronous or even other forms of 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. We also believe that the use of formal specification techniques for designing asynchronous circuits is imperative the design of large-scale asynchronous systems practical. DI circuits are well-suited to formal specification due to the properties cited above. Finally, we believe that the most flexible basis for DI circuit design should be used, since imposition of arbitrary restrictions on the design style for the purposes of simplifying a particular refinement process are likely to lead to inefficiencies in the circuits designed.

There is reason to be concerned, however, that DI circuits require prohibitively large overhead, and perhaps as a corollary require more power and provide lower performance than other types of asynchronous systems. We aim to mitigate this concern by demonstrating that the area overhead can be kept surprisingly small and that impressive performance gains and power savings can be achieved if proper design techniques are used. In this paper, we consider different ways to design and optimize a delay-insensitive modulo- N counter as a case study to several discover principles of design optimization. We show circuit implementations that are comparable in area with their best synchronous counterparts, while at the same time providing higher speed and lower power requirements than competing designs. While this hardly constitutes a general approach to opti-

mizing area, power, and speed for DI circuits, it does serve as an indication that surprisingly good results can be achieved simultaneously in all three dimensions if proper design optimizations are employed.

2 A formal model and a language of specification

The first step in the 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** ([Udd84, vdS85, Hoa85]) for exposition of our methods and for describing our primitive modules.

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 sequential¹ composition, ‘||’ 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.

2.1 Primitive modules and notation

We need a repertoire of atomic modules or **primitives** to build general modules. The subset used in this paper is given in Table 1. Each primitive’s specification is given next to it.

$m \times n$ -*Join* is a useful module 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 & 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.

Example:

We illustrate trace-theory notation using, as an example, a 1×2 -*Join*, whose set of traces is given by: **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 , $a b0$, $a b0 c0$, $a b0 c0 b0$, $b0 a c0$, $b1 a c1 a b1 c1$, $a b1 c1 b0 a c0$.

¹Very 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 [Ebe91].

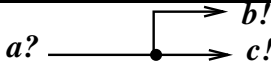
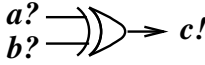
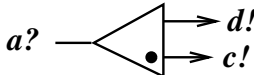
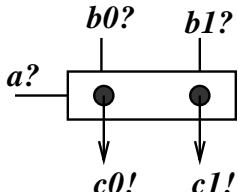
Name	Schematic	Specification
<i>Fork</i>		$\text{pref}(a?(b! c!))^*$
<i>Merge</i>		$\text{pref}((a? b?)c!))^*$
<i>Toggle</i>		$\text{pref}(a?c!a?d!))^*$
$1 \times 2\text{-Join}$		$\text{pref}(((a? b0?)c0!) ((a? b1?)c1!)))^*$

Table 1: A Set of DI Primitives used here

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.

Note that any *extension* of an invalid trace by concatenating symbols on the right is also an invalid trace.

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.

2.1.1 Additional conventions

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. 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 (parity) tree of *Merges*, in our figures.

A design of a module is its *realization* as a DI network ([Ebe89]) of primitives. Response time of a module M is denoted by $\mathcal{T}(M)$. It is the longest time the module’s environment must wait to receive appropriate response signal(s) after supplying a set of necessary input signals to the module. $\mathcal{A}(M)$ denotes the area of a module’s implementation, defined in terms of the constant areas used for the primitives. Similarly $\mathcal{P}(M)$ gives energy consumption of a module in terms of transitions at the inputs and outputs of the primitives making the module. For first order analysis, $\mathcal{T}()$ ignores delays in interconnects between primitives, $\mathcal{A}()$ ignores interconnect area to connect primitives,

and $\mathcal{P}()$ considers only switching events at the primitives' inputs & outputs while ignoring power dissipation due to short-circuit and leakage currents.

3 Specifying a mod- N counter

Subcircuits in a circuit need to transfer information among themselves and this often requires some form of synchronization. If needed, data as well as control information may be transferred within a synchronization event. For conventional clocked circuits, a 'global' clock serves to synchronize, while the asynchronous communication requires some form of **handshake** for synchronization among the communicating modules. Therefore, it is sometimes convenient, though not necessary, to think of circuits as communicating along **handshake channels** [vB92] as a way of 'modularizing' the interface, i.e. a logical communication link is encapsulated by defining hardware components that control/realize it. But, we will not insist on "channels" for internal communication to avoid inefficiency. Accordingly, we will first attempt to specify our mod- N counter as having two channels a and b and which behaves thus:

It repeats the following behavior indefinitely: it participates in N sequential synchronizations along channel a followed by one along b .

An application of such a circuit is where a task needs to be done once every N executions of some other task or every N clock 'ticks.' When the tasks are simple synchronizations, the circuit is called a modulo counter.

Separating a and b into two different handshake channels makes it possible for two physically independent modules to interact through the 'services' of the counter, provided via the two channels. Such separation of control leads to very efficient, structured and modular 'pipelined' designs in our methodology. Hence, our first trace-theoretic behavioral specification of the counter is $S0 = \text{pref}((a^N; b)^*)$.

3.1 Refinement based on choice of handshake protocol

Under the requirement of "delay-insensitivity" a transition on a port must be acknowledged, before another can follow, to avoid **transmission interference** [Udd84]. This makes a protocol for 'handshake' necessary for synchronization. Each input need not be individually acknowledged, but in our case we have two independent interfaces and hence, two independent handshake protocols. Usually two physical wires, logically directed opposite to each other, are used to implement such a channel. As a matter of *our* convention, a channel-name subscripted with a 0 is the name for the *Wire* that undergoes the first transition in a handshake cycle (i.e. the REQUEST line), the other *Wire* (the ACK line), which undergoes the second transition, is subscripted by 1.

We choose a **2-phase** protocol for its simplicity and economy – compared to a 4-phase protocol which is intuitively likely to consume twice as much power and to take twice as much time because of 'reset' transitions – for our first implementation. Later we will demonstrate ample evidence to justify this 'suspicion'. In 2-phase handshaking, a **request** signal from one communication partner is followed by an **acknowledge** signal from the other to complete a synchronization/handshake on a channel. Such a communication involves no transfer of data, but just serves to synchronize the partners.

Having decided on a handshake protocol, we have broken down each asynchronous communication (or, synchronization for our mod- N counter) into smaller atomic events (signal transitions at input or output ports). The logical next step is to decide: (1) What, circuit or environment, initiates a communication ? (2) When is a synchronization attempt deemed 'committed' from circuit's

standpoint ?

Our answer to question (1) is “it can be arbitrarily ‘agreed upon’ between the circuit’s implementor and circuit’s environment.” As a rule of thumb, we suggest that the circuit (environment) should initiate a synchronization if the corresponding channel is an input (output) of the circuit. The answer to question (2) is “when the circuit *receives a handshake* signal from the environment.” That is, the circuit is assured of a synchronization only upon receiving an input signal in accordance with an agreed-upon protocol as in answer (1).

Hence, assuming a 2-phase handshake a straightforward refinement of $S0$ at the level of transitions on connection wires is:

$S1 = \mathbf{pref}((a_0? a_1!)^N b_0! b_1?)^*$ where, without loss of generality, we assume that the environment sends requests on channel a while the counter is the ‘requester’ for b .

But, $S1$ is not a *delay-insensitive specification* as may be checked against Udding’s *conditions* of delay-insensitivity [Udd84], because the two input signals $b_1?$ and $a_0?$ are allowed only in one sequential order. (Informally, because of possible arbitrary delays in the interface it cannot be guaranteed that $b_1?$ is received before the next $a_0?$ if they are causally unrelated.) Considering our answer to question (2) above, and noting the need to preserve the original synchronization semantics of the counter’s specification, we further refine to get:

$S2 = \mathbf{pref}((a_0? a_1!)^{N-1} a_0? b_0! b_1? a_1!)^*$ where a suitable order between $b_1?$ and $a_0?$ is enforced by “enveloping” a handshake on b by the N^{th} handshake on a . At this point, we have precisely stated the DI interface behavior of the circuit and its environment.

4 Design Decompositions

4.1 Design 1

From the theory of algorithm design, we know that **recursion-induction** serves as a powerful tool to solve/analyze a problem indexed by an arbitrary integer. We will apply the same tool in order to decompose this family of state-machines (Mod- N counter) into time-, area-, switching energy-wise economical circuits. In the following, we will recursively apply the well-known technique of **Divide & Conquer** to reduce area complexity of a module. First, we attempt to design a “counter-element” (CE) that can control and reuse a smaller subcounter to realize a bigger one.

The counter in Figure 1(a) counts modulo $2R+1$. The CE — all the logic except the boxed subcounter — distributes $a?$ inputs to the Mod- R subcounter. After the mod- R counter finishes counting up to R — indicated by its request output $m!$ — the CE acknowledges back on m and gets ready to start the second counting cycle for the subcounter. At the end of this second cycle, the mod- $2R+1$ is ready to finish its first cycle by outputting $b!$. The whole process is then ready to repeat itself.

The design above can be modified easily for N even — see Figure 1(b) — by making each, not alternate, transition $m!$ of the subcounter synchronize (and hence ‘count’) with $a?$ at the 2×1 -*Join*. Since we are describing the design recursively, we provide the base-case designs of counters for $N = 2$ and $N = 1$ in Figure 1(c).

Area complexity of both the even and odd counter designs is given by $\mathcal{A}(\text{mod-}N \text{ counter}) = \mathcal{A}(\text{Toggle}) + \mathcal{A}(2 \times 1\text{-Join}) + 2 \times \mathcal{A}(\text{Merge}) + \mathcal{A}(\text{mod-}\lfloor N/2 \rfloor \text{ counter})$, where the first three terms on right of the equality are constants. Therefore, the area complexity is $\mathcal{O}(\log N)$.

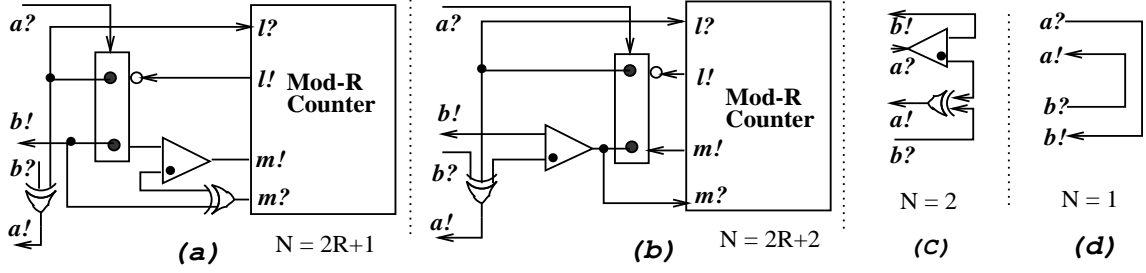


Figure 1: A decomposition of the mod- N counter

4.1.1 Response-time calculations

We will be concerned with both Longest Response Time (LRT) as well as Amortized Response Time (ART). A circuit performs an *operation* to compute a response (set of outputs) to a set of supplied inputs. The difference between the time when the last of the required inputs was supplied and the time when the last of the corresponding result outputs is made available is called the *response time* of the operation involved. LRT refers to the worst-case response time of an operation. ART is the average response time of an operation, in the worst-case [CLR92]. We also define SRT as the Shortest Response Time for an operation. Moreover, we will ignore delays in wires and *Forks* interconnecting primitive modules realizing a circuit in our calculations. This is so because (1) Our designs are linear arrays of counter-elements, and wire delay between adjacent elements can be assumed to be a small constant, (2) DI decomposition ideally does not put any functional requirement on the interconnection delays depend mostly on placement and routing, (3) this simplifies our first order analysis and comparison among designs.

The circuit can be abstractly seen as $k = \lceil \log N \rceil$ communicating and cooperating processes as described below. The synchronization between two adjacent circuit processes are done by a 2×1 -Join which forms a ‘boundary’ between them. A process gets a bit of data from the right and passes it to the left as a $a!$ or $b!$. The delay range $[d-D]$ defines the minimum (d) and maximum (D) time it takes for a data signal to be available to the left neighbor process, as that signal passes through different sequences of primitive modules in different iterations of the loop. These delays are assumed to include the propagation delay of the synchronizing 2×1 -Join as well.

Note: a *CE* does not exactly correspond to a process modeled below – all logic starting from and including a 2×1 -Join up to, but excluding, the immediate left 2×1 -Join – in the ‘unfolded’ figure – is modeled as a process below.

<u>$Proc_i$</u>	<i>/* 0 < i < k */</i>	<u>$Proc_k$</u>	<i>/* Rightmost CE */</i>
Loop		Loop	
Synchronize With $proc_{i-1}$;		Synchronize With $proc_{k-1}$;	
Message and Pass Data Left; <i>/* delay range [d-D] */</i>		Pass Data Left; <i>/* delay range [d-D] */</i>	
Synchronize to GetDataFromRight $proc_{i+1}$;		End.	
End.			
<u>$Proc_0$</u>	<i>/* Environment */</i>		
Loop			
Synchronize Right With $proc_1$;			
Consume Data From Right; <i>/* arbitrary delay */</i>			
End.			

Lemma 1: The ART of the counters in Fig 1 is $\mathcal{O}(1)$.

Proof: Suppose each massaging step in each process is fixed at the maximum over all the iterations in all the processes. This value, D , is clearly a constant. Also suppose that the environment process $proc_0$ has no delay of its own. Now, it is easy to see that it takes $N \times D$ units of time for the $Proc_0$ to iterate N times – all the processes $Proc_i, i > 0$ are perfectly synchronized. Thus, the ART of the counter in this case is $\mathcal{O}(N \times D \div N)$ or $\mathcal{O}(1)$. If all the delays were not at their maximum value, or if $proc_0$ has a positive delay, the response time of the counter can only improve. Hence, the ART remains bounded above by $\mathcal{O}(1)$. \square

Lemma 2: Assuming $proc_0$ (environment) has only a constant delay, the LRT of this counter is $\Theta((D - d) \times \log N)$ if $D > d$.

In spite of the lemma above, the parallelism among the processes ensures that a slow enough $Proc_0$ will perceive the counter to have only a constant LRT. On the other hand, LRT of a ‘ripple’ counter, discussed later, always is $\mathcal{O}(\log N)$.

4.1.2 Power Consumption

Power consumption is the amortized average switching energy used per communication in a sequence of communications between a module and its environment, due to signal transitions at the I/O ports of the primitives realizing a module. This definition has a beneficial characteristic in that the energy efficiency of a circuit is necessarily tied down to its interface specification, making comparisons between implementations (their internal energy consumption) sensible.

From the process description of the counter circuit, it is clear that each synchronization of a process with its left neighbor leads to one synchronization with the right neighbor. Hence, if the environment process synchronizes with $proc_1$ N times, then each of the $\lceil \log N \rceil$ process synchronizes N times, giving us $\Theta(N \times \log N)$ switching transitions, and hence the power consumed is $\Theta(N \times \log N \div N) = \Theta(\log N)$.

Being unhappy with these power and response-time results, we explore further.

4.2 Design 2

If each counter process can service more requests from the left without each time obtaining a data from the right, there would be less frequent demand for data from the right. The processes described below allow more processing time and reduce synchronization overhead compared to the previous attempt. Communication, and hence synchronization, between hardware processes is essential for parallel processing, and therefore, for faster response and smaller area. Synchronization must be judiciously limited for overall performance.

```

Proci                                /* 0 < i < k */
Loop
  Synchronize With proci-1;
  Message and Pass Data Left; /* delay range [d-D] */
  Synchronize With proci-1;
  Create Next Data, Pass Left; /* delay range [d-D] */
  Synchronize to GetDataFromRight proci+1;
End.

```

Lemma 3: LRT of this counter is $\mathcal{O}(1)$ if $D < 2d$, irrespective of the delay in $proc_0$.

Happily for a counter, a process can encode what data (a $a!$ or a $b!$) to pass left in certain states. For example, a process for N even, may pass $a!$ upon every other request from the left,

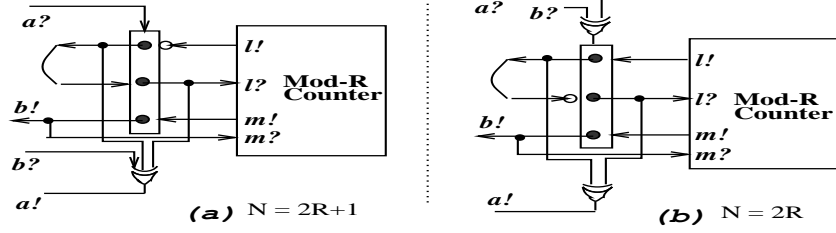


Figure 2: Reducing synchronization

while choosing between $a!$ and $b!$ based on data from the right for the rest of the requests. This idea leads to a design as shown in Figure 2(a & b) for $N = 2r+1$ and $N = 2R$, respectively.

4.2.1 Analysis

The area measure of this design is seen to be about $(\mathcal{A}(3 \times 1\text{-Join}) + 2\mathcal{A}(\text{Merge})) \times \log N$.

Let SD (synchronization delay) be the time lag between an $a?$ occurrence and the corresponding enabling of $3 \times 1\text{-Join}$. Let $\text{SRT}(a?)$ represent the shortest time the requestor on the left can generate $a?$ after receiving response to a previous request, if any. Thus we have,

$$0 \leq \text{SD} \leq \max(0, (\text{LRT}(\text{mod-}R \text{ counter}) - \text{SRT}(a?)))$$

$$\text{LRT}(\text{Mod-}2R+1 \text{ counter})$$

$$\leq$$

$$\begin{aligned} \max(\text{SD} + \text{LRT}(3\text{-input Merge}) + \text{LRT}(3 \times 1\text{-Join}), & /* \text{delay from } a? \text{ to the subsequent } a! */ \\ \text{LRT}(3\text{-input Merge}), & /* \text{delay from } b? \text{ to the subsequent } a! */ \\ \text{SD} + \text{LRT}(3 \times 1\text{-Join})) & /* \text{delay from } a? \text{ to the subsequent } b! */ \end{aligned}$$

Note that if the left process is a CE , $\text{SRT}(a?) \geq 2 \times \text{LRT}(3 \times 1\text{-Join}) + \text{LRT}(3\text{-input Merge})$, hence, SD is zero. Therefore, it follows that $\text{LRT}(\text{Mod-}2R+1 \text{ counter})$ is at most $\text{LRT}(3\text{-input Merge}) + \text{LRT}(3 \times 1\text{-Join})$, which is a constant.

Each counter process described for this design communicates with its right neighbor only half the time it receives a request from the left. The switching action corresponding to one iteration of a loop is clearly constant — about one transition each through the $3 \times 1\text{-Join}$ and the Merge , one response transition to left, and at most one request transition to right. Therefore, the energy consumption for the communication cycle $a^N b$ is

$$E_N = \mathcal{O}\left(\sum_{i=0}^n (N \div 2^i)\right)$$

where $n = \lfloor \log N \rfloor$. Since there are $\mathcal{O}(N)$ external communications with the environment, the power consumption in the circuit is given by $\mathcal{P}(\text{counter}) = E_N / \mathcal{O}(N) = \mathcal{O}(1)$.

4.3 Simplifying the $3 \times 1\text{-Join}$

A $3 \times 1\text{-Join}$ can be decomposed into our basic primitives as shown in Figure 3(a). Figure 3(b) shows the mod- $2R$ counter with this decomposition. Figure 3(c) is arrived at by observing that $l?$ is followed by a $l!$ or a $m!$ from the subcounter, and also that $a?$ is followed by $a!$ or $b!$. A nice property of this optimization is that the top $2 \times 1\text{-Join}$ in Figure 3(c) is the simpler *Toggle* device.

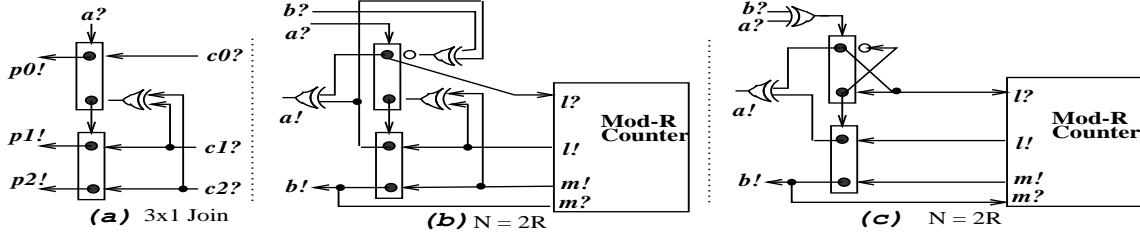


Figure 3: Counter with 3×1 -Join decomposed

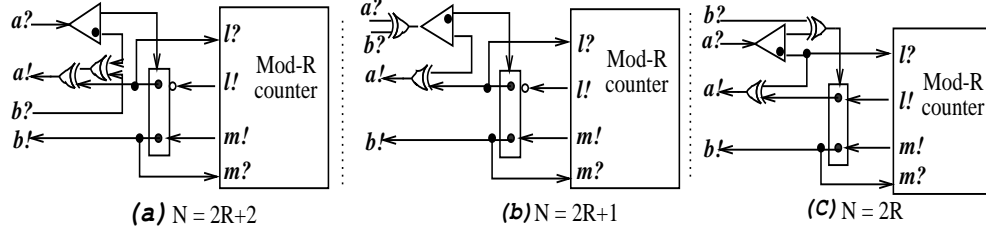


Figure 4: Energy-efficient, and fast mod- N Counter

5 One More Step

Usually a circuit should generate a request as soon as possible and permissible so that the receiving circuit gets more time to process it. This strategy means generating an $l?$ as soon as possible for the right subcounter. But, this would mean decomposing the 3×1 -Joins in Figure 2 in such a way that the row input from the subcounter on the right does not interfere with the self-generated signal competing concurrently to synchronize with the request signal $a?$ from the left. The necessary manipulation results in designs shown in Figure 4, which makes use of the decomposition of 3×1 -Join as in Figure 3.

5.1 Analysis

Below, we calculate the upper bound on the response time of the mod- N counter of Figure 4(a). Let CD be the delay between when the 2×1 -Join generates an output and when the next column input is available.

$$0 \leq SD \leq \max(0, (\text{LRT}(\text{mod-}R \text{ counter}) - \text{CD}))$$

$$\text{LRT}(\text{Mod-}2R+2 \text{ counter})$$

$$\leq$$

$$\max(\text{LRT}(\text{Toggle}) + 2 \times \text{LRT}(\text{Merge}),$$

$$\text{LRT}(\text{Toggle}) + SD + \text{LRT}(2 \times 1\text{-Join}) + \text{LRT}(\text{Merge}),$$

$$2 \times \text{LRT}(\text{Merge}),$$

$$SD + \text{LRT}(2 \times 1\text{-Join}))$$

Note that if the left process is a CE , $\text{CD} \geq 2 \times \text{LRT}(2 \times 1\text{-Join}) + 2 \times \text{LRT}(\text{Merge}) + 2 \times \text{LRT}(\text{Toggle})$. Hence, SD is zero and the penultimate inequality above is satisfied if $\text{LRT}(\text{Mod-}$

$2R+2$ counter) = $\text{LRT}(\textit{Toggle}) + \text{LRT}(2 \times 1\text{-Join}) + \text{LRT}(\textit{Merge})$, which is a constant.

The area and energy requirement (and their analysis) for this circuit remain essentially the same as in the previous section, which are $\mathcal{O}(\log N)$ and $\mathcal{O}(1)$, respectively. Same conclusions are obtained for designs in Figure 4(b,c), but the details are omitted.

Before examining the very interesting property that CD is twice the LRT of the counter, we establish a method for designing ‘ripple’ counters.

Remark: In the abstract, a positive natural number N can be thought of as a product of smaller positive factors, as addition of summands, or as various combinations of both. If counters designed for a particularly suitable set of factors of N are composed together to form a linear asynchronous pipeline, we will have implemented a mod- N counter, with accompanying area and energy efficiency.

When a suitable set of factors cannot be found, a composition of a counter-element (CE) with a subcounter may be attempted. During one cycle of such a mod- N counter, the CE consumes and counts up to L $a?$ inputs, then it keeps passing any further $a?$ input along to a mod- $(N - L)$ subcounter to its right, for some $L, L < N$. The CE resets and the whole process starts when the subcounter to the right finishes its subcycle. Here countings of the two modules are added as opposed to multiplied when you compose them. (End of Remark)

6 Ripple Counters

Let $N = 2^k$ be expressed as a product of k 2’s. A subcounter for each of the factors namely, 2, can be composed to get a “ripple” counter as suggested in Figure 5(a). The important primitive used is a *Toggle* which we have just visited. Note that there is no $b?$ signal defined for this. The LRT is $\mathcal{O}(\log N)$ as is the area. Power consumption is constant. For arbitrary N , consider the following scheme to implement a delay-insensitive ripple counter:

Express N as a binary number \mathcal{B} such that $2^{|\mathcal{B}|} \geq N$. Let $\overline{\mathcal{B}}$ be the 2’s complement of \mathcal{B} . Form a cascade of $|\mathcal{B}|$ *Toggles* such that the non-initial terminal of a *Toggle* feeds the input of the next *Toggle* to the right. This is suggested in Figure 5(a). The non-initial terminal of the last *Toggle* is the $b!$ output port. The first (left most) *Toggle* is designated 0 (1) if the left most bit of $\overline{\mathcal{B}}$ is 0 (1), the second *Toggle* is designated 0 (1) if the second left-most bit of $\overline{\mathcal{B}}$ is 0 (1), and so on. All the initial – the ones marked with a heavy dot – terminals of the *Toggles* designated 1 are merged together with the $a?$ input and fed to the left-most *Toggle*. The rest of the initial terminals are merged with the $b?$ input by a parity tree to form the $a!$ output port.²

For example, let $N = 10$. The 2’s complement of decimal 10 in binary is $\overline{\mathcal{B}} = 0110$. See the decimal ripple counter in Figure 5(b). The middle two bits of $\overline{\mathcal{B}}$ are 1. Hence, the initial terminals of the middle two *Toggles* are merged with $a?$ into a parity tree, etc.

A proof-sketch of the scheme’s correctness follows.

Proof Sketch: A cascade of $|\mathcal{B}|$ *Toggles* as per the scheme in Figure 5(a) gives a mod- $2^{|\mathcal{B}|}$ counter. $\overline{\mathcal{B}}$ gives the number of false $a?$ transitions to be generated ‘endogenously’ so that each power-of-2 cycle of the underlying $2^{|\mathcal{B}|}$ -counter is truncated to N . The k^{th} *Toggle* from the right generates 2^k transitions at its initial terminal in a cycle of the underlying power-of-2 counter. The cascade generates an $a!$ for each $a?$ received, except for the following proviso: A one in bit position k of $\overline{\mathcal{B}}$ from the right means the initial-terminal of the k^{th} right *Toggle* feeds its transition as an endogenous input to the cascade of *Toggles*, instead of producing an $a!$ output.

²The parity trees can be configured to reduce either ART or LRT. ART is a constant if the parity tree of 2-input *Merges* is configured one way, and $\mathcal{O}(\log N)$ if configured another way.

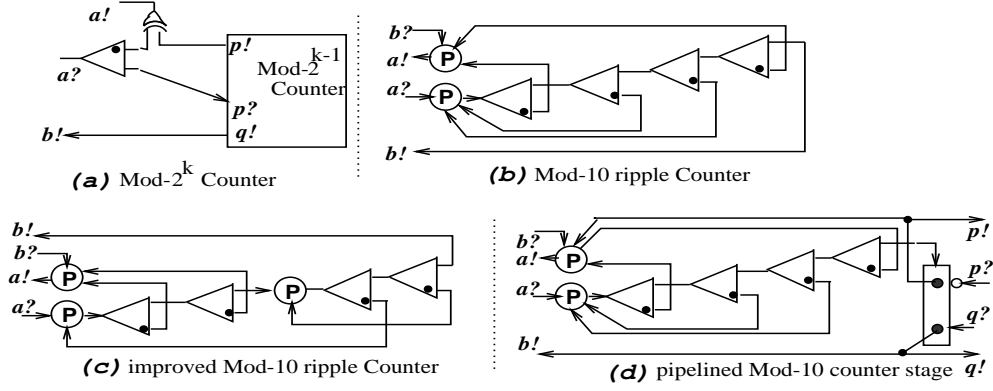


Figure 5: Ripple counters and ripple stages

There is no ‘interference’ to void delay-insensitivity as this whole counter is a *serial* module — each external input generates a single sequence of internal transitions leading to an output signal when it’s ready to accept another input as per specification.
(End of Proof Sketch)

In Figure 5(c), see a design that ‘skips useless states’ of the underlying mod-16 counter more cleverly to count mod-10. This scheme can also be described algorithmically for any N — here some initial terminals may be merged and fed to an intermediate *Toggle*, in stead of the left most.

A surprising use of a ripple counter is provided next.

7 Design Composability – Trading Unused Time

We notice that CD for a *CE* nearly doubles from one element to the immediately next on the right because, the element generates one handshake to the right for every pair of handshakes it does on the left. Therefore, as evident from the response-time calculations in section 5, a *CE* can ‘tolerate’ a twice as slow subcounter on its right. So, we take the approach of composing together counter ‘stages’ (corresponding to factors in an expression for N , as remarked previously), such that the LRTs of adjacent stages are well-matched. A stage is formed by means of a 2×1 -*Join* at the right end to interface with another subcounter. For example, figure 5(d) shows how to build a counter stage from an ordinary mod-10 ripple counter by using a 2×1 -*Join* as a buffer between stages. Buffers of this type give a pipelined structure to an implementation. This way, the considerable area efficiency of ripple counters is brought to bear without sacrificing the response-time of the overall mod- N counter. All this works because any delay-insensitive component/module in a circuit can be replaced by another with the same behavioral specification without causing **delay faults**, while large-scale timing-analysis and redesign is usually needed in a synchronous discipline to allow safe replacements of ‘parts’.

We get a surprisingly area-efficient, fast counter by using intermediate ripple counters stages as follows: A counter stage capable of counting upto $2m$ can be composed to its right with a ripple counter that can count upto 2^{m_i} without slowing down its left neighbor. This is so because the ripple counter is allowed a LRT of m times the constant LRT of the overall counter, without affecting LRT of the left-most counter stage (which is perhaps just a *CE* Therefore, the number of counter stages, and hence, the number of 2×1 -*Joins* in the overall mod- N counter is only $\log^* N$ *asymptotically* (while the longest response time, or LRT, continues to be a constant).³

³The iterated logarithm function \log^* , a very slowly growing function, is defined as $\log^* N = \min\{i \geq 0 : \log^{(i)} N \leq$

8 Comparison with Designs in Literature

From the discussion above, it follows that the total area needed for our final mod- N counter design is about $\log N \times (\mathcal{A}(\textit{Toggle}) + \mathcal{A}(\textit{Merge})\mathcal{A}(2 \times 1\textit{-Join}) \times \log^* N$. This is about $(18 + 6) \times \log N + 32 \times \log^* N$ transistors in total, if we use the transistor designs for the primitives given in section 9. The last term is virtually a constant for large N . Compare this with the synchronous clocked counter suggested in Figure 8.26 of [WE88]. Even when an adder circuit is used to implement a synchronous clocked modulo counter, our design certainly compares well in area. But, the response time of our DI counter does not increase with increasing N , unlike in the synchronous designs. The clock rate for a synchronous counter must be slowed down sufficiently to allow any rippling from least significant bit to the most. Moreover, the clocked synchronous counters cause transitions at clock inputs of all the $\mathcal{O}(\log N)$ flip-flops for every clock tick giving $\mathcal{O}(\log N)$ power dissipation, while for a DI counter the amortized number of signal transitions is only $\mathcal{O}(1)$.

[vB93] shows a design using 4-phase protocol obtained starting from a CSP-based language, Tangram. Optimized implementation therein uses about $26 + 120\lceil \log N \rceil$ transistors. The latency in that implementation is $\mathcal{O}(\log N)$, but, the response time afterwards is a constant. The power consumption is $\mathcal{O}(1)$ per external communication, as in our design. Our designs considerably improve upon [vB93] by reducing the initial response time to a constant simultaneously maintaining all other complexity bounds while making them even tighter by multiplicative constants — e.g., our 2-phase signalling reduces the time to count N by a factor of 2, compared to 4-phase communication.

The independently developed circuit in Figure 3 is similar to the best decomposition for a mod- N counter given in [EP92], which does not discuss the power consumption issue. Nonetheless, our final design gives a much better figure even for area as discussed above.⁴

9 Switch-level Implementation

In the following we show area and time efficient and, yet, robust transistor-design schemes by examples. Figure 6(a, b) shows an implementation for ‘half’ of a $1 \times 2\textit{-Join}$. Figure 6(a) is a 5-transistor *Merge* gate used to derive the internal *Xor* signals. The other half of the $1 \times 2\textit{-Join}$ can be implemented similarly with the p and q signals interchanged in Figure 6(a,b). This scheme can be adapted for other *Join* primitives ([PF93]). Figure 6(c) implements a *Toggle* using pass gates. All reset logic is omitted from the figures to avoid clutter.

The HSPICE simulation of the $1 \times 2\textit{-Join}$ extracted from its unoptimized layout using MAGIC shows a propagation delay of 1.6ns in a 2-micron MOSIS SCMOS process assuming a load of 8 minimum sized inverters at each output. Minimum-size allowable N transistors with matching P transistors were used. The delay for the *Toggle* is about 1ns under the same conditions.

10 Concluding Remarks

We have shown the power of a few simple delay-insensitive primitives and illustrated several circuit design optimizations through manipulation of the primitives’ initial states and composition structures to build robust asynchronous circuits. The few principles of optimization and design methods illustrated here in have lead to many improved circuit properties such as area, response times,

¹}

⁴[EP92] (page 44) suggests that counters for arbitrary N using decompositions shown in their figures 5 and 6(a) yield constant response time. But, we suspect that for N defined as $N = f^k(3)$, where $f(i) = 2i + 1$, and f^k is f applied k times, the decomposition 6(a) will be chosen repetitively. This will lead to a $\mathcal{O}(\log N)$ response time.

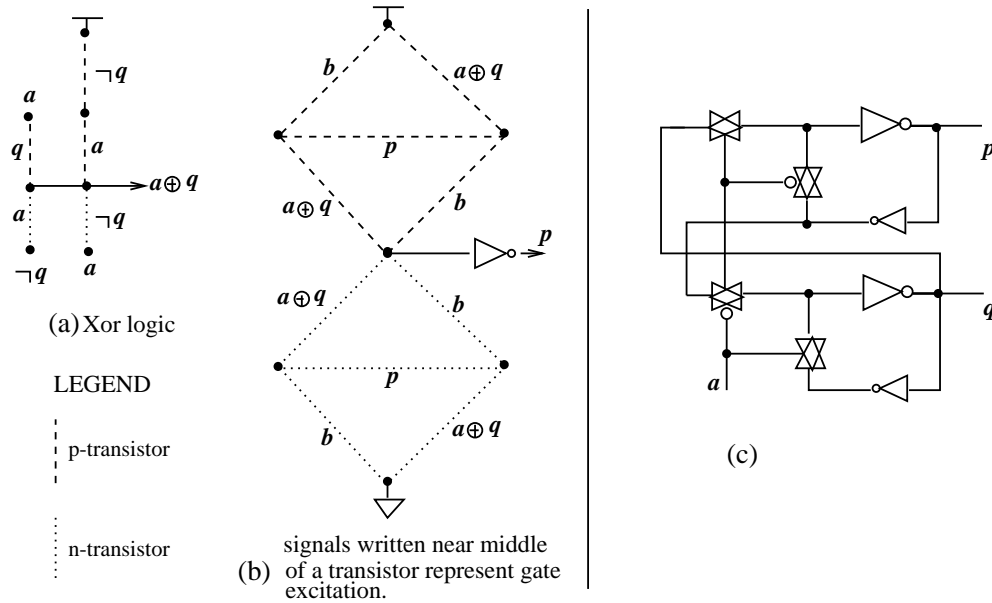


Figure 6: Implementations of 1×2 -Join and Toggle

latency, and power consumption of a modulo counter. The resulting circuit designs are shown to be significantly better in many respects than existing synchronous and delay-insensitive designs in literature. Switch-level designs for the primitives used are also provided.

References

- [BM91] Steven M. Burns and Alain J. Martin. Performance analysis and optimization of asynchronous circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 71–86. MIT Press, 1991.
- [Bro90] Geoffrey M. Brown. Towards truly delay-insensitive circuit realizations of process algebras. In Geraint Jones and Mary Sheeran, editors, *Proceedings of the Workshop on Designing Correct Circuits*, pages 120–131. Springer-Verlag, 1990.
- [Bru91] Erik Brunvand. A cell set for self-timed design using Actel FPGAs. Technical Report UUCS-91-013, Dept. of Comp. Science, Univ. of Utah, Salt Lake City, August 1991.
- [Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.
- [CLR92] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1992.
- [Dil89] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.
- [Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.
- [Ebe91] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.
- [EP92] Jo C. Ebergen and Ad M. G. Peeters. Modulo-N counters: Design and analysis of delay-insensitive circuits. In Jørgen Staunstrup and Robin Sharp, editors, *Designing Correct Circuits*, volume A-5 of *IFIP Transactions*, pages 27–46. Elsevier Science Publishers, 1992.

- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [JU90] Mark B. Josephs and Jan Tijmen Udding. An algebra for delay-insensitive circuits. In Robert P. Kurshan and Edmund M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, 1990.
- [Kel74] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, January 1974.
- [LMBSV92] 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*, pages 568–572. IEEE Computer Society Press, June 1992.
- [Mar92] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.
- [Men88] Teresa H.-Y. Meng. *Asynchronous Design for Digital Signal Processing Architectures*. PhD thesis, UC Berkely, 1988.
- [NDDH93] Steven M. Nowick, Mark E. Dean, David L. Dill, and Mark Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.
- [PF93] P. Patra and D.S. Fussell. Building-blocks for designing DI circuits. Technical report tr93-23, Dept. of Computer Sciences, Univ of Texas at Austin, November 1993.
- [Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.
- [Udd84] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.
- [vB92] Kees van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.
- [vB93] Kees van Berkel. VLSI programming of a modulo-N counter with constant response time and constant power. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 1–11. Elsevier Science Publishers, 1993.
- [vdS85] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [WE88] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, 1988.