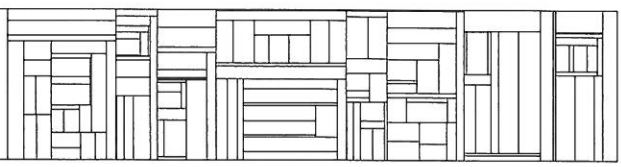
ISSN 0105-8517

Self-Timed Iteration

M.R. Greenstreet
J. Staunstrup
T.E. Williams

DAIMI PB – 228
August 1987

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 83 55 *Telex: 64767 aausci dk*



Self-Timed Iteration

M.R. Greenstreet, T.E. Williams, and J. Staunstrup
Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark

Abstract

This paper describes a technique for specifying, analyzing and implementing a series of computations using iterative, self-timed circuits. Even though the circuits iterate, they operate without clocking. The circuits do not require arbiters, have no possibility of synchronization failure, and function correctly independent of time delays. Each computation in the iteration can be a general function and is implemented using asynchronous, delay-independent logic which indicates its completion. A general implementation for self-timed iteration is presented; then, simplifying assumptions for specific implementations are shown which can reduce the amount of hardware required.

1 Introduction

Self-timed circuits operate without external clocking and perform their operations as fast as the technology, temperature, and specific data inputs allow. These circuits indicate their completion along with an output value by either providing a signal from a matching worst case path or encoding completion as part of the result[6]. However, most previous usages of self-timing techniques in integrated circuits have been very localized and have not performed iteration (cycling) within the self-timed domain. For example, though the self-timed PLA[9] has become common, it is typically embedded in a synchronous system, and though it operates repeatedly, the self-timing does not extend beyond one evaluation of the PLA; the iteration is controlled by clocked circuitry. Although iterative asynchronous designs are presented in [1], those circuits perform synchronization with arbiters. This paper shows that it is possible to build a system which iterates within the self-timed domain, and without any external clocking, arbiters, timing hazards, or constraints on propagation delays.

The analysis of self-timed iterations was inspired by, and is a generalization of, the technique used to implement a division chip in [10]. Using self-timed iterations is particularly attractive and practical for arithmetic processors, such as the divider, because they involve a long sequence of calculations which can proceed without further external inputs after initialization. The self-timed iterations can provide a performance advantage over a system whose iterations are controlled by an external clock because each iteration can begin as soon as the previous is finished; whereas, a clocked design must wait for the worst possible delays in each cycle.

In order to specify and analyze communication in an asynchronous environment, a notation called "Synchronized Transitions" [7] is used. After introducing the notation, a self-timed iterative algorithm is given and shown to operate correctly independent of delays. A general hardware implementation for the self-timed algorithm is shown which corresponds directly to the specification. This is followed by a discussion of optimizations which can be performed based on bounds for relative timing in specific implementations.

2 Synchronized Transitions

A system is typically constructed by hierarchically partitioning it into smaller blocks of circuitry with defined communication between the blocks. In synchronous designs, this communication is based on clocks which are common to the communicating circuits. In self-timed designs, communication between blocks must be asynchronous because there are no common clocks to control communication actions.

State changes are modeled as **transitions**. Transitions are described using an imperative notation similar to what is found in many high-level programming languages. For example,

```
TRANSITION evaluate(s:  state; z:  BOOLEAN)
< s.x → s.x, s.y := FALSE, f(z) >
```

The transition `evaluate` is performed only in a state satisfying `s.x` and leads to a state where `s.y = f(z)` and `s.x = FALSE`. The transition is atomic, which means that it *appears* to be indivisible. Asynchronous blocks communicate by participating in the same transition. This is expressed by including state variables from two or more blocks as actual parameters to the same transition. The general form for a transition is:

```
TRANSITION name(n1, n2 ...  )
< C(n1, n2 ...) → A(n1, n2 ...) >
```

- `n1, n2 ...` are the state variables used by the transition.
- `C(n1, n2 ...)` is the **precondition** of the transition; it is a boolean condition on the values of `n1, n2 ...` The transition can only be performed when its precondition is satisfied.
- `A(n1, n2 ...)` is the **action** of the transition. This is an assignment which specifies the **state transformation** made by this transition. It may only change the values of `n1, n2 ...`

It is not required that a transition is performed immediately after its precondition has been satisfied, and there is no upper bound on when it takes place. The precondition may become `false` again without the transition having been performed; however, restrictions presented in section 2.1 prevent this in the self-timed implementations we are considering. The execution order and duration of enabled transitions is unspecified.

A transition is atomic as indicated by the notation < ... >. This means that the transition appears to be executed indivisibly; this may, however, not be the way it is implemented.

Initiation of a transition is written as a procedure call, e.g. `invert(a, b)`. Transitions are repeated indefinitely once they are initiated. Usually, many transitions must be initiated to describe the desired computation. This is done by giving a list of transition initiations separated by `||` (to indicate concurrent executing of the transitions):

```
invert(ready, reset) || evaluate(si, input) ||
evaluate(so, yi) ...
```

This type of description gives an abstract model (specification) for the behavior of a design. A more detailed description of "Synchronized Transitions" and their applications is given in [7]. A similar notation using "Production Rules" for the synthesis of self-timed circuits is presented in [5].

2.1 Implementation rules

The "Synchronized Transitions" notation provides a **specification** of a system in terms of atomic actions (transitions). However, it is not required that an **implementation** perform *only* one transition at a time. To do so would be both impractical and inefficient. Therefore, the implementation should perform many transitions simultaneously, while preserving the appearance of atomicity.

This section presents criteria for correspondence between a specification using the "Synchronized Transitions" notation and an implementation where the transitions are non-atomic. Sufficient conditions for a non-atomic implementation to be correct are given, and some relationships between these conditions and the physical behavior of circuits are described. This presentation is informal; a rigorous derivation is beyond the scope of this paper.

In a non-atomic implementation of a transition, $< C \rightarrow A >$, the appearance of atomicity must be maintained locally, since there are no global synchronization signals to enforce indivisible operations. This means that the condition C must be strong enough (together with all other preconditions) to prevent inconsistent executions, and that some restrictions must be imposed on the allowable collections of transitions.

Executions where two transitions simultaneously write the same state variable must be prohibited. Such writing could result in variables with undefined values. We do not use arbiters to resolve these conflicts, because an arbiter cannot be both error free and guaranteed to decide in bounded time[4]. By observing the following restriction, simultaneous writing is avoided:

> *Exclusive Write*: Each state variable must only be written by a single transition.

This is a very strong condition; weaker conditions could be sufficient. However, this condition is simple, and it is satisfied by the algorithms presented in this paper.

The values of variables must be stable while a transition is (possibly) using them. If the value of a variable read by a transition were to be changed while the transition was modifying another variable, the write variable of the transition could receive an undefined value. For example, a transition could be disabled when it had half-way changed the value of its write variable. Thus, we further restrict the allowable set of transitions with the following condition:

> *Stable Read*: Let `t1` and `t2` be two transitions such that `t1` writes a variable which `t2` reads, then `t1` and `t2` may not be simultaneously **active**.

A transition, $\mathtt{t}, (< \mathtt{c_t} \rightarrow \mathtt{l_t} := \ \mathtt{f}(\mathtt{r_t}) >)$ is defined to be **active** if $\mathtt{c_t} \wedge (\mathtt{l_t} \neq \mathtt{f}(\mathtt{r_t}))$. Like exclusive-write, this condition is stronger than necessary.

"Synchronized Transitions" descriptions which meet the above conditions can be implemented by hardware which never produces metastable values. Metastability occurs when an intermediate value (neither `true` nor `false`) is written into a storage element. This can occur either by attempting to concurrently write different values, or by inhibiting a write before the stored value has been completely changed. The first situation is prevented by the exclusive-write condition. The stable-read condition prevents the second.

Furthermore, "Synchronized Transitions" descriptions which meet the above two restrictions can be implemented by hardware which functions correctly independently of delays in logic elements and wires. Arbitrary delays in logic elements are modeled by the property that enabling

elapse before it actually does. Wire delays influence the order in which inputs arrive to a transition. This is equivalent to altering the order in which the other transitions wrote these variables. The stable read condition guarantees that the values assigned by a transition are independent of the order in which enabled transitions were executed. Thus, hardware implementations satisfying these conditions are independent of both wire and logic element delays.

3 Iterative Self-timed Algorithms

In this section, two algorithms will be presented for self-timed iteration. The first algorithm is for a simple **ring-oscillator**, but it demonstrates many of the key features of more complex iterations. The second algorithm is a generalization of the ring-oscillator; arbitrary functions can be computed, yet the algorithm retains the simple underlying structure of the ring-oscillator. The correctness of both algorithms is demonstrated by establishing invariants which they maintain.

In an iterative design, computation progresses as a wave traveling around the loop of stages. The rising edge of the wave corresponds to computing a new result, the falling edge corresponds to resetting the hardware element in preparation for the next computation. The wave can be represented by a single bit, in which case the circuit oscillates, but computes no other result. By representing the wave with more than one bit, data-values can be encoded into the wave, and useful computation can be performed as the wave progresses. For proper computation, it must be ensured that *the rising edge of the wave never overtakes the falling edge* (i.e. attempting to perform a computation on a stage which is not reset from the previous computation), and *that the falling edge never overtakes the rising edge* (i.e. resetting all stages and entering a dead-state).

3.1 Simple Iteration

In the case of a simple ring-oscillator, a single bit circulates around the loop. Such a ring-oscillator is shown in figure 1. This oscillator differs from traditional ring-oscillators in that it is built from Muller-C elements (see figure 2) instead of inverters. The forward path along which a pulse

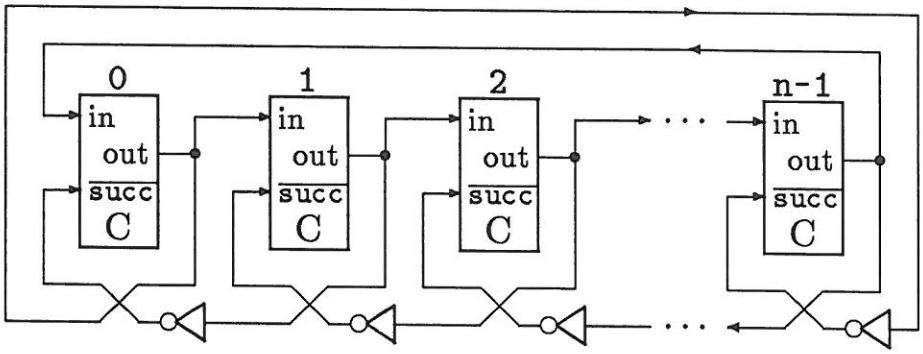


Figure 1: Ring-Oscillator

Truth Table

in	$\overline{\text{succ}}$	out
L	L	L
L	H	Unchanged
H	L	Unchanged
H	H	H

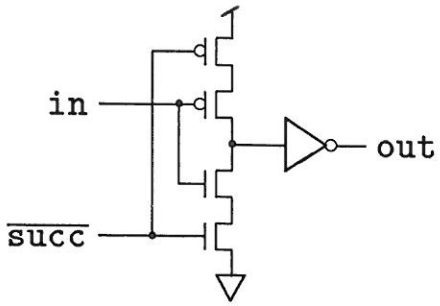


Figure 2: The Muller-C Element

progresses around the oscillator results in no overall inversion; thus, the number of stages, n, can be either even or odd.

The operation of the oscillator can be understood by, initially, assuming that the lower ($\overline{\texttt{succ}}$) input of each C element is always the same as the upper input. In this case, the C element functions as a buffer; thus, the oscillator works as a ring of buffers. The C elements are divided into two non-empty, contiguous groups: those whose outputs are `true` (the crest of the wave) and those whose outputs are `false` (the trough of the wave). The wave progresses around the ring as the buffers on the boundary between the crest and the trough transfer a `true` value forward and by the complement action at the other boundary.

If the circuit was simply a ring of buffers, the oscillation would eventually die out. In real circuits, propagation delays are different for rising and falling values and for different instances of the same circuit. Thus, in a ring of buffers, either the trough will eventually overtake the crest, or the crest will overtake the trough, and the circuit will enter a dead-state. The ring of C elements avoids this problem. This is the purpose of the

```
ringosc(n:  3..∞)  (* the loop has n stages *)
   VAR y:  ARRAY [0..n-1] OF BOOLEAN
   TRANSITION C(in, out, succ:  BOOLEAN)
   (* C-element with one input inverted *)
   < in <> succ → out := in >
BEGIN
   ||_{i=0}^{n-1} C(y[i⊖1], y[i], y[i⊕1]);
END ringosc.
```

Where: i $\oplus$ 1 = (i + 1) mod n and i $\ominus$1 = (i - 1) mod n.

Figure 3: "Synchronized Transitions" Program for the Ring-Oscillator

lower ($\overline{\texttt{succ}}$) input to the C elements. This input ensures that the output of one element can only become `true` when the output of the successor element is `false`. This guarantees that the crest cannot overtake the trough. Likewise, the output of an element can only become `false` when the output of the successor element is `true`.

The preceding arguments assumed that the delays of the wires and inverters were insignificant in comparison to the delays of the C elements. In particular, it was assumed that the value of the lower input to a C element was the complement of the value of the output from the successor stage. To account for arbitrary delays, we can only assume that the value at an input is a value which the supplying output had *sometime in the past.* To consider this more general case, it is useful to employ the "Synchronized Transitions" notation. This will also be useful when the algorithm is generalized by replacing the C elements with circuits which compute a useful result in the course of the oscillation. A program for the ring-oscillator using "Synchronized Transitions" notation is shown in figure 3.

This program satisfies the restrictions given in section 2.1 and can be expected to function correctly independently of the delays of wires or logic elements. Since each state variable is written by exactly one transition, the program satisfies the exclusive-write condition. The dynamic behavior of the program must be considered to show that the stable-read condition is satisfied.

In the self-timed paradigm, there is no global time reference (such as

a clock) to allow reasoning about the state of the entire system. Instead, we can reason about individual transitions, and show that certain pairs of transitions cannot be simultaneously active. This allows us to reason about the values of variables in *local* sets of transitions, and from this, *global* properties of the system can be demonstrated.

Invariant 1 *The stages whose outputs are* `true` *form a non-empty set of adjacent stages. The stages whose outputs are* `false` *form a non-empty set of adjacent stages.*

To see that this is an invariant, assume that it holds at some point of the execution. This means that there exists a stage `y[i]` such that:

$$(\texttt{in} = \texttt{true}) \wedge (\texttt{out} = \texttt{succ} = \texttt{false}) \quad \texttt{or} \quad (\texttt{in} = \texttt{false}) \wedge (\texttt{out} = \texttt{succ} = \texttt{true})$$

In both cases, performing `out := in` maintains the invariant. [1]

From the invariant and the requirement that $\texttt{n} \geq 3$, it follows that the stable read condition is satisfied. Observe that, if there is an element whose output is `true` followed by two elements whose outputs are `false`, there must be an active transition (to set the middle output to `true`). A similar argument applies for an element whose output is `false` followed by two elements whose outputs is `true`. This leads to one more invariant, which shows that the ring-oscillator never enters into a dead-state.

Invariant 2 *If there are at least three C elements in the loop, then there exists an active transition.*

In this section, we have shown how a description using "Synchronized Transitions" may be used to demonstrate the correctness of a self-timed design. This was done by checking the implementation rules of section 2.1 and establishing invariants, which we believe is a much more reliable way of demonstrating properties of a design than simulation or exhaustive case analysis.

3.2 A General Algorithm for Iteration

A tail-recursive function can be computed by repeatedly applying a simpler kernel to the result of the previous application (or input). A direct implementation of such a function, $\mathcal{F}$, uses a separate instance of the

[1] Strictly speaking, it should also be demonstrated that the initial state satisfies the invariant.

$$y_i = \begin{cases} \texttt{input} & i = 0 \\ \mathcal{F}(y_{i-1}) = \mathcal{F}^i(\texttt{input}) & k \geq i > 0 \end{cases}$$

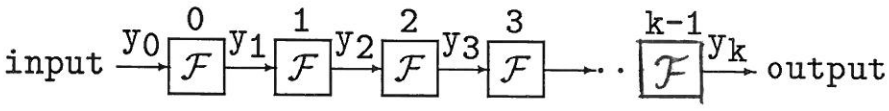


Figure 4: Fully-Combinatorial Implementation

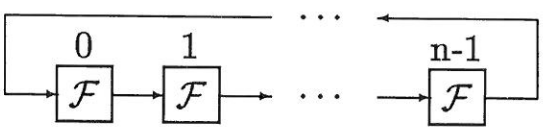


Figure 5: Iterative Implementation

hardware for each application of the kernel. This is shown in figure 4. Such implementations are used, for example, in fully-combinatorial multipliers, where the kernel is a shift and a conditional addition. Given an input value y_0, the circuit computes a sequence of values $y_1, y_2, ..., y_k$. Each stage computes a new value, y_i, of the sequence by applying some fixed function $\mathcal{F}$ on the value received from its predecessor, y_{i-1}. The exact nature of $\mathcal{F}$ is not important for our analysis. In the divider [10], which inspired this work, $\mathcal{F}$ is a function which given a divisor and a partial remainder computes a quotient digit and a new partial remainder.

Iteration allows the computation of the desired sequence $(y_1, y_2, ..., y_k)$ by looping the evaluation around a small, fixed number of stages arranged in a ring. This is possible because it is the same function $\mathcal{F}$ which is used repeatedly. Such an implementation is shown in figure 5. For simplicity, issues of input to and output from such a loop are deferred to section 5.

In this paper, we are concerned with self-timed implementations. Let F be a physical (self-timed) implementation of $\mathcal{F}$. Let in and out be the input and output of a stage, and let succ be the output of the succeeding stage (as in figure 1). The circuit F may require some time to compute a new output given a new input. During this time, it may be the case that $F(y_{in}) \neq \mathcal{F}(y_{in})$. The output is said to be **invalid** during this time. To use F in a self-timed system, it must be possible to determine if the output is **valid**, from the value of $F(y_{in})$. This can be done by adding one or more

```
iteration(n:  3..∞)  (* the loop has n stages *)
   VAR y:  ARRAY[0..n-1] OF alphabet;
         (* alphabet is a self-completion-indicating
         signal *)
   TRANSITION C'(in, out, succ:  alphabet)
   <(valid(in) ∧ reset(succ)) ∨ (reset(in) ∧ valid(succ))
   → out := F(in)>
BEGIN
   ||_{i=0}^{n-1} C'(y[i ⊖ 1], y[i], y[i ⊕ 1]);
END iteration.
```

It is assumed that F will produce a reset output when `reset(in)`.

Figure 6: "Synchronized Transitions" Program for Iteration

reset values to the range of F. Such representations of values are called **self-completion-indicating**. A completely delay-independent implementation requires self-completion-indicating signals; sending a separate completion signal would introduce a timing dependency into the design. In particular, the delays of the wires carrying the actual data and the wire(s) carrying the completion signal would have to be matched.

The block F can be used in an iterative system as follows. First, the input of F is set to a reset value. When the output of F becomes reset, we know that the circuit is ready to begin a new computation. Second, the input of F is set to a valid value. When the output of F becomes valid, we know that the circuit has computed a new result. These operations can be performed repeatedly as required by the iterative algorithm. The computation progresses like the wave in a ring-oscillator. A valid value represents the crest of the wave, and a reset value represents the trough. This is a generalization of the simple oscillator in section 3.1.

The "Synchronized Transitions" program for such a system is shown in figure 6. The predicate `valid(X)` is true if the value of the signal X is valid, and the predicate `reset(X)` is true if it is reset. During the time a signal X is transitioning due to the resetting or evaluation of its source, neither `reset(X)` nor `valid(X)` will be true. The validity of this program can be verified using invariants analogous to those presented with the simple oscillator.

set of adjacent stages.

Invariant 4 *If there are at least three C' elements in the loop, then there exists an active transition.*

Furthermore, the program in figure 6 computes the desired sequence of values as demonstrated by the next invariant. Let `j` be an index for the stages ($0 \leq$ `j` $<$ `n`) and `i` be an index for the values ($0 \leq$ `i` $\leq$ `k`). Because each stage computes many values, `i` increases as the computation progresses.

Invariant 5 `y[j]` $= \mathcal{F}^{\texttt{i}}$`(input)` $\vee$ `reset(y[j])`

Assume that the invariant holds before performing an instance of a `C'` transition writing to `y[j]`. There are two cases to consider depending on which clause of the precondition is satisfied. If `valid(in)` $\wedge$ `reset(succ)`, it follows from the invariant that there is an `i` such that `in = y[j`$\ominus$`1] =` $\mathcal{F}^{\texttt{i}-1}$`(input)`; hence, performing `out := F(in)` establishes `y[j] =` $\mathcal{F}^{\texttt{i}}$`(input)`. Otherwise, `reset(in)` $\wedge$ `valid(succ)`, and performing `out := F(in)` establishes `reset(y[j])`.

4 Hardware Implementation

Self-timed hardware can be designed which corresponds to the algorithm described in the previous section. First, designs are presented which are technology independent and preserve all the properties of the algorithm. In particular, they function correctly regardless of the delays of logic elements or wires. Next, simplifications for specific implementations are suggested. Then, it is shown how more efficient designs can be derived when some timing relationships are known.

4.1 Direct Implementation

Figure 7 shows a block diagram corresponding directly to the self-timed iteration algorithm presented above. The function `F` computes a new value for `out` from the value of `in` when the `enable` input is `true` (corresponding to the precondition of a transition enabling the action); otherwise, the old value of `out` is retained. The enabling can be implemented with a transparent latch on the output of a purely combinational function

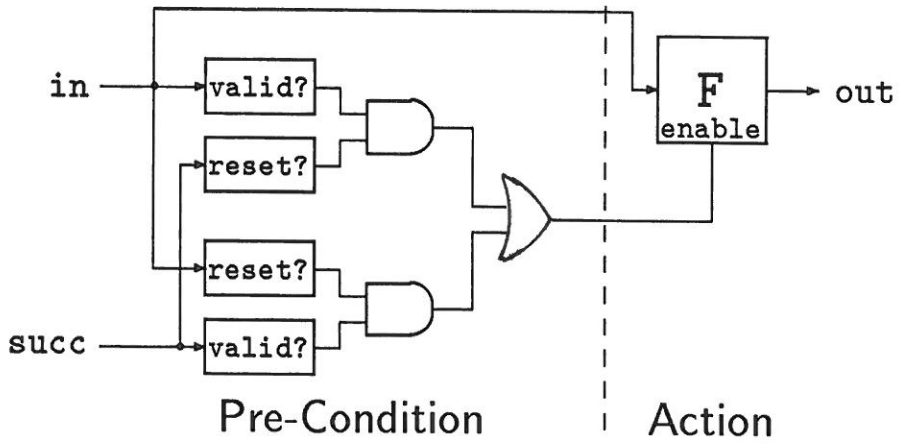


Figure 7: Direct Implementation of an Iteration Stage

which produces a reset value for out when in is reset, and out $= \mathcal{F}$(in) when in is valid.

The signals for in, out, and succ to the function F are self-completion-indicating. A simple encoding is to let each bit, X, in such a signal be represented on two wires, X^T and X^F. When both wires are low, the value is reset (e.g. not yet valid), and when either of the wires becomes high, the boolean value encoded on the pair is valid. The value for X is indicated by the superscript on whichever of the two wires X^T and X^F is high. The wire pair must return to the reset condition before the transmission of another value; so, there is never a high on both wires simultaneously. The two wires can be OR'ed together to produce the status of the signal: high if the pair represents a valid value, and low for a reset value. This representation of a boolean value can be generalized to a multi-bit representation. A multi-bit value is valid if each of its bits is valid (i.e. the AND of the status of all the bits) and reset if each bit is reset (i.e. the complement of the OR of the status of all the bits). These are the functions performed by the boxes labeled valid? and reset? in figure 7.

4.2 CMOS Implementation

Figure 8 shows a CMOS two-input NAND gate which uses the encoding described above. If both inputs are reset (A^T, A^F, B^T, and B^F are low), the stacks of P-channel transistors will pull the internal nodes (I and J) to vdd resulting in both Y^T and Y^F being low (i.e. a reset value on the

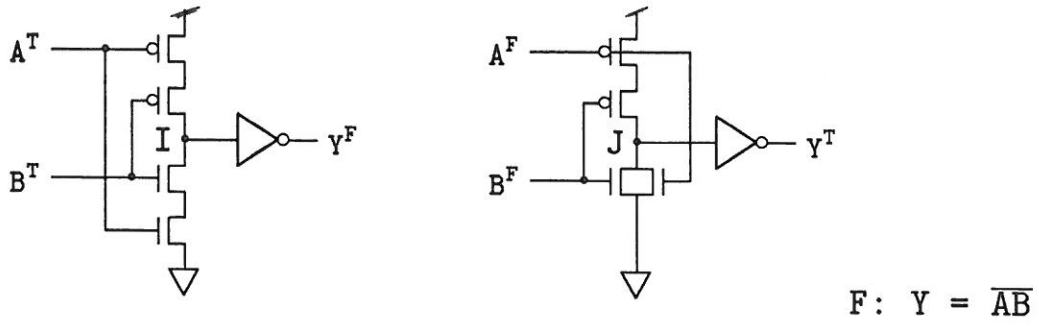


Figure 8: Self-Completion Indicating Two-input NAND Gate

output). If either A^F or B^F goes high, node J will be pulled to gnd by the corresponding N-channel transistor, resulting in the Y^T output going high. This implements the function of the NAND gate that if either input is `false` the output is `true`. A similar analysis of the other half of the circuit shows that, if both inputs are `true`, the output is `false`. Arbitrary logic functions can be implemented by these techniques to implement the function, F, for the iteration algorithm. As both the `true` and `false` signals are available from every logic element, appropriate networks of N-channel devices can be designed to implement whatever function is desired. Furthermore, logic implemented as the NAND gate above makes *monotonic* transitions. If each bit of the input remains stable after transitioning from a reset value to a valid value (or valid to reset), each bit of the output will also make a single transition from a reset value to a valid (or valid to reset).

This form of logic implementation has properties which allow simplification of the hardware design. It is possible for neither the pull-up network nor the pull-down network of such logic networks to be active. In this situation, the old value of the output is retained by the capacitance of nodes I and J. This dynamic storage can be utilized to merge the hardware for F and the transparent-latch.

The monotonicity of all circuitry makes it possible to implement the tests for `valid(in)` and `reset(in)` implicitly, within the the hardware for F. Such tests do not need to be separately implemented for the precondition. Moreover, the tests for `valid(succ)` and `reset(succ)` can be combined into a single test, `status`, which has memory as to whether `valid(succ)` or `reset(succ)` were last true:

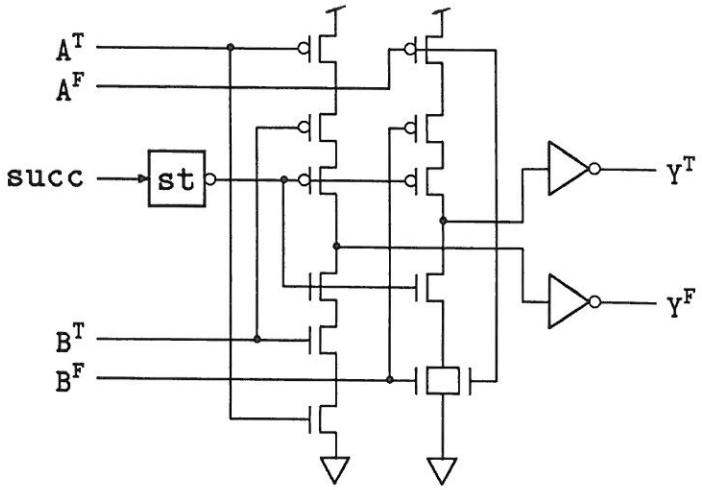


Figure 9: A CMOS Implementation of an Iteration Stage

status(X) = TRUE if X were last valid
= FALSE if X were last reset

Figure 9 shows a single stage for the iterative algorithm simplified by the observations above. The function F is a two-input NAND gate to provide a simple example. The transparent-latch is combined with the hardware for computing F by adding an extra N-channel and an extra P-channel transistor to each stack. Tests for `valid(in)` and `reset(in)` are implicitly performed by the logic implementing the NAND gate, and the tests for `valid(succ)` and `reset(succ)` are replaced by the single test for `status(succ)` as above.

The "Synchronized Transitions" algorithm guarantees that the stages interact correctly independently of any timing delays. Each stage corresponds to an atomic transition. This requires that inputs to F have sufficiently short rise and fall times that the various transistors connected to the signal interpret it as the same boolean value. Furthermore, when an output changes, it must be guaranteed that it makes a complete change between voltage levels corresponding to the logical values so that, if the value is used by more than one stage, it will be interpreted consistently. Schmidt-triggers[3] can be used to buffer the inputs and output of each stage such that these conditions are met. The Schmidt-trigger can be designed such that, for a monotonically changing input, once the output enters the undefined region (voltages between high and low), the output

is guaranteed to complete the transition in bounded time.[2] The Schmidt-triggers isolate the internal nodes to guarantee bounded wiring delays and rise and fall times within each stage regardless of the properties of the other stages and the inter-stage wiring.

These observations are made to show that truly time-delay independent circuits can be built for self-timed iteration (the largest isochronic region is a Schmidt-trigger and the internal node to which it is connected). However, it is often possible to derive reasonable bounds for timing delays. Thus, much more efficient implementations are possible as described in the next section.

4.3 Optimization

Optimization seeks to simplify the hardware and lessen the execution delay and silicon area. If adequate relations are known about relative time delays, more efficient designs can be produced than those described in the previous section. When an implementation is fabricated on a single integrated circuit, reliable delay judgements can usually be made based on transistor sizing and the similarities of transistors fabricated on the same chip. If such is the case, then it can be assumed that faster paths have already finished when a known slower path is observed to finish. This assumption means that the hardware to explicitly check such conditions can be removed, yielding a simpler design.

In the case of the CMOS implementation of figure 9, the P-channel transistors controlled by A^T, A^F, B^T, and B^F serve to confirm that the reset value on all of a stage's input wires is received from its predecessor before the stage itself can reset. If it is known that one particular input wire is the slowest, then these stacks of transistors can each be replaced by a single transistor.

The generation of `status(out)` can be simplified in a similar manner. If a particular bit of out, represented on the wire pair `out.slowV`, is known to be the last to become valid, and another wire pair, `out.slowR`, is known to be the last to become reset then:

$$\texttt{status(out)} = (\texttt{out.slowV}^T \vee \texttt{out.slowV}^F) \copyright (\texttt{out.slowR}^T \vee \texttt{out.slowR}^F)$$

[2]Note that this is not using a Schmidt-trigger to solve a decision or arbitration problem. There can be input voltages for which the Schmidt-trigger takes unbounded time to change; however, once the output has changed far enough to enter the undefined region, the time to complete the transition is bounded.

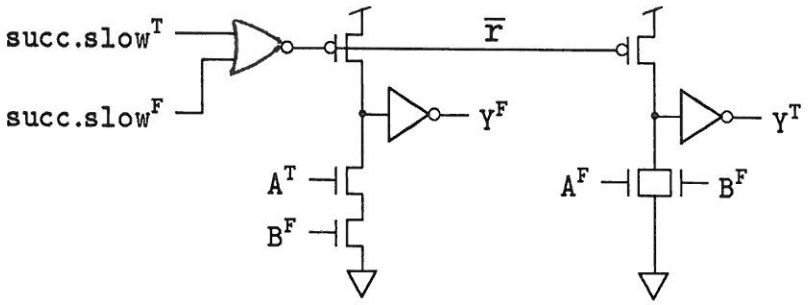


Figure 10: Optimization of an Iteration Stage Based on Known Timing Relations

where the symbol © denotes the Muller-C function. If `out.slowV` and `out.slowR` are the same wire pair (e.g. the pair driving the largest capacitance), denoted `out.slow`, then this relation reduces to

$$\texttt{status(out)} = \texttt{out.slow}^T \vee \texttt{out.slow}^F$$

Thus, the circuit which computed `status` in the previous implementation (which required logic elements for each wire in `out`) can be replaced by just a NOR gate to sense the completion of the slowest wire pair.

Further optimizations are possible when the stages are the same and they can be reset faster than they evaluate. This is often the case because evaluation may require several levels of logic, whereas resetting can be done in parallel to all of the nodes within a stage. If resetting and the associated wire propagation is always faster than evaluation, then all of the P-channel transistors which confirm the reception of the reset inputs from a stage's predecessor can be eliminated. Similarly, the N-channel transistors connected to the `st` box delay the evaluation of the stage until the reception of the status that the next stage has been reset. If resetting is always faster than evaluation, then these N-channel transistors are unnecessary because the successor to a stage will have had plenty of time to reset. The resulting circuit is shown in figure 10. It is noted that in this implementation that the invariants of the algorithm are maintained as a consequence of known time-delays in the circuit and are not explicitly enforced by the logic.

The iteration algorithm concurrently performs the resetting of one stage and the evaluation of another when there are more than three stages. However, there is no concurrency with only the minimum of three stages. When resetting is known to be faster than evaluation for

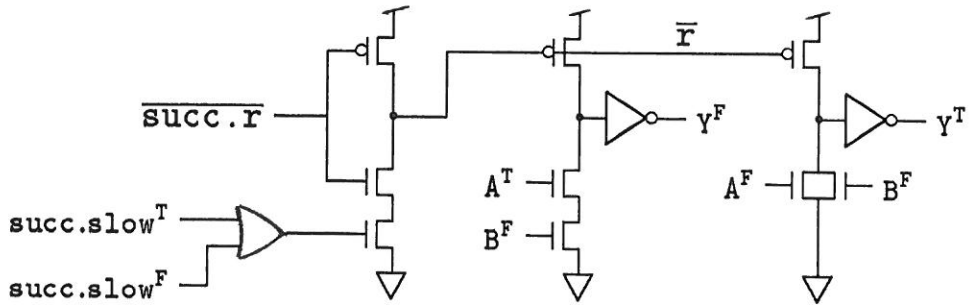


Figure 11: Modifications to Achieve Concurrency with only Three Stages

similar stages, concurrency can be achieved for the three stage case as well. In figure 10, a stage was actively reset as long as the output of the successor stage is valid. This can be expressed as:

`r = valid(succ)`

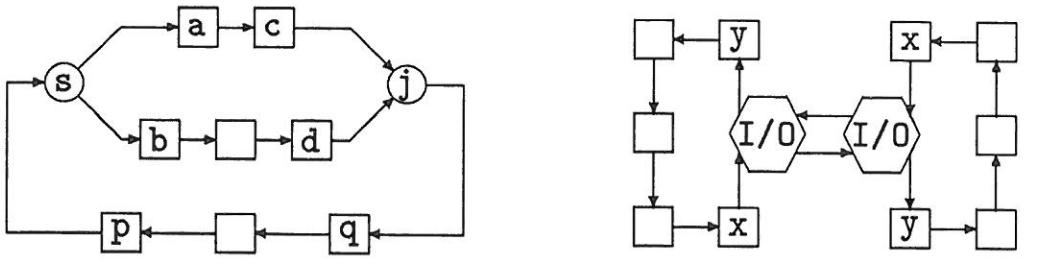
Hence, the signal $\overline{r}$ to reset a stage is low (active) throughout the time it takes for the second successor to evaluate and then for the successor to reset. If resetting is assumed to be faster than evaluation, the reset signal for a stage can be removed as soon as the stage's second successor finishes evaluation. This is because it is known that the resetting of a stage will be already done by the time the second successor has finished evaluating. Since the successor begins resetting after the second successor finishes evaluating, the reset control for a stage can be changed to:

`r = valid(succ)` $\wedge$ `¬succ.r`

By thus removing the reset signal of a stage sooner, the evaluation of that stage may proceed concurrently with the resetting of its successor. This modification, shown in figure 11 for the CMOS implementation, is the circuit used in [10] which achieves the execution speed improvement from concurrent reset and evaluate even with three stages.

5 Generalizations

In the simplest form of the algorithm, only limited concurrency is realized (e.g. one stage can compute a new valid value while another resets). Greater concurrency is possible by implementing several parallel datapaths or several communicating loops. Diagrams for these configurations are shown in figure 12.



Loop With Parallel Paths

Loops With I/O Elements

Figure 12: Systems With Increased Concurrency

As shown in the left half of figure 12, parallel data-paths may be implemented with the addition of **split** and **join** operators. The split operator, `s`, sends data from `p` to both `a` and `b` and combines the status signals from `a` and `b` to send back to `p`. In particular, `s` indicates a valid (reset) successor when *both* `a` and `b` have valid (reset) outputs as shown below:

```
valid(s) = valid(a) ∧ valid(b)
reset(s) = reset(a) ∧ reset(b)
```

The join operator, `j`, performs the complementary function of the split. The data values from `c` and `d` are concatenated to provide the input for `q`, and `valid(q)` and `reset(q)` are sent to both `c` and `d`.

The right half of figure 12 shows two communicating loops. Each loop implements the iteration algorithm. The I/O element can perform one of two operations based upon the value of its input (e.g. a particular bit of the value may specify the operation). One operation is to copy the input (from `x`) to the output (i.e. to `y`). This allows the two loops to iterate independently. The other operation exchanges a pair of values between the two loops. In this operation, the I/O element sends its input to the I/O element of the other loop, waits to receive a value from the other I/O element, and then sends this value to `y`. Because each element waits to receive a valid input before performing its next action, the exchange can be performed without arbitration[2]. This provides a mechanism for I/O operations within the self-timed domain without any possibility of synchronization failure. This also provides a method for communicating with circuits designed by other (e.g. synchronous) methods; however, the other system may be susceptible to timing hazards if adequate precautions are not taken.

if adequate precautions are not taken.

The algorithms presented in this paper provide a basis for self-timed iteration which is independent of the function performed by each stage. Different functions may be implemented for different applications. Iteration is well suited for dedicated co-processors such as chips for multiplication, division [10], transcendental functions (e.g. the CORDIC algorithm [8]), and data encryption/decryption. Using the generalizations presented in this section, self-timed iteration could be applied to arrays of processors or cellular automata, where individual cells can operate at much higher rates than it is practical to distribute a global clock.

6 Conclusions

We have presented an algorithm for iteration in the self-timed domain using only asynchronous circuits. Fabricated and tested VLSI arithmetic chips [10] using these circuits have previously verified and demonstrated the usefulness of self-timed iterations. The algorithm in this paper can be applied to systems of any size independent of communication and processor delays. Because it is self-timed, particular hardware implementations will operate as fast as the technology, temperature, and data values allow.

References

[1] T.S. Anantharaman, E.M. Clarke, et al., "Compiling path expressions into VLSI circuits," Distributed Computing, vol. 1, no. 3, pp. 150-166, 1986.

[2] D.M. Chapiro, "Globally-Asynchronous, Locally-Synchronous Systems," Ph.D. thesis, Department of Computer Science, Stanford University, October 1984.

[3] L.A. Glasser and D.W. Dobberpuhl, **The Design and Analysis of VLSI Circuits**, pp. 280-282, Addison-Wesley, 1985.

[4] L.R. Marino, "General Theory of Metastable Operation," IEEE Transaction on Computers, vol. 30, no. 2, pp. 107-115, February 1981.

[5] A.J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits," Distributed Computing, vol. 1, pp. 226-234, 1986.

[6] C.L. Seitz, "System Timing," Chapter 7 of **Introduction to VLSI Systems** by C. Mead and L. Conway, Addison-Wesley, 1978.

[7] J. Staunstrup and A.P. Ravn, "Synchronized Transitions," DAIMI PB-219, Computer Science Department, Aarhus University, Denmark, (submitted for publication) January 1987.

[8] J.S. Walther, "A Unified Algorithm for Elementary Functions," 1971 Spring Joint Computer Conference, AFIPS Proceedings, vol. 38, pp. 379-385, 1971.

[9] N. Weste and K. Eshraghian, **Principles of CMOS VLSI Design, A Systems Perspective**, p. 372, Addison-Wesley, 1985.

[10] T.E. Williams, M. Horowitz, et al., "A Self-Timed Chip for Division," Proceedings of the Conference on Advanced Research in VLSI, Stanford University, March 1987.