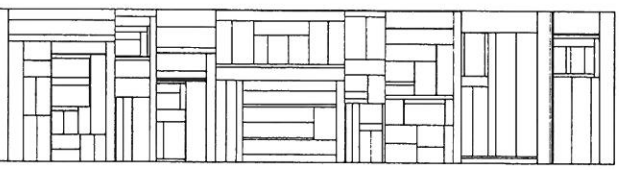
ISSN 0105-8517

From High-level Descriptions to VLSI Circuits

M. R. Greenstreet
J. Staunstrup

DAIMI PB – 255
June 1988

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88 *Telex: 64767 aausci dk*



From High-level Descriptions to VLSI Circuits

J. Staunstrup
Computer Science Department
Aarhus University, Ny Munkegade
DK-8000 Aarhus C
Denmark

M.R. Greenstreet
Computer Science Department
Princeton University
Princeton, NJ 08540
USA

To appear in BIT 1988

Abstract

This paper presents a high-level language for describing VLSI circuits designed as a collection of asynchronous concurrent processes. This language is called "Synchronized Transitions," and it can be used to describe designs from very high levels of abstraction down to the gate level design. Both synchronous and asynchronous/self-timed circuits can be described, and it is not necessary to choose a particular type of circuitry in the early phases of a design. "Synchronized Transitions" programs may be used for experimenting with (simulating) a design at several levels, e.g., to explore different high-level decisions or to verify the gate level. By observing certain constraints in a "Synchronized Transitions" program, it is possible to systematically transform the program into an efficient layout.

CR categories: B6, C.1.2, C.5.4, D.1.3, D.3.3.

expressed explicitly in the preconditions of transitions (i.e., as an expression of the system state). Thus, high-level descriptions in "Synchronized Transitions" do not depend on details of the low-level implementation. This makes "Synchronized Transitions" programs both clearer and easier to write.

The development of "Synchronized Transitions" was motivated by research in applying VLSI to parallel processing. Traditional VLSI architectures are based upon a synchronous model of parallel computation which require the timing details of the low-level implementation to be reflected in its high-level descriptions. Systolic algorithms [9] are an example of this and are well suited for many applications such as matrix operations. However, systolic algorithms are poorly suited for applications where the computation to be performed depends on the input data or requires response to inputs at times that cannot be statically determined. Such computations are naturally expressed as interactions of *asynchronous* processes which can be specified using "Synchronized Transitions".

At the other end of the design spectrum, circuits for VLSI have traditionally been designed using a synchronous (i.e. clocked) approach. In many fast circuits, clock distribution imposes the dominant speed limitation, and they cannot operate as fast as would be allowed if only gate delays were significant. Self-timed/asynchronous designs promise to overcome such limitations [14]. However, self-timed designs have typically required intricate analysis of the timing details of the circuits. A promising alternative is to analyze the functionality of self-timed circuits independently of timing details by viewing the operation of the circuit as the execution of a collection of asynchronous parallel processes as can be described with "Synchronized Transitions" [6].

"Synchronized Transitions" programs are well suited for designing application specific VLSI circuits (so-called ASICs). ASICs are usually designed to satisfy requirements of size, maintainability, speed, or special function which cannot be met by off-the-shelf components. Because ASICs are typically produced in small quantities, it is important to reduce design costs wherever possible. The need for ASICs typically arises from a specific application, and to meet the requirements of the application, creative architectures and alternative systems approaches need to be considered. Tools developed for particular classes of VLSI circuits such as microprocessors, signal processors, and PLAs can be too specialized for

the design of ASICs. "Synchronized Transitions" provides the flexibility needed to explore alternative implementations and produce useful ASIC designs.

This paper presents the "Synchronized Transitions" language and its application to the design of VLSI circuits. Section 2 introduces the notation, and section 3 describes an application, the problem-store which is used as an example throughout the rest of the paper. A particular implementation of the problem-store called a storage-ring is described in section 4. The use of "Synchronized Transitions" programs for simulation is presented in section 5, and the simulation of the storage-ring is discussed. In section 6, transformations from "Synchronized Transitions" programs to synchronous circuits are described, and these transformations are applied to the storage-ring in section 7.

2 Synchronized Transitions

This section gives a brief introduction to the "Synchronized Transitions" notation. Like a VLSI circuit, a "Synchronized Transitions" program consists of state variables and transitions (combinatorial logic). **Transitions** specify state changes using multi-assignments. For example,

$$\ll \texttt{from} \neq \texttt{E} \wedge \texttt{to} = \texttt{E} \rightarrow \texttt{to, from} := \texttt{from, E} \gg$$

defines a transition which is performed only when `from` $\neq$ `E` and `to` = `E` holds, and it leads to a state where `to` = $\texttt{from}^{old}$ and `from` = `E` ($\texttt{from}^{old}$ denotes the value of `from` before the transition is performed).

In general, transitions are of the form

$$\ll \textit{precondition} \rightarrow \textit{action} \gg .$$

The **precondition** is a boolean expression. A transition can only be performed when its precondition is satisfied. The **action** is a multi-assignment which specifies the **state transformation** made by the transition. Transitions are atomic as indicated by the notation $\ll \ldots \gg$. This means that each transition *appears* to be executed indivisibly.

It is not required that a transition be performed immediately after its precondition becomes satisfied, and there is no upper bound on when it takes place. This is an abstraction of delays in hardware. For example,

$$\ll \texttt{TRUE} \rightarrow \texttt{y} := \texttt{a} \vee \texttt{b} \gg$$

describes an OR gate. The precondition, TRUE, specifies that it is always allowed to set the output, y, to the OR of the inputs, a and b; however, an arbitrary delay may elapse between changing the inputs and the changing of the output. In fact, other transitions can change the values of variables while a transition is enabled. Thus, the precondition of a transition may become false again without the transition having been performed.

2.1 Combinators

A transition describes the behavior of a subcircuit (e.g., an area of the chip). Once such a subcircuit is fabricated, it is never removed, and it is continuously in operation. This behavior is modeled in a "Synchronized Transitions" program by a **transition instantiation** which results in a transition that is performed repeatedly, i.e., every time the precondition is satisfied, the action may be performed. To describe any significant circuit, a large number of such transitions are needed. Two operators are provided for combining simple transitions into descriptions of substantial circuits. The asynchronous combinator, ||, combines two transitions which execute independently. For example,

$$\ll \texttt{a < b} \rightarrow \texttt{a, b := b, a} \gg \| \ll \texttt{b < c} \rightarrow \texttt{b, c := c, b} \gg$$

is a description with two independent transitions each of which may be performed whenever its preconditions is satisfied. This computation sorts a, b, and c into ascending order. All transitions combined with || are independent; there is no global thread of control determining the order of execution.

The synchronous combinator, $*$, creates transitions which are always performed together. For example,

$$\ll \texttt{c1} \rightarrow \texttt{a := b} \gg * \ll \texttt{c2} \rightarrow \texttt{b := c} \gg$$

is equivalent to

$$\ll \texttt{c1 AND c2} \rightarrow \texttt{a,b := b,c} \gg .$$

Synchronous composition is used to specify that two operations are always performed simultaneously (e.g., under control of a global clock); whereas, no such assumptions are made with asynchronous composition. The || and

$*$ combinators have the following properties:

$$\begin{array}{rrcl} \text{Commutative:} & t_1 \| t_2 & \equiv & t_2 \| t_1, \qquad t_1 * t_2 \equiv t_2 * t_1 \\ \text{Associative:} & (t_1 \| t_2) \| t_3 & \equiv & t_1 \| (t_2 \| t_3), \quad (t_1 * t_2) * t_3 \equiv t_1 * (t_2 * t_3) \\ \text{Precedence:} & t_1 \| t_2 * t_3 & \equiv & t_1 \| (t_2 * t_3) \\ \text{Distributive:} & t_1 * (t_2 \| t_3) & \equiv & t_1 * t_2 \parallel t_1 * t_3 \end{array}$$

These properties of $\|$ and $*$ give "Synchronized Transitions" great expressive power in describing the relationships of transitions. A few special forms of transitions are used to simplify forming expressions:

$\ll$ *Multi-Assignment* $\gg$
: This is a transition whose precondition is always true. For example, $\ll$ `y := a` $\vee$ `b` $\gg$ is equivalent to: $\ll$ `TRUE` $\rightarrow$ `y := a` $\vee$ `b` $>$ $>$.

$\ll$ *Boolean Expression* $\gg$
: This is a transition which has a preconditon, but performs no action. For example, $\ll$ `c1` $\gg$ $*$ $\ll$ `a := b` $\gg$ is equivalent to: $\ll$ `c1` $\rightarrow$ `a := b` $\gg$.

Frequently, the same operation is required in many different places in a "Synchronized Transitions" program. In each place, the operation is performed on a different set of state variables. "Synchronized Transitions" provides a mechanism for parameterizing and naming transitions which is analogous to functions of traditional languages. For example,

```
TRANSITION copy(from, to: element);
≪ from ≠ E ∧ to = E → to, from := from, E ≫
```

Several **instances** of a transition can be created using the operators described above and supplying actual parameters. For example,

```
copy(a, b) || copy(x, y).
```

2.2 Sets, Qualifiers, and Quantifiers

Often it is necessary to create a number of similar instances of a transition (operating on a set of state variables, for example, an array). This is expressed as follows:

```
|| { 0 <= i < n:  copy(a[i], a[i+1]) },
```

which is equivalent to

```
copy(a[0], a[1]) || copy(a[1], a[2]) || copy(a[2], a[3])
   || ...|| copy(a[n-1], a[n])
```

This notation can be used in general to combine a set of transitions. In its simplest form, a set is written {a, b, c}, and an expression such as || {a, b, c} is equivalent to a || b || c. Sets may also be used in binary expressions. a || {b, c} is equivalent to a || b || c, and a || {} (where {} denotes the empty set) is equivalent to a.

A set element may be qualified by preceding it with a predicate. For example, the set

```
{d > 1:  a }
```

includes a if d > 1. This allows **conditional instantiation** of transitions.

A set element may be quantified by preceding it with the range for an index variable. For example, the set {0 <= i < n: a(i)} is equivalent to

```
{a(0), a(1), a(2), ..., a(n-1)},
```

and * {$0 <= i < n$: a(i)} is equivalent to

```
{a(0) * a(1) * a(2) * ...* a(n-1)}
```

The predicates and range expressions used in sets are evaluated statically (when the transition is instantiated). Transitions, like hardware, are not dynamically created and destroyed.

2.3 Cells

Cells are used for subdividing a large program into smaller components. An example of a cell is given in figure 1. It consists of n registers forming a ring. The ring circulates the elements in its registers. The cell heading

```
CELL ring(p: ARRAY[0..n-1] OF element);
```

gives the name of the cell and its formal parameters. The corresponding actual parameters form the **interface** by which the cell communicates with other cells. A cell may contain declarations of local state variables, transitions and cells. The body of a cell (enclosed by BEGIN END) describes

```
CELL ring(p: ARRAY[0..n-1] OF element);
        (* n gives the size of p *)
   TRANSITION copy(from, to: element);
   ≪ from ≠ E ∧ to = E → to, from := from, E ≫
BEGIN
   || {0 <= i < n:  copy(p[i], p[(i+1) MOD n])}
END ring.
```

Figure 1: A ring of registers

the instantiations which take place when the cell itself is instantiated. These are described using the combinators || and * introduced above.

Instantiation of a cell is written in the same way as the instantiation of a transition, by giving its name and actual parameters. Such a cell instantiation creates the state variables of the cell and performs the internal instantiations of the cell, which in turn instantiate all its transitions and possible further subcells. Cell instantiations may appear in expressions using the || and * combinators, just like transition instantiations. A cell may instantiate itself recursively, but recursion is a description mechanism only. The recursion takes place during instantiation. Cells may have STATIC parameters which are fixed when the cell is instantiated and may be used to dimension arrays, control recursive instantiation, etc.

The remainder of this paper presents an example showing how "Synchronized Transitions" may be used to derive a non-trivial VLSI design. The design is done as a sequence of refinements. Each refinement is written as a "Synchronized Transitions" program, with increasing implementation detail.

3 An Example: The Problem-store

A **problem-store** is a multi-set with one or more independent ports, see figure 2. At each port processors may change the contents of the problem-store by inserting or retrieving elements. When retrieving an element, a processor may get any element previously inserted; there is no requirement on the order in which the problem-store returns elements. However, elements may not disappear or be duplicated by the store. Each

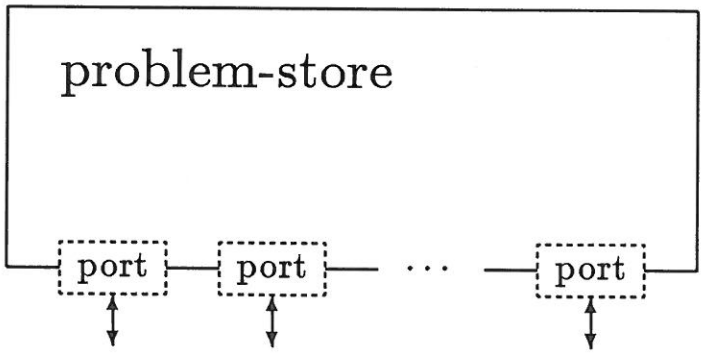


Figure 2: Problem-store with multiple ports

element in the problem-store is associated with one or more categories. To retrieve an element of a certain category, the processor supplies a pattern which the retrieved element must match. The essential operations on a problem-store are:

- `out`: inserts an element into the store, and
- `in(p)` retrieves an element matching the pattern `p` (when there is one).

The operation names `in` and `out` are taken from the tuple operations in Linda [3]. This reflects the strong resemblance between the tuple space of Linda and a problem-store.

Algorithms using a problem-store have been implemented on several general purpose multiprocessors [3][5][11]. The motivation for studying a specialized problem-store is to achieve speed improvements beyond what is possible on general purpose multiprocessors.

The interface to the problem-store is a number of ports. Each port can be manipulated independently. It contains three registers: `inreg`, `pattern`, and `outreg`, see figure 3.

- An element is inserted in the store by placing it in `outreg`.
- An element is retrieved by placing a pattern in the register `pattern`. An element matching pattern will appear in `inreg` (if there is one).
- For all three registers it is assumed, that it can be detected when they are empty (e.g., by inspecting a status variable).

The rest of this paper discusses a particular implementation of a problem-store, called a **storage-ring**. It is intended to be implemented as one or

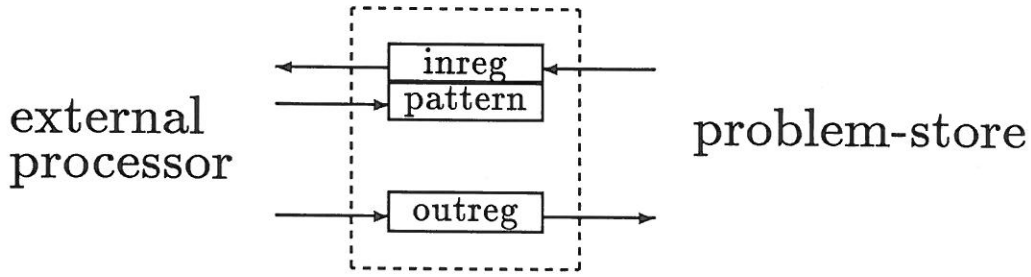


Figure 3: A port into the problem-store

more VLSI chips. Naturally, this leads to a very different architecture than the S/Net's Linda kernel [3], which is implemented on top of a local area network. Even at the VLSI level, there is a wide span of possible implementations [1] [12]. The one considered here is a ring of storage elements. Problems circulate around until they are retrieved at a port (where an `in` operation with a matching pattern is performed).

4 Storage-ring

To illustrate the concept of a storage-ring, consider the simple design shown in figure 4. It consists of n stages each of which has a port (`p[i]`) and a register (`x[i]`). The `out` and `in` transitions correspond directly to the `out` and `in` operations for the problem-store. The stages form a ring by connecting the registers of adjacent stages.

```
wire(x[i], x[(i+1) MOD n])
```

All elements stored in the ring keep moving (as long as there is at least one register which is empty). When an element passes a port, `a`, which has a matching pattern (`match(...)` is satisfied), it is retrieved.

The problem-store implementation shown in figure 4 has several deficiencies, among others a limited storage capacity and a potential for blocking operations for long periods when the store starts filling up. These problems can be solved by using a **rubber-wire**. This is a connection which can appear to stretch, increasing the storage capacity to more than the two registers found at the ends of a wire in the program shown in figure 4. When both registers are full, a new pair of empty register appears between the two. The third register has been there all the time;

```
CONST E = ...  ;              (* value of an empty register *)
TYPE port = RECORD
               inreg, outreg: element;
               pat: pattern;
     END;

CELL storage_ring(p: ARRAY[0..n-1] OF PORT);

  STATE  x: ARRAY [0..n-1] OF element;

  TRANSITION out(a: port; b: element);
  << (a.outreg <> E) AND (b = E) -> a.outreg, b := E, a.outreg >>

  TRANSITION in(a: port; b: element);
  << (a.inreg = E) AND match(a.pat, b) -> a.inreg, b := b, E >>

  TRANSITION wire(from, to: element);
  << (from <> E) AND (to = E)  ->  from, to, := E, from >>

BEGIN
  || {  0 <= i < n:  wire(x[i], x[(i+1) MOD n]) ||
                      out(p[i], x[i]) || in(p[i], x[i])  }
END storage_ring.
```

Figure 4: A simple storage-ring

```
CELL rubber_wire(A, B: element; STATIC d: INTEGER);
(*                Au --rubber_wire--> Bu
                  |                   |
                  |                   |
                  A ----------------> B       *)
 STATE Bu, Au: element;

 TRANSITION stretch;
 << (A<>E) AND (Au = E) AND (B<>E) -> A, Au:= E, A >>
BEGIN
  wire(A, B) ||  stretch ||  wire(Bu, B) ||
    { d > 1: rubber_wire(Au, Bu, d-1),
      d = 1: wire(Au, Bu) }
END rubber_wire.
```

Figure 5: Rubber-wire

it is bypassed when it is not needed. In figure 5, a "Synchronized Transitions" description of such a rubber-wire is shown. The STATIC parameter d specifies the height of the wire. The rubber-wire acts as a normal wire when B is empty, but when it is not, elements are sent upwards to Au, the terminal of another rubber-wire. The argument d specifies the maximal height of the rubber-wire. Note that the rubber-wire instantiates itself recursively.

The rubber-wire can be used to create an improved version of the storage-ring by replacing the wire connecting the registers of two ports, `wire(x[i], x[(i+1) MOD n])`, with a rubber-wire connecting the same two registers, `rubber_wire(x[i], x[(i+1) MOD n])`.

5 Program Prototypes

The "Synchronized Transitions" language is well-suited for describing hardware as parallel programs. Descriptions written in "Synchronized Transitions" can be used to derive program prototypes (simulations) and the final physical implementation (a circuit). Program prototypes are useful for experimenting with and analyzing the properties of a design at an early stage of its development. The prototype exhibits the same

functional behavior as the "Synchronized Transitions" description, and therefore as the physical implementation. For example, program prototypes of the storage-ring have been used to evaluate the effectiveness of rubber-wires.

Each transition in a "Synchronized Transitions" program performs only two simple operations: expression evaluation and assignment. These operations are typical of imperative high-level languages, and it is straightforward to translate "Synchronized Transitions" programs to a more traditional language (e.g., Modula-2 or C). The concurrent execution of transitions is accomplished by providing a module which implements lightweight processes. Each transition is executed as a separate process. This results in a very large number of concurrent processes; however, the requirements of these processes are so simple that efficient implementations are readily achieved.

The program prototype for the storage-ring was derived by hand translating the "Synchronized Transitions" program to Modula-2. The "Synchronized Transitions" language deliberately has many similarities with Modula-2 which facilitate this approach to translation. Declarations, expressions, and parameter mechanisms do not require any modification. Cells are translated into procedures which perform the necessary instantiations, and transitions are implemented as lightweight processes.

A compiler to translate "Synchronized Transitions" programs to C is currently being implemented. Although C has slightly different forms for declarations, expressions, etc., the translation process is straightforward. This compiler will be used to perform experiments with exploiting the intrinsic parallelism of "Synchronized Transitions" programs on several different multiprocessors.

5.1 Simulations

A translation such as the one just described may be used to transform the program given in section 4 into a simulation aimed at giving a quantitative evaluation of the design. The evaluation given here will focus on the significance of the rubber-wires, so several versions of a storage-ring with different heights (the parameter d) are compared.

The quantitative comparisons are based on the execution times needed to finish an **input batch**. An input batch consists of n sequences, where n is the number of ports of the storage-ring. Each sequence contains in

random:

lower bound	d = 8	d = 4	d = 2	upper bound
45	205	205	205	1200

biased :

lower bound	d = 8	d = 4	d = 2	upper bound
51	115	188	549	1200

Figure 6: Simulation results, n = 32

and out operations. A simulation consists of running the storage-ring with a given batch. Each port attempts to perform the operation at the front of its sequence in the batch, and does it as soon as it is possible. An in operation is possible if there is an element to retrieve from a particular port. Similarly, an out is possible if there is room to insert the element at the port.

The execution time is measured by counting the number of time steps required to process the entire batch. In every time step, each transition is performed at most once depending on its precondition. It is quite simple to give both an upper and a lower bound on the execution time. If the problem-store is represented in a traditional common store, at most one operation from one batch can be performed at a time. Thus, the sum of the lengths of all input sequences is an upper bound. With an ideal problem-store, out operations (from any number of ports) would always be possible and in operations (from any number of ports) would be possible whenever there are matching elements in the store. The execution time of such an ideal problem-store can be determined from the sequences in the input batch. This time is used as a lower bound on the execution time of a batch.

The simulations were done on two types of batches, in the first, called random, ins and outs were distributed evenly on all ports. In the second, called biased, the batch has more outs than ins (ratio 3:1) in its first half, and more ins than outs (also ratio 3:1) in the second half. The execution times for two batches (one of each kind) are shown in figure 6 (the execution times for the biased batches show considerable variation, but the trend is the same in all the simulations that have been performed). For the random batches the height of the rubber-wire is not significant. One would expect that most of the time there are relatively few elements in the store, because in and out operations are in balance. The benefit of

the rubber-wire is to absorb bursts of out operations, where the rubber-wire works as a buffer. This is clearly demonstrated by the simulation results of the biased batches.

The conclusion from these preliminary simulations is that the rubber-wires effectively smooth out variations in the insertion rates. The height of rubber-wires needed does of course depend on the expected variations, but a few levels (< 10) seems to be sufficient for realistic batches.

6 VLSI Implementations

The previous sections have shown how the functional behavior of a circuit may be specified, analyzed, and verified using "Synchronized Transitions." This facilitates exploring a design at a high level of abstraction which is very important for evaluating capabilities and trade-offs. Just as "Synchronized Transitions" specifications can be transformed to program prototypes, they can also be used to derive the final physical implementation (layout). "Synchronized Transitions" is not tied to a specific type of circuit. For example, it is possible to transform a "Synchronized Transitions" program into either a synchronous or an asynchronous/self-timed circuit. We will illustrate this by deriving a synchronous implementation of a storage-ring with rubber-wires. The use of "Synchronized Transitions" with self-timed circuits is presented in [6].

The synchronous design is based on a two-phase clock. Each clock phase is controlled by a global signal available in all transitions. The clock signals alternate between high and low, and the high periods of the two clocks are guaranteed not to overlap. The essential property of the clocks is this period of non-overlap which separates computations that must not be performed simultaneously. Consider the following transition:

$$\mathtt{t} :\ll \mathtt{C}^{\mathtt{t}} \rightarrow \mathtt{l}^{\mathtt{t}} := \mathtt{F}(\mathtt{r}^{\mathtt{t}}) \gg$$

The reading and writing of state variables may be separated by doing the reads during the high interval of one phase (called φ_1 pulses) and the writes on the complementary clock pulses (called φ_2). The transition t can be implemented as follows using a two-phase non-overlapping clock:

$$\ll \varphi_1 \rightarrow \mathtt{temp}, \mathtt{enable} := \mathtt{F}(\mathtt{r}^{\mathtt{t}}), \mathtt{C}^{\mathtt{t}} \gg$$
$$\ll \varphi_2 \wedge \mathtt{enable} \rightarrow \mathtt{l}^{\mathtt{t}} := \mathtt{temp} \gg$$

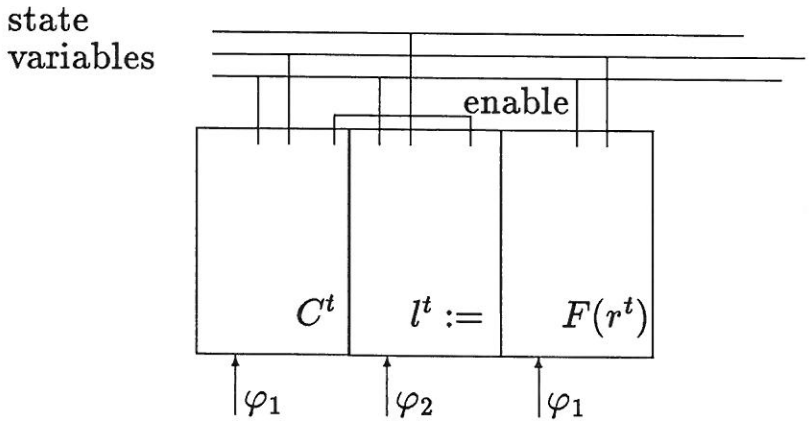


Figure 7: Two-phase implementation

This is called a **two-phase implementation.** The direct implementation of this pair of transitions is shown in figure 7. In the two-phase implementation, there is a block such as the one shown in figure 7 for each transition. The state variables are implemented as wires connected to all the transitions using them.

6.1 Implementation conditions

To transform a "Synchronized Transitions" program to a circuit (using the two-phase implementation described above), state variables are realized as wires (connected to latches) and transitions as combinational functions. However, a mechanical application of such transformations can result in a circuit which does not exhibit the same behavior as the "Synchronized Transitions" program. This is because transitions may be performed simultaneously by the hardware in a way that violates the atomicity of the "Synchronized Transitions" specification. For example, two circuits could simultaneously attempt to assert different values on a wire (state variable); whereas, an atomic execution assumes that these two writes of the variable would happen one at a time. Likewise, in the hardware, a value could be changed by one circuit while it was being used by another resulting in an undefined value, but the atomic model assumes a distinct sequence of reads and writes, so that each transition reads consistent and well-defined values. We chose to restrict the specifications to ones for which such anomalous behaviors cannot occur and mechanical transformations from specifications to circuits are valid. Such restrictions are called **implementation conditions.** In the design pro-

cess, the "Synchronized Transitions" program is shown to satisfy a set of implementation conditions, and the implementation is guaranteed to be correct.

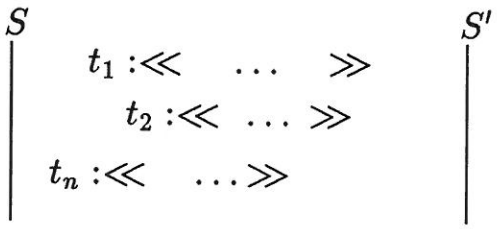
In the two-phase implementation, *all* enabled transitions are performed in every clock cycle. This is a correct way of implementing a "Synchronized Transitions" program where there are never dependent transitions enabled simultaneously. Two transitions are *dependent* if one writes a state variable which the other reads or writes. This implementation condition can be defined more precisely as follows: Let $R^i = \{r_1, r_2, ..., r_{k_t}\}$ be the set of variables read by a transition $\mathtt{t}^i$, and $W^i = \{w_1, w_2, ..., w_{m_t}\}$ be the set of variables written by $\mathtt{t}^i$.

Concurrent Read Exclusive Write Condition (CREW):
For all distinct pairs of transitions instantiations $\mathtt{t}^1$ and $\mathtt{t}^2$:

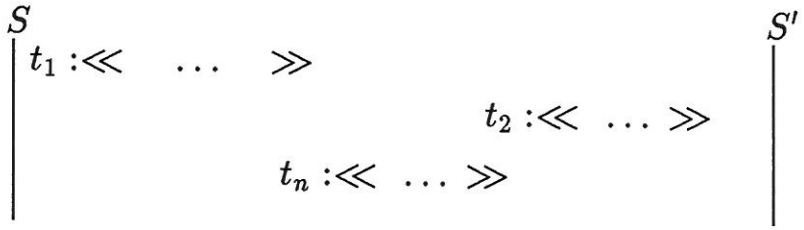
$$(R^1 \cap W^2 \neq \emptyset) \vee (R^2 \cap W^1 \neq \emptyset) \vee (W^1 \cap W^2 \neq \emptyset) \Rightarrow \neg(\mathtt{c}^1 \wedge \mathtt{c}^2)$$

The CREW condition says, that it must not be possible to satisfy the precondition of a transition while another dependent transition is enabled. Establishing $\neg(\mathtt{c}^1 \wedge \mathtt{c}^2)$ may require more analysis than just a local argument using boolean algebra. Typically, an invariant capturing some global property of the "Synchronized Transitions" program is needed.

It is important to note *that this condition is applied to the "Synchronized Transitions" program.* Because the description is based on a model of atomic transitions, established proof techniques such as invariants are directly applicable to demonstrate that the conditions are satisfied [4]. The implementation, on the other hand, may be constructed from hardware which does not preserve atomicity; however, such an implementation will preserve the appearance of atomicity (i.e., the possible behaviors of the implementation will correspond to possible atomic executions). Starting in a certain state S, a two-phase implementation of a description satisfying CREW can only lead to states S', which could also have been obtained by a sequence of atomic transitions starting from S. To see this, consider one full clock cycle (both phases) starting in a state S and leading to a state S'.



Let $(s_1, s_2, ..., s_m)$ be the state variables written in this clock cycle. Because it is assumed that CREW is satisfied, each is written by exactly one transition, t_i, and none of them are read by any of the other transitions (t_1, t_2, ..., t_{i-1}, t_{i+1}, ..., t_n). Hence, writing $(s_1, s_2, ..., s_m)$ (by the same transitions) in some serial order (atomically) cannot lead to a different state than S'.



Thus, any state which is obtained by a two-phase implementation of a program satisfying CREW can also be obtained by performing the transitions one at a time (atomically). This means that the two-phase implementation is sound. Therefore, CREW is a *sufficient* condition for using the two-phase implementation, but it is not necessary. There are other (weaker) conditions that allow transitions to be performed simultaneously and still have valid two-phase implementations. The argument given above indicates what such a weaker condition could be, namely one which ensures that when a set of transitions is performed simultaneously, the same result could have been obtained by executing the transitions atomically. Any condition which ensures the existence of such a sequence is sufficient.

Even when CREW is satisfied, there may be physical problems in a two-phase implementation. For example, the duration of a clock phase may be to short for all signals to propagate through the circuit and for transistors to switch. Such physical problems are not discussed in further detail here.

6.2 Transformations to satisfy CREW

The CREW condition is a rather strong condition and many programs may not satisfy it immediately. However, in this section, a few transformations are shown which can be used to obtain a program satisfying CREW.

Consider two transitions not satisfying CREW.

$$\mathtt{t}^1 : \ll \mathtt{C}^1 \rightarrow \mathtt{A}^1 \gg$$
$$\mathtt{t}^2 : \ll \mathtt{C}^2 \rightarrow \mathtt{A}^2 \gg$$

Hence, $\mathtt{t}^1$ and $\mathtt{t}^2$ are dependent and the conjunction: $\mathtt{c}^1 \wedge \mathtt{c}^2$, may be satisfied. Somewhere, a decision must be made about which transition to perform when both preconditions are satisfied. Let us represent the outcome of this decision by a boolean, b. If b is true, the first transition is performed, otherwise the second. The two transitions can be transformed as follows:

$$\mathtt{t}^1 : \ll \mathtt{C}^1 \wedge (\neg \mathtt{C}^2 \vee \mathtt{b}) \rightarrow \mathtt{A}^1 \gg$$
$$\mathtt{t}^2 : \ll \mathtt{C}^2 \wedge (\neg \mathtt{C}^1 \vee \neg \mathtt{b}) \rightarrow \mathtt{A}^2 \gg$$

Now the conditions are disjoint, and the transitions may be implemented as above. Regarding b, there are several alternatives. The simplest is to make a static choice about which transition should be performed, e.g., if b is statically true:

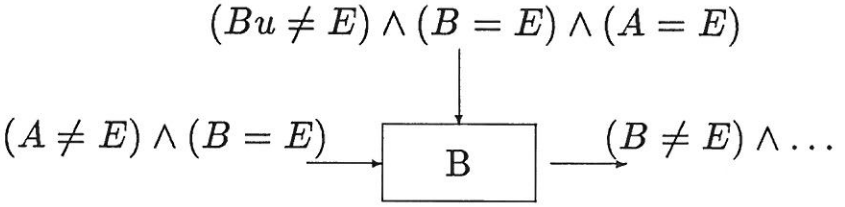
$$\mathtt{t}^1 : \ll \mathtt{C}^1 \rightarrow \mathtt{A}^1 \gg$$
$$\mathtt{t}^2 : \ll (\mathtt{C}^2 \wedge \neg \mathtt{C}^1) \rightarrow \mathtt{A}^2 \gg$$

If this is not adequate, one can introduce a boolean variable assigned in the two transitions as follows:

$$\mathtt{t}^1 : \ll \mathtt{C}^1 \wedge (\neg \mathtt{C}^2 \vee \mathtt{b}) \rightarrow \mathtt{A}^1, \mathtt{b} := \mathtt{FALSE} \gg$$
$$\mathtt{t}^2 : \ll \mathtt{C}^2 \wedge (\neg \mathtt{C}^1 \vee \neg \mathtt{b}) \rightarrow \mathtt{A}^2, \mathtt{b} := \mathtt{TRUE} \gg$$

Finally, b could also be implemented by circuitry producing a random value; then, the choice of which transition to perform is non-deterministic.

The rubber-wire may be used to illustrate how CREW is used. Consider the transition operating on one of the B-registers.



The last conjunct on the vertical transition, $\ldots \wedge (A = E)$, is included to satisfy CREW. It is an example of the static choice mentioned above.

7 Synchronous Implementation

The storage-ring of sections 3 and 4 can be implemented using the methods just described. The derivation process offers many opportunities for refining the design; for example, the storage-ring presented in this section is based upon bit serial communication. The bit-serial approach was chosen to simplify wiring both on and off chip and to reduce the number of pins required. The bit serial design is assumed to be controlled by a global clock and all enabled transitions are performed in each clock cycle (two-phase implementation). The global control signal, `transfer`, is true for ww consecutive clock cycles (to enable a shift from one register into another). So, ww is the word width of a register. The signal `transfer` is false for exactly one clock cycle (in which state changes take place). The transfer of values from one register to another may be described as follows:

```
TRANSITION copy(from, to: register);
<< transfer >> * << to.r[ww-1]:= from.r[0] >>
        * {0 <= i < ww-1:  << to.r[i] := to.r[i+1] >> }
        * {0 <= i < ww-1:  << from.r[i] := from.r[i+1] >>  }
```

This transition is the only one which needs to be different in the descriptions of a bit serial and a bit parallel design. In figure 8, a detailed description of a bit serial version of the rubber-wire is given. The other parts are similar. Note that this program assumes that the two-phase implementation is used.

This specification of a rubber-wire satisfies CREW; thus, it can be implemented with a two-phase clocked implementation as described in the preceding section. It would be too space consuming to give a description of how a complete layout is derived; however, a small example will illustrate the technique. Consider the following transitions (part of a rubber-wire):

```
TRANSITION enable(from, to: register);
<< from.s AND NOT to.s >>
```

```
CONST
  ww = ... ;    (* # bits in an element (word width) *)
TYPE
  element  = ARRAY [0..ww-1] OF bit;
  register = RECORD
               s: BOOLEAN;   (* TRUE = full, FALSE = empty *)
               r: element;
             END;

  TRANSITION enable(from, to: register);
  << from.s AND NOT to.s >>

  TRANSITION newstate(from, to: register);
  << NOT transfer >> * << from.s, to.s := F, T >>

  TRANSITION copy(from, to: register);
  << transfer >> * << to.r[ww-1]:= from.r[0] >>
      * {0 <= i < ww-1:  << to.r[i] := to.r[i+1] >> }
      * {0 <= i < ww-1:  << from.r[i] := from.r[i+1] >>  }

  TRANSITION wire(from, to: register);
  enable(from, to) * ( newstate(from, to) || copy(from, to) )

  CELL rubber_wire(A, B: register; STATIC d: INTEGER)
   (*          Au --rubber_wire--> Bu
               |                   |
               |                   |
               A ----------------> B            *)
     STATE Au, Bu: register;
  BEGIN
     wire(A, B)                  ||
     wire(A, Au) * < B.s >       ||
     wire(Bu, B) * < NOT A.s >   ||
     {d>1: rubber_wire(Au, Bu, d-1), d=1: wire(Au, Bu) }
  END rubber_wire.
```

Figure 8: Bit serial description of rubber-wire

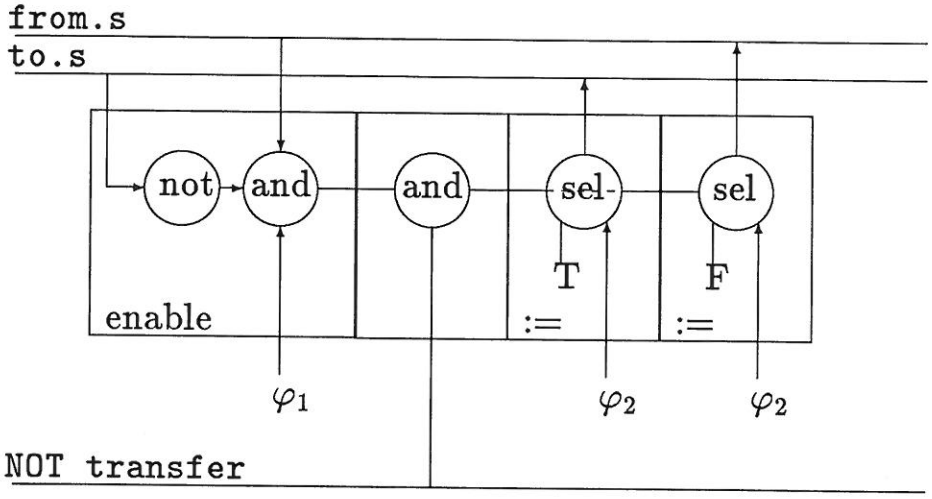


Figure 9: Two-phase implementation of newstate transition

```
enable(from, to) * << NOT transfer >> *
                   << from.s, to.s:= F, T >>
```

The layout for the last transition (part of the wire in figure 8) follows the outline given in figure 7 with standard building blocks for each major part of a transition (precondition, left hand side of :=, and right hand side of :=), see figure 9. In figure 10, a layout of the circuit in CMOS is shown. A complete layout of a rubber-wire following this approach has been completed.

8 Conclusion

This paper has introduced a high-level language, "Synchronized Transitions," for describing VLSI designs. The following key points have been stressed in this paper.

- A VLSI chip is described as a massively parallel computation.
- A systematic derivation of a VLSI design from a high-level program.
- Restrictions applying to a particular technology are formulated as implementation conditions which can be checked on the high-level description.

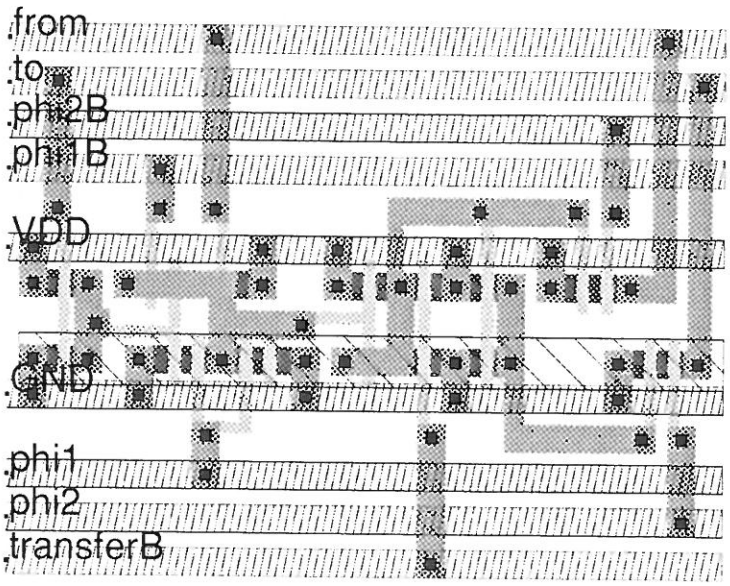


Figure 10: Layout of newstate transition

- Use of program prototypes for early experimentation/simulation of a design.

Another important property is the potential for formal verification of designs, e.g., invariance proofs. This has been described in a separate paper [4]. An important application of this is doing automatic check of implementation conditions such as CREW.

The detailed design of a so-called storage-ring was used to illustrate the "Synchronized Transitions" language. A storage-ring is a specialized store aimed at representing a problem-heap [11] or a tuple space [3]. Both of these concepts have been used to program a variety of different multiprocessor algorithms. The motivation for studying a specialized problem-store is to achieve speed improvements beyond what is possible on general purpose multiprocessors.

Acknowledgements

Anders P. Ravn provided valuable insight and inspiration in the early phases of the development of "Synchronized Transitions." The first author is grateful to "The University of Washington", Seattle for inviting him as a visitor, supported in part by the National Science Foundation under Grant No. CCR-8619663, and providing excellent conditions for undertaking most of the work presented in this paper.

References

[1] F. Barrett, **Problemhobe og VLSI**, Masters Thesis (in Danish), Computer Science Department, Aarhus University, Denmark, October 1986.

[2] K.M. Chandy and J. Misra, **A Foundation of Parallel Program Design**, Prentice Hall, 1987.

[3] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," **ACM Transactions on Computer Systems 4, 2**, May 1986.

[4] S. Garland, J. Guttag and J. Staunstrup, Verification of VLSI circuits using LP, **Proceedings of the IFIP WG 10.2 Worshop on 'Design for Behavioural Verification'**, North Holland 1988. unpublished

[5] A. Gottlieb, B. Lubacevsky and R. Rudolph, "Basic techniques for the efficient coordination of very large number of cooperating sequential processors," **ACM Transactions on Programming Languages 5, 1**, January 1983.

[6] M.R. Greenstreet, T.E. Williams, and J. Staunstrup, "Self-Timed Iteration," in **Proceedings from VLSI-87**, Vancouver, North Holland 1987.

[7] D.D. Hill and D.R. Coelho, **Multi-level Simulation for VLSI Design**, Kluwer Academic Publishers, 1986.

[8] C.A.R. Hoare, "Communicating Sequential Processes," **Communications of the ACM**, vol. 21, no. 8 (August 1978), pp. 666-677.

[9] H.T. Kung and C.E. Leiserson, "Systolic Arrays (for VLSI)," in **Sparse Matrix Proc.**, 1978, I.S. Duff and G.W. Stewart (eds.), pp. 256-282, SIAM 1979

[10] J.D. Morison, *et al.*, "ELLA: Hardware Description or Specification," in **Proceedings of 1984 IEEE ICCAD**.

[11] P. Møller-Nielsen and J. Staunstrup, "Problem-heap. A paradigm for multiprocessor algorithms," **Parallel Computing 4**, February 1987, North Holland.

[12] J. Staunstrup, F. Barrett, M. Greenstreet and P. Møller-Nielsen, "The Design of a Problem-mesh," **Proceedings from Comp-Euro 87, VLSI and Computers**, IEEE 1987.

[13] "1986 Survey of Logic Simulators," **VLSI Systems Design**, Vol. 4, no. 2, Feb. 1986, pp. 32-40.

[14] 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.