Logic Transformations for Synchronous Logic Synthesis

Giovanni De Micheli and Roger Yip
Computer Systems Laboratory
Stanford University

Abstract

This paper presents a new approach to logic synthesis of digital synchronous sequential circuits. We describe here algorithms for minimizing i) the area of synchronous combinational and/or sequential circuits under cycle time constraints and ii) the cycle time under area constraints. Previous approaches attacked this problem by separating the combinational logic from the registers and by applying circuit transformations to the combinational component only. We show in this paper instead how to optimize concurrently the circuit equations and the register position by a set of algorithms based on logic transformations. A computer implementation of the algorithms in program Minerva is described and experimental results are reported.

1 Introduction

Logic synthesis has shown to be of pivotal importance in the computer-aided design of integrated circuits. Logic synthesis systems have been the object of extensive investigation and commercial implementations have shown to be practical for product-level design of digital circuits.

Most circuits of interest in digital design are **synchronous** logic circuits, that are interconnections of logic gates and registers with synchronous clocking. Feedback connections are restricted to be through synchronous registers, to guarantee race-free design. Semi-custom circuit implementations, such as standard-cells and sea-of-gates, have motivated the use of multiple-level (or multiple-stage) logic synthesis techniques. In particular, such implementations have shown to be more flexible and faster than two-level implementations, such as Programmable Logic Arrays. As a result, several techniques for multiple-level logic synthesis techniques have been investigated and clever algorithms for combinational logic synthesis have been reported in the literature [1] [2] [3] [4] [5].

However, techniques for synthesizing synchronous logic circuits have been lagging behind, due to the additional complexity of handling registers and feedback connections. Most logic synthesis systems deal with such circuits by partitioning them into an interconnection of a combinational logic component and registers. The combinational portion of the circuit is optimized by combinational logic algorithms. Then registers are added back to the circuit. Needless to say, such optimization techniques are limited in their scope by this partitioning strategy. Recently dynamic partitioning techniques have been proposed [6], aiming at extracting the largest portion of a synchronous circuit that can be dealt with by combinational logic techniques.

We attempt in this paper to solve the synchronous logic synthesis problem by considering algorithms that operate on the entire sequential circuit, i.e. that do not separate registers from the combinational component. For this reason, we introduce the concept of synchronous Boolean network and we study transformations on this network that preserve I/O equivalence and that optimize i) the circuit area under cycle time constraints and ii) the cycle time under area constraints. Some of these transformations are extensions of those used in combinational logic synthesis and operate within and across the register boundaries. Therefore a logic synthesis package that exploits these extended transformations in conjunction with those available in the combinational logic synthesis programs [2] [4], may synthesize circuits whose quality is at least as good as that obtained by the previous techniques.

The register position is determined as a by-product of these circuit transformations. It is important to remember that a technique to position the registers in a network, called **retiming**, was introduced by Leiserson and Saxe [7] in a different context, where logic synthesis transformations were not considered. This paper presents a model for synchronous logic synthesis that combines retiming with combinational logic synthesis techniques. Then algorithms that minimize the circuit area and cycle time are described. The algorithms are implemented in computer program Minerva, that performs combinational and sequential logic synthesis. Experimental results are then reported.

2 Basic concepts and definitions

We consider synchronous circuits that are interconnections of combinational logic gates and clocked registers. We assume first that all the registers are driven by one clock (i.e. single phase circuits) and that the latching is always positive (or always negative) edge-triggered. (Master-slave registers consisting of a cascade interconnection of latches gated by the clock and its complement fall in this class.) We assume that the clock has a period T (cycle time), and that the clock skew, the register setup, hold and propagation times are negligible. We will remove these simplifying assumptions in Section 5.

We model synchronous circuits by **synchronous Boolean networks**. A synchronous Boolean network is described in terms of Boolean variables and Boolean functions. Each Boolean variable corresponds to either a primary input/output of the circuit or to the output of a combinational logic gate. A positive integer label on a variable (superscript) denotes the synchronous register delay, if any, of the corresponding signal with respect to the primary input or combinational logic gate that generates it. Zero-valued labels are omitted for the sake of simplicity. Each Boolean function specifies the value of a variable in terms of other variables, i.e. it is a multiple-input single-output combinational logic function. It is represented by an equation, whose left term is a variable with zero-valued label and whose right term is an expression, e.g. the equation at vertex v_i is represented by $i = \mathcal{I}$, where $\mathcal{I}$ is a Boolean expression in terms of other (labeled) variables.

The network is modeled by the **synchronous network graph,** that is a directed weighted multi-graph $G(V, E, W)$, whose vertex set $V = V^I \cup V^G \cup V^O = \{v\}$ is in one-to-one correspondence with the variables corresponding to the set of primary inputs, logic gates and primary outputs respectively. The edge set E and the edge weight set W are defined as follows. There is an edge between v_i and v_j with weight k when variable i appears in the expression $\mathcal{J}$ for vertex v_j with label k. Zero-valued weights are not indicated by convention. There is a (weighted) edge to each output vertex in V^O from the vertex in V^G corresponding to the gate generating that output signal. For each pair of vertices joined by a path in $G(V, E, W)$, the **path weight is the sum of the weights along the path. We assume that each** cycle (i.e. closed path) has strictly positive weight, to model the restriction of breaking combinational logic cycles by at least one register. An example

0073-1129/90/0000/0407$01.00 © 1990 IEEE

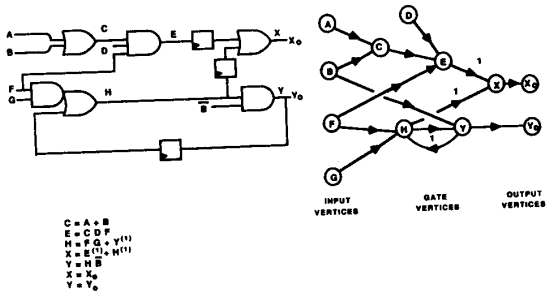

Figure 1: Synchronous Boolean Network and its representation.

of a synchronous Boolean network and its representation is shown in Fig. 1.

In general, a synchronous Boolean network may have cyclic dependencies, i.e. its corresponding graph be cyclic. A network is called **unidirectional** when the graph $G(V, E, W)$ is acyclic. It models a pipelined combinational circuit. Note that the combinational Boolean network (without synchronous registers) introduced by Brayton [1] is just a special case of the synchronous Boolean network that is acyclic and whose labels are all zeroes.

The (direct) **fanin** set of a vertex v_i is the subset of vertices that are tail of an edge (with zero weight) incident to v_i and it is denoted by $FI(v_i)$ ($DFI(v_i)$). Similarly the (direct) **fanout** set of a vertex v_i is the subset of vertices that are head of an edge (with zero weight) incident to v_i and it is denoted by $FO(v_i)$ ($DFO(v_i)$). Each vertex of the graph $v_i \in V^G$ (i.e. corresponding to a gate) has as attributes an area estimate l_i in terms of literal count [1] and a positive propagation gate delay d_i. Each input and each output vertex has zero delay.

The propagation delay model captures the difference in speed of gates implementing various Boolean expressions. Therefore it is a function of the structure of the Boolean expression. For example, in the case of CMOS, such a structure is characterized by the maximum number of N-type and P-type devices in series. The delay function is assumed to be always a monotonically increasing function of l_i. It is important to remark that an accurate gate propagation delay model should include loading factors and device sizes. We assume that the choice of device sizes for a gate is done in a successive stage of logic design, the technology mapping, so that it compensates for loading factors. Therefore this propagation delay model includes an average loading factor. [1]

Each vertex v_i has a **data ready time** t_i, that is the time at which the signal generated by the corresponding gate is ready with respect to the clock edge [8]. We assume the primary inputs to be synchronized to the clock positive edge and therefore their data ready time is zero. For any other vertex v_i, the data ready time is the sum of its propagation delay d_i to the largest data ready time of its inputs that are not registers, i.e. $t_i = d_i + max_{v_j \in DFI(v_i)}(t_j)$. Since the subgraph representing the direct fanin relation is acyclic, the data ready time can be computed by topological sort.

Given a cycle time T, a synchronous network is a **timing-feasible** implementation if all the data ready times are bounded from above by the cycle time, i.e. $T \geq max_{v_i \in V}(t_i)$. Each vertex v_i has a **slack** s_i representing the additional delay that the vertex can tolerate while preserving timing-feasibility of the network for a given T [8]. In a timing-feasible network a vertex is **critical** if its slack is null.

[1] Our model has shown to be accurate enough when the loading factors is between one and four gates. The cases in which the loading factor is higher are rare, and need to be addressed by using buffering schemes for high performance design. Therefore modeling delay in the case of large loading factors becomes irrelevant.

The area taken by a network implementation depends on the total number of literals and registers required. For each variable i, let m_i be the maximum of the labels that the variable takes in the network representation. Then m_i represents the number of synchronous registers that are connected in cascade at the output of the corresponding gate. An area estimate can be computed as: $A = \alpha \sum_{v_i \in V^G} l_i + \beta \sum_{v_i \in V} m_i$, where α and β are coefficients taking into account the relative area cost of a literal and a register. Given an area bound A_{max}, a network is an **area-feasible** implementation if $A_{max} \geq A$, and it is a **feasible** implementation if it is both area-feasible and timing-feasible.

3 Logic transformations in synchronous logic synthesis

The problem of minimizing the area (cycle time) of a synchronous Boolean network implementation, possibly under cycle time (area) constraints, is difficult and no efficient exact solution method is known. Most techniques for multiple-level logic optimization are based on network transformations, that preserve the I/O equivalence of the network, and achieve area/time optimal solutions with respect to some local criterion. Transformations are classified as **local** and **global**. Transformations are said to be local when they modify the representation of a Boolean function at a network vertex at a time (e.g. factoring or Boolean simplification). Such transformations have been presented in [1] [2] for combinational logic synthesis and can be used (without significant extensions) in synchronous logic synthesis, because they do not depend on the network model. Global transformations target more than one vertex at a time and attempt to improve the network by restructuring the global interconnections (e.g. elimination, resubstitution and extraction). We consider here global transformations extended to synchronous logic synthesis in relation with network retiming.

Retiming [7] is a technique that determines a register assignment in a network (i.e. a set of weights in $G(V, E, W)$) so that it is a feasible implementation for a given cycle time T, if such an assignment exists. [2] In our context, the retiming of variable i by an integer r corresponds to adding r to its label, and the retimed variable is denoted by $i^{(\cdot+r)}$, where the dot in the superscript represents the label of variable i before retiming (e.g. for variable i with label 2, fully denoted by $i^{(2)}$, a retiming by $r = 3$ yields $i^{(2+3)} = i^{(5)}$). Similarly, the retiming of an expression $\mathcal{I}$ by an integer r corresponds to adding r to the labels of all its operands and it is represented by $\mathcal{I}^{(\cdot+r)}$. The positive (negative) retiming of a gate vertex v_i by r_i is the shift of r_i register delays from its outputs (inputs) to its inputs (outputs). It corresponds to retiming by r_i the expression $\mathcal{I}$ of v_i and to retiming by $-r_i$ the variable i in the expressions of the vertices of $FO(v_i)$. The retiming of an input vertex is just the retiming by $-r_i$ of the variable i in the expressions of the vertices of $FO(v_i)$. The retiming of an output vertex is just the retiming by r_i of the expression $\mathcal{I}$ of v_i. An example is shown in Fig. 2.

It was shown in [7] that retiming preserves the I/O behavior of the network, provided that the original network and the retimed one can be set in a corresponding initial state. This requirement can be satisfied by assuming that the registers of interests are controllable by an appropriate primary input signal, i.e. that the networks are designed to have a reset condition. Note that since labels cannot be negative by definition, the retiming of a vertex is valid only for some restricted values of r_i. A retiming of the vertices of a Boolean network is **feasible** for a cycle time T, if the retimed network is a timing-feasible implementation with non-negative labels and I/O equivalent to the original network.

Leiserson and Saxe proposed an algorithm in [7] that searches for the minimum T for which such an assignment exists. The corresponding networks are said to be **optimal with respect to retiming**. If this technique were the only available to optimize the cycle time, then its result would represent a global optimum solution. However retiming does not change the structure of the network (i.e. the vertex and edge sets in $G(V, E, W)$), and therefore better results may be achieved by combining it with other transformations

[2] There are different variations of Leiserson and Saxe's retiming algorithm. A retiming algorithm can also be used to find the register assignment that minimizes the area cost of the network, when this is measured in terms of registers only. We refer to retiming in this paper according to its original notion of a technique that determines a register assignment in a network so that it is a feasible implementation for a given cycle time T, if such an assignment exists.

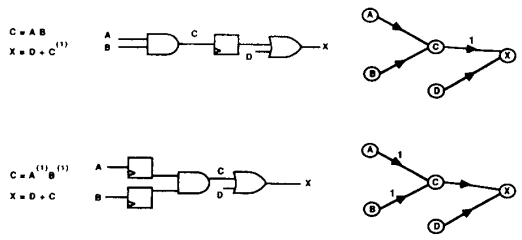

Figure 2: Retiming vertex v_c by +1.

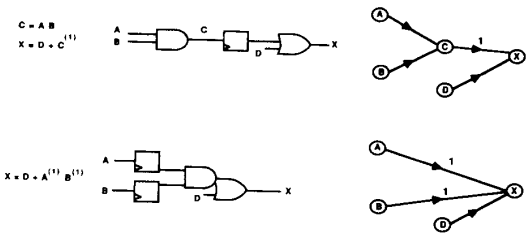

Figure 4: Resubstitution of v_x into v_y.

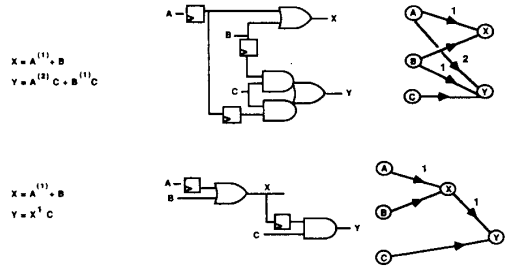

Figure 3: Elimination of vertex v_b .

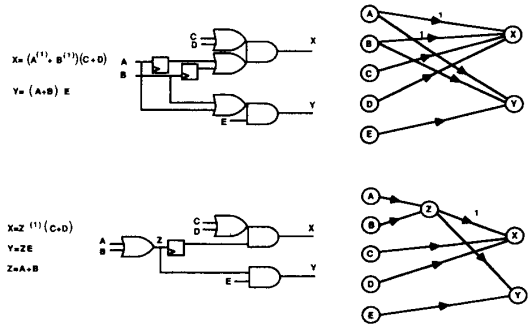

Figure 5: Extraction of v_z.

that modify the network structure. For this reason we consider here the following transformations.

The **elimination** of a variable with label k is the replacement of the variable by its corresponding expression retimed by k. Given two gate vertices v_i and $v_j \in FI(v_i)$, the elimination of v_j into v_i is the elimination of variable j in all its occurrences in the expression $\mathcal{I}$ for v_i (Fig. 3). The elimination of vertex v_j is its elimination into all the vertices in $FO(v_j)$. Note that the elimination of a variable with label zero is equivalent to the elimination used in combinational logic synthesis [1] [2]. The elimination of a variable with non-zero label corresponds to merging two logic gates that are separated by a register, by shifting the register to the inputs of the gate corresponding to the variable being eliminated.

Let $\mathcal{I}, \mathcal{J}, \mathcal{Q}$ and $\mathcal{R}$ be Boolean expressions. Then $\mathcal{J}$ is a **synchronous divisor** of $\mathcal{I}$ if $\exists r \geq 0$ such that $\mathcal{I} = \mathcal{J}^{(+r)}\mathcal{Q} + \mathcal{R}$ and $\mathcal{J}^{(+r)}\mathcal{Q} \neq 0$. Note that the product $\mathcal{J}^{(+r)}\mathcal{Q}$ may have the algebraic or Boolean flavor, as defined in [1]. Given two gate vertices v_i and v_j such that the expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{I}$, the **resubstitution** of v_j into v_i is the factoring of $\mathcal{I}$ as $j^{(+r)}\mathcal{Q} + \mathcal{R}$. Note again that the divisors defined in [1] are a subset of the synchronous divisors and therefore resubstitution with null retiming (i.e. $r = 0$) is equivalent to resubstitution in combinational logic. The resubstitution of a variable with non-zero retiming corresponds to adding one (or more) register between two gates to simplify the latter (Fig 4).

The **extraction** of a common sub-expression of expressions $\mathcal{I}$ and $\mathcal{J}$ corresponding to two vertices v_i and v_j is the addition to the network of a vertex v_l (with the related edges) corresponding to a common synchronous divisor of $\mathcal{I}$ and $\mathcal{J}$ and to the factoring of $\mathcal{I}$ and $\mathcal{J}$ in terms of the new variable l (Fig 5).

There are different ways of decomposing a Boolean expression. In this paper we define **decomposition** of an expression $\mathcal{I}$ its replacement by the expression: $j^{(+r)}\mathcal{Q} + \mathcal{R}$, where j is a new variable, its corresponding expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{I}$ and $j^{(+r)}\mathcal{Q} \neq \emptyset$. The decomposition of a vertex v_i implies the addition to the network of vertex v_j (Fig 6). Decomposition can be applied recursively to v_i and v_j.

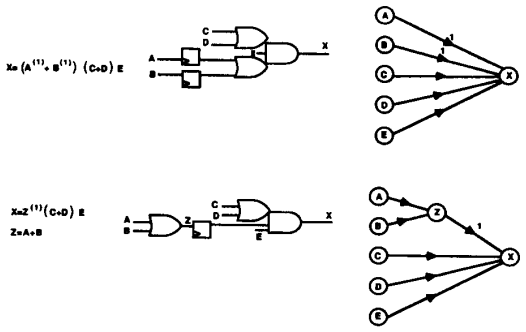

Figure 6: Decomposition of v_i.

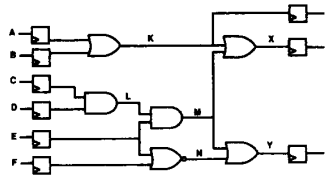

Figure 7: Transformation in a critical path.

4 Algorithms for synchronous logic synthesis

We consider here algorithms for optimizing digital networks according to four major strategies: area minimization without/with cycle time constraints and cycle time minimization without/with area constraints. We concentrate here on logic transformations that operate across register boundaries, because transformations on combinational networks have been extensively described [1] [2] [4] [3] [8]. Nevertheless the techniques described here apply to combinational networks and to registers-less portions of synchronous logic networks as well.

While the details of the logic transformations are presented in the following subsections, we would like to comment here on the general strategy in applying the transformations to achieve a given goal. We conjecture that the problems of optimal synchronous logic synthesis is at least as difficult as the problem of finding optimal combinational logic networks. Therefore heuristic optimization is done as in combinational logic synthesis by iterating an operator on a network (i.e. a set of transformations) until local optimality with respect to this operator is found. Then a different operator is applied.

For area minimization, the general frame of the algorithm is as follows. Vertices of the synchronous logic networks are examined in pairs and a transformation is applied if suitable.

```
transform {
   While (some candidate pair is found) {
      (v_i, v_j) = select-candidates;
      If (constraints are satisfied) transform(v_i, v_j);
   };
}
```

The selected candidates are such that the chosen transformation can be applied to them with a decrease of the figure of merit of interest. Consider for example the problem of unconstrained area minimization. Then, the candidate selection is driven by the variation of the estimate of the area cost $\delta_A = \alpha\delta_l + \beta\delta_m$, where δ_l is the variation in the number of literals and δ_m is the variation in the number of registers. The computation of δ_l and δ_m is specific to a transformation, and therefore it will be detailed in the sequel.

The problem of minimizing area under timing constraints is approached under the assumption that a timing-feasible network is given, whose area estimate we want to minimize. We constrain the transformations to preserve timing-feasibility and therefore we reject candidates whose transformation would lead to a non timing-feasible configuration. In the case of combinational networks, a necessary and sufficient condition for preserving timing-feasibility was shown to be that any increase of the data ready time of any vertex be bounded by its slack [8]. While the sufficiency of this condition still holds in synchronous logic synthesis, its necessity no longer does. Indeed, a transformation followed by retiming may preserve timing-feasibility and therefore a retiming of a network is attempted before rejecting a transformation.

Example: Consider the circuit of Fig. 7.

Assume that the cycle time is set equal to the propagation delay through the longest path, say the path (v_f, v_n, v_y). Suppose, for example, that we want so reduce the circuit area, by eliminating v_l into v_m. It may be the case that the increased propagation delay through v_m introduces a longer critical path (v_e, v_m, v_x), or equivalently that the slack at v_x becomes negative. If the position of the register storing x is fixed, then the elimination has to be rejected. Otherwise it may be possible to find a feasible retiming (for example by trying to retime v_x by $+1$) so that the elimination can be accepted •

The problem of minimizing the cycle time T, is approached by generating a sequence of networks that are timing feasible for decreasing values of T [8]. For each network in this sequence the critical vertices are identified, and transformations are applied to such vertices. It is important to detect whether the transformations affect the optimality with respect to retiming. If this is not the case, then the network cycle time can be further reduced by retiming.

In addition, when area constraints are enforced, transformations are subject to the additional check that the area bound A_{max} is not violated. Therefore, an area cost for each transformation ($\delta_A = \alpha\delta_l + \beta\delta_m$) is computed and added to the current value of A. If the result is larger that A_{max} the candidates are rejected.

In the following sections we describe the network transformations in detail. Because of their interrelations with retiming, we describe first an implementation of the retiming algorithm that suits the synchronous network model and supports incremental changes in a network.

4.1 Retiming

The following algorithm can be used to check whether a synchronous network implementation is feasible for a given cycle time T. It is derived from an algorithm described in [9] for networks without multiple I/O vertices, and it differs by having the subroutine *set-outputs*, that is not present in the original algorithm. In this paper we are concerned with networks with multiple I/Os, under the assumptions that all inputs are synchronous to the system clock. Such model better conforms to synchronous digital circuits that need to be interconnected among each other. It is important to note that a retiming of an output vertex increases all the path weights from the inputs to that vertex. In this case, if the graph $G(V, E, W)$ is connected, a necessary condition to preserve equivalence is to delay all the other outputs (to keep them in phase with the retimed output) and to recover the delay by

subtracting a register delay from all the inputs.

```
retime {
  For (k = 1; k = |V|; k++) {
    Compute t_i for each vertex v_i ∈ V;
    M = {m | t_m > T};
    If(M = ∅)
      return (TRUE);
    else {
      If(exit)return (FALSE);
      Retime by 1 all vertices in M;
      set-outputs;
    }
  }
}

Set-outputs {
  If ( ∃m ∈ M | v_m is a primary output) {
    Retime by 1 all primary output vertices not in M;
    S = {v ∈ V | ∃ a zero weight path from an input vertex to v};
    Retime by 1 all vertices in S;
  }
}
```

Assume that procedure $exit$ returns true when $k = |V|$. It is obvious that procedure $set\text{-}outputs$ returns immediately in the case that the network has no primary output explicitly defined, as in [7] [9]. The following theorem applies to such networks.

Theorem 1: [9] Given a cycle time T, algorithm $retime$ returns TRUE iff a feasible retiming exists•

Let us consider now synchronous Boolean networks with multiple I/Os. Assume again that procedure $exit$ returns true when $k = |V|$. It can be easily noted that when the algorithm returns TRUE, a feasible retiming is constructed by the algorithm such that all the data ready times are bounded by the cycle time. Since every time that a primary output vertex is retimed, all the other outputs and all the inputs vertices are retimed, then the length of all the I/O path is preserved. In addition, since $t_m > T$ implies $t_i > T\ \forall v_i \in DFO(v_m)$, then the retiming of a vertex implies the retiming of all the vertices on zero-weighted paths originating from it as well. Therefore no negative weights (labels) can be introduced. Furthermore it can be shown that no feasible retiming exists if the algorithm returns FALSE.

Theorem 2: For any synchronous Boolean network described by $G(V, E, W)$ and a given cycle time T, algorithm $retime$ returns TRUE iff a feasible retiming exists•

Proof: To prove the theorem, it is sufficient to note that running algorithm $retime$ on any multiple I/O network $G(V, E, W)$ is equivalent to running the same algorithm on a modified network without I/Os. Consider a modified network obtained by merging the input and output vertices into a dummy vertex v_h, with $d_h = T$, and by adding one to the weights of all edges incident to v_h. For any feasible retiming of both networks, the data ready time is the same for each pair of corresponding gate vertices. Indeed a retiming of the modified network cannot remove the synchronous register delays from the dummy vertex v_h to any vertex depending on a primary input and therefore the data ready time of these vertices is preserved. In addition, since any retiming of the modified network does not change the cycle weights in the corresponding graph [7], then all the I/O path weights are preserved in the original network. Therefore a feasible retiming of the modified network co-implies a feasible retiming of the original network. Consider now algorithm $retime$. The retiming of a primary output vertex in the original network corresponds to retiming v_h in the modified network and therefore to retiming all other primary output vertices. In turn, the retiming of v_h causes the retiming of all the vertices in the set S. Therefore running algorithm $retime$ on any multiple I/O network is equivalent to running the same algorithm on the corresponding modified network and the claim follows from Theorem 1•

The theorem shows that the existence of a feasible retiming can be computed in $O(|V||A|))$ time for general synchronous Boolean networks, because each of the $|V|$ iterations involves the computation of the data ready times, which can be done by topological sort ($O(|A|)$). In some cases, the algorithm can terminate earlier.

Theorem 3: If at any iteration of the algorithm, $\exists v_m \in M \cap S$ and v_m is a primary output, then no feasible retiming exists•

Proof: In this case, there is a zero weighted path from some input vertex to v_m and $t_m > T$. Since the path weight must be preserved, then t_m cannot be reduced•

This theorem provides an early exit condition which is incorporated into procedure $exit$ of algorithm $retime$.

```
exit {
  If ( k = |V| ) return (TRUE);
  S = {v ∈ V | ∃ a zero weight path from an input vertex to v};
  If ( ∃m ∈ M ∩ S | v_m is a primary output) return (TRUE);
  return (FALSE);
}
```

Algorithm $retime$ has several advantages over the original retiming algorithm [7]. First, the description of a synchronous Boolean network structure in terms of a (sparse) graph suffices to implement the algorithm. This contrast the requirements for the algorithm in [7], that needs two full square matrices of dimension $|V|$. Second, $retime$ is an incremental algorithm, and so it can be applied in connection with network transformations that make small modifications to the network to check feasibility. Circuit transformations affecting the structure of the graph may require local rippling of the registers around the modified area, and in many cases it is likely that the algorithm completes in a number of iterations much smaller than $|V|$.

The algorithm requires the update of the data ready times at each iteration. Note that not all the data ready times need to be recomputed at each iteration. Therefore, the algorithm can be made more computationally efficient by scheduling the set of vertices that are target of the transformation (i.e. M and S). Then the following steps are iterated until the schedule is empty: i) selecting the subset of the scheduled nodes whose direct fanin set is not scheduled; ii) updating their data ready time; iii) scheduling their direct fanout set if the data ready time has changed; iv) deleting them from the list of scheduled vertices.

A network can be made optimal with respect to retiming by running algorithm $retime$ for decreasing values of T. In particular, Leiserson and Saxe suggested to compute the propagation delays between all vertex pairs, and to binary search among these values for the minimum value for which $retime$ returns TRUE [9]. While the computation of all-pair delays may be computationally expensive, a convenient heuristic to solve the problem is to decrease T by fixed increments, so that its value can be a practical choice for the cycle time.

4.2 Elimination

The $elimination$ algorithm follows the outline of that presented in [1] and fits the frame of algorithm $transform$ described above. Candidate vertices are selected according to some criterion and the elimination takes place if some constraints are satisfied. Elimination terminates when no candidate vertices can be found.

We concentrate here on the selection and acceptance criteria for synchronous networks. Let us consider first the area cost (or value) of an elimination, say of v_j into v_i. An elimination changes the total number of literals in a network by δ_l. This number can be computed as $\delta_l = n_{ji}(l_j - 1) - l_j$, where n_{ji} is the multiplicity of variable j in expression I [1] [2]. When elimination is performed across a register boundary, then it is important to compare the saving in terms of literals with the possible increase of registers. This can be computed as follows. Recall that m_k was defined to be the maximum label of variable k in all expressions. Then, for each vertex $v_k \in FI(v_j)$, let $m'_k(I)$ be the maximum label of variable k in the expression I after the elimination. Then additional registers are needed to delay variable k if $m'_k(I) > m_k$. Fewer registers may be needed at the output of v_j. In particular, let $m'_j(\overline{I})$ be the maximum label of variable j in the expressions corresponding to $FO(v_j)$ different from I. Then the register saving is: $m_j - m'_j(\overline{I})$. The total variation in registers is:

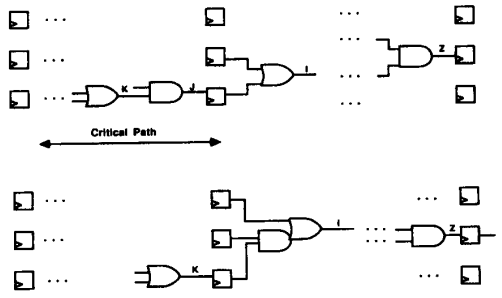

Figure 8: Elimination at the head of a critical path.

$\delta_m = (\sum_{v_k \in FI(v_j)} max(0, m'_k(\mathcal{I}) - m_k)) - (m_j - m'_j(\overline{\mathcal{I}}))$. The area cost of an elimination is then $\delta_A = \alpha\delta_l + \beta\delta_m$.

Example: Consider the circuit of Fig. 3. The variation in the number of literals is: $\delta_l = n_{cx}(l_c - 1) - l_c = 1(2-1) - 2 = -1$, i.e. one literal is saved. Assume that variable c is not used in any other expression and that $m_a = m_b = 0$, i.e. no register is present at the output of v_a and v_b. After the elimination one register is needed to delay a and b, i.e. $m'_a(\mathcal{X}) = m'_b(\mathcal{X}) = 1$ and no register is needed at the output of v_c, that is deleted from the network. Then $\delta_m = (1-0) + (1-0) - (1-0) = 1$ and $\delta_A = -\alpha + \beta$•

Then, for unconstrained area minimization, candidates are selected to either to minimize δ_A, or to be such that δ_A is less than a threshold usually set to zero. When timing constraints are enforced, the new slack s_i is computed. If this value is positive, the elimination is accepted. Else, its acceptance is conditional to finding a retimed feasible network of non superior area cost.

Let us consider now the problem of minimizing the cycle time T. Let us assume that the network is optimal with respect to retiming (by using the $retime$ algorithm for decreasing values of T) and with respect to elimination within register boundaries (as described in [8] and in [2]). We assume that T is the minimum cycle time achieved by these techniques and we address the problem of reducing it by attempting elimination across register boundaries.

In particular, we consider as candidates for elimination the critical vertices whose gate is connected to a register, i.e. at the head of a critical path (Fig. 8). Let us assume, for the sake of simplicity that there is only one such candidate, say v_j and that it is critical (i.e. its slack $s_j = 0$ or equivalently its data ready time $t_j = T$). The elimination of such a vertex shortens the critical path and it is beneficial if no other longer critical path is introduced in the circuit. Therefore, to verify the feasibility of the elimination of a candidate vertex v_j, we must consider the increase of data ready time of each vertex $v_i \in FO(v_j)$. If such increases are all strictly bound by the corresponding slack, then the elimination is accepted because there is a cycle time $T' < T$ for which the network is a feasible implementation after the elimination. If the increase of the data ready time at some vertex is not bound by its slack, then the elimination is accepted under the condition that a feasible retiming is found.

Since we would like to speed up the computation time of the elimination algorithm as much as possible, we seek conditions to avoid to retime a network to check feasibility. Consider first the case that the inputs of the gate corresponding to v_i are registers (i.e. $DFI(v_i)$ is empty). Since $t_i = d_i$, its variation is much easier to compute. If, in addition, $t'_i > d_i + d_j$, then the elimination can be rejected outright. Indeed no feasible retiming can be found for $T' < T$ because, if it were so, vertex v_j could be retimed by -1, and then the network would not be optimal with respect to retiming, as assumed before.

Consider now the case when an elimination is accepted in this context. Then the circuit cycle time can be reduced to the new maximum data ready time T'. It is important to know if the network is still optimal with respect to retiming for this new cycle time.

Theorem 4: Given a synchronous network that is optimal with respect to retiming for a cycle time T, assume that a vertex v_j with $t_j = T$ is eliminated. If after the elimination $\exists v_k \in FI(v_j)$ such that the maximum data ready time is $t_k = T' < T$, then the network is optimal with respect to retiming for cycle time T'•

Proof: Before the elimination, the network is optimal with respect to retiming for a cycle time T implies that no feasible retiming exists for a smaller cycle time and that the cycle time is bounded from below by the data ready time of a vertex which is the head of a critical path. Such node was necessarily v_j, because $t_j = T$ and the vertex was unique because after its elimination the maximum data ready time decreases. Since $v_k \in FI(v_j)$, then v_k was on the critical path before the elimination and it becomes the head of the critical path thereafter. Then the critical path does not change, but for v_j. After the elimination, suppose that a feasible retiming exists for a cycle time $T'' < T'$. This would correspond to shortening the path whose head is v_k. But then, such a retiming could have been applied before the elimination, contradicting the assumption of optimality•

Example: Consider the circuit of Fig. 8. Assume that after the elimination, the maximum data ready time is t_k. Then, the network after elimination preserves optimality with respect to retiming, because the elimination has not introduced another critical path.

4.3 Resubstitution

The $resubstitution$ algorithm follows the outline of that presented in [1] and [2] and fits the frame of algorithm $transform$ described above. Candidate vertices are selected in pairs, say v_i, v_j, so that the expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{I}$. The resubstitution of v_j into v_i is performed if the constraints, if any, are satisfied. The algorithm terminates when no candidate pair can be found.

We consider here only algebraic division [1]. The condition that one expression is a synchronous divisor of another one is checked by routine $synchronous\text{-}divisors$, that iterates algebraic divisions. Algebraic division of two expression is performed by procedure $alg\text{-}div$, which is described in [1] [2].

```
synchronous-divisors(I, J) {
    QR = ∅;
    II = expand(I);
    For (r = 0; ; r++) {
        JJ = expand(J^(+r));
        If(exit(I, J^(+r)))return
        QR = QR ∪ alg-div(II, JJ);
    }
}
```

Candidate pairs are searched among all possible pairs of gate vertices. Note that an expression $\mathcal{J}^{(+r)}$ may divide an expression $\mathcal{J}$ for more than one value of r. Therefore, the algorithm stores all non-trivial quotients $\mathcal{Q}$ and remainders $\mathcal{R}$ in $\mathcal{QR}$. If multiple choices are possible, a greedy strategy is used to select the most convenient resubstitution. To allow multiple resubstitutions, the algorithm will add to the candidate list the pairs q, j and r, j. If $\mathcal{QR}$ is empty, the candidate pair is rejected.

Algorithm $synchronous\text{-}divisors$ operates as follows. Procedure $expand$ replaces every variable with non zero label by a new variable. Therefore expressions $\mathcal{II}$ and $\mathcal{JJ}$ are polynomials that can be divided by algorithm $alg\text{-}div$ [1] [2]. Procedure $exit$ returns true if any variable in $\mathcal{J}^{(+r)}$ has a label larger than the maximum of the labels that the corresponding variable takes in $\mathcal{I}$. In this case, no non-trivial divisor can be found, because expression $\mathcal{JJ}$ contains a literal not in $\mathcal{II}$ and therefore $\mathcal{JJ}$ cannot divide $\mathcal{II}$ [1] [2]. Clearly this condition is true for any value of r larger than the current index of the loop of the algorithm. Note that when both expressions $\mathcal{I}$ and $\mathcal{J}$ have no labels, then $\mathcal{II} = \underline{\mathcal{I}}$ and $\mathcal{JJ} = \mathcal{J}$, the

algorithm performs just the algebraic division as in [1] [2] and returns after one iteration.

To choose among candidate pairs, it is important to evaluate the local change in area due to resubstitution. When resubstituting v_j into v_i, the variation in literals can be computed as $\delta_l = -n_{ji}(l_j - 1)$, where n_{ji} is the multiplicity of variable j in expression $\mathcal{I}$ [1] [2]. The number of registers in the network is affected only by resubstitutions across register boundaries (i.e. when $r > 0$). In this case, resubstitution may increase or decrease the number of registers according to the circumstances. For example, when resubstituting v_j into v_i as $i = j^{(+r)}\mathcal{Q} + \mathcal{R}$, we require r register delays for variable j and r fewer register delays on some inputs to v_i. The total variation in register count δ_m, can be computed from the local variation as follows. First note that additional registers may be needed at the output of v_j, namely $max(0, m'_j(\mathcal{I}) - m_j)$. Registers may be spared on the inputs $FI(v_i)$, where the fanin set is computed before the resubstitution. For each vertex $v_k \in FI(v_i)$, the register saving is $m_k - m'_k$. Then $\delta_m = (max(0, m'_j(\mathcal{I}) - m_j)) - (\sum_{k \in FI(v_i)}(m_k - m'_k))$.

Example: Consider the circuit of Fig. 4. The variation in the number of literals is: $\delta_l = -n_{xy}(l_x - 1) = -1(2 - 1) = -1$, i.e. one literal is saved. (Note that the original expression for y could be factored as $c(a^{(2)} + b^{(1)})$). Assume that $m_x = 0$ and that no additional delayed values of a and b are needed to gates other than those shown in Fig.4. Then $\delta_m = 1 - ((2 - 1) + (1 - 0) = -1$ and $\delta_A = -\alpha - \beta\bullet$

For unconstrained area minimization, the candidates are selected so that either $\delta_A = \alpha\delta_l + \beta\delta_m$ is minimized or it is less than a given threshold. Note that $resubstitution$ reduces the number of literals (i.e. $\delta_l < 0$), whenever the expression for v_j is non trivial, i.e. whenever $l_j > 1$. Therefore resubstitutions on non-trivial expressions that are within register boundaries (i.e. $\delta_m = 0$) are always selected.

Consider now area minimization under cycle time constraints. Note that a resubstitution of vertex v_j into vertex v_i decreases the literal count l_i and it is likely to decrease its propagation delay d_i. However, the data ready time t_i may depend now on t_j, if $v_j \in DFI(v_i)$ after the resubstitution. In this case ($v_j \in DFI(v_i)$), the transformation can be accepted if the increase in t_i is bounded by the slack s_i. Otherwise, a feasible retiming must be searched for. On the other hand, when a register delay is inserted between v_j and v_i ($v_j \notin DFI(v_i)$), then t_i cannot increase and it is likely to decrease. Then the transformation can be accepted without further checks.

The problem of minimizing the cycle time T is analyzed under the previous assumptions: i.e. the network is optimal with respect to retiming and to resubstitution within register boundaries. We also assume that T is the minimum cycle time and we address the problem of reducing it by attempting resubstitution of two vertices, say v_j into v_i across register boundaries. In this case, the data ready time t_i cannot but decrease and t_j remains constant. Then candidates for resubstitution are a critical vertex v_i, which is the tail of a critical path and $v_j \in FI(v_i)$. Candidates are selected to minimize locally the cycle time T. Since an upper bound on the decrease of T is the variation in propagation delay d_i, this is used as a quick way of choosing a candidate.

Example: Consider for example the circuit of Fig. 9. The critical path has as a tail vertex v_i. The resubstitution of vertex v_j into v_i decreases the propagation delay d_i, and therefore reduces the data ready time of the vertex at the head of the critical path. If the maximum value of the data ready time is attained at that vertex only, then the cycle time T can be reduced.

4.4 Extraction

The $extraction$ algorithm consists of detecting common sub-expressions and implementing them as additional gates, possibly under timing constraints. Candidate sub-expressions can be found by means of a global search as in [2], where all the kernel intersections are computed and ranked. Alternatively, candidate sub-expressions can be detected by examining pairs of expressions, one pair at a time. Such a search is justified when synthesis is driven by timing considerations, because critical candidates may be easily detected. We have implemented the latter strategy, that can still be described by the frame of the $transform$ algorithms described above. Selected candidate vertices are such that they share a common sub-expression. Common sub-expressions, including common cubes, are detected by solving a modified rectangular covering problem [10]. The algorithm terminates when no candidate pair can be found.

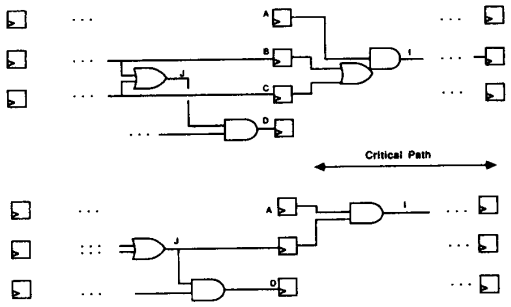

Figure 9: Resubstitution at the tail of a critical path.

The local change in area due to extraction is: $\delta_l = -n(l_l - 1) + l_l$, where usually $n = 2$ because vertex v_l is extracted from $n = 2$ other vertices. Then the variation δ_l is negative (i.e. favorable) every time that $l_l > 2$. The number of registers in the network is affected only by extraction across register boundaries, and can be computed as follows. When extracting v_l from v_i and v_j, then m_l additional registers are needed, where m_l is the maximum label of variable l in the expressions $\mathcal{I}$ and $\mathcal{J}$. Registers may be spared on the inputs $FI(v_i)$ and $FI(v_j)$, where the fanin sets are computed before the extraction. For each vertex $v_k \in FI(v_i) \cup FI(v_j)$, the register saving is $m_k - m'_k$. Then $\delta_m = m_l - (\sum_{k \in FI(v_i) \cup FI(v_j)}(m_k - m'_k))$.

Example: Consider the circuit of Fig. 5. The variation in the number of literals is: $\delta_l = -2(l_i - 1) + l_i = -2(2-1)+2 = 0$, i.e. the number of literals is constant. The variation in register is: $\delta_m = 1 - ((1 - 0) + (1 - 0)) = -1$ and therefore $\delta_A = -\beta\bullet$.

For unconstrained area minimization, the candidates are selected so that either $\delta_A = \alpha\delta_l + \beta\delta_m$ is minimized or it is less than a given threshold.

In the case of area minimization under cycle time constraints, we must verify the data ready times t'_l, t'_i and t'_j after the transformation. Assume first that the extraction is within register boundaries. Then, t'_l is bounded from above by the previous data ready times t_i and t_j. Therefore we must check the values t'_i and t'_j only. Note that an extraction of vertex v_l from vertices v_i and v_j decreases the literal counts l_i, l_j and it is likely to decrease the propagation delays d'_i, d'_j. However an extra stage of delay through v_l is added, which affects the data ready times t'_i and t'_j. In this case the transformation can be accepted if any increase in t'_i and t'_j is bounded by their slacks. Otherwise, a feasible retiming must be searched for. On the other hand, when a register delay is inserted between v_l and both v_i and v_j then t'_i and t'_j cannot increase and they are likely to decrease. Then the transformation can be accepted under the condition that $t_l \leq T$.

The problem of minimizing the cycle time T is analyzed under the previous assumptions. We also assume that T is the minimum cycle time and we address the problem of reducing it by attempting extraction of a vertex, say v_l from v_i and v_j across register boundaries. Then candidates vertices are pairs of vertices v_i, v_j, at least one of which being critical and the tail of a critical path.

4.5 Decomposition

The general frame of the $decomposition$ algorithm is similar to the $transform$ algorithm described before, with the exception that single vertices are target of the transformation. $Decomposition$ of a vertex can be seen as the extraction of a single cube or sub-expression. Therefore it may be applied repeatedly to the same vertex. Note that decomposition increases

the number of literals δ_l. For this reason, decomposition is used in combinational logic synthesis only to break large expression that have no efficient implementation or to satisfy timing goals. However, decomposition in synchronous logic synthesis can lead to a reduction of the number of registers and therefore be beneficial for area reduction as well.

Let us consider the local change in area due to a decomposition step in detail. The variation in literals is $\delta_l = 1$. The number of registers in the network is affected only by decomposition across register boundaries, and can be computed as follows. First $r = m_j$ registers are needed at the output of gate v_j. Registers may be spared on some of the vertices $FI(v_i)$ that become inputs to v_j, where the fanin sets are computed before the decomposition. For each vertex $v_k \in FI(v_i) \cap FI(v_j)$, the register saving is $m_k - m'_k$. Therefore $\delta_m = m_l - (\sum_{k \in FI(v_i) \cap FI(v_j)} (m_k - m'_k))$.

Example: Consider the circuit of Fig. 6. The variation in the number of literals is +1. The variation in register is: $\delta_m = 1-((1-0)+(1-0)) = -1$ and therefore $\delta_A = \alpha - \beta$•

For area minimization the candidate vertex is selected so that either $\delta_A = \alpha\delta_l + \beta\delta_m$ is minimized or it is less than a given threshold. In the case of area minimization under cycle time constraints, note that the literal count l_i decreases and therefore the propagation delay d_i is likely to decrease. However, the data ready time t_i is affected by the additional stage through v_j. When decomposition does not introduce a register at the output of v_j, (i.e. $m_j = 0$), then we must verify the slack s'_i after the transformation, and attempt a retiming in case it becomes negative. Instead, when decomposition is across a register boundary, the data ready time t_i cannot increase and only s_j must be checked.

The problem of minimizing the cycle time T is analyzed under the previous assumptions. We also assume that T is the minimum cycle time and we address the problem of reducing it by attempting decomposition of a vertex, say v_i. Then candidates vertices are critical ones. Candidate divisors are chosen so that $FI(v_j)$ does not include critical vertices, because of the additional stage of delay added. In the case that decomposition is across a register boundary, the same criterion is used. The rationale is that when such a decomposition is feasible, then $FI(v_j)$ is not likely to include critical signals, because such signals were feeding v_i through registers before the decomposition. Therefore the purpose of the decomposition is still to reduce d_i. Candidate divisors are then chosen to include non critical vertices.

5 Extensions to other synchronous delay models

We consider now synchronous Boolean networks that are interconnections of combinational logic gates and positive (negative) edge-triggered registers with worst-case **setup** time t_s, **hold** time t_h and clock-to-output **register propagation** delay t_p. We also assume that the worst case clock skew is ΔT, i.e. the maximum delay of a rising (falling) clock edge with respect to the nominal time.

The extension of the previous techniques to the case in which only the setup and register propagation delay are non zero is trivial. Indeed, the input to the registers must be available at least t_s units of time before the clock edge and the inputs to the combinational circuit will be available t_p units of time thereafter. Therefore it suffices to consider a reduced effective cycle time $T - t_s - t_p$, which represents an upper bound to the propagation delay through the combinational logic.

Conversely, non negligible hold times require the signal at a register input to be stable after the clock edge for t_h units of time. Therefore, the propagation delay through the combinational logic must be bounded from below as well. Therefore, a timing feasible implementation of a network must be such that the data ready time t_i of any vertex v_i with non-zero label ($m_i > 0$), i.e. at any register input, satisfy the following inequalities:

$$\begin{aligned} t_i + t_p + t_s &\leq T - \Delta T \\ t_i + t_p - t_h &\geq \Delta T \end{aligned}$$

Since $t_h - t_p + \Delta T$ is in general a small quantity, some simple solutions can be applied to satisfy the second inequality. For example, if the second equation is not satisfied at some vertex v_i, then the vertices in $FO(v_i)$ can be retimed by -1. when possible. Otherwise technology mapping solutions with synchronous elements can be used, like inserting active delay elements (inverter pairs or simple inverters feeding the inverted input).

Consider now the case in which each synchronous delay element is implemented by a level sensitive gated latch (instead of an edge triggered one.) Then the following inequalities must be satisfied for each data ready time t_i of any vertex v_i with non-zero label:

$$\begin{aligned} t_i + t_p + t_s &\leq T - \Delta T \\ t_i + t_p - t_h &\geq T_1 + \Delta T \end{aligned}$$

where T_1 is the gating time (which tends to zero in the edge-triggered case). Even though single-phase gated latch design is convenient in terms of silicon area, because the latch implementation requires fewer devices, the design space is limited by these inequalities. (Note that $T_1 + t_h - t_p + \Delta T$ may not be a small quantity.) The techniques for synchronous logic synthesis presented in the previous sections still apply. However retiming techniques must be extended to cope with upper and lower bounds on the propagation delays between register pairs. The retiming problem can be cast as a mixed integer-linear program that is an extension of that presented in [7]. Unfortunately there is no solution to this extended problem, to our knowledge, by means of an iterative algorithm, such as $retime$ of Section 4. Therefore, other solution methods should be used, possibly requiring higher computational cost.

Polyphase design can be seen as a further extension of these techniques. The simplest case is the one in which registers are edge-triggered, the time period between leading (trailing) edges of adjacent phases is T and inputs are available on the first phase and outputs on the last. Then polyphase networks can be derived from single phase ones by replacing each register by as many as the number of phases. Retiming can be used to optimally distribute the the registers, so that combinational path delays are shortened.

More complex clocking schemes, that do not need to satisfy these restrictions may be handled by expressing the timing constraints by an appropriate set of inequalities. For example, design with two-phase non-overlapping gated latches, can be handled in this perspective [11]. In general, synchronous logic synthesis techniques can be extended to generic synchronous delay models and clocking schemes by providing a set of logic transformations in conjunction with methods for solving inequalities.

6 Implementation and Results

Logic synthesis of synchronous digital circuits is supported by programs **Minerva**, **Janus** and **Ceres**, which share the same data structure, I/O formats and user interface. Minerva is a workbench to develop and test the algorithms on un-mapped logic networks described in the previous sections. Janus provides fundamental transformations to handle synchronous logic networks as well as an interface to the simulation enviroment and to the netlist formats of some commercial uncommitted arrays. Ceres performs technology mapping of synchronous networks.

Circuit specifications can be entered to the programs by specifying synchronous Boolean networks in a hierarchical way in the Structure/Logic Intermediate Format (SLIF) developed at Stanford University [12]. Such a specification can be generated automatically from circuit descriptions in the Hardware Description Language **HardwareC**, that can be compiled into the SLIF format by the **Hercules** and **Hebe** programs [13]. These programs are a part of the Olympus synthesis system developed at Stanford University (Fig 10), that has been used to synthesize three chips designs to date [14] [15].

The logic synthesis tools, and in particular Minerva, are interfaced to the MIS-II program [2], that provides an excellent set of routines for optimizing and mapping combinational sub-components of the circuit being designed. They can isolate these components and interface them with MIS-II in a bidirectional way. Minerva supports hierarchical circuit input descriptions, that may including specific circuit macros (such as bus drivers, tristate elements, or generic combinational black boxes). However the algorithms for synchronous logic synthesis are applied, to date, to a flattened circuit description. Minerva is programmed in C and consists of approximatively 6000 lines of code.

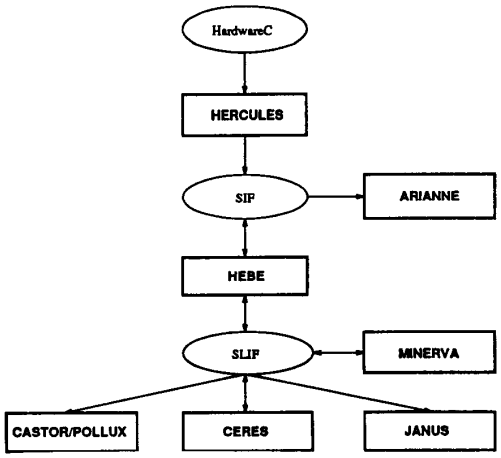

Figure 10: The Olympus Synthesis System.

Minerva is an interactive program, that can operate also in batch mode. The user interface of Minerva can be set to one of three levels, according to the level of expertise of the user. In the first level, pertinent to the novice user, only those commands that operate on the entire network are visible and executable from the user, e.g. global retiming, elimination , etc . In the second level, additional commands can be invoked to perform transformations selectively to some vertices of the network. The third level supports the fundamental transformations and it is used for algorithm and program development only. Minerva supports the different optimization goals described in Section 4. A strategy variable can be set to the desired goal.

The algorithms have been tested on benchmark circuits, derived from standard benchmarks. In particular, the examples ex3-7 are derived from the MCNC fsm examples ex3-7. The first examples is a pipelined ALU, derived from Alu2 and the second is the phase decoder of the DAIO chip [14]. It is important to remark that standard multiple-level synchronous networks are to date not available and therefore the starting points for our experiments have been derived from two level representations by an arbitrary, but fixed sequence of logic synthesis steps. Area and timing variations are computed from these starting points.

Some experimental results are reported in Tables 1 and 2 to validate the proposed procedures. Computing time is in the order of a fraction to a few seconds on a DECstation 3100 computer. In Table 1 we compare the area optimization achieved by Minerva versus MIS-II. The circuit area is a linear combination of the literal and register cost. For a fair comparison, Minerva does not apply transformations across registers. The comparative quality of the results is reported in the Δ columns. Not surprisingly, global operations performed by MIS-II do better than the pair-wise operations done by Minerva. However, the difference in area is small.

In Table 2 we compare the results of using Minerva with fixed register position versus Minerva with floating registers, i.e. performing operations across register boundaries. The comparison shows the relative advantage that can be achieved in reducing the cycle time when the registers are floating. For this case, we report the cost in additional area to be paid for the decreased cycle time and reported in the Δ columns. Note that the increased area cost is mainly due to an increase in the number of registers. Note also that the timing results are dependent on the delay model.

7 Concluding remarks and future directions

This paper has presented a new approach to the optimal logic synthesis of digital synchronous circuits, based on the concurrent optimization of the circuit equations and the register positions. This method, which combines retiming techniques with network restructuring operations, can achieve results that are at least as good as those obtained by other logic synthesis approaches that separate the combinational logic from the registers. Algorithms for circuit transformations within and across register boundaries have been studied and implemented in program Minerva.

This research as shown the feasibility of approaching sequential logic design from a new perspective, based on a stepwise refinement of a logic representation. We think that this approach can encompass the register allocation problem for sequential logic and therefore classical problems, such as the state assignment problem, can be cast in this setting. However several problems are not yet solved and deserve further research. First a study of the appropriate set of logic transformations for synchronous sequential logic, with particular reference to the possibility of reaching all the possible circuit configurations with equivalent I/O behavior. Second the search of efficient retiming techniques supporting an extended propagation delay model with explicit fanout dependency as well as satisfying both upper and lower bounds on propagation delays. Such an extension would support logic synthesis techniques for circuit designs with gated latches and with non-negligible clock skew. Third the study of technology mapping techniques that take advantage of the information contained in the synchronous Boolean network and the application of retiming techniques to mapped networks.

8 Acknowledgements

This research has been sponsored by NSF under contracts MIP-8710748 and MIP-8719546. We would like to acknowledge the stimulating discussions with Andrew Fox, Michiel Ligthart and Frederic Mailhot. Thierry Klein developed some of the ideas of the retiming and elimination algorithms and implemented them in Minerva. Frederic Mailhot developed the SLIF format, the front and back ends of program Minerva and the Janus program that provides the interfaces of SLIF to other formats.

References

[1] R.Brayton, "Algorithm for Multilevel Synthesis and Optimization" in G.De Micheli, A.Sangiovanni-Vincentelli and P.Antognetti, Editors, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff, 1987.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD/ICAS*, Vol. CAD-6, No. 6, November 1987, pp. 1062-1081.

[3] J.Darringer, D.Brand, J.Gerbi, W.Joyner and L.Trevillyan, "LSS: A System for Production Logic Synthesis", *IBM Journal of Res. and Dev.*, Vol 28, No 5, pp. 537-545, Sep 1984.

[4] K.Bartlett, W.Cohen, A.De Geus and G.Hachtel, "Synthesis and Optimization of Multilevel Logic under Timing Constraints" *IEEE Transactions on CAD/ICAS*, Vol CAD-5 No. 4, pp.582-596, Oct. 1986.

[5] S.Muroga, Y.Kambayashi, H.Lai and J.Culliney, "The Transduction method - Design of Logic networks based on permissible functions", *IEEE Transactions on Computers* Vol 38, No. 10, pp 1404-1424, Oct. 1989.

[6] S.Malik, E.Sentovich, R.Brayton and A.Sangiovanni, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *Proceedings on the International Workshop on Logic Synthesis*, Research Triangle Park, May 1989.

[7] C.Leiserson, F.Rose and J.Saxe "Optimizing Synchronous Circuitry by Retiming", in R.Bryant, Editor *Third Caltech Conference on VLSI*, Computer Science Press, 1983.

[8] G. De Micheli, 'Performance-oriented synthesis in the Yorktown Silicon Compiler', *IEEE Trans on CAD/ICAS*, Vol CAD-6, NO 5, Sept 1987, pp.751-765.

[9] J.Saxe "Decomposable Searching Problems and Circuit Optimization by Retiming: Two Studies in General Transformations of Computational Structures" *Ph. D. Dissertation*, Department of Computer Science, Camegie Mellon University, 1985.

[10] R.Brayton, R.Rudell, A.Sangiovanni and A. Wang, "Multi-level logic optimization and the Rectangular Covering Problem", *ICCAD- Proceedings of the International Conference on CAD*, Santa Clara, 1987, pp. 66-69.

[11] M.Dagenais and N.Rumin, "On the Calculation of Optimal Clocking Parameters in Synchronous Circuits with level Sensitive Latches" *IEEE Trans on CAD/ICAS*, Vol CAD-8, No. 3 , March 1989, pp 268-278.

[12] G.De Micheli, "Algorithms for Synchronous Logic Synthesis", *Proceedings of the Internatinal Workshop on Logic Synthesis*, Research Triangle Park, 1989.

[13] G.De Micheli and D.Ku, "HERCULES - A system for High- Level Synthesis", *Proceedings of the 25th Design Automation Conference*, Anaheim, pp. 483-488, 1988.

[14] M.Ligthart, A.Bechtolsheim, G.De Micheli and A.El Gamal "Design of a Digital Audio Input Output chip", *Proceedings of the Custom Integrated Circuit Conference*, San Diego, 1989.

[15] V.Rampa and G.De Micheli, "Computer Aided Synthesis of a Discrete Cosine Transform Chip", *Proceedings of the International Symposium on Circuits and Systems*, Portland, 1989.

Example	Original			Minerva				MisII			
	Lits	Regs	Area	Lits	Regs	Area	Δ	Lits	Regs	Area	Δ
Ex1	615	6	662	615	6	662	0	587	6	635	-27
Ex2	651	51	1059	649	51	1057	-2	506	51	914	-145
Ex3	168	12	264	138	12	244	-20	137	12	233	-31
Ex4	180	23	364	159	23	343	-21	129	23	313	-51
Ex5	157	11	245	130	11	218	-27	127	11	215	-30
Ex6	192	16	320	191	16	319	-1	171	16	299	-21
Ex7	177	12	273	151	12	247	-26	149	12	245	-28

Table 1: Comparative area variation.

Example	Minerva fixed registers				Minerva floating registers				Δ	
	Time	Lits	Regs	Area	Time	Lits	Regs	Area	Area	Time
Ex1	53.7	615	6	662	27.0	615	60	1095	+433	-26.7
Ex2	42.1	649	51	1057	35.4	651	73	1235	+178	-6.7
Ex3	16.3	138	12	244	15.0	168	12	264	+20	-1.3
Ex4	22.9	159	23	343	20.0	159	25	359	+16	-2.9
Ex5	13.7	130	11	218	13.7	130	11	218	0	0
Ex6	21.2	191	16	319	15.8	181	23	365	+46	-4.4
Ex7	15.7	151	12	247	14.9	177	12	273	+26	-0.8

Table 2: Comparative timing and area variation.