Algorithms for Synchronous Logic Synthesis

Giovanni De Micheli
Thierry Klein
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 how to optimize concurrently the circuit equations and the register position. This method is novel and can achieve results that are at least as good as those obtained by previous methods. A computer implementation of the algorithm in program Minerva is described.

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 research 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 mitigated 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 have been investigated and clever algorithms for combinational logic synthesis have been reported in the literature [1],[2],[3],[4].

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. 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 1/0-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 a superset of those used in combinational logic synthesis and operate within and across the register boundaries. Therefore the potential quality of the optimized circuits is at least as good as that obtained by the previous techniques that were constrained to operate on the combinational component only.

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 [5] 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.

2 Basic concepts and definitions

We consider synchronous circuits that are interconnections of combinational logic gates and single-clock positive-edge-triggered registers with negligible setup times. 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 labeled on a variable (or expression) 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 $r_i$ is represented by $i = f$, where $f$ is a Boolean expression in terms of other (labeled) variables.

The network is modeled by the synchronous network graph, that is a directed weighted multigraph $G(V, W)$, where $V = 1^0U2^0U3^0U\ldots = \{r\}$ 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 $r_i$ and $r_j$ with weight $k$ when variable $r_j$ appears in the expression $f$ for vertex $r_i$ with label $k$. Zero-valued weights are not indicated by notation. There is a (weighted) edge to each output vertex in $1^0$ from the vertex on $1^0$ corresponding to the gate generating that output signal. For each pair of vertices joined by a path in $G(V, W)$, 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 of a synchronous Boolean network and its representation is shown in Fig. 1.

In general, a synchronous Boolean network may have cycle dependencies, i.e. its corresponding graph be cyclic. A network is called unidirectional when the graph $(G(V, W))$ is acyclic. It models a
piped 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 $F_I(v_i)$ ($DF_I(v_i)$). 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 $F_O(v_i)$ ($DOF_O(v_i)$). Each vertex of the graph $v_i \in V^R$ (i.e., corresponding to a gate) has as attributes an area estimate $l_i$ in terms of literal count [1] and a positive gate delay $d_i$, which depends on the logic expression and which is a monotonically increasing function of $l_i$. Each input and each output vertex has zero delay.

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 [6]. 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., $t_i = d_i + \max_j FOP_j(l_j)$. 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 above by the cycle time, i.e., $\Phi \geq \max_j FOP_j(l_j)$. 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 $\Phi$ [6]. In a timing-feasible network a vertex is critical if its slack is null.

The area taken by a network implementation depends on the total number of literals and registers required. For each variable $y$, we denote the maximum of the labels that the variable takes in the network representation. Then $m_y$ 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: $A = \sum \sum a_i \times l_i + \sum \sum b_i \times m_y$, where $a$ and $b$ are coefficients taking into account the relative area cost of a literal and a register. Given an area bound $\Phi$, a network is an area-feasible implementation if $\Phi \geq A$, and it is a feasible implementation if it is both area-feasible and timing-feasible.

A better model of gate delay would include loading effects due to fanout [6]. We neglect this dependence here for the sake of simplicity.

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 not efficient exact solution 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 re-structuring the global interconnections (e.g., elimination, restructuring, and extraction). We consider here global transformations extended to synchronous logic synthesis in relation with network retiming.

Retiming [9] 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.

In our context, the retiming of variable $y_i$ by an integer $r_i$ corresponds to adding $r_i$ to its label, and the retimed variable is denoted by $\Phi + r_i$. Similarly, the retiming of an expression $E$ by an integer $r_i$ corresponds to adding $r_i$ to the labels of all its operands and it is represented by $E + r_i$.

The positive (negative) retiming of a gate vertex $v_i$ is the shift of $v_i$ register delays from its outputs (inputs) to its inputs (outputs). It corresponds to retiming by $r_i$ the expression $I$ of $v_i$, and to retiming by $-r_i$ the variable $i$ in the expression of the vertices of $FPO(v_i)$. The retiming of an input vertex is just the retiming by $-r_i$ of the variable $i$ in the expression of the vertices of $FPO(v_i)$. The retiming of an output vertex is just the retiming by $r_i$ of the expression $E$ of $v_i$. An example is shown in Fig. 2.

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 a search technique that finds the minimum $\Phi$ for which such an assignment exists [9]. The corresponding network is said to be optimal with respect to retiming. If this technique were the only available to optimize the cycle time, then its
result would be a global optimum solution. However retimeing 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 that modify the network structure.

For this reason we consider here the following transformations.

The elimination of a variable with label \( k \) in the replacement of the variable by its corresponding expression delayed by \( k \). Given a two gate vertices \( v_i \) and \( v_j \in \hat{F}(v_i) \), the elimination of \( v_j \) into \( v_i \) is the elimination of \( v_j \) in all its occurrences in the expression \( F \) for \( v_i \) (Fig. 3). The elimination of vertex \( v_i \) is its elimination into all the vertices in \( F \).\( F(v_i) \). Note that the elimination of a variable with label \( k \) is equivalent to the eliminations used in combinatorial logic synthesis [1] [2]. The elimination of a variable \( v_i \) with label \( k \) is equivalent 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 \( I, F, Q \) and \( R \) be Boolean expressions. Then \( I \) is a synchronous divisor of \( I \) if there exists \( Q \in \mathbb{N} \) such that \( I = I \cdot Q + R \) and \( I \cdot Q \neq \emptyset \). Note that this product \( I \cdot Q \) may have the algebraic or Boolean flavor, as defined in [3]. Given two gate vertices \( v_i \) and \( v_j \), such that the expression \( L \) is a synchronous divisor of \( I \), the recombination of \( v_j \) into \( v_i \) is the factoring of \( I = I \cdot F + R \). Note again that the divisors defined in [1] are a subset of the synchronous divisors and therefore recombination with null retiming (i.e., \( k = 0 \)) is equivalent to recombination in combinatorial logic. The recombination 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 \( I \) and \( J \) corresponding to two vertices \( v_i \) and \( v_j \) is the addition to the network of a vertex \( v_i \) corresponding to a common synchronous divisor of \( I \) and \( J \) and to the factoring of \( I \) and \( J \) in terms of the new variable \( I \).

4 Algorithms for synchronous logic synthesis

4.1 Retimeing

The following algorithm can be used to check whether a synchronous network implementation is feasible for a given \( F \). It is derived from an algorithm described in [1] for networks with 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 assumption 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 feasibility 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.

```
retiming {
  for (int k = 1; k <= |V|; k++) {
    for (int i = 1; i <= |V|; i++) {
      if (M = a[i][k] > 0) {
        set_outputs;
      } else {
        set_outputs = (FALSE);
        retiming (i) all vertices in M;
        set_outputs;
      }
    }
  }
}
```
since \( t_0 \to \Phi \) implies \( t > \Phi \Phi \forall \in DF(t_0) \), the retained of a vertex implies the retaining of all the vertices on the zero-weighted path originating from it as well. Therefore no negative weights (labels) can be introduced. Furthermore it can be shown that no feasible retaining exists if the algorithm returns FALSE.

**Theorem 2:** For any synchronous Boolean network described by \( G(V,E,W) \) and a given cycle time \( \Phi \), algorithm retain returns TRUE if a feasible retaining exists.

**Proof:** To prove the theorem, it is sufficient to note that running algorithm retain on any multiple I/O network \( G(V,E,W,F) \) is equivalent to running the same algorithm on a modified network without I/Os. Consider a modified network obtained by negating the input and output vertices into a dummy vertex \( \delta \), with \( \delta \equiv \Phi \), and by adding out-edges from each of the grounded inputs to \( \delta \). For feasible retaining of both networks, the data ready time is the same for each pair of corresponding gate vertices. Indeed a retaining of the modified network cannot remove the synchronous register delays from the dummy vertex \( \delta \) to any vertex, depending on a primary input and therefore the data ready time of these vertices is preserved. In addition, since any retaining of the modified network does not change the cycle weights in the corresponding graph \( G \), then all the I/O path weights are preserved in the original network. Therefore a feasible retaining of the modified network co-implies a feasible retaining of the original network. Consider now algorithm retain. The retaining of a primary output vertex in the original network corresponds to retaining \( \delta \) in the modified network and therefore to retaining all other primary output vertices. In turn, the retaining of \( \delta \) causes the retaining of all the vertices in the set \( S \). Therefore running algorithm retain 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 a feasible retaining can be computed in \( O(|V|^4)|E| \) time for general synchronous Boolean networks, because each of the \( |V| \) iterations involves the computation of the data ready times of the vertices which can be done by topological sort \( G(t_0) \).

**Theorem 3:** If at any iteration of the algorithm, \( S_0 \equiv \delta \cap S \) and \( S_0 \to \Phi \), no further retaining exists.

**Proof:** In this case, there is a zero-weighted path from some input vertex to \( S_0 \), and \( \delta \to \Phi \). Since the path weight must be preserved, then \( \delta \) cannot be retained.

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

**Algorithm retain** has several advantages over the original retaining algorithm \([5]\). First, the description of a synchronous Boolean network structure in terms of a (sparse) graph suffices to implement the algorithm. This contrasts the requirements for the algorithm in \([5]\) that needs two full square matrices of dimensions \(|V| \times |V|\). Second, retain 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 re-pinning 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 \(|V|^2\).

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 (e.g., \( M \) and \( S \)). Then the following steps are iterated until the schedule is empty: (i) selecting the subset of the scheduled nodes whose direct (main) set is not scheduled; (ii) updating their data ready time; (iii) scheduling their direct fanout vertices whose data ready time has changed; (iv) deleting them from the list of scheduled vertices.

### 4.2 Elimination

The elimination algorithm follows the outline of that presented in \([1]\) and \([2]\). 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.

Let us consider first the area cost (or value) of an elimination. An elimination changes the total number of literals in a network, say \( h \). This number can be computed by the formula given in \([1]\) \([2]\). 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. Assume that we are eliminating vertex \( v \) into vertex \( v' \). Then, for each vertex \( z \in DF(\delta) \), let \( m_z(\delta) \) be the maximum label of variable \( z \) in the expression \( z \) after the elimination. Similarly, let \( m_z(\delta) \) be the maximum label of variable \( v \) in the expression \( z \) before the elimination.

In particular, since \( m_z(\delta) \) is not affected by the elimination. Then additional registers are needed to delay variable \( k \) if \( m_z(\delta) > m_z(\delta) \) at the \( z \) for all \( z \). Therefore, for unconstrained area minimization, candidate vertices are selected to minimize \( h \), which is required to be less than a threshold usually set to zero.

In the case of area minimization under cycle time constraints, we assume that the network is timing-feasible before the transformation and that it is required to remain such thereafter. In previous work \([6]\), 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. Indeed an elimination of vertex \( v \) into vertex \( v' \) increases the literal count \( l \) by \( l \) and is in evidence to the propagation delay \( d \) and data ready time \( t \). While the sufficiency of the above condition still holds in this setting, its necessity no longer does; because a feasible retaining of the network may exist after the elimination. Therefore any elimination that does not satisfy the above sufficient condition is followed by retaining. If a feasible retaining is not found, then the transformation is rejected. The candidate selection is based on the previous criterion.

Let us now consider the problem of minimizing the cycle time \( \Phi \). Let us assume that the network is optimal with respect to retaining (i.e., using the retain algorithm for decreasing values of \( \Phi \) as described in \([5]\) and in \([7]\)) and with respect to elimination within register boundaries (as described in \([8]\) and in \([9]\)). We assume that \( \Phi \) is the maximum 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 computed by a register in the area of a critical path. Let us assume, for the sake of simplicity that there is only one such candidate, say \( v_z \) and that it is critical (i.e., its slack \( s \) is not less than zero). Since increasing the number of critical vertices by \( v_z \) would increase the area of the network, the slack is less than the area of the slack. 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 \( v_z \), we must consider the increase of data ready time of each vertex \( v = DF(v) \) if such increases are strictly bounded by the corresponding slack, then the elimination is accepted because there is a cycle time \( \Phi' < \Phi \) for which the network is a feasible implementation after this elimination. If the increase of the data ready time at some vertex is not bounded by its slack, then the elimination is accepted under the condition that a feasible retaining 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 testing a network to check feasibility. Consider first the case that the inputs of the gate corresponding to \( v_z \) are registers (i.e., \( DF(v) = \emptyset \)). Since \( t_i = s_i \), its variation is much easier to compute. If, in addition, \( t' > t_1 + d \), then the elimination can be rejected outright. Indeed no feasible retaining can be found for \( \Phi' < \Phi \) because, if they were so, vertex \( v_z \) could not be retained by \( \delta \), and then the network would not
be optimal with respect to retiming, as assumed before.

Consider now the case where all-elimination is accepted in this context. Then the current cycle time can be reduced to the new maximum data ready time, \( \Phi \). It is important to know if the network is still optimal with respect to retiming for this new cycle time.

**Proof:** Given a synchronous network that is optimal with respect to retiming for a cycle time \( \Phi \), assume that a vertex \( v \) with \( \delta(v) > \Phi \) is eliminated. If after the elimination \( |E| / \Phi \) such that the input data ready time \( w \) is greater than \( \Phi \), then the network is optimal with respect to retiming for cycle time \( \Phi \).

**Proof:** Before the elimination, the network was optimal with respect to retiming for a cycle time \( \Phi \). This implies that there is 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. Each node was eliminated because the cycle time is the maximum of its data ready time. Since \( v \) is a node in \( E \), then \( \delta(v) \) 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 \( v \). After the elimination, suppose that a feasible retiming exists for cycle time \( \Phi \) < \( \Phi \). This would correspond to shortening the path whose head is \( v \). But then, such a retiming could be applied before the elimination, contradicting the assumption of optimality.

The problem of minimizing the cycle time by elimination under area constraints are selected on the basis of a possible reduction of the cycle time \( \Phi \) and the acceptance of the transformation is based on the above considerations. In fact, the same conclusion holds, and on the check that the area bound \( \Phi \) is not violated. Therefore, any area bound for each elimination is computed and added to the current value of \( \delta \). If the result is larger than \( \Phi \), the candidate elimination is rejected. The area bound may also be used as a tie rule to choose among candidates yielding the same cycle time reduction.

### 4.3 Resubstitution

The resubstitution algorithm follows the outline of that presented in [1] and [2]. In this version, operations are selected in pairs, say \( v \), \( w \), so that the expressions \( f \) and \( g \) in a synchronous divisor of \( I \). The resubstitution of \( v \) into \( w \) is performed if some constraints are satisfied. The algorithm terminates when no candidate pair can be found. We concentrate here on the selection of synchronous divisors and on the check that the area bound \( \Phi \) is not violated.

We consider here only algebraic divisions [1]. Synchronous divisors are searched by iterating algebraic divisions, performed by procedure \( \text{syn-div} \), described in [1] [2].

### 4.3.1 The Algorithm

#### 4.3.1.1 The Algorithm

The algorithm stores the quotient and the remainder of the division in \( QR \), which is initialized empty, when non trivial ones are found. The algorithm proceeds replacing every variable with non zero label by a new variable. The algorithm returns true if any variable in \( f \) has a label larger than the maximum of the labels of the corresponding variable taken in \( I \). If both expressions \( I \) and \( J \) have no labels, the algorithm returns after one iteration and performs just the algebraic division in [1] [2].

#### 4.3.1.2 The Algorithm

The algorithm returns true if any variable in \( f \) has a label larger than the maximum of the labels of the corresponding variable taken in \( I \). If both expressions \( I \) and \( J \) have no labels, the algorithm returns after one iteration and performs just the algebraic division in [1] [2].

### 4.3.2 Resubstitution

The resubstitution algorithm has been implemented as a part of program Minerva. Minerva is a computer program to support logic design and optimization of large-scale synchronous digital circuits. The program can be applied to the problem by specifying the circuit equations and interconnection in a hierarchical way. An appropriate format, called SLM, is used to support the circuit description. Alternatively, the circuit can be specified in a Hardware Description Language, Hardware, that can be compiled into the circuit equations by program Herebe. The output of Minerva is an optimized circuit description in the SLM format. Minerva can provide an output compatible with the Hasten hardware tools and with the OCC logic view. Minerva is interfaced to the MIS-II program [2], that provides an efficient set of routines for optimizing and mapping combinational sub-components of the circuit being designed.

Minerva can include these components and interface them with MIS-II in a hierarchical way. Minerva is programmed in C and consists of approximately 12,000 lines of code. Minerva in a part of the Olympia synchronous system developed at Stanford University (Fig. 5).

Wide Minerva can handle a hierarchical circuit description, including specific circuit modules (such as latches, toplevel elements etc.), and the algorithm for synchronous logic synthesis is applied, to date, to a flat circuit description. Four different optimization goals, constrained and unconstrained minimization of area and cycle time, correspond to four different strategies of the elimination and resubstitution algorithms.

In the program implementation, care has been taken in avoiding to
Figure 5: The Olympus synthesis system.

...circuits, which otherwise not necessary to reduce computation time. However, when the desired goal is to minimize the cycle time, then a frequent use of retiming has shown to be beneficial. Algorithms refining generally return true, when it does, in few iterations. Computing time is in the order of few seconds to few minutes on a L-MIP workstation.

Experimental results on the MCNC benchmarks have shown that these techniques are used in avoiding some low-valued local optimal solutions that are found by other logic synthesis algorithms applied to the combinational component of the circuit (without registers). Results are heavily dependent on the delay model, and therefore comparisons are useless in the absence of a standardization of the delay parameters.

To date Minerva has been used successfully to support the synthesis of three large scale digital designs at Stanford University [8] [10].

6 Concluding remarks and future work

This paper has presented a new approach to the optimal logic synthesis of digital synchronous sequential 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. These algorithms have been studied and implemented in program Minerva.

This research as shown the feasibility of approaching sequential logic design from a global perspective that considers synchronous register delays and gate propagation delays. The choice of realistic delay models for both registers and combinational gates is of pivotal importance. Furthermore, contemporization in the delay model complicate the algorithm for retiming, elimination and resynthesis. An extension of the algorithm to cope with function-dependent gate delay is under progress. We are also currently researching the application of other logic synthesis transformations in synchronous logic synthesis, the analysis of multiple clocking phases and the use of different register types.

7 Acknowledgements

This research has been sponsored by NSF and ARPA, under contracts MHT-8710248, MIP-8710440 and N00014-87-K-0038. We would like to acknowledge the stimulating discussions with Andrew Fox and Michel Lichtenst. Frederic Mailhot developed the front and back ends of program Minerva.

References


