Synchronous Logic Synthesis

Giovanni De Micheli
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. 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 algorithms 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 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].

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 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 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-phase-clock positive-edge-triggered registers with negligible setup and hold 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 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 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 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. $t_i = d_i + max_{v_j \in DFI(v_i)}(t_j)$. 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 [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 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, etc.). We consider here global transformations extended to synchronous logic synthesis in relation with network retiming.

Retiming [5] 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 i by an integer r corresponds to adding r to its label, and the retimed variable is denoted by $i^{(+r)}$. 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 $\mathcal{I}^{(+r)}$. 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.

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 T for which such an assignment exists [5]. 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 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 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).

Other global transformations, such as **extraction** and **decomposition**, can be defined in a similar way for synchronous Boolean networks. We defer the analysis of such transformations to a later paper and we concentrate here on algorithms for synchronous logic synthesis based on elimination and resubstitution that exploit retiming techniques. An algorithm for retiming a synchronous Boolean network, derived from the one presented by Saxe [7], is fully detailed in [9].

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] [6]. 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.

We report in this section on two operators for synchronous logic synthesis: elimination and resubstitution. The algorithms have the general frame described in [1] [2]. They differ from those used for combinational logic synthesis in the cost function evaluation and in the selection criteria for the candidate vertices for a transformation. 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 [6]. 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. 5. 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 [6]. 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.

4.1 Elimination

The *elimination* algorithm follows the outline of that presented in [1]. 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 $\mathcal{I}$ [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 $m'_k(\mathcal{I})$ be the maximum label of variable k in the expression $\mathcal{I}$ after the elimination. Then additional registers are needed to delay variable k if $m'_k(\mathcal{I}) > m_k$. Fewer registers may be needed at the output of v_j. In particular, let $m'_j(\overline{\mathcal{I}})$ be the maximum label of variable j in the expressions corresponding to $FO(v_j)$ different from $\mathcal{I}$. Then the register saving is: $m_j - m'_j(\overline{\mathcal{I}})$. The total variation in registers is: $\delta_m = (\sum_{v_k \in FI(v_j)} max(0, m'_k(\mathcal{I}) - m_k)) - (m_j - m'_j(\overline{\mathcal{I}}))$. 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: $\delta_l = n_{c,x}(l_c - 1) - l_c = 1(2 - 1) - 2 = -1$, 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 $\delta_m = (1 - 0) + (1 - 0) + (0 - 1) = 1$ 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 feasible-retimed 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 [5] [9]) and with respect to elimination within register boundaries (as described in [6] 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. 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.

4.2 Resubstitution

The *resubstitution* algorithm follows the outline of that presented in [1] and [2]. We consider here only algebraic division [1]. The condition that one expression is a synchronous divisor of another one is checked by routine *synchronous-divisors*, that iterates algebraic divisions. Algebraic division of two expression is performed by procedure *alg-div*, which is described in [1] [2].

```
synchronous-divisors {
    $\mathcal{QR} = \emptyset$;
    $\mathcal{II} = expand(\mathcal{I})$;
```

```
or (r = 0; ; r + +) {
  $\mathcal{J}\mathcal{J} = expand(\mathcal{J}^{(+r)})$;
  If(exit($\mathcal{I}.\mathcal{J}^{(+r)}$))return
  $\mathcal{QR} = \mathcal{QR} \cup alg\text{-}div(\mathcal{II}.\mathcal{JJ})$;
```

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-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-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 one in the loop of the algorithm. Note that when both expressions $\mathcal{I}$ and $\mathcal{J}$ have no labels, then $\mathcal{II} = \mathcal{I}$ 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 $\delta_A = -\alpha - \beta$●

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 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.

5 Minerva

Minerva is a computer program to support logic design and optimization of large scale synchronous digital circuits. Circuit specifications can be entered to the program by specifying the circuit equations and interconnection in a hierarchical way. An appropriate format, called SLIF, is used to support the circuit description. The SLIF is format description is reported in the Appendix. Alternatively the circuit can be specified in a Hardware Description Language, HardwareC, that can be compiled into the SLIF format by program Hercules [8]. The output of Minerva is an optimized circuit description in the SLIF format. Minerva in a part of the Olympus synthesis system developed at Stanford University.

A twin program, called Janus, provides an interface between SLIF and other formats, foreign to the Olympus system. For example, Janus can generate a logic view compatible with the netlist format of the LSI Logic tools and with the OCT database. Interface to the Thor and Lsim simulators are also supported by Janus. Minerva and Janus are both 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. Minerva can isolate these components and interface them with MIS-II in a bidirectional way. This feature allowed us to avoid to duplicate the algorithms of program MIS-II into Minerva and to concentrate on the implementation of algorithms specific to synchronous logic synthesis. Minerva is programmed in C and consists of approximatively 12000 lines of code.

At present, Minerva is used as a workbench to test algorithms for synchronous logic synthesis. In particular, algorithms *elimination* and *resubstitution* have been implemented among others as a part of program Minerva. The algorithm *retime* presented in [9] has also been implemented in Minerva. Four different optimization goals, constrained and unconstrained minimization of area and cycle time, correspond to four different strategies of the algorithms described in the previous sections. Experimental results on the MCNC FSM benchmarks (ex1-ex7) have shown that the are gain by performing elimination is often larger than the gain by using resubstitution. Timing improvements by combining elimination, resubstitution and retiming is small, but justified by the fact that the examples are too small to allow radical changes in the structure of the network. Running times are under one second on a DEC 3200 workstation.

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 logic expressions 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. The algorithm for elimination and resubstitution within and across register boundaries, as well as a retiming algorithm [9], 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. At present we are extending Minerva to cope with other circuit transformation, as well as supporting more general models for synchronous storage elements.

7 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 and Michiel Ligthart. Thierry Klein and Roger Yip provided insight into the algorithms and implemented Minerva. Frederic Mailhot developed the framework of Minerva, the SLIF format and wrote program Janus.

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] 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.

[6] 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.

[7] 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, Carnegie Mellon University, 1985.

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

[9] G.De Micheli and T.Klein, "Algorithms for Synchronous Logic Synthesis" *Proceedings of the International Symposium on Circuits and Systems*, Portland, May 1989.

8 Appendix: the SLIF format

NAME

SLIF – Structure and Logic Intermediate Format

DESCRIPTION

SLIF is a concise format used by tools in the *Stanford Olympus Synthesis Project* to describe logic circuits and their interconnections. It is an hierarchical, non-procedural notation that is described in ASCII files. It supports the description of combinational logic in terms of *Boolean factored forms*, as well as *don't care sets*. It supports the notion of *registers* and *three-state elements* and therefore it is well suited to describe sequential circuits.

SYNTAX

SLIF is a free-format notation; i.e., statements may begin at any point on a line, and whitespace may be used freely. Each statement must be terminated by a semicolon. Statements may appear in any order within the description of a model, with the restriction that inputs, outputs, inouts and types must be declared before they are used and that the last statement in the model description must be the *.endmodel* statement (see the COMMANDS section below for more details).

Identifiers are character strings restricted to alphanumeric characters and the following symbols:

: ^ % [ ] _ . / ˉ -

Variables, model names and instance names are all identifiers. There are two constants, "1" and "0", which represent the logic values TRUE and FALSE, respectively.

Commands in *SLIF* are command words preceded by a period (e.g., *.library*). and are summarized in the next section. Any declaration that does not begin with a command is a **logic statement** and has the form

var = expression ;

where *var* is an identifier and *expression* is an expression in Boolean form, consisting of variables and **operators**. The operators +, * and ' represent Boolean sum, product and inversion (i.e., AND, OR and NOT), respectively; the '*' operator is optional and may be omitted. e.g.,

```
out = reset' + clock * (in0' + (in1 * in2)) ;
```

is equivalent to

```
out = reset' + clock (in0' + in1 in2) ;
```

An expression, like a literal, may be complemented using the prime (i.e., apostrophe) symbol; e.g.,

```
x = (a (b + c)' + d)' ;
```

By default, the *expression* represents the *ON SET* of the variable *var*. Two symbols, ' and ˉ are appended to *var* to indicate the *expression* is its *OFF SET* or *DON'T CARE SET* respectively. The ˉ can also be used in the *expression* to indicate the *DON'T CARE SET* of a variable. Used alone, ˉ means the global *DON'T CARE SET* of the surrounding model.

There are two **built-in functions**. The arguments of these functions must be variables (not expressions). The built-in functions are:

D(a,c,e) A flow-through D-type latch, which has input *a*, is clocked by *c*, and is optionally enabled by signal *e*.

T(a,b) A tristate latch whose output is *a* when *b* is true, or high-impedance otherwise.

The use of a built-in function is indicated by the '@' symbol; e.g.,

```
out1 = @ D (sig1, clock') ;
```

In product to built-in functions, library functions may be called; these are defined as a separate model (see *.library* below).

Comments are identified by the symbol '#'. This symbol indicates that the remainder of the line is to be ignored by any program reading the *SLIF* description.

COMMANDS

.attribute *type_name variable_name parameters* ;

Specifies parameters for one variable (or one instance), named *variable_name*. The parameters consist of a sequence of strings, integers and floats, defined in the type *type_name*. If the type used allows for a variable number of parameters, the corresponding list has to be enclosed in parentheses "(" and ")".

.call *instance_name model_name (inputs ; inouts ; outputs)* ;

Creates an instance *instance_name* of the *SLIF* model *model_name*, which may be described in the same file or in a file specified by a *.search* statement. The called model may be a library element. Variables are linked according to the parameter listing; *inputs*, *inouts* and *outputs* are lists of variables separated by commas, which must agree in number and order with those in the called model.

.date *time_stamp* ;

Specify the time of the last modification (optional). The *time_stamp* format is *YYMMDDHHmmSS* where YY is the year, MM the month, DD the day, HH the hour, mm the minutes, and SS the seconds. Each element of the *time_stamp* is a two-digit number.

.endmodel *name* ;

Terminates the model. Each model has to be terminated by this declaration. There may be more than one model within the same file.

.global_attribute *type_name parameters* ;

Specifies parameters valid for an entire model.

.include *file_name* ;

Indicates that the information in *file_name* will be read as if it was part of the current file.

.inouts *var1 var2 ... varn* ;

Declares variables *var1 ... varn* as global bidirectional "inouts."

.inputs *var1 var2 ... varn* ;

Declares variables *var1 ... varn* as global inputs.

.library ;

Identifies the model as a library element.

.model *name* ;

Indicates the beginning of a new model and assigns it name *name*. Each model has to be declared using this declaration. Multiple models may be described in a single file.

.net *var1 var2 ... varn* ;

Lists variables that are connected together. The net will be named after one of the variables. If there are global inputs, outputs or inouts then the net will be inherit one of their names; otherwise it will be named after *var1*.

.outputs *var1 var2 ... varn* ;

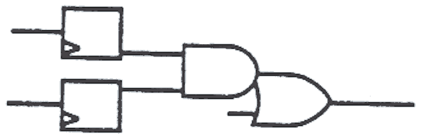
Declares variables *var1 ... varn* as global outputs.

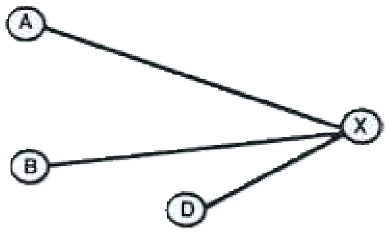
.search *file_name* ;

Indicates that models included in *file_name* may be used, if they are needed. Users are encouraged to use the absolute path to the file.

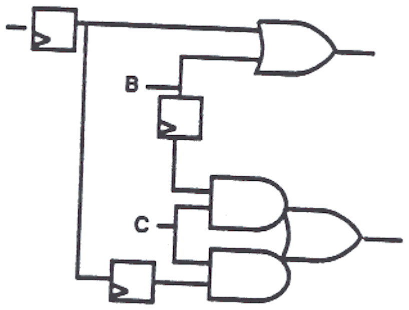
.type *type_name spec1 spec2 ... specn* ;

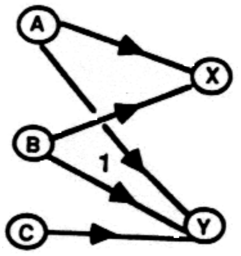
Declares a type *type_name* as a sequence of specifications *spec1 spec2 ... specn* where *spec* is any of *%d %f %s* (integer, float or string). A number may be used in front of a *spec*, to tell how many *specs* are to be used. A *spec* or set of *specs* can also be included inside parentheses, to indicate a variable number of that *spec* (or set of *specs*). A type is used whenever a *.attribute* or *.global_attribute* command is used. The type defines all the information that follows the type name. For *.attribute*, a string HAS to be inserted between the type name and the typed information. This string indicates the variable (or instance) to which the attribute will be attached.



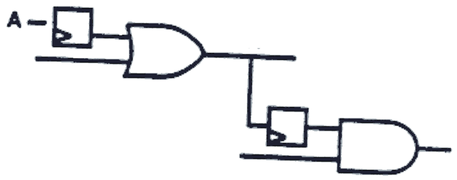


FIG

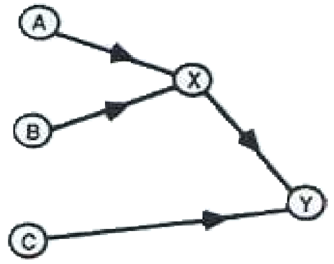


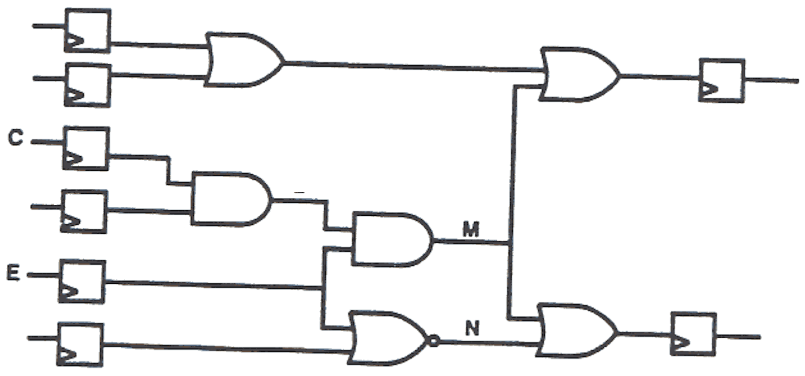


X =



FIG





FIG