Reencoding for Cycle-Time Minimization under Fixed Encoding Length

Balakrishnan Iyer Maciej J. Ciesielski
Department of Electrical & Computer Engineering, University of Massachusetts, Amherst, MA 01003.
{biyer,ciesiel}@ecs.umass.edu

Abstract— **This paper presents efficient reencoding and resynthesis algorithms for cycle-time minimization of multilevel implementations of synchronous finite state machines (FSMs) under a fixed encoding length. The proposed technique is applicable to both gate-level and technology independent synchronous network representations. We present two algorithms for identifying useful reencodings – one is based on Boolean cube representation applicable to technology independent synchronous networks and the other employs recursive learning techniques appropriate for gate netlists. We show that the proposed XOR/XNOR based reencoding technique explores a suffciently rich set of encodings to identify implementations with smaller cycle-times. The Boolean and structural interpretations of reencoding are explored and its relationship to isomorphic sequentially redundant faults is presented. We also show that the reencoded circuit always has a valid initial state and present a simple procedure to derive it. The effectiveness of the proposed technique is illustrated on a large set of benchmark circuits which indicates an average cycle-time improvement of 15.26% for a small area overhead of 3.56% over that of performance-driven combinational logic optimization.**

I. Introduction

In this paper, we describe an iterative sequential reencoding and resynthesis technique for minimizing the cycle-time of multilevel circuit implementations of finite state machines (FSMs) while maintaining the same number of registers. It has been observed [1], [2] that the area increase in retiming and resynthesis is often due to the increase in the number of registers in the circuit. In our approach, this problem is avoided by restricting the reencoding to a fixed number of registers. While this limits the scope of area-performance tradeoffs, we show that a sufficiently rich choice of solutions is available for a large number of circuits.

Previous approaches to performance driven synthesis of FSMs can be classified into 2 categories: *(i)* Optimization and speed-up of the combinational logic component[3], [4] which ignores the interaction of gates across register boundaries, *(ii)* Gate-level retiming which involves moving registers across combinational logic blocks so as to minimize the cycle-time[5]. The drawback of gate-level retiming is that it is guided by the minimization of cycle-time based on a precomputed function of the location of the registers in the network and does not consider any prospective logic optimization. There have been several attempts to overcome the limitations of gate-level retiming[6], [1], [7] by combining retiming with combinational logic resynthesis. However, no satisfactory solutions have emerged – either the cycle-time improvement has been marginal or it comes at a prohibitive area overhead.

In an orthogonal direction multilevel and sequential logic optimization techniques that leverage the advances in automatic test pattern generation (ATPG) have been proposed[8], [2]. These approaches employ ATPG techniques as a Boolean reasoning engine. ATPG based redundancy elimination represents an efficient implicit technique to minimize a circuit with respect to a don't care set[9].

Reencoding transformations similar to those presented in this paper have been applied to reduce the size of the reduced ordered binary decision diagram (ROBDD)[10] representations of FSMs[11], [12]. They show that the XOR transformation is the most promising among the set of all reencoding transformations.

II. Retiming and Resynthesis

Our approach is based on the following theorem due to Malik *et. al.*[6]:

Theorem 1: [6] Given a machine implementation M_1 with a STG G, and a state assignment S_1, it is always possible to derive a machine M_2 with the same STG G, and a state assignment S_2 by applying only a series of resynthesis and retiming operations on M_1.

The proof of the theorem is outlined in the transformations shown in Fig. 1. The combinational component of machine M_1 is augmented by an identity logic block $I = C \cdot C^{-1}$, where C is the mapping between the states of machine M_1 and M_2 and C^{-1} is the inverse mapping. The registers R_1 are then retimed backward across C^{-1}, and the combinational logic resynthesized, leading to the machine M_2 with encoding S_2 and register set R_2. The block C is often referred to as the *encoder block* and the block C^{-1} is called the *decoder block* in the sequel. The retiming and resynthesis approach outlined above is limited to circuits with the same STG and with the same encoding length. The reason is that both C and C^{-1} must exist, i.e. they must be Boolean functions. This is trivially true when C is an injective (one-to-one) mapping, and in particular when the encoding lengths of machines M_1 and M_2 are identical. This is exactly the type of transformation considered in this paper. While the synchronous behavior of machines M_1 and M_2 are equivalent, their corresponding combinational blocks have different functionalities and cost(area, performance, power). In this paper, we propose techniques to find an encoder block C which on retiming and resynthesis minimizes the cycle-time of machine M_2.

This work has been supported in part by a grant from NSF under contract No. MIP-9613864.

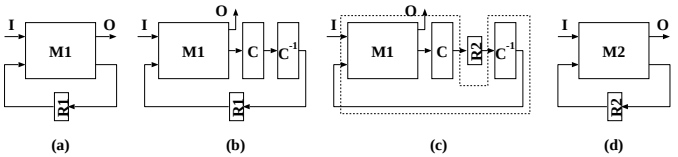

Fig. 1. Reencoding by Retiming and Resynthesis: (a) original machine M_1, (b) machine M_1 with an identity block $C \cdot C^{-1}$ appended, (c) Retiming across C^{-1} block, (d) Resynthesized machine M_2.

The reader is referred to [13], [14] for the definitions and notations used in logic synthesis and testing. In the sequel, we assume synchronous implementations using edge-triggered registers (D-flip-flops). In this paper we use redundancy removal techniques to simplify the circuit. It has been shown that in the presence of false paths, i.e. paths that cannot propagate signals under any circumstances, redundancy removal and Boolean simplification can actually increase the delay of the circuit by making the long false path active. Fortunately, redundancies can be removed from circuits with false paths with no adverse effect on delay but at the cost of area increase using the KMS algorithm[15]. Thus, without loss of generality, we assume that the longest topological path is true.

III. Motivating Example

In this section, we illustrate the problem by means of a simple motivating example. Consider the gate-level implementation of a FSM shown in Fig. 2(a).

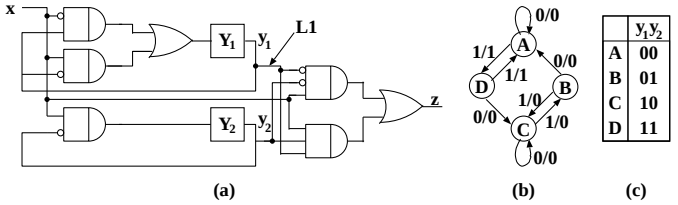

Fig. 2. Motivating example for cycle-time minimization (a) Gate-level implementation of a FSM, (b) the underlying STG, and (c) the state encoding.

The underlying STG and the state encoding are also indicated. The bubbles at the inputs of the gates indicate inversion. Assume that the target library provides two and three input AND/OR gates with all possible input polarities and that each two input gate has one unit of delay and the three input gate has two units of delay. The two boxes in the gate-level representation are the registers, annotated with the next state lines Y_1 and Y_2. The present state lines are indicated by y_1 and y_2. The primary input (PI) to the circuit is x and the primary output (PO) of the circuit is z. The circuit has a delay of 3 on the output node z. Note that the cycle-time cannot be reduced below this value by conventional gate-level retiming since the critical-path is the input-output path. Moreover, the cycle-time cannot be improved by the "retiming bottleneck removal" method presented in [7].

Now, consider the same STG with a different encoding of the states: $A \leftarrow 00$,

$B \leftarrow 01$, $C \leftarrow 11$ and $D \leftarrow 10$. In the new encoding we have swapped the codes of states C and D. A resynthesized circuit under the new state assignment is

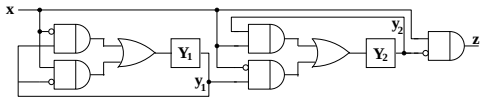

Fig. 3. Motivating example: Gate-level implementation of the reencoded FSM.

shown in Fig. 3. The reencoded circuit has a delay of 2 units.

The above reencoding can be put in the retiming and resynthesis perspective of [6], by noting that the required swapping of encodings between states C and D can be achieved by appending two XOR gates on the next state lines of the original machine in Fig. 2 as shown in Fig. 4(a). The reader may verify that the first XOR transforms the code of state C from 10 to 11 while the reverse is true

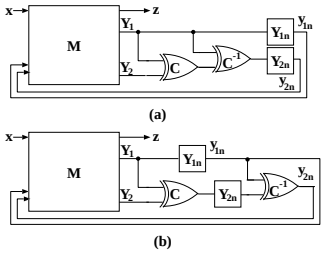

Fig. 4. Retiming and resynthesis perspective: (a) Appending XOR encoder and decoder blocks, (b) Retiming backward across the decoder XOR.

for state D. The second XOR exists to restore the state encodings of states C and D to their original encoding. Thus, the first XOR gate labeled C corresponds to the *encoder block* shown in Fig. 1 and the second XOR gate labeled C^{-1} is the *decoder block* in Fig. 1. Next we retime backwards across the second XOR gate as shown in Fig. 4(b) and resynthesize the resulting combinational circuit to get the circuit in Fig. 3 with the same number of registers.

Clearly it is quite impossible even for reasonably small machines to try all possible reencodings. In the sequel, we will present efficient algorithms for discovering such encodings leading to smaller cycle-times. We show how we arrive at the above reencoding from two perspectives: One is based on Boolean techniques and the other is based on recursive learning based implications[16] on the gate-level netlist.

A. Reencoding through Boolean Techniques

The Karnaugh maps (K-maps) for the next state functions Y_1, Y_2 and PO z for the example in Fig. 2 is shown in the top row of Fig. 5. We use Karnaugh map for illustration purposes – our algorithm and implementation is not based on Karnaugh maps. As can be seen from the K-map for z in the top row of Fig. 5,

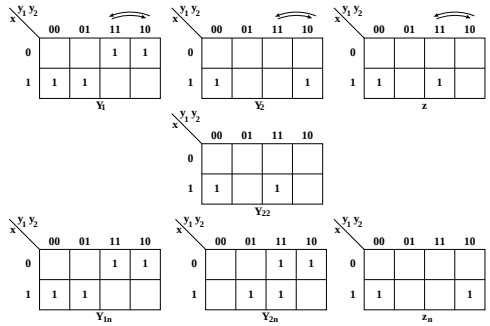

Fig. 5. Boolean perspective: Top row shows the K-maps for the original machine, second row is an intermediate step and the third row gives the K-map for the reencoded machine.

we can drop the dependency of z on y_1 by swapping the columns $y_1y_2 = 11$ and $y_1y_2 = 10$. This corresponds to swapping codes 11 and 10 in the design. In other words, we present 11 in the new circuit whenever 10 was presented in the original circuit, and vice-versa. Obviously, this cannot be done in isolation – the corresponding columns have to be swapped in the K-maps corresponding to Y_1 and Y_2 as indicated by the bidirectional arrows in Fig. 5. After the swapping of the rows indicated by the arrows, we have the K-map of z in the new circuit shown as z_n in Fig. 5. The K-map Y_{1n} in the new circuit remains unchanged from that of Y_1 after swapping the columns. However, computing the K-map corresponding to Y_2 in the new circuit is a little more involved. Swapping the columns in the K-map of Y_2 gives the K-map denoted Y_{22} in the figure. The new K-map Y_{2n} is computed as

$$Y_{2n} = Y_{22} \oplus Y_{1n} \quad (1)$$

To see why this is the case, note that the swapping of the columns in the original K-maps is synonymous to swapping the corresponding present state inputs in the original state transition table (STT). In other words, we have modified the circuit such that whenever the registers have a value of 11(10) a value of 10(11) is presented to the original combinational block. However, in order to leave the synchronous behavior of the circuit unaltered we have to ensure that whenever a value of 11 is produced at the next-state lines the value stored in the registers is modified to 10 and vice-versa. This reflects the change in the encoding of the states C and D in the STG. This is achieved by the XOR operation in Eq. 1. The K-maps for the reencoded machine are shown in the bottom row of Fig. 5.

B. Reencoding through Recursive Learning

The latest arriving signals on the critical path to z in the circuit of Fig. 2(a) are x, y_1 and y_2. If we can make a *stuck-at* fault on any of these lines undetectable and hence redundant then the circuit can be simplified by removing the redundancy. Since we are concentrating on *stuck-at* faults on the latest arriving signals, any simplification due to redundancy of these faults would lead to a new circuit with delay at least as small as the original circuit if not smaller. Consider a *stuck-at-0* fault on the line $L1$ in the original circuit of Fig. 2(a). This D-fault is indeed untestable if it cannot be propagated to the output z for **any and all possible** values of the primary input x. Since an input of $x = 0$ would trivially block the propagation of the fault, we will only consider the interesting case when $x = 1$. Also for any possible value of the present-state line y_2, the value of D should not be propagated to the output z, i.e. it must be a 0 or 1 for both faulty and fault-free circuits. To learn the conditions for redundancy of the *stuck-at-0* fault on the line $L1$, we perform two rounds of recursive learning. In the first iteration we learn the implications of choosing $z = 0, x = 1$ and $y_1 = D$. This gives a mandatory assignment of $y_2 = \overline{D}$. Similarly, the implications of choosing $z = 1, x = 1$ and $y_1 = D$ leads to a mandatory assignment of $y_2 = D$. This implies that the propagation of D to the output z would be blocked if $y_2 = \overline{D}$ or $y_2 = D$. Thus, we have to inject a "fault" on line y_2. This can be accomplished by introducing the XOR gate marked $X2$ as shown in Fig. 6 – whenever the *stuck-at-0* fault is activated, i.e. $y_1 = 1$ and $y_2 = 1(0)$ in the "good" circuit, the output of $X2$ is $0(1)$ which corresponds to $D(\overline{D})$. However, the addition of this gate alters

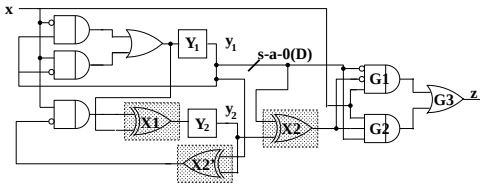

Fig. 6. Reencoding through Recursive Learning: new reencoded machine where the *stuck-at-0* fault is undetectable.

the functionality of the circuit. To compensate for this altered functionality we propagate $y_2 = \overline{D}$ or $y_2 = D$ back across the register marked Y_2 into the previous time frame. This is accomplished by adding the XOR gate $X1$, which in turn is compensated by the XOR gate $X2'$ (c.f. Fig. 6). Now the functionality of the circuit in Fig. 6 is identical to that of the original fault-free circuit. Noting that the *stuck-at-0* fault shown in the figure is redundant we can simplify the output logic by removing gates $X2$, G2 and G3 and the connection from y_1 to $G1$. Redundancy removal and resynthesis leads us to the final reencoded circuit shown in Fig. 3.

The STG of the reencoded FSM is identical to that of the original FSM except for a relabeling of the states with different encodings. In this light, reencoding can be seen as a technique to induce isomorphic sequentially redundant faults (SRF)[17] such that the new circuit has better delay properties.

IV. Reencoding Algorithms

For a design with n register variables, there are $2^n!$ possible encodings and the problem of finding the *globally* optimal encoding is intractable. In this section we present iterative algorithms to discover encodings resulting in smaller cycle-times. Each step in the iteration involves reencoding one of the latch variables by using the previously explained XOR/XNOR transforms. Each of these steps can be seen as a *two-bit reencoding*[11]. Our algorithms for reencoding are based on the following observations:

1. The number of minterms in the on-set of the functions computing the next-state values and the primary outputs remains invariant under the reencoding.
2. Interchange of minterms and cubes corresponding to different primary input alphabets (transition predicates) is not permitted. It can be shown that such interchanges lead to a increase in the number of registers in the design[18]. Thus, the

permissible reencoding transformations are restricted to swaps in the present-state bit locations in the K-maps. In the recursive learning paradigm of Section III-B, this translates to the observation that the redundancy of an injected fault does not require a D or a $\overline{D}$ on any of the primary inputs.

3. In the course of reencoding we may swap the code assigned to a state with an unused code, i.e. one of the unused codes is now used and one of the codes in the used set becomes an unused code. However, the unused state codes are typically used as don't cares in the optimization of the combinational logic. After the code swap the original don't care set used to optimize the circuit is partially invalidated and has to be replaced by a new don't care set. To ensure design correctness we need to propagate these changes throughout the design. We assume that the reachable state set (used codes) are available as part of the design specification as in SIS[19] or have been computed using implicit state traversal techniques[13], [20].

It has been shown in [11] that the iterative application of the XOR/XNOR transforms provide a good basis for the design of effective reencodings. They use counting arguments to prove the following result:

Theorem 2: [11] The number of possible encoding transformations, $t(n)$ achieved by the iterative application of the XOR/XNOR is given by:

$$t(n) = \prod_{i=0}^{n-1} \left(2^n - 2^i\right) \tag{2}$$

where, n is the number of bits in the encoding. ■

Although, this is much smaller than the number of all possible reencodings, the XOR/XNOR operators provide a rich set of reencoding transformations. In fact, all of the $4!(=24)$ one-to-one (bijective) mappings over 2 Boolean variables can be expressed as a composition of 3 basic operators: *(i)* the XOR/XNOR, operation over the 2 variables *(ii)* the register-variable permutation (or renaming), and *(iii)* the identity (no reencoding) operator[11]. Of these, the register renaming and identity transforms have no effect on the cycle-time of the implementation. Thus, the XOR/XNOR transform is the *only* possible one-to-one mapping over two Boolean variables. This is expressed in the following theorem.

Theorem 3: An encoding/decoding function pair implementing a two-bit reencoding of a synchronous network is valid if both members of the pair take one of the following two forms:

$$y_k = f(Y) \cdot (y_k \oplus y_l) \tag{3}$$

OR

$$y_k = f(Y) \cdot (y_k \overline{\oplus} y_l) \tag{4}$$

where, $y_i, \forall i$ are register variables and $f(Y)$ is some Boolean function, with $Y = \{y_i | i \notin \{k,l\}\}$.

Proof: The proof follows from the following observations: the function $f(Y)$ in Eq. 3, 4 serves as a "restriction" which limits the reencoding to states whose encodings satisfy the function $f(Y)$. Thus, without loss of generality we can concentrate on the XOR/XNOR forms involving y_k and y_l in the above equations. The rest of the proof follows from previous arguments. ■

Note that this theorem provides only the necessary condition – in fact, there are many reencodings not explored by the proposed transform. However, the above theorem constitutes a necessary and sufficient condition when the original STG is reduced, has 2^n states and is encoded in n (minimum number) bits.

A. Algorithm based on Boolean Techniques

The algorithm is based on "truth-table permutations" to cluster the cubes used to compute the next-state and primary outputs. This can be computed in a straightforward manner on a ROBDD representation of the functions[12]. Next we iterate over the cubes in the next-state/primary output function(s) with the largest delay. In each iteration we try to merge the cubes and evaluate the impact of the transform on the circuit delay using the user specified delay model. If the transformation reduces the delay then it is accepted. The process is iterated until no improvement in delay results. Convergence of the procedure is ensured since delay is strictly decreasing and the slacks on non-critical paths are decreasing. Thus, the process tends to equalize the delay of all paths. If the don't care conditions are changed then the don't care set is updated and the network is resynthesized using the new don't care set. The algorithm is outlined in Algorithm 1.

B. Algorithm based on Recursive Learning

The algorithm based on the ATPG technique using recursive learning as the implication engine is shown in Algorithm 2. The algorithm proceeds by identifying the stems on the critical path in the network. Stuck-at-0 and stuck-at-1 faults are induced on these stems and recursive learning is used to learn the conditions for the redundancy of these faults. Stuck-at controlling value faults on these stems are attempted first failing which non-controlling values are tried. If the fault is deemed redundant for all primary input values then the circuit is transformed and resynthesized using redundancy removal. If this involves swapping codes with a previously unused state code then part of the old don't care set is invalidated. Optimization under the new don't care network is accomplished using ATPG techniques presented in [9].

Inputs: Encoded FSM M, Delay Model D, Initial Delay t_d.
Outputs: Reencoded FSM M' with smaller delay t_r.
loop
  Do a critical path trace. Record initial circuit delay (t_c).
  Compute C as the set of maximal cube clusters for all outputs.
  Select output o_i with critical delay.
  S = Maximal Cube Clusters for o_i.
  for $i = 0$ to $|S|$ **do**
    Merge cube clusters.
    if Don't care (DC) network has changed **then**
      Remove old DC, resynthesize network with new DC.
    end if
    Evaluate t_n = network delay with delay model D.
    if $t_n < t_c$ **then**
      Accept the reencoding. $t_c = t_n$.
    end if
  end for
  If $t_c \geq t_d$ then **break**.
end loop
Return reencoded machine.

Algorithm 1: Reencode machine M using Boolean Techniques.

Inputs: Gate-level FSM M, Gate library L, Initial delay t_d.
Outputs: Gate-level FSM M', with delay $t_c \leq t_d$.
loop
  Trace critical path P. $S =$ stems on P.
  for each stem in S **do**
    $c =$ controlling value of stem.
    for v in $c, \bar{c}$ **do**
      Induce fault s-a-v.
      Use recursive learning to compute the reencoding, if any.
      If reencoding found **then** break;
    end for
    If no reencoding found **then** continue;
    Transform circuit. Resynthesize (with new DC, if DC has changed). break;
  end for
  If no improvement in delay **then** break.
end loop
Return reencoded machine.

Algorithm 2: Reencode machine M using Recursive Learning.

V. Initial State Computation

Since we perform backward retiming across the decoder block we have to ensure that the retimed circuit has a valid initial state. The following theorem proves that a valid equivalent initial state always exists for the resynthesized circuits produced by the proposed technique.

Theorem 4: Given a initial implementation M with initial state S_0, the new implementation M' produced by the proposed technique always has a valid equivalent initial state S_0'. If the initial implementation M has a synchronizing (initializing) sequence that brings the machine into a known starting state then the same initializing sequence would bring the machine into the same starting state.

Proof: Refer to [18] for the proof. ■

In the proposed technique, only backward retimings are performed across the gates and fanout stems thus satisfying the conditions for *0-cycle safe replaceability*[21].

VI. Results

We implemented a prototype version of the proposed algorithms within the SIS[19] framework. The results for the proposed approach on the *LGSynth89* and *LGSynth91* benchmark suites are presented in Table I. As explained earlier in the paper, we leverage the existing techniques for performance driven synthesis of combinational logic blocks[3] and available in SIS[19]. Thus, in the sequel, we present the area overheads and the speedup achieved by the proposed technique over and above that achieved by performance driven synthesis of the combinational portion[3]. The initial circuit netlist was generated by minimizing the state machine using *stamina*[22], state assignment using *jedi*[23], and logic optimization using SIS[19] (using the *script.delay*[3] and mapped onto the *lib2* library). The critical path(s) of the mapped circuit was traced and reencoding and resynthesis was performed to reduce the delay of the critical path(s). The resulting circuit was remapped for performance onto the same target library. Gate-level retiming was done using the *retime* command in SIS.

The results for the proposed technique, in general, shows a super-linear improvement in performance with respect to the area increase. In summary, the proposed approach achieves an average speedup of **14.91%** in the combinational delay and **15.26%** in cycle-time with an area overhead of **3.56%** over that of circuits synthesized with *script.delay*. Thus, the proposed technique is effective in improving the performance of state machine implementations with a modest increase in the silicon area. Also, the technique compares very favorably, both in terms of the area overhead and cycle-time, with gate-level retiming techniques that have been traditionally employed to improve performance.

TABLE I
RESULTS FOR REENCODING

FSM Name	$\|S\|$	$\|O\|$	$\|R\|$	Script.delay			Reencoded			% Inc	% Spd	% Spd	SD + Retime			Reenc. + Retime		
				Area	Delay	Cycle	Area	Delay	Cycle	Area	Delay	Cycle	$\|R\|$	Area	Cycle	$\|R\|$	Area	Cycle
bbara	7	5	3	73776	7.15	9.91	70992	6.24	8.63	-3.77	14.58	14.83	16	134096	8.83	8	94192	8.05
bbsse	13	11	4	142448	9.66	12.53	158224	7.96	11.58	11.07	21.36	8.20	-	-	-	11	190704	11.11
bbtas	6	5	3	35728	4.34	5.26	40368	4.09	5.62	12.98	6.11	-6.41	-	-	-	-	-	-
beecount	4	6	2	46400	4.45	6.27	46400	4.45	6.27	0.00	0.00	0.00	-	-	-	-	-	-
cse	16	11	4	275616	12.67	18.19	299744	8.37	11.60	8.75	51.37	56.81	-	-	-	-	-	-
dk14	7	8	3	135488	9.99	14.74	113680	8.63	12.76	-16.10	15.76	15.52	5	144768	13.74	7	132240	12.03
dk15	4	7	2	94192	11.19	16.02	114608	9.25	12.16	21.67	20.97	31.74	7	117392	14.96	-	-	-
dk16	27	8	5	330832	17.24	23.86	330832	17.24	23.86	0.00	0.00	0.00	19	395792	23.20	-	-	-
dk17	8	6	3	93264	8.86	11.54	88160	8.36	12.17	-5.47	5.98	-5.18	4	97904	11.37	5	97440	11.25
dk27	7	5	3	41296	5.42	8.07	41296	4.66	6.70	0.00	16.31	20.44	5	50576	7.28	-	-	-
dk512	15	7	4	85840	6.31	9.40	85840	6.31	9.40	0.00	0.00	0.00	10	113680	9.27	-	-	-
dvram	35	21	6	243136	7.76	11.86	255200	6.90	10.09	4.96	12.46	17.54	19	303456	11.20	14	292320	9.89
ex1	18	24	5	354960	9.64	13.22	352640	8.88	12.32	-0.65	8.56	7.31	-	-	-	-	-	-
ex3	5	5	3	40368	5.53	6.02	39904	4.42	5.86	-1.15	25.11	2.73	-	-	-	-	-	-
ex4	14	13	4	101152	7.45	10.43	102080	6.24	8.42	0.92	19.39	23.87	8	119712	9.91	7	116000	8.20
ex5	4	4	2	25984	3.44	3.83	25984	3.44	3.83	0.00	0.00	0.00	-	-	-	-	-	-
ex6	8	11	3	116928	12.49	16.01	130384	8.76	11.11	11.50	42.58	44.10	-	-	-	-	-	-
ex7	4	4	2	41760	7.10	9.15	40832	6.41	8.99	-2.22	10.76	1.78	4	51040	8.97	4	50112	8.56
lion	4	3	2	26912	4.52	5.65	26912	4.52	5.65	0.00	0.00	0.00	-	-	-	-	-	-
planet	48	25	6	676976	13.24	19.07	642640	11.44	15.84	-5.07	15.73	20.39	-	-	-	-	-	-
sand	32	14	5	676048	12.75	16.88	771632	11.06	15.17	14.14	15.28	11.27	-	-	-	-	-	-
scf	97	63	7	1039824	10.18	14.48	1033328	9.93	13.26	-0.62	2.52	9.20	24	1118704	14.09	-	-	-
styr	30	15	5	610624	12.48	16.97	610624	12.48	16.97	0.00	0.00	0.00	-	-	-	-	-	-
tbk	16	7	4	342432	14.97	23.84	324336	11.67	18.43	-5.28	28.28	29.35	11	374912	22.95	20	398576	15.99
train4	4	3	2	31088	4.64	5.52	31088	4.64	5.52	0.00	0.00	0.00	-	-	-	-	-	-
s1488	48	25	6	793440	13.21	17.63	766528	11.40	14.90	-3.39	15.88	18.32	14	830560	17.22	-	-	-
s1494	48	25	6	781840	14.08	17.90	809680	13.26	16.79	3.56	6.18	6.61	-	-	-	-	-	-
s27	5	4	3	33408	5.41	6.45	39440	5.10	6.03	18.06	6.08	6.97	6	47328	5.30	-	-	-
s298	135	14	8	1645344	16.75	22.59	1859712	12.57	17.18	13.03	33.25	31.49	-	-	-	27	1947872	16.64
s386	13	11	4	163328	9.09	12.88	159152	8.72	12.08	-2.56	4.24	6.62	7	177248	12.29	12	196272	11.49
s420	18	7	5	140128	10.36	13.90	133632	7.91	9.99	-4.64	30.97	39.14	-	-	-	-	-	-
s510	47	13	6	387440	14.24	20.39	428272	11.00	14.93	10.54	29.45	36.57	12	415280	20.02	23	507152	14.65
s820	24	24	5	373520	10.68	13.64	387440	9.19	12.21	3.73	16.21	11.71	12	406000	13.23	-	-	-
s832	24	24	5	391616	9.64	13.20	402288	9.01	11.76	2.73	6.99	12.24	7	400896	12.68	8	416208	11.58
Sum				**10393136**	**326.93**	**447.30**	**10763872**	**284.51**	**388.08**									
Avg.				**305680**	**9.62**	**13.16**	**316584**	**8.37**	**11.41**	**3.56**	**14.91**	**15.26**						
Min										**-16.10**	**0.00**	**-6.41**						
Max										**21.67**	**51.37**	**56.81**						

LEGEND: $|S|$ = # of states, $|O|$ = # of outputs and $|R|$ = # of registers. Columns 5 through 7 indicate the area, delay and cycle-time achieved with *script.delay*. Columns 8 through 10 give the corresponding values for the reencoding scheme. Column 11 gives the % area increase over *script.delay* due to reencoding (negative values indicate decrease in area). Columns 12 and 13 give the % speedup in combinational delay and cycle-time, respectively. Columns 14 through 16 indicate the # of registers, area and cycle-time after retiming the circuits produced by *script.delay*. Columns 17 through 19 give the corresponding numbers for retiming the circuits produced by reencoding. A "-" in the last 6 columns indicates that retiming failed to improve the cycle-time.

VII. CONCLUSIONS

In this paper we have presented reencoding and resynthesis algorithms based on XOR/XNOR reencoding transformations for cycle-time minimization of FSM implementations. The proposed technique satisfies the constraint that the number of registers in the circuit remain constant. We also proved that a valid initial state always exists for the resynthesized circuit. Techniques to identify potentially useful reencodings were presented for synchronous network representations as well as gate-level netlist representations. The Boolean and structural interpretations of reencoding were explored and its relationship to isomorphic sequentially redundant faults was presented. In summary, the reencoding and resynthesis for performance can be seen as a technique for "slack redistribution" – reducing the depth of long paths while increasing the depth of short paths. The effectiveness of the technique is attested by the results on the benchmark circuits.

REFERENCES

[1] G. De Micheli, "Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization," *IEEE Trans. on CAD*, vol. 10, pp. 63–73, Jan. 1991.

[2] L. Entrena and K.-T. Cheng, "Combinational and Sequential Logic Optimization by Redundancy Addition and Removal," *IEEE Tr. on CAD*, vol. 14, pp. 909–916, July 1995.

[3] H. Touati, H. Savoj, and R. Brayton, "Delay Optimization of Combinational Logic Circuits through Clustering and Partial Collapsing," in *Intl. Workshop on Logic Synthesis*, 1991.

[4] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Timing Optimization of Combinational Logic," in *ICCAD*, pp. 282–285, 1988.

[5] N. Shenoy and R. Rudell, "Efficient Implementation of Retiming," in *Proc. ICCAD*, 1994.

[6] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques," *IEEE Trans. on CAD*, vol. 10, pp. 74–84, Jan. 1991.

[7] S. Chakradhar, S. Dey, M. Potkonjak, and S. Rothweiler, "Sequential Circuit Delay Optimization using Global Path Delays," in *Proc. 30th DAC*, pp. 483–489, June 1993.

[8] W. Kunz, D. Stoffel, and P. Menon, "Logic Optimization and Equivalence Checking by Implication Analysis," *IEEE Tr. on CAD*, vol. 16, pp. 266–281, Mar. 1997.

[9] D. Brand, R. Bergamaschi, and L. Stok, "Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions," Tech. Rep. RC 20127, IBM T.J. Watson Research Center, Yorktown Heights, NY, Sep. 1995.

[10] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Tr. on Comp.*, vol. C-35, pp. 677–691, Aug. 1986.

[11] C. Meinel and T. Theobald, "Local Encoding Transformations for Optimizing OBDD-Representations of finite state machines," in *Proc. FMCAD'96*, pp. 404–418, 1996.

[12] M. Fujita, Y. Kukimoto, and R. Brayton, "BDD Minimization by Truth Table Permutations," in *Proc. IWLS'95*, 1995.

[13] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[14] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[15] K. Keutzer, S. Malik, and A. Saldanha, "Is Redundancy Necessary to Reduce Delay?," in *Proc. DAC*, pp. 228–234, June 1990.

[16] W. Kunz and D. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems – Test, Verification and Optimization," *IEEE Tr. on CAD*, vol. 13, pp. 1143–1158, Sep. 1994.

[17] A. Ghosh, S. Devadas, and A. Newton, *Sequential Logic Testing and Verfication*. Kluwer Academic Publishers, 1992.

[18] B. Iyer and M. Ciesielski, "Reencoding for Cycle-Time Minimization under Fixed Encoding Length," Tech. Rep. TR-98-04, Department of ECE, University of Massachusetts, Amherst, MA 01003., 1998.

[19] E. Sentovich *et al.*, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley., 1992.

[20] G. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[21] C. Pixley, V. Singhal, A. Aziz, and R. Brayton, "Multi-level Synthesis for Safe Replaceability," in *Proc. ICCAD'94*, 1994.

[22] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby, "Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines," *IEEE Tr. on CAD*, vol. 13, pp. 167–177, Feb. 1994.

[23] B. Lin and A. Newton, "Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages," in *Proc. VLSI'89 Conf.*, Aug. 1989.