Wave Steered FSMs

Luca Macchiarulo, Shih-Ming Shu and Malgorzata Marek-Sadowska
University of California, Santa Barbara

Abstract

In this paper we address the problem of designing very high throughput finite state machines (FSMs). The presence of loops in sequential circuits prevents a straightforward and generalized application of pipelining techniques, which work so well for combinational circuits, to increase FSM performance. We observe that appropriate extensions of the "wave steering" technique [17,18] are possible to partially overcome the problem. Additionally we use FSM decomposition theory to decouple state variable dependencies. Application of these two techniques to MCNC benchmarks resulted in a factor of 3 average throughput increase as compared to a standard cell implementation, at the expense of factor 3.7 area and less than factor 2 latency penalties.

1. Introduction

Pipelining has always been ranked among the most powerful techniques to enhance performance without resorting to better technologies. However, in classic literature very few attempts have been pursued to use this powerful methodology in an FSM context.

One of the most lucid accounts on this issue, [16], points out the main problem of pipelining sequential machines. Any pipelining scheme will trade off latency for throughput, but the throughput of a sequential system is inherently limited by an iteration bound related to the logical behavior of the system rather than physical constraints. To clarify the question, let's take a look of figure 1a, which depicts a typical sequential system. It has memory of the past, or in other words, its outputs are not uniquely determined by the primary inputs alone, but also by some history condensed in the state variables. If an attempt is made to pipeline this system, the result looks like figure 1b, where the single stage of the computation is completed at a very fast rate, but the change to the next state is bounded by the latency. It is so because the next primary input vector has to be processed together with the state bits; therefore no matter how fast the single phases of computation are executed, the final speed of the system will be dictated by the overall latency τ. There have been different approaches to overcome this obstacle ([11], [16]), but all of them target particular problems or classes of problems.

We think, though, that a systematic way of dealing with the iteration bound exists, and is related to the deep logic structure of the FSMs. In fact, even if the "black box" description of an FSM (figure 1a) entails a hard constraint on iteration bound, it is easy to see that in particular cases the machine works like the one shown in figure 1c, where not all the state variables are needed from the first stage of computation. If such decoupling of complete state knowledge can be extended far enough, a situation like that in figure 1d is feasible, where the performance-limiting loop has been split into smaller loops and local machines are enabled to act much faster. This leads almost naturally to the classical theory of machine decomposition as detailed by Hartmanis [10]. However, even if such decompositions existed for each machine, a second, different problem, arises: there is no guarantee that a system like the one depicted in figure 1d will always be better (in terms of clock cycle, because the latency will be degraded as in any pipelined scheme) than its "black box" counterpart. In fact, even if there is a rough connection between complexity of logic and critical path delay, this relation is not trivial.

The solution we propose here relies on a design method for FSMs which guarantees better results if an appropriate decomposition is found. This is in turn related to the concept

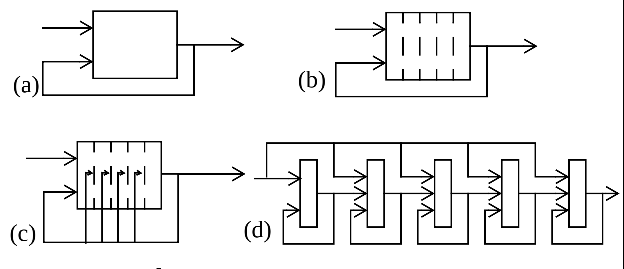


Figure 1. Pipelining FSM.

of **decomposition's throughput**. Given this definition we can explore the space of "good" decompositions with a clear cost function. Some good decompositions, including the cascaded one, are detailed. To verify feasibility of our approach, we have applied the classical decomposition theory, while more recent developments (like [1], [6], [22]) will be integrated in the future. We developed a novel partition lattice traversal technique to visit the solution space. We also briefly mention implementation issues related to the problem of BDD based encoding and simplification techniques ([12]). We finally present experimental results for a subset of MCNC benchmarks and discuss the pros and cons of the proposed method along with possible enhancements.

All this reasoning stands (or falls) on the existence of practical systems which could take advantage of high throughput machines. According to [16] such systems arise naturally in the field of signal and image processing, and many FSMs are cycle-intensive in a sense that it is less important to reduce the latency between inputs and outputs than it is to reduce the time interval between consecutive legal outputs. In general our approach will be advantageous to all systems with relatively loose loops connecting data with control parts, or FSMs occurring in general data-dominated problems [11].

Previous work related to the problem studied here, besides [11] and [16], can be found in early papers like [5] or [8], where the possibility of high throughput FSMs was clearly addressed, but didn't find immediate application, we believe due to the lack of an implementation framework like the wave steering proposed here. In the present work we are also indebted to all the papers on machine decomposition, from the classical (and always inspiring) [10] to the more recent works [1][2][7][14].

The main point that differentiates our work from the previous is the perspective that sees decomposition as a tool which can immediately provide performance enhancements rather than an encoding aid.

2. Wave steering of combinational and sequential circuits

Recently the Wave Steering Technique has been proposed [17][18] which tremendously increases the throughput of combinational logic. The idea is to allow several signal waves to co-exist in combinational circuit. Each wave contains encoded information about a subset of input variables. As a particular wave travels through the circuit it is enhanced by information carried by consecutive variables of the same input vector.

The idea of wave steering is based on the possibility of computing a function taking the effect of one variable at a time to enhance the information carried by the variables seen so far. This is allowed through the use of a BDD (Boolean Decision Diagram) scheme of computation. There are standard ways of mapping BDD representations of a Boolean functions into Pass Transistor Logic (with minor variations: see [3][4][19][21]). Each node of a BDD is substituted by a multiplexer realized by a couple of pass transistors controlled by the signal corresponding to the variable and its negation. To achieve a wave steering scheme it is fundamental to electrically decouple different BDD levels, by controlling the adjacent levels with alternating phases of clock signals.

The wave steering principle requires that at any given clock phase every other level is active, that is, any multiplexer connects one of its inputs with the output, while the other half of the levels are idle (they maintain memory of the previous computation). In order for this scheme to work properly the input bits must be applied at the right time, such that they act on the right wave of partial outputs. This implies input skewing: the input bits controlling higher levels in the BDD structure have to be delayed with respect to the one controlling lower levels. This is achieved through the use of dynamic flip-flops.

Previous works ([17] and [18]) show the good performance of this methodology with the use of a particular decision diagram, while more recent results indicate the feasibility of the approach with regular PTL mapped BDDs (with some modification to account for placement and routing problems), particularly for arithmetic functions. In general it seems that the approach is advantageous whenever it is acceptable to trade-off area and latency for throughput.

3. Wave steered FSM

As stated in the introduction, the possibility of a pipelined FSM is challenged by the iteration bound, but solvable if the "big loops" could be substituted by many small ones. Applying the wave-steering technique detailed in the previous section directly to the combinational portion of an FSM may not lead to substantial throughput improvement because of necessity to feedback the state bits.

We define *combinational latency* of an FSM whose combinational portion is implemented in wave steering technique as the time needed to compute the next state bits. It is proportional to |input bits|+|state bits|. As an example, consider a machine in fig. 2a. Its state is coded into 4 bits and the combinational part is wave steered. If we fix the order of

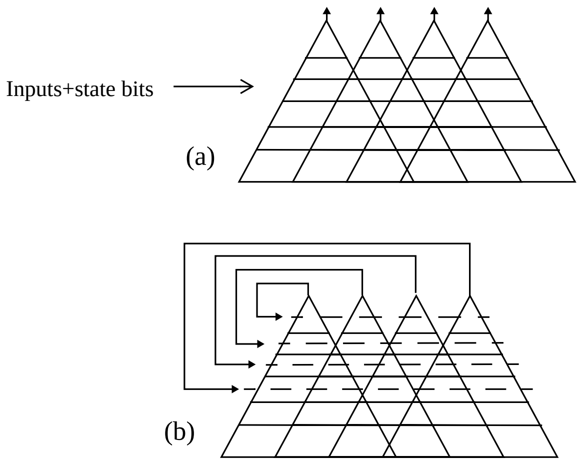


Figure 2. Wave-steered FSM.

the variables in such a way that state bits appear on the top of the BDDs (see fig.2b), the **sequential latency**, that really limits the throughput by stalling the pipeline, becomes simply dependent on the number of state bits.

Such a scheme, however, is acceptable only if the number of state bits is relatively small. It is possible to apply the Wave Steering idea hierarchically to achieve the desired speedup. The top level wave steering is an appropriate FSM decomposition which results in iteration loops with small number of bits. The bottom level wave steering works as usual, by accumulating partial information on the local

BDDs that eventually give partial results on the next state. The top level wave contains the information about all the input variables and a partial information about the present state, that is used to compute the partial information on the future state, to close the first loop. The information on the next state is used by the next sub-machine which can process it together with a complete information on the input to refine the knowledge on future state. It is clear that in this way the next-state output is not the result of a one-shot computation but rather a progressive refinement of partial information. This means that the entire next state vector will be available only after all the machines have closed their loops. As every machine can act independently, the overall throughput of the system is no longer limited by the total number of state bits but rather by the maximum number of state bits to be processed inside a single machine (see figure 3).

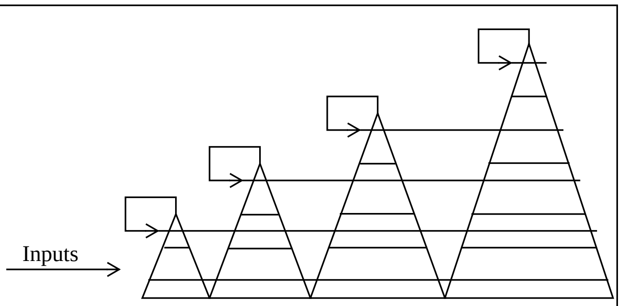


Figure 3. Wave steered decomposed FSM.

It is clear that the existence of such a topology is related to the possibility of finding decompositions of the original machines. We however have an advantage with respect to the usual way decomposition theory is applied: while ordinary FSMs have cost functions relating the quality of a decomposition to the result of a very long and complex process (comprising state assignment, logic minimization, and technology mapping), in our case the quality of decomposition can be assessed on purely logical grounds, namely by the number of bits in the critical loops. This brings us to defining the **decomposition's throughput** as the maximum number of bits which have to be processed simultaneously by any machine in the decomposition. With the cost function given by the throughput, we can proceed to analyze the feasible decompositions, and build algorithms to identify them.

4. Decomposed FSMs

4.1 Good decompositions

Here we list some machine decompositions that best fit the wave-steering scheme. The crucial point is that they have to reduce the number of bits needed to represent state information produced and processed in a clock cycle. The scheme detailed in figure 4a corresponds to the cascaded decomposition ([1][10]) and Figure 4b shows a generic loop-free decomposition. In both of these cases the throughput gain is directly related to the maximum number of state variables of the component machines. A more complex situation, where feedback is present, is shown in figure 4 c. In this case the iteration bound is given not only by the number of state bits computed by the single machine, but also by the communication complexity between different connected machines. The same principle applies to generic topologies like those of figure 4 d. Even though cases c and d cannot rely on simplistic figures of merit like the maximum number of bits, nonetheless the performance can be always computed on the basis of purely structural information, without considering encoding, synthesis, or mapping data.

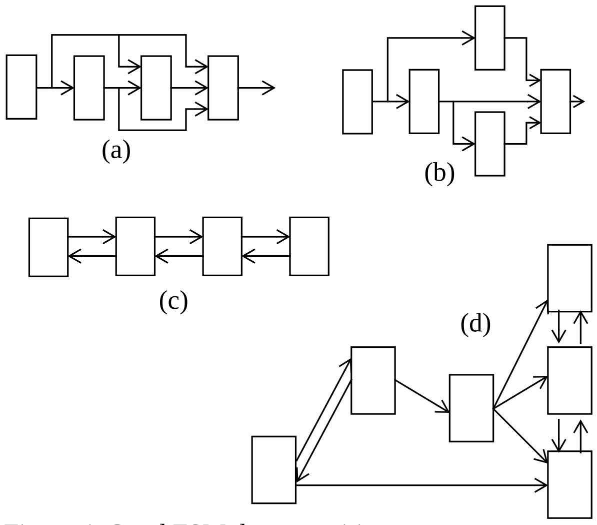


Figure 4. Good FSM decompositions.

In general, we accept any multiway decomposition which guarantees the presence of short iteration loops defined in terms of bits to be processed.

4.2 FSM serial decomposition

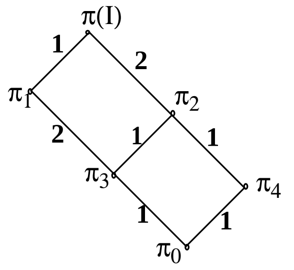
We have implemented our ideas for FSMs decomposed into cascades. Therefore, from now on, we will focus on FSM serial decomposition only. In this section we will rephrase pertinent definitions, recall theoretical background from [10],[13] and explain our implementation.

Definition 1: A finite state machine (FSM) is a 5-tuple $M = (S, I, O, \delta, \lambda)$, where S is a finite non-empty set of states, I a finite non-empty set of inputs and O a finite non-empty set of outputs. $\delta: S \times I \rightarrow S$ is called the transition (or next state) function and $\lambda: S \times I \rightarrow O$ the output function of M.

The functions δ and λ are represented by a state transition table. Figure 5 shows transition table of machine A, an example machine used throughout this paper.

PS	NS / OUT	
	x = 0	x = 1
0	4/0	1/1
1	4/0	0/1
2	3/0	0/0
3	2/1	5/0
4	5/1	2/0
5	4/0	2/0

Figure 5. The example Machine *A*



$\pi(I) = \{\overline{0,1,2,3,4,5}\}$
$\pi_1 = \{\overline{0,1,2,5}; \overline{3,4}\}$
$\pi_2 = \{\overline{0,1}; \overline{2}; \overline{3}; \overline{4,5}\}$
$\pi_3 = \{\overline{0,1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}\}$
$\pi_4 = \{\overline{0}; \overline{1}; \overline{2}; \overline{3}; \overline{4,5}\}$
$\pi_0 = \{\overline{0}; \overline{1}; \overline{2}; \overline{3}; \overline{4}; \overline{5}\}$

Figure 6. The lattice *L*(*A*) of machine *A*.

Definition 2: [10] Let the expression $s \equiv t\ (\pi)$ mean that states s and t are in the same block of partition π.
Then a partition π on the set of states of the machine *M* is said to have the *substitution property* (S.P.) if and only if, for all states s and t, $s \equiv t\ (\pi)$ implies that $\delta(s, i) \equiv \delta(t, i)(\pi)$ for all *i* in *I*.

In other words, the partition π on *S* of *M* has the substitution property if and only if each input maps blocks of π into the blocks of π. For example, $\pi_1 = \{\overline{0,1,2,5}; \overline{3,4}\}$ is a S.P. partition of the machine A.

We denote a partition consisting of a single block as π(*I*), and the partition consisting of blocks each of which has only one state as π_0. Each machine has only one π(*I*) and one π_0.

Since the operation of *M* determines unique block to block transformations on S.P. partition π, we can think of these blocks as the states of a new state machine defined by π and *M*.

Let π_1, π_2 be two partitions on *S*, and "≤" (*equal or smaller than*) denotes the partial order operator, then $\pi_1 \leq \pi_2$, if and only if each pair of elements which are in a common block of $\pi_1$are also in a common block of π_2.

It has been shown in [10] that if π_1 and π_2 are S.P. partitions on the set of states of a sequential machine, then so are the partitions $\pi_1 \cdot \pi_2$ and $\pi_1 + \pi_2$, where the two operators represent appropriate meet and join operations. With partial ordering, the set of all S.P. partitions (including π_0 and $\pi(I)$) of a sequential machine *M* forms a lattice *L*(*M*), where each node in the lattice is a partition and the edge represents the partial ordering between the nodes. Node π_1 is drawn on a lower level than the node π_2 whenever $\pi_1 \leq \pi_2$.

A serial decomposition of a machine *M* is a cascade chain of sub-machines $m_1, m_2, \ldots, m_n$, in which the outputs of any sub-machine with lower index number may be used as inputs to sub-machines with higher index number. [10] has shown that each path in *L*(*M*) starting from π(*I*) and ending in π_0 corresponds to a cascaded decomposition of the machine *M*. Every edge in *L*(*M*) maps into a sub-machine in the cascaded decomposition. A sub-machine is derived from a partition $\tau_{i,j}$ such that $\pi_i \cdot \tau_{i,j} = \pi_j$. Figure 6 shows the Hasse diagram of the lattice *L*(A) and all S.P. partitions of the machine *A*. We will use a possible serial decomposition of the machine *A* as an example to illustrate how to build the cascade machine. The path we choose traverses π_1, π_3, π_0. Partition of the first sub-machine, $\tau_{I,1} = \{\overline{0,1,2,5}; \overline{3,4}\}$, consists of two states, therefore can be realized by one variable y_0. The second sub-machine $\tau_{1,3}$ is $\{\overline{0,1,3}; \overline{2,4}; \overline{5}\}$. It has three states and needs two variables, $y_1\ y_2$, to encode them. The last sub-machine $\tau_{1,0}$ is $\{\overline{0,2,3,4,5}; \overline{1}\}$, and needs one variable, y_3, to represent its states.

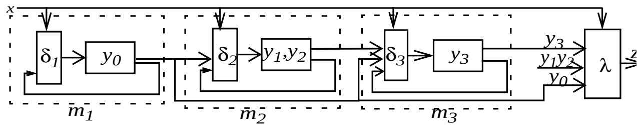


Figure 7. Machine *A* decomposed into a cascade

An assignment based on the above partitions will yield the following functional relationships, where $(y_0', y_1', \ldots y_n')$ is the encoded next state, $(y_0, y_1, \ldots y_n)$ is the present state, and *z* is the output:

$y_0' = \delta_1(x, y_0)$
$(y_1', y_2') = \delta_2(x, y_0, y_1, y_2)$
$y_3' = \delta_3(x, y_0, y_1, y_2, y_3)$
$z = \lambda(x, y_0, y_1, y_2, y_3)$

The schematic diagram of this realization is shown in Figure 7.

The number of bits needed to represent the sub-machine states is labeled on the edges of *L*(*A*) of Figure 6. For a detailed treatment of theoretical background we refer to [10] and [13].

Next we will show how to get all the S.P. partitions of a machine. Let $\pi s_i s_j$ be the *smallest* nontrival S.P. partition, in terms of partial order, containing state s_i and s_j in one block. We refer to the placing of s_i and s_j in one block as *identifying* them. To determine $\pi s_i s_j$, we first identify s_i and s_j. This identification implies that we must also identify the successors $\delta(s_i, i_k)$ and $\delta(s_j, i_k)$, for every input i_k in I. The states $\delta(s_i, i_k)$ and $\delta(s_j, i_k)$ are said to be *implied* by s_i and s_j. Whenever a state s_i is identified with s_j and s_k, the transitive law must be applied so that (s_i,s_j,s_k) are placed in the same block of π. If we repeat the above procedure and find the smallest closed partition $\pi s_i s_j$ for every pair of states $s_i s_j$, we obtain a set of partitions which are called the *basic* partitions. The partitions in the second lowest level of the lattice will all be basic partitions, therefore serving as the building blocks of the whole lattice.

Once we have the complete basic partitions, we then use the union operation on these basic S.P. partitions to generate all S.P. partitions of the machine [9], [10]. Finally we insert them, starting from the partitions with smallest number of blocks, into the lattice in a depth-first manner.

4.3 Extraction of a good decomposition

A straightforward way to build a serial decomposed machine A is to pick a path from $L(A)$ such that the maximum weight among its edges is minimum among all the possible paths. For example, we can either pick π_1, π_3, π_0, or π_2, π_3, π_0, etc. from $L(A)$ in Figure 6, which all require three submachines. This strategy certainly will give us a cascaded machine with the best possible throughput (in terms of sequential latency). However, if we look closer, we find that actually we only need two submachines, $\rho_{I,2}$ and $\rho_{2,0}$, to realize the original machine. In doing so we literally decrease the number of sub-machines and still maintain the highest throughput possible. Thus the problem of finding a path with minimum sequential latency leads to the problem of finding a path that has both minimum sequential latency and minimum number of nodes in the path.

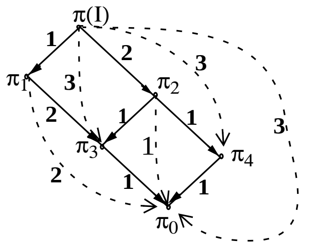


Figure 8. Complete information about the lattice L(A).

In order to capture this essence we propose the following algorithm:

1.Build the connected graph G (complete lattice), where each node represents an S.P. partition of M, and each partial order pair nodes are connected with a directed edge.

2.Let w be the number of bits used in distinguishing the partial order pair nodes in G. Assign each edge the cost of w.

Figure 8 shows a directed graph for lattice $L(A)$ obtained in steps 1 and 2. The graph G will give us a complete information about the cost of different paths, which will be otherwise difficult to get from the Hasse diagram alone.

3.Assign the node $\pi(I)$ a value of 0, and the rest of the nodes the value of infinity. Let u, v be two nodes in G, and $c(u,v)$ be the cost from u(source) to v(target). Do a breadth-first-traversal on G, starting from $\pi(I)$. For each visited node u, assign each connected target node v a value = max(min(v value, $c(u,v)$), u value).

4.Remove all the edges in G that have weight greater than the node value of π_0.

5.Use $\pi(I)$ and π_0 as source and destination respectively, find the shortest path between them.

This algorithm is exact. Step 3 and 4 together guarantee that the path we found will have the minimum of maximum bits. This guarantees that we are effectively choosing only decompositions with the best possible throughput. Step 5 ensures that the path we found consists of minimum number of nodes, thus minimizing the number of the machines in the decomposition.

5. BDD implementation

After extracting the machine, to ensure the feasibility of the approach, we devised a strategy to address the problem of area minimization. While the performance in terms of delay is basically fixed during the logic step of the synthesis, the area is still strictly related to the choice of good state assignment and logic minimization has to bear in mind the different target architecture which is BDD based as opposed to the traditional standard cell multilevel realization. As for now we don't have any encoding scheme appropriate to the BDD structures, we used a classic algorithm for symbolic encoding (JEDI[15]), starting from the head machine and using the results of previous encoding as we proceed towards the tail machine. An outline of the algorithm is as follows:

Given any extracted machine:

1. Encode through JEDI
2. Pass it through a BDD package (CUDD [20])
3. Minimize it safely ([12])
4. Do the placement of the BDDs

Step 3 uses a well-known algorithm to simplify a BDD with don't care conditions ([12]); in our case, don't cares are both external (the machine is not completely specified) and internal (the encoding for an intermediate machine specifies only some combinations of bits). This has shown to greatly improve the overall node count of the BDDs.

The last step consists in adding some intermediate dummy nodes to the BDDs in order to ensure functionality (those nodes practically act as flip flops).

6. Results

In this section we detail the experimental results performed on a set of MCNC benchmarks from the 1993 Logic Synthesis Workshop.

Table1. Cascade decomposition of MCNC benchmarks

FSM	pi	po	s	bp	tp	m	#b	CPU
bbara	4	2	10	5	5	3	2	1
dk27	1	2	7	4	4	3	1	0.06
dk512	1	3	15	13	44	3	3	2
ex1	9	19	20	8	67	2	4	71
ex7	2	2	10	2	2	2	3	0.37
kirkman	12	6	16	3	3	4	1	159
opus	5	6	10	2	2	3	3	0.67
s1	8	6	20	4	8	2	4	51
s208	11	2	18	9	9	5	1	42
s27	4	1	6	4	5	3	1	0.22
s420	19	2	18	9	9	5	1	42
shiftreg	1	1	8	8	28	3	1	0.24
tav	4	4	4	1	1	2	1	0.17
tbk	6	3	32	10	16	3	4	490
train11	2	1	11	8	26	2	3	1

Table 1 contains decomposition results for the benchmarks that have nontrival serial decompositions. “pi” is the number of primary inputs. “po” is the number of primary outputs. “s” is the number of states. “bp” is the number of basic partitions. “tp” is the number of total partitions. “m” is the number of decomposed sub-machines. “b” is the maximum number of bits in a submachine, which is directly related to the throughput of the whole machine. CPU column gives runtimes in seconds for Sun 4 Sparc workstation.

Table 2 contains results of BDD implementation for the benchmarks from Table 1. “jedi” is the area of the FSM encoded through JEDI. “random” is the area of the FSM encoded randomly. “S.C.A.” and “S.C.Del.” are the area and clock cycle of the FSMs after logic minimization and mapping obtained through SIS with the following script:

```
state_minimize stamina
state_assign jedi
extract_seq_dc
source script.rugged
source script.delay,
```

followed by automated placement and routing, together with a static timing analysis (column “S.C.Del.”) of the corresponding standard cell implementation.

Table2. Area and delay comparisons.

FSM	jedi $10^3\mu m^2$	random $10^3\mu m^2$	S.C.A. $10^3\mu m^2$	Delays ns	Lat. ns	S.C.Del ns
bbara	71.6	68.7	10.3	1.6	8.8	5.5
dk27	5.8	5.8	5.38	1.6	4.0	4.1
dk512	26.2	30.7	12.0	3.2	12.8	6.3
ex1	282.7	334.5	64.7	3.2	24.0	9.7
ex7	33.8	37.6	6.21	3.2	12.8	4.3
kirk-man	117.6	114.6	38.1	1.6	12.8	12.3
opus	60.5	69.0	17.7	3.2	17.6	7.1
s1	251.4	310.3	38.9	3.2	20.8	9.8
s208	44.2	44.2	19.8	1.6	9.6	7.5
s27	15.1	10.7	8.36	1.6	5.6	4.6
s420	44.2	44.2	16.9	1.6	9.6	6.6
shiftreg	0.7	0.7	3.73	1.6	2.4	3.8
tav	10.8	10.8	6.02	1.6	4.8	3.4
tbk	118.1	117.5	45.6	3.2	19.2	12.6
train11	31.0	31.7	7.15	3.2	12.8	4.2
Total	1113.7	1231.0	300.85	35.2	177.6	101.8

Both implementations are in a 0.5 μm technology. The basic cells needed for the wave steering implementation have been simulated, taking into account parasitics, under different operating conditions, and proven to be working at a frequency of 625 MHz. This explains why the delays reported are all multiple of 1.6 ns. Electrical simulations show that the two phase clocking scheme permits to accept two state bits without stalling the pipeline; therefore, only submachines with more than 2 state bits force a slow-down of the system. One key feature of the new design technique is that the performance of the implementation depends on the speed of the elementary cell (buffered multiplexer) and on the quality of the decomposition, but not on the overall complexity of the machine. Therefore we hope, through different decomposition schemes, to get even better results for bigger FSMs. In the current implementation the state space behavior is considered and optimized separately from the output behavior. The results show that, with respect to standard cell realizations, the overall area of the wave-steered implementation increases on the average by a factor 3.7 (4.1 for the random encoding), and the throughput increases by almost a factor of 3. A different customization of the basic cells is possible that will provide more compact implementations. Note also the relatively mild, but for special cases, degradation in latency. The use of an encoding algorithm results in only minor improvement on the area. We think this is due to the substantial difference between a standard multi-level optimization, for which JEDI is well suited, and the BDD representation targeted here. We believe that an encoding customized for BDDs may improve the results.

7. Conclusions.

In this work we have demonstrated the feasibility of wave steering, a novel design technique, in building high-throughput FSMs. The results on some of the examples in the MCNC benchmark suite are encouraging. We believe that the latency and area of wave-steered FSMs can be decreased and more examples can be handled when different than just cascade decompositions are considered. We also expect to improve the results by allowing state splitting and developing appropriate encoding techniques. However, for this approach to gain full acceptance, the system issues caused by the difference between throughput and latency in the behavior of those machines have to be solved. We are currently working on these problems.

Aknowledgements.

This work was supported in part by MARCO/DARPA GSRC and in part by NSF through grant CCR-9811528.

References:

[1] P. Ashar, S. Devadas and A.R.Newton: Optimum and Heuristic Algorithm for an Approach to Finite State Machines Decomposition, IEEE TCAD, March 1991.

[2] L. Benini, E. Macii, M. Poncino, and G. De Micheli: Telescopic Units: A New Paradigm for Performance Optimization of VLSI Designs, IEEE TCAD vol. 17 no. 3, March 1998.

[3] V. Bertacco, et al.: Decision Diagrams and Pass Transistor Logic Synthesis, IWLS'97, Lake Tahoe, May 1997.

[4] P. Buch, A. Narayan, A.R. Newton, A. Sangiovanni-Vincentelli: Logic Synthesis for Large Pass Transistor Circuits, ICCAD'97, San Jose, November 1997.

[5] C. Chao and H.H. Loomis: High Rate Realization of Finite-State Machines, IEEE Trans. on Comp. vol. C-24, July 1975.

[6] G. De Micheli: Synchronous Logic Synthesis: Algorithms for cycle time minimization, IEEE TCAD, Jan 1991.

[7] S. Devadas and A.R.Newton: Decomposition and Factorization of Sequential Finite State Machines, TCAD, November 1989.

[8] A.D. Friedman: Feedback in Synchronous sequential Switching Circuits, IEEE Trans. on Comp. vol EC-15, June 1966.

[9] M. Geiger, T. Muller-Wipperfurth: FSM Decomposition Revisited: Algebraic Structure Theory Applied to MCNC Benchmark FSMs, Proc. 28th Design Automation Conf., San Francisco, 1991, pp. 182-185.

[10] J. Hartmanis, R. E. Sterns: Algebraic Structure Theory of Sequential Machines, Prentice Hall, Englewood Cliffs, 1966.

[11] A. Hertwig, H.-J. Wunderlich: Fast Controllers for Data Dominated Applications, ED&TC 97, Paris, March 1997.

[12] Y. Hong, P.S. Beerel, J.R. Burch and K.L. McMillan: Safe BDD Minimization using Don't Cares, DAC'97, Anaheim, June 1997.

[13] Z. Kohavi: Switching and Finite Automata theory, McGraw-Hill, New York, 1970.

[14] K. Lam and S. Devadas: Performance-Oriented Decomposition of Sequential Machines, ISCAS '90, New Orleans, May 1990.

[15] B. Lin and A.R. Newton: Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages, IFIP Int.l Conf. on VLSI, August 1989.

[16] H.-D. Lin, D.G. Messerschmitt: Finite state machine has unlimited concurrency. IEEE Transactions on Circuits and Systems, vol.38, (no.5), May 1991.

[17] A. Mukherjee, R. Sudhakar, M. Marek-Sadowska, S.I. Long: Wave Steering in YADDs: A Novel Non-iterative Synthesis and Layout Technique, DAC'99, New Orleans, June 1999.

[18] A. Mukherjee, M. Marek-Sadowska, S.I. Long: Wave Pipelining YADDs, CICC'99, San Diego, 1999.

[19] M. Shamanna, K. Cameron, S.R. Whitaker: Multiple-input, Multiple-output Pass Transistor Logic, Int'l J. Electronics vol. 79 n. 1, July 1995.

[20] F. Somenzi: CUDD: CU Decision Diagram Package Release 2.3.0, University of Colorado at Boulder, 1998.

[21] K. Taki: A Survey for Pass-Transistor Logic Technologies, ASP-DAC'98, Yokohama, February 1998.

[22] T. Villa: NOVA: state assignment of finite state machines for optimal two-level implementation, IEEE TCAD, Sept. 1990.