Synthia: Synthesis of Interacting Automata targeting LUT-based FPGAs

George A. Constantinides[1], Peter Y. K. Cheung[1], Wayne Luk[2]
(*george.constantinides@ieee.org*)

ABSTRACT

This paper details the development, implementation, and results of *Synthia*, a system for the synthesis of Finite State Machines (FSMs) to field-programmable logic. Our approach uses a novel FSM decomposition technique, which partitions both the states of a machine and its inputs between several sub-machines. The technique developed exploits incomplete output specifications in order to minimize the interconnect complexity of the resulting network, and uses a custom Genetic Algorithm to explore the space of possible partitions. User-controlled trade-off between logic depth and logic area is allowed, and the algorithm itself during execution determines the number of sub-FSMs in the resulting decomposition. The results from MCNC benchmarks applied to Xilinx XC4000 and Altera FLEX8000 devices show a typical speedup of 35% to 37% combined with a typical area reduction of 26% to 33% over a standard one-hot encoding of the original circuit. Final results will be available within weeks and incorporated into the paper.

1. Introduction

Finite State Machine (FSM) Decomposition is the implementation of a large FSM as a network of smaller interacting FSMs, a problem that has been studied since the late 1960s [2]. In VLSI architectures, FSM decomposition is useful for a number of reasons. By controlling the topology and manner in which the machine is decomposed, it is possible to aim for an implementation with characteristics such as: high speed, as decomposition can usually lead to a reduction in logic depth [6]; low area, as state-encoding heuristics will often cope more efficiently with each of the smaller sub-FSMs than with the large lump-FSM [3]; reduction in interconnect complexity [7], and I/O minimization [8].

This paper describes one such approach to FSM decomposition, which specifically targets LUT-based FPGA implementations. The technique developed heuristically partitions the states of a large FSM between several sub-machines, while also partitioning its inputs. Output don't-care conditions are exploited in order to minimize the interconnect complexity of the resulting network, and a custom Genetic Algorithm is used to explore the space of possible input partitions. A user-controlled trade-off between logic depth and logic area is allowed, and the state-partitioning algorithm itself, during execution, determines the number of sub-FSMs in the resulting decomposition.

The following points summarise the part of this work that is believed to be original.

- *Synthia* extends and modifies the work of Yang [7] in order to exploit incomplete specifications in state-machines.
- *Synthia* uses a novel decomposition topology, which can be thought of as a combination of the earlier work of Feske [3] with the above.
- *Synthia* exploits a Genetic Algorithm approach to the problem of searching the space of possible input partitions.

[1] Dept. of Electrical and Electronic Engineering, Imperial College, London, UK.
[2] Dept. of Computing, Imperial College, London, UK.

The paper is subdivided into several sections. Section 2 introduces some of the background work into FSM decomposition which has been built upon. Section 3 details the concepts and algorithms that form the *Synthia* system. Section 4 presents experimental results obtained from running the system on several benchmark circuits, targeting both Xilinx XC4000 [10] and Altera FLEX8000 [12] devices. Section 5 then presents the conclusions of this work.

2. Background

Feske et al. [3], have developed an approach to FSM decomposition which operates at the State Transition Graph level. It has been shown that decomposition strategies that partition the STG allow a wider solution space to be searched by the following phases of synthesis [6]. The work in [3] proceeds by breaking the STG into a number of sub-STGs, each containing an extra 'wait' state representing all states outside the sub-STG's domain. The sub-FSMs must pass messages to each other when a transition occurs which would cross a sub-STG boundary, indicating which state to enter. Unlike most other techniques for FSM decomposition, [3] not only allows an arbitrary number of sub-FSMs to be formed, but decides upon that number itself, through adaptation as the algorithm executes. However, the simplicity of the messages passed means that more complex don't care conditions, arising from the association of particular inputs with particular groups of states, are not exploited. Even so, the authors report significant improvements compared to a one-hot implementation: reduction in circuit depth (29%) and number of logic blocks (38%) when mapped to a Xilinx XC4000 FPGA.

Yang, et al. [7], have developed an FSM decomposition to minimize the complexity of VLSI interconnection. The first step of this *n*-way decomposition is to partition the set of inputs between *n* sub-FSMs, each containing all the states of the lump FSM. The additional communication between sub-FSMs necessary for each sub-FSM to calculate its correct (next-state, output) combination is then found. Outputs are partitioned between sub-FSMs, leading to possible redundancy in states. This is followed by a state-minimization on each sub-FSM. Because the messages passed between sub-FSMs are only functions of machine inputs (not states), the logic to generate them is combinatorial and tends to be very simple compared to the sequential logic for the sub-FSM. This technique has the advantage that more complex don't-care conditions can be exploited, leading to a significant reduction in interconnect complexity. This is important for FPGA design, as routing resources tend to be limited. However, as developed in [7], little advantage is taken of incompletely specified FSMs. In addition, the user is required to specify the number of sub-FSMs to decompose to in advance of algorithm execution. In a later paper [9], the authors apply their technique to FPGA implementation on the Xilinx XC4000, claiming a significant speedup at the cost of a 44% increase in the number of logic blocks.

3. *Synthia*

3.1 Decomposition Topology

Our technique, which forms the basis of the *Synthia* system, performs a partitioning of both inputs and states between sub-FSMs. The state partitioning is performed by adapting the linear partitioning approach described in [3]. The constraint that each lump-FSM state is assigned to exactly one sub-FSM is retained. The output of the lump FSM is produced by a mapping $\mathbf{X} \times \mathbf{S} \rightarrow \mathbf{Y}$, where **X** is the set of input values, **S** the set of states, and **Y** the set of output values. Thus to partition **S** across several sub-FSMs implies that each output may need to be produced by each sub-FSM. For this reason, a block of logic is required to recombine outputs from each sub-FSM, as shown in Fig. 1(a). However, the constraint that every input must be available to every sub-FSM [3], is relaxed. Instead, our

approach introduces two types of messages which may be passed between sub-FSMs. Messages from one sub-FSM can either instruct another sub-FSM to enter a particular state, or inform another sub-FSM of the status of its inputs. It is worth noting that the two different types of message are never needed simultaneously, since if a given sub-FSM is in its wait-state, then it is only waiting for next-state type messages, whereas if it is not in its wait-state, it will never receive next-state type messages. This observation allows message assignments to be shared between the two types of message, if desired, leading to fewer communication wires between sub-FSMs. For our implementation, this option was not taken, as in order to obtain the best possible performance, the messages were implemented by small combinatorial sub-circuits. Shown in Fig. 1(b) is a single sub-FSM.

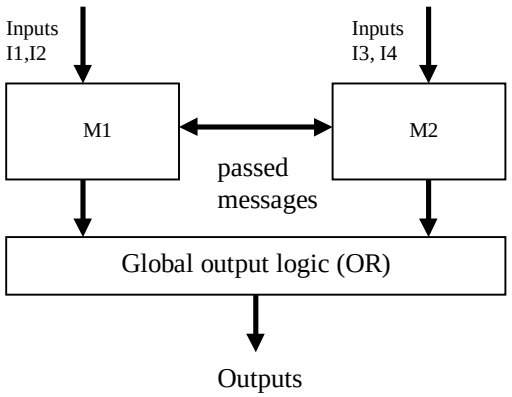


Figure 1(a): A 2-way *Synthia* decomposition

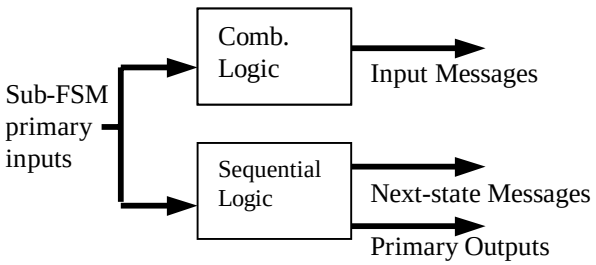


Figure 1(b): A single sub-FSM

3.2 Input Messages

Yang [7] uses *n*-dimensional 'state-transition matrices' in order to find what messages need to be passed between *n* sub-FSMs. In our present paper we extend the idea of state-transition matrices to better cope with two phenomena not present in [7]:

- In our decomposition, states belong to only one sub-FSM.
- The presence of incompletely specified outputs.

Firstly, assume that the input and state partitioning has already been performed using the method described in section 3.3. An (n+1)-dimensional matrix is then built for each sub-FSM *M*. One dimension corresponds to the state of *M*. The other *n*-dimensions correspond to possible input cubes seen by each sub-FSM, and the data value at each co-ordinate represents a next-state/output combination. An example will illustrate the point. Consider the state-machine specification shown in Fig. 2(a). Let us assume that st0 and st1 are contained in sub-FSM M1, the other states belonging to other sub-FSMs. Further, assume that the first two inputs are sent to sub-FSM M1, the next two are sent to sub-FSM M2, and the final one to sub-FSM M3. A 4-D state-transition matrix is constructed for M1 (shown in Fig. 2(b) as two 3-D matrices).

It may now be possible to merge axis headings on each of the (n-1) dimensions not representing M1, while still retaining all necessary information. This is equivalent to reducing the number of interconnect wires necessary between sub-FSMs. We aim to minimize interconnect complexity for two reasons:

1. routing resources are scarce in FPGAs
2. small sub-FSMs are likely to make heavy use of fast local FPGA interconnect, whereas interconnect between sub-FSMs is likely to make more heavy use of slower, non-local routing resources.

When there are don't care conditions on some of the output specifications, it is possible that there is no single unique

solution to the problem. For example, the reduced matrix could take the form of Fig 2(c) or 2(d) equally. This situation arises because don't-care conditions may be expanded into concrete '0' or '1' specifications as a result of one merge, which would conflict with the concrete value arising from another possible merge. A heuristic has been implemented, as detailed below, to select which merges to implement.

```
.i 5
.s 4
.o 2

0-1-- st0 st0  0-
0-01- st0 st1  01
1-1-- st0 st0  -0
1-01- st0 st1  01
0-1-0 st1 st2  0-
0-1-1 st1 st2  -0
0-01- st1 st3  -1
1-1-0 st1 st2  -1
1-1-1 st1 st2  0-
1-01- st1 st3  1-
...
...
```

Figure 2(a): Partial KISS specification

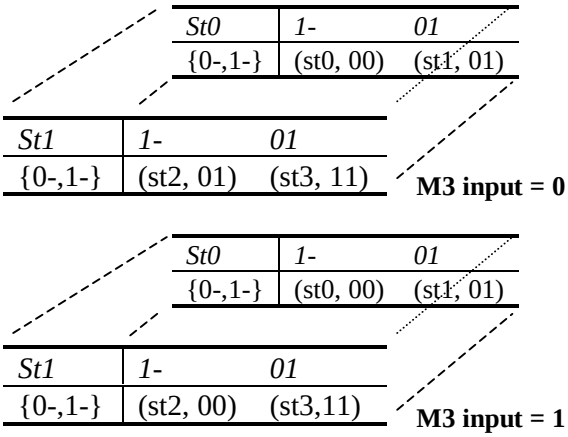


Figure 2(c): One possibility for optimization of Figure 2(b).

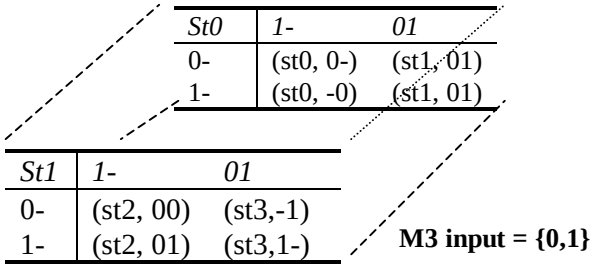


Figure 2(d): Another possibility for optimization of Figure 2(b).

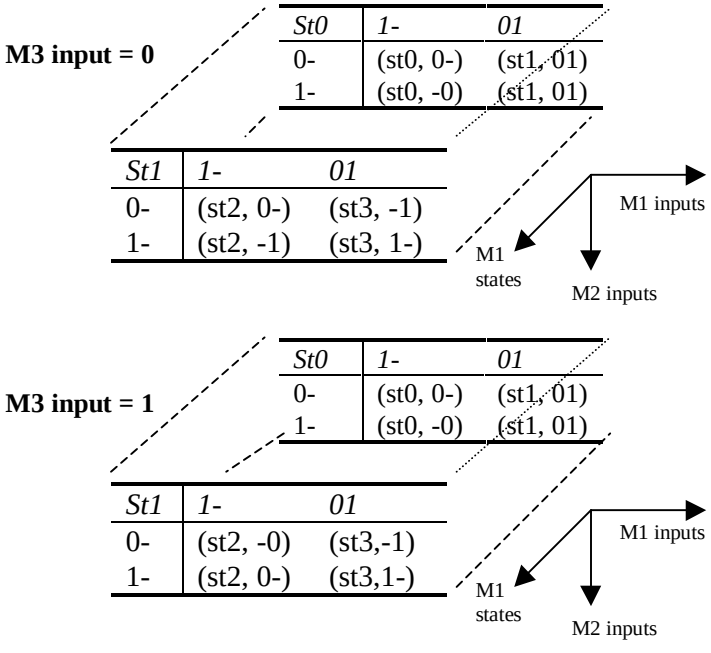


Figure 2(b): A state-transition matrix with output don't-cares

3.3 Primary Input and State Partitioning

The previous section answered the question of what input messages are necessary for communication between sub-FSMs, given a partition of states and inputs between them. This section will address the problem of how to partition states and inputs between sub-FSMs.

If there are n sub-FSMs, and p primary inputs, then each input could be assigned to each sub-FSM, leading to a total of n^p different partitions. A Genetic Algorithm (GA) approach is used to search this large space of possible input partitions. We may represent the mapping between primary inputs and sub-FSMs by a vector **I**, indexed by primary input number, and with entries in the range 0 to (n-1), indicating which sub-FSM that input is assigned to. When viewing the problem in this way, a GA approach is natural: a clear demarcation already exists between alleles in a chromosome - the index of the vector

I. With a larger allele size there is not full genetic control over the search-space, whereas with a smaller size (say bits, more typically used in a GA) no extra information is present. The unusual properties of the GA applied are listed below:

- valid allele values are 0 to (*n*-1), and *n* may vary during algorithm execution
- mutation consists of randomly choosing an allele value between 0 and (*n*-1)

The above has addressed the problem of choosing a suitable input partition given a partitioning of states between sub-FSMs. Rather than extending the GA approach to the partitioning of states, certain *a-priori* information may be used effectively: states that are successors or predecessors of other states in the STG are more likely to benefit from incorporation within a single sub-FSM than any two states chosen at random. Such sub-FSM networks are likely to have fewer inter-FSM transitions, and in addition, the results of [3] indicate that particular inputs tend to be associated with particular sub-graphs of the STG. Indeed, the algorithm described in [3] is a heuristic incorporating elegantly this prior knowledge. The approach taken in that paper has been modified to incorporate the changes in decomposition topology, the two different message types, and the presence of an extra phase - that of input assignment.

3.4 Algorithms

3.4.1 The Core

At the core of the algorithm employed is the integration of the two phases: an STG-level heuristic for state partitioning, and a GA for input partitioning. A naive approach is either to optimize the state partition for each input partition or to optimize the input partition for each state partition. The computational load from such an approach is unrealistic, and so the solution taken is to interleave the two algorithm phases in a way that is unlikely to impact greatly on the performance.

The resulting top-level algorithm is shown in pseudo-code below.

```
OptimizeFSM( stg, start_its,
 next_its, popsize, pcross, pmut )
{
  decomp=init_state_partition(stg);
  popul=ga_initpop(decomp, popsize,
            pcross, pmut );

  ga_optimize( popul, start_its );
  newCost = 1.0 / best_fitness;

  do {
    oldCost = newCost;
    foreach sub-FSM B {
     do {
       gain = optimize_state_part(B,
              best_chromosome );
      } while( gain > 0.0 );
     }
     cleanup_popul( popul, decomp );
     ga_optimize( popul, next_its );

     newCost = 1.0 / best_fitness;
  } while( newCost < oldCost );

  write result file
}
```

The algorithm starts with an initial state partition of one state per sub-FSM (a one-hot state-encoding on the lump-FSM). After performing a GA optimization with respect to that state-partition, the main loop is entered. `Optimize_state_part` works on one block of the state partition at a time, trying to suck-in any state which has a predecessor or successor state within another block. If this results in a positive gain, the move is retained. If this results in an empty block, all references to that block in the chromosome **I** are set to point instead to the new block containing that state. (A similar function is performed by `cleanup_popul`, above). It is worth noting that two, possibly different, numbers of iterations are used in the GA - one inside the main loop body and one outside. This is because it is quite likely that the population of input partitions before entering the state-partitioning heuristic is a good *first-guess* for the GA after exiting the heuristic. This insight was confirmed by the results collected, which show good performance can be achieved at little computational cost by having a higher number of iterations outside the loop body.

3.4.2 Transition Matrix Manipulation

The cost estimation used by the state- and input-partitioning algorithms has two phases:

1. construct a transition matrix for each interacting sub-FSM of the network
2. estimate delay and area of the resulting network

A pseudo-code for the cost-estimation function is shown below.

```
get_cost( decomp, chromosome )
{
  foreach sub-FSM B {
    M = construct_matrix(B, decomp );
    optimize_matrix( M, B );
    add matrix M to array 'Matrices'
  }
  return estimate_cost_from_matrices(
       Matrices,decomp,chromosome );
}
```

After constructing an unoptimized state-transition matrix (as discussed in section 3.2), the algorithm proceeds to optimize that matrix by merging axis headings as much as possible. The heuristic designed iteratively merges axis headings, two at a time. It is a non-backtracking step-by-step approach: the merge chosen at any given step is the one judged 'most likely' to result in the largest interconnect complexity reduction, and once that choice is made it will not be changed at a later stage. For each axis in the given state-transition matrix, a *compatibility table* is constructed. This table indicates which axis headings are mergible with which others (to be determined by searching the matrix entries). Three example compatibility tables are shown in Fig 3. Note that these are lower triangular matrices with binary entries (the compatibility relation is bi-directional and Boolean). The heuristic proceeds by counting the number of possible merges on each axis (3 for M0, 1 for M1 and 2 for M2). The axis with the greatest number is chosen for merging, in the hope that some merges will still be possible after the current one. Then the heuristic picks the heading H_1 with the largest number of possible merges (in this case either -00, 011 or 010). Finally, the pair to merge is completed by choosing the heading with the next largest number of possible merges H_2, subject to the constraint that H_1 and H_2 are mergible. After merging the two axis headings, and adjusting their entries (replacing don't-cares by '0' or '1') as necessary, the heuristic is ready for its next iteration.

M0	**-00**	**011**	**010**	**001**
-00				
011	TRUE			
010	TRUE	TRUE		
001	FALSE	FALSE	FALSE	

M1	**--0**	**001**
--0		
001	TRUE	

M2	**000**	**100**	**110**	**111**
000				
100	FALSE			
110	FALSE	TRUE		
111	TRUE	FALSE	FALSE	

Figure 3: Three Compatibility Tables

After optimization, the matrices form a complete specification for the network of sub-FSMs, which is then passed to a function in order to calculate a cost estimate. This proceeds in four stages:

1. encode input messages
2. encode state messages
3. construct STGs for each sub-FSM
4. estimate delay and area of each sub-FSM

The message assignments are arbitrary minimum length encodings. The delay and area of each sub-FSM are estimated separately from its neighbours for speed of execution. This is done using SIS [5] procedures for LUT-based FPGAs, as developed by Murgai [4], using a JEDI [11] minimum-length state encoding on each individual sub-FSM.

The worst-case logic depth (and therefore the estimated worst-case delay) of the resulting network occurs when more than one sub-FSM is involved in a transition. The maximum number of sub-FSMs that may be involved in a transition

is two. Hence if the logic depths of all sub-FSMs are written as d_0, d_1, ... d_n, with $d_0 \geq d_1 \geq ... \geq d_n$, then the worst-case logic depth is bounded above by $d_0 + d_1$ and below by d_0. Thus the returned delay estimate returned is $d_0 + 0.5d_1$. Similarly, the area estimate returned for the network is simply the sum of all area estimates for the sub-FSMs. The two costs (area and logic depth) are combined in a linear manner with a user-controlled factor in order to return a single cost-function value.

4. Experimental Results

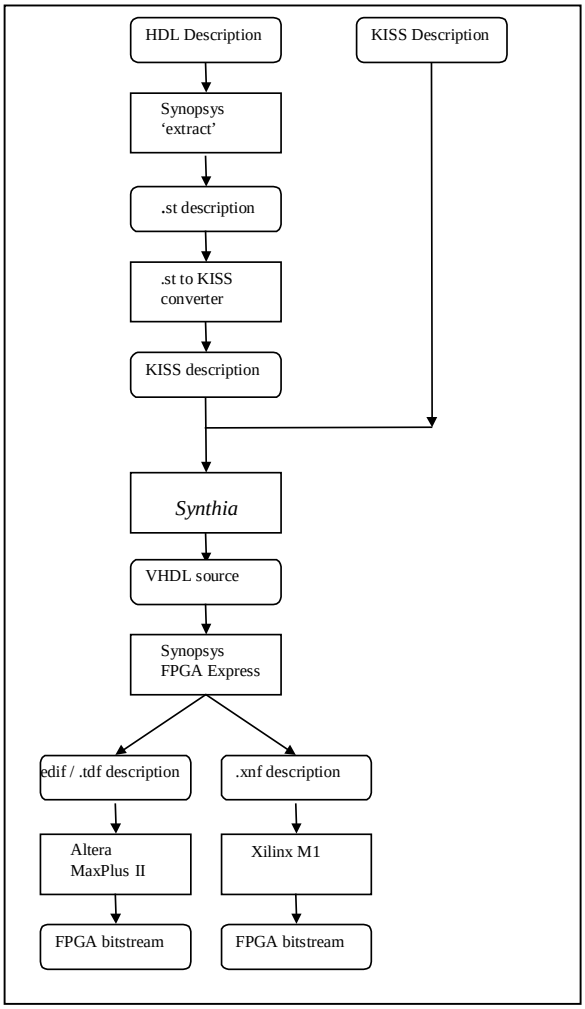
The algorithm described was implemented and integrated within the SIS [5] logic synthesis package. The implementation takes an STG as input and produces a file of hierarchical VHDL source-code describing the network of interacting FSMs.

A diagram illustrating a typical design flow when using *Synthia* is shown on the right in Figure 4. The format for describing FSMs in SIS is the KISS format. This can be acquired either directly from design specification or through extraction from an HDL description. Synopsys software [14] has one possible extraction algorithm, 'extract'. The state-machine format used by Synopsys is another abstract description, '.st', for which we have written a converter to KISS.

A subset of the MCNC Logic Synthesis Workshop benchmarks was used in order to explore the parameter-space of the algorithm, looking for a good operating point. Once this point was found, another subset of the MCNC benchmark set, and the two FSMs from the PREP [13] benchmark set were decomposed using the algorithm. The MCNC benchmarks are available as KISS FSM-specification files, whereas the PREP benchmarks were first converted to KISS.

The resulting VHDL code was compiled by Synplify [1] for both the Altera FLEX8000 (EPF8282A-2) [12] and the Xilinx XC4000 (XC4003EPC84-1) [10], once aiming for a high-speed result and once for a low area result. The resulting number of Altera LCs and Xilinx CLBs were collected, alongside the reported maximum clock frequencies. These **preliminary** results are tabulated in the *Synthia* columns of Tables 1 and 2. In addition, run-times are reported in Table 3.

For the purposes of comparison, each benchmark was also encoded in a one-hot style by JEDI, automatically written as



VHDL code, and compiled as above by Synplify. In the case of the PREP benchmarks, the KISS file was re-encoded in VHDL, rather than using the original VHDL. These one-hot results are shown in the one-hot columns of Tables 1 and 2.

The tabulated results are additionally shown in graphical form in Figure 5.

B'mark	One-hot		*Synthia*	
	Clock	#LCs	Clock	#LCs
bbara	45.7	37	108.7	14
beecount	57.1	22	84.7	19
cse	24.8	122	47.2	94
dk14	37.9	64	66.2	46
dk27	47.8	22	79.4	13
dk512	34.2	51	73.5	27
ex6	32.9	73	57.1	54
lion	54.3	18	104.2	9
mark1	42.6	60	41.7	43
opus	39.4	53	43.5	61
s27	55.6	28	90.1	11
shiftreg	76.3	22	85.5	16
tav	70.9	14	58.5	26
PREP3	51.3	43	52.9	35
PREP4	24.9	114	62.5	68

Table 1(a): Area-optimized results for FLEX8k

B'mark	One-hot		*Synthia*	
	Clock	#CLBs	Clock	#CLBs
bbara	51.3	53	108.7	14
beecount	59.9	26	84.7	19
cse	32.8	151	47.2	94
dk14	40.8	89	66.2	46
dk27	60.6	32	79.4	13
dk512	48.8	75	73.5	27
ex6	36.2	92	57.1	54
lion	73.0	20	104.2	9
mark1	41.8	82	41.7	43
opus	45.0	63	43.5	61
s27	65.4	28	90.1	11
shiftreg	82.6	29	85.5	16
tav	82.0	20	58.5	26
PREP3	52.6	53	78.1	37
PREP4	34.1	156	68.5	72

Table 1(b): Speed-optimized results for FLEX8k

B'mark	One-hot		*Synthia*	
	Clock	#LCs	Clock	#LCs
bbara	69.4	15	73.0	5
beecount	63.7	10	68.0	8
cse	39.7	48	53.5	36
dk14	49.5	27	58.1	23
dk27	68.0	9	76.3	5
dk512	53.5	26	68.0	11
ex6	45.2	31	56.5	25
lion	62.5	6	88.5	3
mark1	52.9	24	38.9	26
opus	47.6	19	52.4	25
s27	69.4	11	77.5	4
shiftreg	62.9	9	74.1	5
tav	67.6	5	64.5	11
PREP3	57.5	15	62.1	14
PREP4	45.0	46	59.2	31

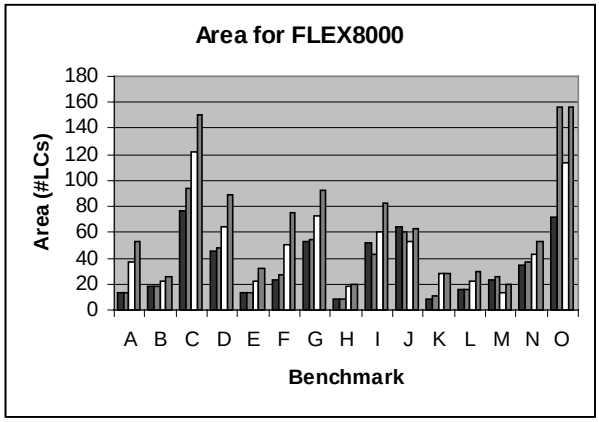
Table 2(a): Area-optimized results for XC4k

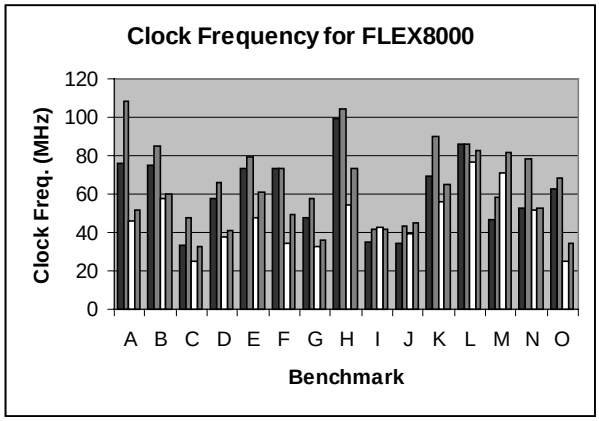
B'mark	One-hot		*Synthia*	
	Clock	#CLBs	Clock	#CLBs
bbara	66.7	19	73.0	5
beecount	64.5	11	68.0	8
cse	47.8	63	53.5	36
dk14	51.0	32	58.1	23
dk27	64.9	10	76.1	5
dk512	57.5	27	68.0	11
ex6	51.3	38	56.5	25
lion	70.9	7	88.5	3
mark1	58.8	29	38.9	26
opus	54.3	25	52.4	25
s27	69.0	12	77.5	4
shiftreg	73.0	9	74.1	5
tav	73.0	6	64.5	11
PREP3	62.1	22	57.5	17
PREP4	49.5	56	60.2	31

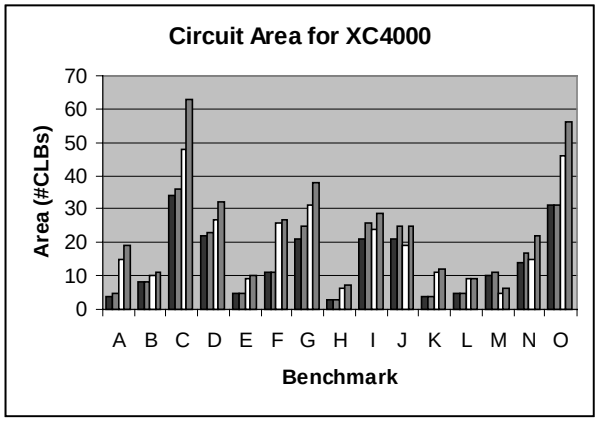
Table 2(b): Speed-optimized results for XC4k

B'mark	Run-time	B'mark	Run-time
bbara	4m 6s	beecount	1m 44s
cse	13m 49s	dk14	6m 50s
dk27	1m 57s	dk512	5m 39s
ex6	To be collected	lion	To be collected
mark1	To be collected	opus	To be collected
s27	To be collected	shiftreg	To be collected
tav	To be collected	PREP3	To be collected
PREP4	To be collected		

Table 3: Synthia Run-Times







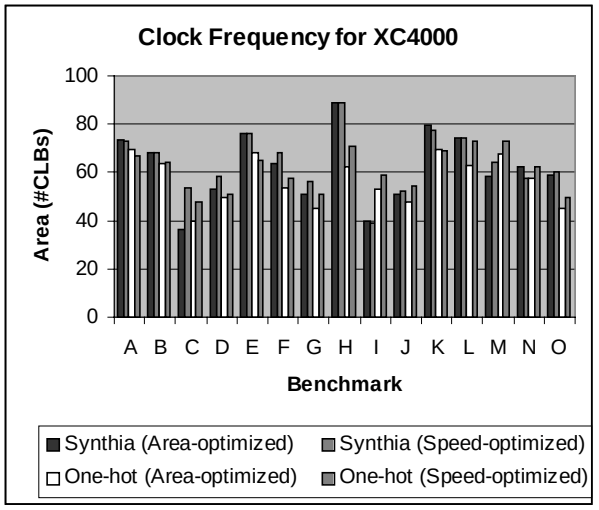


Figure 5: Graphs of Experimental Results

Key to benchmarks			
A	bbara	I	mark1
B	beecount	J	opus
C	cse	K	s27
D	dk14	L	shiftreg
E	dk27	M	tav
F	dk512	N	PREP3
G	ex6	O	PREP4
H	Lion		

The preliminary results show an average speedup of 35% for FLEX8k targeting speed, and 6% for XC4k targeting speed. These figures change to 37% and 10% respectively, when counting negative speedups as 0 (i.e. when either one-hot or *Synthia* is used, whichever is faster). Also shown is an average area reduction of 26% for FLEX8k and 27% for XC4k (changing to 33% and 35% respectively when modified as above).

Importantly, it is clear that all of the speedups have been *accompanied* by a significant reduction in area, and all of the area reductions (except one) by a significant speedup.

5. Conclusion

A novel algorithm for the automatic decomposition of FSMs into networks of interacting sub-FSMs has been presented. The technique developed uses a combination of a Genetic Algorithm and a state-partitioning heuristic, along with a further heuristic which exploits output don't-care conditions in order to minimize the interconnect complexity of the resulting network.

Synthia has significantly improved the performance of the later stages of synthesis, leading to a simultaneous significant reduction in circuit area, and significant increase in maximum permissible clock frequency.

The speedups gained with the Xilinx XC4k, though quite impressive, are relatively small compared to those gained with the FLEX8k. This is almost certainly at least in part due to the approximation made at the cost estimation stage, of a circuit as a network of 4-LUTs. The Xilinx logic block, being a combination of more

than one type of LUT, is less suited to this approximation than the Altera device.

The question of choosing an efficient message encoding for inter-FSM communication stands out as a missing part of this work. If the results here can be achieved with arbitrary message encoding, it is clear that a more intelligent approach could achieve even better performance. This question is related to the ongoing research topic of hierarchical synthesis of sequential circuits.

6. Acknowledgements

The authors would like to acknowledge the support of Alan Marshall, Nick Wainwright and John Lumley of Hewlett Packard Laboratories, Bristol, UK. In addition, donations of software tools were gratefully received from Xilinx and Altera.

7. References

[1] *Synplify Users Manual*, Synplicity.

[2] J. Hartmanis and R.E. Stearns, "Algebraic Structure Theory of Sequential Machines," *Prentice-Hall*, 1966.

[3] K. Feske, "Technology-Driven FSM Partitioning for Synthesis of Large Sequential Circuits Targeting Lookup-Table Based FPGAs," in *Proc*. FPL 1997, *Springer-Verlag*, 1997.

[4] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Synthesis for Field Programmable Gate Arrays," *Kluwer Academic Publishers*, 1995.

[5] E. M. Sentovich, et al., "SIS: A System for Sequential Circuit Synthesis," UCB/ERL M92/41, May 1992. sis@eecs.ucb.edu.

[6] P. Ashar, S. Devadas, and A. R. Newton, "Sequential Logic Synthesis," *Kluwer Academic Publishers*, 1992.

[7] W. L. Yang, R. M. Owens, and M. J. Irwin, "Multi-way FSM decomposition based on interconnect complexity," in *Proc*. EURO-DAC, 1993.

[8] M. T. Kuo, L. T. Liu, and C. K. Cheng, "Finite State Machine Decomposition for I/O Minimization," in *Proc*. ISCAS, 1995.

[9] W. L. Yang, R. M. Owens, and M. J. Irwin, "FPGA-Based synthesis of FSMs through Decomposition," in *Proc*. 4th Great Lakes Symposium on VLSI Design, 1994.

[10] *XC4000 Data Book*, Xilinx Inc., 1991.

[11] B. Lin and A. R. Newton, "Synthesis of Multiple Level Logic from Symbolic High-Level Description Languages," in *Proc. Int. Conference on VLSI*, August 1989.

[12] *FLEX8000 Handbook*, Altera Corp., 1994.

[13] *PREP Benchmarks*, http://www.prep.org.

[14] *FPGA Express Online Help*, Synopsys inc.