SYNTHESIS OF ASYNCHRONOUS SEQUENTIAL LOGIC IN THE SPATIAL AND TEMPORAL DOMAINS

by

David S. George

B.S. in Electrical Engineering and Mathematics, Carnegie Mellon University, 1984

Submitted to the Graduate Faculty
of the Graduate School of Engineering
in partial fulfillment of
the requirement for the degree of
Master of Science
in
Electrical Engineering

University of Pittsburgh
1991

The Author grants permission
to reproduce single copies.

____
Signed

COMMITTEE SIGNATURE PAGE

Steven P. Levitan, Ph.D.	
Advisor	Signature

ACKNOWLEDGMENTS

I would especially like to thank Steve Levitan for his guidance and patience throughout the development of this tool.

ABSTRACT

Signature________________________

SYNTHESIS OF ASYNCHRONOUS SEQUENTIAL LOGIC IN THE SPATIAL AND TEMPORAL DOMAINS

David S. George, M.S.

University of Pittsburgh

This thesis describes the background, motivation, and performance of an asynchronous sequential circuit synthesis tool called SASS, which synthesizes VHDL architectures tuned in either the temporal or spatial domain. From a directed acyclic graph description language, SASS generates structural VHDL architectures, which can be used by the KEYSTONE layout synthesis tools to generate CMOS circuits. Although many academic and commercial tools have been developed for synchronous finite state machine synthesis, no asynchronous sequential logic synthesis tools exist that will generate VHDL architectures from a flow graph description. It is shown that the asynchronous sequential circuits synthesized and tuned in either the temporal or spatial domain by SASS are competitive in terms of KEYSTONE generated layout area to functionally equivalent synchronous sequential circuits.

DESCRIPTORS

Asynchronous Sequential Circuits

Design Automation

Asynchronous Sequential Logic

Logic Synthesis

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

ABSTRACT iii

LIST OF FIGURES viii

LIST OF TABLES x

1.0 Introduction 1

1.1 The Motivation for an Asynchronous Sequential Design Automation Tool 1

1.2 SASS as a CAD Solution for Asynchronous Sequential Circuit Design 2

1.3 Contributions of SASS 3

2.0 Survey of Asynchronous Sequential Design Methods 6

2.1 Methods Yielding Spatial Efficiency Circuits 7

2.1.1 Huffman State Assignment 7

2.1.2 One-Hot State Assignment 8

2.2 Methods Yielding Temporally Efficient Circuits 10

2.2.1 Tan Algorithm for STT Equation Generation 11

2.2.2 Detailed Tan Algorithm for STT Equation Generation 11

2.3 Robustness of State Assignment Algorithms 14

2.3.1 One-Hot 14

2.3.2 Tan 16

2.4 Survey of Existing Tools and Approaches 18

2.4.1 System Synthesis 18

2.4.2 Circuit Synthesis 19

2.5 Summary Reasoning for the State Assignment Algorithms used in SASS 20

3.0 Syntactic Structure and Semantic Requirements for the DAG Input Language 23
3.1 Determination of Root State 23
3.2 Acyclic Description Requirement 24
3.3 Implicit Fundamental Mode Operation 24
3.4 Redundant Functionality 25
3.5 Disjoint DAG 25
4.0 General Synthesis Algorithm For SASS 28
4.1 Front-End Routines 28
4.1.1 Clique Partition Algorithms 28
4.2 Back-End Routines 32
5.0 Experimental Results 39
5.1 Layout Area Comparison with Synchronous Equivalents 39
5.2 Performance Comparison Between One-Hot and Tan Circuits 41
6.0 Future Work 43
6.1 Summary of Contributions 45
APPENDIX A 47
APPENDIX B 51
APPENDIX C 59
BIBLIOGRAPHY 67

LIST OF FIGURES

Figure No.		Page
1	CAD Environment of SASS	5
2	Sequential Function with Huffman State Assignment	8
3	Sequential Function with One-Hot State Assignment	9
4	Flow Table With Inherent Race Condition	15
5	Algorithmically Correct Flow Table for One-Hot Synthesis	16
6	Directed Acyclic Graph For A Sequential Function	26
7	Corresponding DAG Language Description	26
8	Flow Table with Redundant Functionality	27
9	Common Front-End Routines	29
10	Results of Flow Table Merging Routines	33
11	Condensed Flow Table For DAG	34
12	Back-End Equation Synthesis Routines	35
13	One-Hot VHDL Synthesized By SASS	36
14	Type A and B Partitions Generated by SASS	36
15	Initial Partitions Generated by SASS	37
16	Tan VHDL Synthesized by SASS	38
17	Syntax Diagrams for DAG Language	56
18	Syntax Diagrams for DAG Language (continued)	57
19	Syntax Tree Example	58
20	MEG File Input	62
21	MUSTANG File Input	63
22	MISII File Input	64
23	EQN File Produced by MISII	65

24 Corresponding EQN File Produced by SASS 65

LIST OF TABLES

Table No. Page

1 Comparison of KEYSTONE Layout Area Statistics 40

2 Comparison of Performance Statistics 42

3 Statistics for Circuits from Equally Minimal Flow Tables 45

4 Symbol Table for LEX Token Definition 52

1.0 Introduction

For the task of designing complex digital systems, such as microprocessors and bus controllers, either synchronous or asynchronous sequential logic can be used to implement control mechanisms. If Mealy or Moore [(1)*] synchronous logic is used, a periodic reference clock will determine the duration of the states, and initiate the transitions between states. Conversely, asynchronous sequential logic does not need a reference clock because a change in one of the input signals will trigger any possible change in state or output values immediately after propagation delays through the logic.

Both types of sequential logic have their niche for the task of controller design in complex digital systems. Synchronous logic is primarily used for internal control of the processes of a digital system that reference a common clock. For example, a microprocessor could use synchronous logic to control the sequence of steps needed to fetch, decode, and execute a machine instruction. On the other hand, asynchronous logic is often used for the control of the external interfaces of a digital system component, where a common clock is unavailable for different independently synchronous components of a large system. For example, a microprocessor can use asynchronous logic for an interprocessor communications bus controller. Microprocessors usually do not run in lockstep and will communicate asynchronously across a bus, using a handshake protocol which is ideally implemented with asynchronous sequential logic.

1.1 The Motivation for an Asynchronous Sequential Design Automation Tool

Most kinds of asynchronously controlled processes can also be synchronously controlled. The advantage to using synchronous logic is that it can typically be implemented with less logic than a functionally equivalent asynchronous counterpart. An asynchronous circuit, however, will respond more quickly to an input event than its

*Parenthetical references placed superior to the line of text refer to the bibliography.

synchronous equivalent. In synchronous circuits a certain amount of latency, based on the reference clock frequency, is inherent in its performance when trying to detect asynchronous input events. This latency time might not be acceptable in some applications.

An asynchronous circuit is often more difficult to design than the synchronous equivalent. Asynchronous sequential logic generation requires certain state assignment schemes and next-state memory variable equation generation methods that eliminate race conditions and hazards [2]. These schemes thus employ additional logic to insure correct circuit behavior. Conversely, the Mealy and Moore synchronous design approaches have less stringent state assignment schemes and equation generation methods, and have been incorporated into numerous academic and commercial tools. Notably, the PEG, MEG, and MUSTANG [3] tools developed at Berkeley can accept textual finite state machine (FSM) descriptions and generate next-state logic equations for either a Mealy or Moore FSM, which can then be used for programmable logic array synthesis or fed into other logic optimization tools.

Although many well understood approaches have been developed for general purpose asynchronous sequential circuit design, and similar work is currently being done at other universities [4], there still do not exist tools to automate the task of designing such circuits which are comparable to those that exist for synchronous FSM design. Most asynchronous design methods are tedious and complex enough to warrant the need for a tool to automate the synthesis procedure.

1.2 SASS as a CAD Solution for Asynchronous Sequential Circuit Design

A Sequential Asynchronous Subcircuit Synthesizer (acronym SASS), is a logic design automation tool ideally intended for the synthesis of asynchronous sequential circuits, which are tuned for either smaller layout area or faster performance. The circuits produced by SASS can be applied to any type of digital control task, provided that different asynchronous input events directed toward the circuit do not coincide.

This operational constraint, known as *Fundamental Mode Operation* [1], will guarantee correct behavior of the circuit and insure that it is free from metastability problems.

The function of an asynchronous sequential circuit is described by a directed acyclic graph (DAG) language description. From this description, a set of VHDL next-state memory variable equations and output variable equations are eventually produced [5]. The VHDL memory and output variable equations are incorporated into a VHDL architecture, and an entity description is also produced for the VHDL architecture. Within each architecture, the unique initialization logic necessary for each particular set of next-state memory variable equation is contained; thus, an asynchronous circuit derived from the equations can be easily incorporated into the initialization scheme of a larger system design.

Figure 1 shows the flow of CAD options that can interface with SASS. Along with a corresponding VHDL entity, SASS can generate architectures that are tuned in the spatial domain for reduced circuit complexity and layout area, or architectures tuned in the temporal domain for decreased latency. Either type of architecture can be used by the KEYSTONE layout synthesis tools [6] to generate a CMOS asynchronous sequential circuit. The circuit layout is contained in a MAGIC file that can be translated into a file format for linear model transistor-level, and switch-level simulation by IRSIM. If the architecture merely needs to be simulated for functionality, it can also be compile into format by VCOMP for gate-level logic simulation by VSIM.

1.3 Contributions of SASS

This thesis explains the theoretical logic design techniques used in SASS (Sequential Asynchronous Subcircuit Synthesizer), which was developed for the synthesis of asynchronous sequential logic. SASS enables a logic designer to synthesize a hardware description of an asynchronous sequential function from a directed acyclic graph language description. It is also demonstrated that the asynchronous circuits that SASS produces are layout area competitive with their synchronous counterparts for general purpose CMOS layout synthesis.

The remainder of this thesis is organized as follows:

Chapter 2 surveys the various asynchronous sequential state assignment algorithms. An explanation is provided for the choice of algorithms that are used in SASS, and an analysis of the robustness of these algorithms is performed. Finally, relevant existing tools and approaches are compared to SASS.

Chapter 3 explains the syntactic structure and the semantics of the language that is used to describe asynchronous sequential circuits to SASS.

Chapter 4 illustrates the general synthesis algorithm used in SASS. The resulting metamorphosis of a sequential input language description into a VHDL architecture is shown after various phases of program execution.

Chapter 5 compares the asynchronous circuits generated by SASS against synchronous equivalents in terms of size. Also, the asynchronous circuits tuned for size and speed are compared against one another in terms of speed.

Chapter 6 describes the future enhancements to SASS that would convert it from a research tool into a commercial-style product.

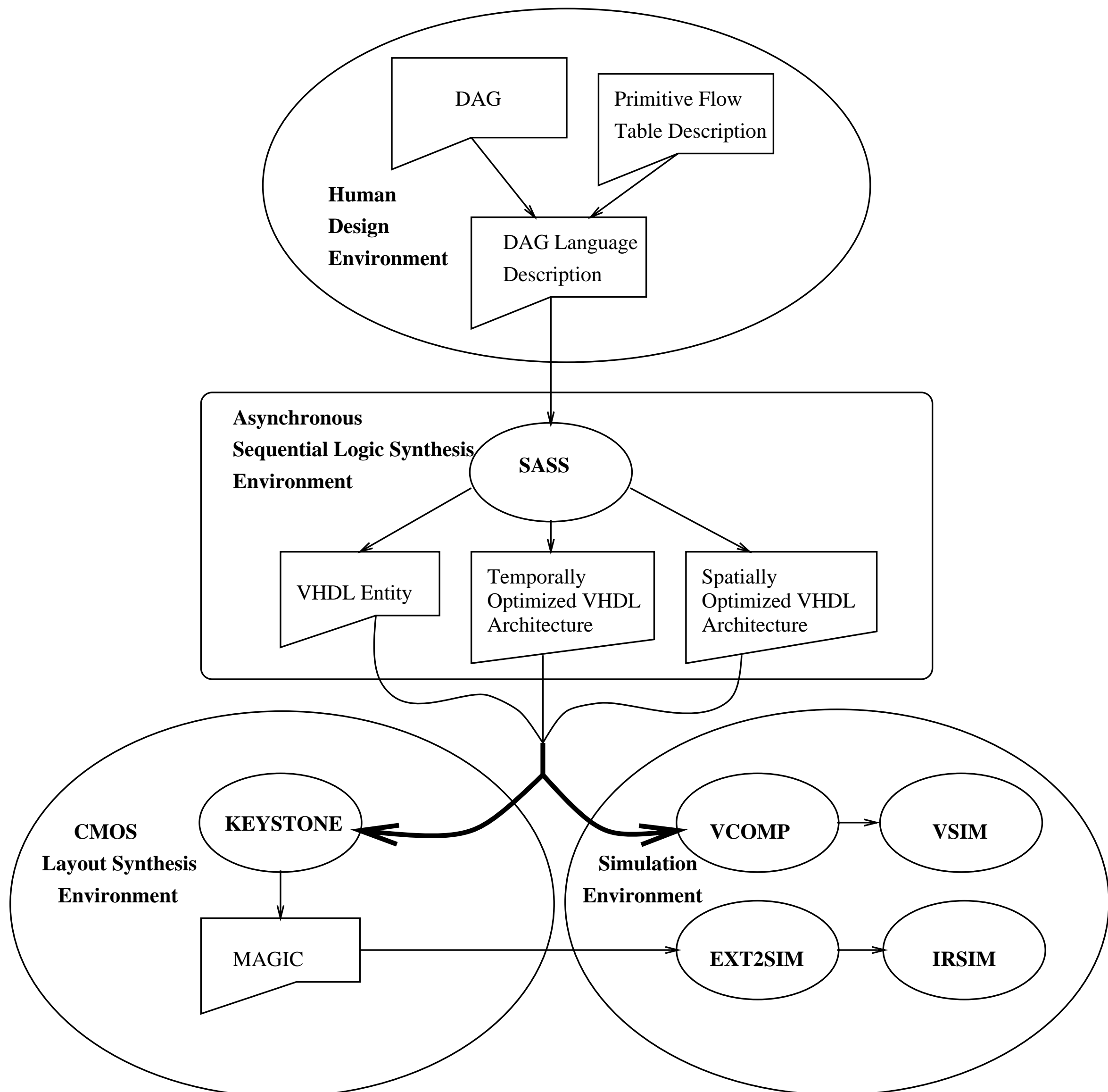


Figure 1 CAD Environment of SASS

2.0 Survey of Asynchronous Sequential Design Methods

In this chapter, the various state assignment algorithms that have been considered for SASS are divided into two categories: spatially efficient, and temporally efficient algorithms. The spatially efficient methods employ adjacent temporary states to form a bridge over erroneous stable states, during a desired state to state transition between two non-adjacent states in a memory feedback circuit [(1)]. Only a single next-state memory variable is permitted to change during each transition between temporary states. These temporary states prevent critical races from occurring between correct stable states and erroneous stable states within a circuit, but at the cost of increased latency.

In temporally efficient methods, the state assignment is such that critical race-free state to state transition can occur without temporary buffer states. State assignment methods of this nature that allow direct state to state transitions are known as STT (single transition time [(7)]). When a transition occurs between two states that are non-adjacent, all the next-state memory variables are permitted to change simultaneously [(8)]. Thus, the latency of non-adjacent state transitions is decreased because the circuit does not have to repeatedly wait for a single memory variable to change and then restablize into a new temporary state. The improvement in speed comes at the expense of more next-state memory variables, which usually will result in more logic circuitry than in spatially efficient methods.

The spatially efficient method that will generally yield the fastest circuit, called One-Hot design; and the temporally efficient method that will derive the smallest circuit, a variant of Lui state assignment, were chosen as the methods of asynchronous sequential circuit synthesis for SASS. The following subsections justify the choices of One-Hot and Tan from among the alternatives.

2.1 Methods Yielding Spatial Efficiency Circuits

2.1.1 Huffman State Assignment

A method developed by D. A. Huffman in the mid-1950's uses temporary states as buffers to control the transition from an initial stable state to a desired stable state in a memory feedback circuit [2]. In some cases, where two states are adjacent, a transition can occur directly. For example, in in the flow table in Figure 2, a transition can be made directly from state **2** to **3** when the input condition changes from I_4 to I_3. The state assignment, which is given by the f_n memory variables, changes by only a single variable, f_3, during the transition. All the other memory variable remain constant during the transition.

Non-adjacent states are those states whose memory variables differ in more than one memory bit. For two non-adjacent states, a number of buffer states might be necessary to enable a race-free transition. If multiple temporary states are employed to buffer a transition between two non-adjacent stable states, the response of the circuit will be slower than for a transition between two adjacent states.

The flow table in Figure 2 has a state assignment that employs multiple temporary state to control the state to state transitions to state 8, 5, and 2. For example, when a transition from stable state **1** to stable state **5** occurs, the circuit must first jump down to row m in the flow table, then from m to i, then from i to j, and finally from j to f. The state assignment given by f_n is contrived such that only one memory variable changes during each jump between rows. During each jump between temporary states, there is no chance of encountering an erroneous stable state because the circuit will remain in a given temporary state until the single memory variable that changed has a chance propagate back through the circuit and invoke the jump to the next temporary state.

The rows of the flow table can often be shuffled or switched such that transitions can be made adjacent to one another. If rows b and f were switched, the transition from stable state **1** to **5** would be faster; however, the transition between **5** and **10** would then be non-adjacent and require a temporary buffer state.

	I_1	I_2	I_3	I_4	f_1	f_2	f_3	f_4
a	**1**	5		2	0	0	0	0
b	1	**4**	**3**	**13**	0	0	0	1
c	8	12	3	**2**	0	0	1	1
d	1	**12**	**6**	**7**	0	0	1	0
e	1	**9**		**11**	0	1	0	0
f	10	**5**	6	13	0	1	0	1
g	**10**			11	0	1	1	1
h	**8**	9		11	0	1	1	0
i		5			1	1	0	0
j		5			1	1	0	1
k	8				1	1	1	1
l	8				1	1	1	0
m		5		2	1	0	0	0
n				2	1	0	0	1
p	8			2	1	0	1	1
q					1	0	1	0

Note: states in boldface are stable states

Figure 2 Sequential Function with Huffman State Assignment

2.1.2 One-Hot State Assignment

An offshoot of Huffman's first method, which is commonly called $One-Hot$ design [2], utilizes exactly one temporary state for every transition. For circuits in which every state to state transition occurs between adjacent states, Huffman's original method will be faster than the One-Hot method. However, for many sequential functions, the adjacency between transitional states is usually impossible to achieve for every possible combination of transitions. The flow table in Figure 2 shows a sequential function in which all the transitions cannot be made adjacent to one another. Since all One-Hot state transitions employ a single temporary state, the response will be more consistent and generally faster than Huffman's first method.

For the same sequential function represented in Figure 2, the One-Hot state assign-

	I_1	I_2	I_3	I_4	Q_a	Q_b	Q_c	Q_d	Q_e	Q_f	Q_g	Q_h
a	**1**	5		2	1	0	0	0	0	0	0	0
b	1	**4**	**3**	**13**	0	1	0	0	0	0	0	0
c	8	12	3	**2**	0	0	1	0	0	0	0	0
d	1	**12**	**6**	**7**	0	0	0	1	0	0	0	0
e	1	**9**		**11**	0	0	0	0	1	0	0	0
f	10	**5**	6	13	0	0	0	0	0	1	0	0
g	**10**			11	0	0	0	0	0	0	1	0
h	**8**	9		11	0	0	0	0	0	0	0	1

Note: states in boldface are stable states

Figure 3 Sequential Function with One-Hot State Assignment

ment is shown in Figure 3. The next state memory variable equations are composed of a hold term and an number of transition terms. If a circuit is in a particular stable state, and a particular memory state variable is asserted, the hold term will cause that variable to remain asserted during a state to state transition until the variable for the next memory state becomes asserted. The transition terms will force the next memory variable to become asserted. For each row of the table, a next-state memory variable equation can be derived by inspection by following the basic algorithm:

- Generate the holding term by forming the product of the asserted state variable, which represents the current row, and the inverted state variables, which represent the stable states in the rows that can be reached directly from the current row.
- Generate the transition terms by forming the product of the input variable condition and the memory state variable that represents that row from which transitions can be made to the current row.
- Form the summation of the holding term and all the transition terms for a particular row.

By applying the previous algorithm to the sequential function in Figure 3, the following memory variable equations can be derived:

$$
\begin{array}{llll}
 & \mathit{Holding} & \mathit{Transition} \\
Q_a = & (Q_a * Q'_f * Q'_c) + & (I_1 * Q_b) + (I_1 * Q_d) + (I_1 * Q_e) \\
Q_b = & (Q_b * Q'_a) + & (I_3 * Q_c) + (I_4 * Q_f) \\
Q_c = & (Q_c * Q'_h * Q'_d * Q'_b) + & (I_4 * Q_a) \\
\vdots & &
\end{array}
$$

The One-Hot method will generally yield simpler next-state memory variable equations than the first method. Although Huffman's first method will result in fewer memory state variables, those equations often contain more product terms, and the typical product term will usually be more complex than the terms in One-Hot equations.

2.2 Methods Yielding Temporally Efficient Circuits

Two similar STT state assignment methods that were developed by C. N. Lui and J. H. Tracey [9, 10] enable asynchronous sequential circuits to be derived that are faster than those from the One-Hot method. Both the Lui and Tracey method partition a given reduced flow table by grouping the rows according to how they are covered by either the different aggregates of numerically equivalent stable and unstable states, or by the unstable to stable state transitions in each particular column. The partitions then are used to generate a set of state assignments which guarantee that during the course of a state transition, no critical race conditions with erroneous stable states will occur. Although circuit complexity depends on how readily the flow table can be partitioned into certain types of partitions, circuits derived from the Lui method tend to be implemented more efficiently than Tracey circuits [7].

A definition and proof of STT state assignment, which is applicable to both Lui and Tracey state assignment methods, is presented in Appendix A.

2.2.1 Tan Algorithm for STT Equation Generation

An iterative next-state memory variable equation generation algorithm developed by C. J. Tan yields equations based on a Lui state assignment directly from a minimized flow table description [8]. Tan's method bypasses the cumbersome task of generating the Lui state assignment, and then deriving memory variable equation from a next-state transition table. A minimized flow table is partitioned into two different types of partitions, and from these partitions a set of initial memory equations is derived. The initial equations are used to extrapolate a minimal, though complete set of equations that will implement the desired function of the flow table.

The iterative algorithm for generating the Lui equations is summarized as follows:

- Generate Type A partitions
- Generate Type B partitions
- Compute the value of each partition in terms of it ability to cover the columns of the flow table
- Generate the set of initial partitions
- Generate the set of initial equations from the initial partitions
- Iteratively generate the complete set of next-state memory equations from the set of initial equations

This algorithm is implemented in C-language, and constitutes a majority of the code written for the entire SASS tool. A more detailed description of the algorithm and definitions of terms are contained in the next section.

2.2.2 Detailed Tan Algorithm for STT Equation Generation

A detailed description of the Tan algorithm for generating Lui next-state memory variable equations is given as follows:

1. Generate Type A and Type B partitions in each column of the flow table.

Type A formed by grouping the rows under a column such that they cleanly dichotomize the aggregate set of rows into two blocks. For a partition to cleanly dichotomize the rows under a column, it must contain every row that has the same stable or unstable state numeral. It is acceptable for the partition to have rows that represent more than one state numeral. If the block can cleanly dichotomize two or more columns, then the partition is Type A.

Type B formed by cleanly dichotomizing the rows under a particular column into a partition, and then checking the partition's rows under every other column to determine if any row contains a stable state. If there is at least one column that does not contain the partition's rows with stable states, then the partition is Type B. It is possible for a partition to be both Type A and Type B.

2. Compute the W_p of each partition according to the equation: $W_p = M_p + M_I(N_p - 1)$

   W_p is a measure of how well the memory variable equation that can be generated from the partition will implement STT state assignment.

   M_p is the number of columns which do not contain a stable state at the row positions indicated in the partition. A Type B partition will have $M_p \geq 1$.

   M_I is the number of flow table columns.

   N_p is the number of columns that can be cleanly dichotomized by a particular partition. A Type A partition will have $N_p > 1$. A partition that is exclusively Type B will have have $N_p = 1$.

3. Generate the set of initial partitions by chosing the Type A or Type B partition with the highest W_p in each column. If no Type A or B partitions are present under a column, then choose any partition that cleanly dichotomizes the rows under the column. Since it is possible for Type A and B partitions to cover more than one column, only include the partition once in the set of initial partitions even if it happens to be the best for multiple columns.

4. Store the initial partitions in a memory equation variable list.

5. From the initial partition set, iteratively generate the initial set of next-state memory variable equations.

    - The product terms for each equation are formed by the input of a particular column, and the product partition composed of rows holding equivalent stable and unstable states.
    - The rows for the product term partition are derived by examining the rows in the initial partition. Under a particular column, if the row given in the initial partition holds a stable state, then every row in the flow table with the same state number is partitioned into the product term.
    - If the aggregate set of rows is contained in the product term partition, then only include the input column condition number in the product term.
    - Add every unique product term partition that was formed during memory variable equation generation to a list that holds potential memory variables.
    - Rank each partition in the potential memory variables list according to how many new product terms its potential memory variable equation would contain if it were generated. The partition with the lowest ranking has the least number of new product terms, and is thus chosen to generate another memory variable equation.

6. Choose the product term partition with the lowest ranking from the potential memory variable partition list.

    - If the product partition can be composed of the inverse dichotomies of partition terms of current memory variable equations, then replace every instance of the product partition in the existing memory variable equations with the inverted partition composition.
    - If the product term cannot be composed of the inverted current memory variable terms, then generate a new memory variable equation from the term.

7. Continue choosing product partition terms from the list of potential memory variables until the list is empty

   - As new memory variable equations are generated, any new product term partitions formed in the process are added to the list of potential memory variables.

2.3 Robustness of State Assignment Algorithms

For every sequential function, the One-Hot and Tan algorithms will not always produce sets of memory variable equations that are free of hazards and race conditions. Certain sequential functions produce minimized flow tables that lead to One-Hot circuits with inherent race conditions. For Tan circuits, the logic used to generate the input conditions must be free of glitches, or all the memory feed-back variables will become permanently unasserted after a change in input. The causes and cures for the problems are explained further in the next sections.

2.3.1 One-Hot

The One-Hot algorithm, for certain condensed flow table descriptions, will generate next-state memory variable equations that have inherent hazards for certain transitions. For example, the flow table in Figure 4 has an asynchronous sequential circuit implementation that will fail during the transition from stable state **2** to **3**. The reason for this failure it that the two memory variable equations that control the transition contain each others' inverted memory variable in their respective holding terms.

For the flow table in Figure 4, the following equations:

```
q2 <= (( abar and b and q4 )or(a and  bbar and q5 )or
      (a and  bbar and q1 )or( q3b and  q1b and q2 ))and resetbar;
q1 <= (( abar and  bbar and q2 )or( q2b and q1 ))and resetbar;
```

control the transitions from state **2** to **3**. Signals that end with '*b*' or '*bar*' are the complemented versions. When in state **2**, $q2$ is asserted; however, when input a becomes unasserted, a transition should be made from **2** to a temporary state, in which both $q1$ and $q2$ are asserted, and then finally to state **3**, in which only $q1$ is asserted. This sequence of state transitions is correct and desirable.

$a' * b'$	$a * b'$	$a' * b$	$a * b$
3	2	**4**	**9**
3	**2**	**8**	6
1	**11**	7	**6**
5	11	8	**10**
1	2	**7**	10

Note: states in boldface are stable

Figure 4 Flow Table With Inherent Race Condition

In the actual circuit, the race condition is initiated by the transition to the temporary state. When the circuit is in state **2** and input a becomes unasserted, $q1$'s transition term, ($abar\ and\ bbar\ and\ q2$), will cause $q1$ to become asserted. The circuit is then in the temporary state with both $q1$ and $q2$ asserted. Since the inverted $q1$ signal, $q1b$, is contained in $q2$'s holding term, ($q3b\ and\ q1b\ and\ q2$), it will force $q2$ to become unasserted. When $q2$ becomes unasserted, $q1$ erroneously becomes unasserted again because $q1$'s transition term, ($abar\ and\ bbar\ and\ q2$) becomes unasserted before the holding term, ($q2b\ and\ q1$), has a chance to lock $q1$ into an asserted state. This behavior was verified in both VSIM and IRSIM simulations of the complete asynchronous circuit derived from the flow table in Figure 4.

Fortunately, correct asynchronous sequential circuits can sometimes be derived by condensing the primitive flow table of the original function in a different manner. For example, the same functionality for the previous table can be derived by the table in Figure 5, and a circuit can be derived without race conditions in any of the transitions. However, the circuit will be slightly larger because of the addition of another memory state variable.

2.3.2 Tan

The Tan equations are susceptible to hazards formed by the 2^n combinations of input signals, where n is the number of input signals. If a function has two inputs, and those inputs are used to form four different input conditions to the circuits, then hazards are inherent unless interlocked logic is used to generate the input conditions. For the table in Figure 4, there are four possible input conditions for the two inputs, a and b:

```
abar <= not a;
bbar <= not b;
I1 <= abar and bbar ;
I2 <= a    and bbar ;
I3 <= abar and b ;
I4 <= a    and b ;
```

Since an inverter must be used to generate the complementary signals in a physical realization, it is possible for both a signal and its complement to simultaneously have the same value. If the input conditions $I1$ through $I4$ are implemented literally as in the equations above, then it is possible for none of the input conditions to be asserted at some instant in time when either a or b are changing from high to low.

The Tan equations are such that if all the input conditions become unasserted, then every memory variable will also become unasserted. For the table in Figure 4,

$a' * b'$	$a * b'$	$a' * b$	$a * b$
1	2	**7**	10
3	**2**	4	6
5	11	8	**10**
1	**11**	7	**6**
3		**8**	6
3	2	**4**	**9**

Note: states in boldface are stable

Figure 5 Algorithmically Correct Flow Table for One-Hot Synthesis

the following subset of Tan equations can be derived such that $q5$ is unasserted and $q12$ is asserted when the circuit is in stable state **2**:

```
q12 <= ((I4 and q9 ) or (I3 and q12 ) or
        (I2 and q15 ) or (I1 )) or reset;
q5  <= ((I3 and q5 ) or (I1 and  q2b )) or reset;
```

The input condition $I2$ is asserted since the circuit is in state **2**. If a transition from state **2** to **3** is desired, then a must become unasserted, which should then cause a transition to **3** where both $q5$ and $q12$ are asserted. When a becomes unasserted, the input condition is merely changing from $I2$ to $I1$. However, since $I2$ is composed of (a *and bbar*) and $I1$ is composed of (*abar and bbar*), then both condition will momentarily become unasserted when a goes from high to low, which will cause $q12$ and $q5$ to become permanently unasserted. The Tan equations are particularly susceptible to input signal hazards because they have no holding term that keeps individual memory signals stable during input signal transitions, as do the One-Hot equations.

To eliminate the input hazards, the equations $I1$ through $I4$ should be implemented as follows:

```
I1 <= (abar and bbar) or ((not a) and (not b));
I2 <= (a and bbar) or ((not abar) and (not b));
I3 <= (abar and b) or ((not a) and (not bbar));
I4 <= (a and b) or ((not abar) and (not bbar));
```

Although these equations have redundant terms, the physical implementation will interlock the true and complement input signals of a and b, so that at least one input condition term, $I1$ through $I4$, will be asserted during a high to low transition of a or b. The true and complement signal inputs must be treated as four independent inputs to the asynchronous circuit.

2.4 Survey of Existing Tools and Approaches

Three more recent approaches were considered for their relevance to asynchronous sequential logic design. The first approach surveyed is suited for communications interface synthesis, and not applicable to the synthesis of general purpose asynchronous sequential circuits. It synthesizes the binary handshake protocol between asynchronously connected independently synchronous modules. The second approach surveyed is suitable for the synthesis of general purpose asynchronous circuitry; however, it might not be as spatially efficient as other synthesis methods. Notably, neither of the approaches surveyed indicated how the asynchronous sequential circuits they synthesized are initialized.

Also, the suite of Berkeley tools that perform $One - Hot$ state assignment for synchronous sequential circuits were analyzed for their adaptability to synthesizing asynchronous sequential circuits. They were not readily adaptable because their state assignment algorithms are merely subsets of the algorithms that perform asynchronous state assignment in SASS. A more detailed explanation of the analysis is contained in Appendix C.

2.4.1 System Synthesis

In the first approach, a tool written in Franz Lisp synthesizes asynchronous interconnections between the synchronous modules of a digital signal processor [11]. The approach utilizes the fact that a four-phase handshake protocol is able to facilitate general asynchronous communication between two processor modules. A signal transition graph representation of the handshake protocol is constructed such that it is semi-modular; which means that every loop within the graph contains pairs of signals that initiate each others' transitions between logic levels. From this graph, a semi-modular circuit implementation can be found that is free of hazards and allows maximum concurrent operation between the interconnected modules.

To implement the semi-modular circuit, a specialized differential cascode voltage switch logic (DCVSL) circuit is utilized so that two or more modules can asynchronously communicate. The functionality of DCVSL circuits can also be repre-

sented in boolean algebra by C-elements, which have the property of the output signal following the level of the input signals only if the logic levels of all the input signals are identical. The use of DCVSLs enables efficient generation of asynchronous interface mechanisms for a digital signal processor design. However, other kinds of system design might require components that have asynchronous interfaces which are not semi-modular. Also, since the KEYSTONE layout tool does not support DCVSL circuits, semi-modular circuit implementations derived from C-element equations might not have any greater area or speed advantage when compared to asynchronous sequential circuits derived from other methods.

2.4.2 Circuit Synthesis

Unlike the formerly mentioned tool, which is intended for the synthesis of system or module interfaces, a recently developed design methodology can generate general purpose asynchronous sequential circuits, implemented with duplicate structures built of MOS pass-transistors (7). Within each structure are sub-circuits composed of either a pull-up or pull-down transistor followed by a chain of pass-transistors, which define a state to state transition path for a next-state memory variable. A number of such subcircuits may be wired in parallel to form the complete transition function for a particular memory variable. Aside from having different configurations of pull-up and pull-down transistors, the pass-transistor circuitry necessary to generate a particular next-state memory variable is structurally identical to the circuitry of any other memory variable. For a minimized flow table representation that has either a Lui (9) or Tracey (10) state assignment, with M next-state memory variables, a single pass-transistor structure repeated M times will suffice to implement the function.

Since the core pass-transistor structure is not logically minimized, the density of the layout area will depend greatly on the quality of the placement of the duplicated structures and their routing to to one another. For the same flow table representation and state assignment methods, the logic equations derived by means of a next-state transition table are much less complex. Consequently, a more compact layout might result with restored CMOS logic, as opposed to building multiple core pass-transistor structures.

Also, as the complexity of the sequential function increases, the length of the series chains of pass-transistors will grow, and thus the speed of the overall circuit will decrease. The length of the series chains of pass-transistors is determined by the number of inputs in the sequential function. If a sequential circuit has more inputs, each particular state will have more possible states to which it can transverse because their will be more input conditions that can change. A state to state transition path will need to have more series transistors in order to differentiate it from other paths.

2.5 Summary Reasoning for the State Assignment Algorithms used in SASS

Of the numerous state assignment methods considered for SASS, none produce a guaranteed minimal solution in terms of either size or speed. Before the code for SASS was written, manual experimentation was conducted by applying the Huffman, One-Hot, Lui, Tracey, Tan, and Whitaker [7] (see Section 2.4.2) algorithms to an example sequential function. In order to intuit which algorithms would be best for SASS, the sets of logic equations that were produced by the various algorithms were compared in terms of:

- size of the equation set
- complexity of product terms
- circuit layout area

Only a single sequential function was used for experimentation, however, it was sufficient to achieve an understanding of how the algorithms functioned, and what were the characteristics of the memory variable equations that they produced. As far as manually determining the best algorithms with absolute certainty, the sheer complexity of the algorithms made a massive manual experimentation impractical and prohibitively time-consuming with a statistically significant quantity of different functions.

Upon comparing the next-state memory variable equations that result from the One-Hot and Huffman algorithms, the former appeared to produce the best equations in terms of speed and area. Reguardless of the sequential function, the One-Hot equations only utilize a single temporary state during every state to state transition; whereas the Huffman equations might utilize multiple temporary state during a transition. Although the Huffman algorithm might yield faster equations for some trivial sequential function with few states, it was thought that for more complex functions the One-Hot would produce circuits that were consistently faster. From manual observation of the One-Hot and Huffman equations, it was also concluded that the One-Hot equations were simpler, and thus apt to produce smaller circuit layouts. Although the Huffman algorithm produces state assignments with less memory variable equations, those equations often contain more product terms, and the product terms were larger.

Upon comparing the memory variable equations produced by the STT state assignment algorithms, the Tan algorithms yielded the equations that generally had the least complex logic terms. The Tan algorithm uses a Lui state assignment, which was shown to generate simpler logic equations [7] than Tracey, in the formation of its memory variable equations. It was thought that more efficient equations might also be faster when synthesized into circuit layout.

The Whitaker algorithm, which uses either Lui or Tracey state assignment, did not seem applicable to general purpose CMOS logic synthesis. The algorithm produces multiple duplicated pass-transistor structures that implement the sequential function. Upon analyzing the pass-transistor memory variable equations, it was noticed that they translate into unoptimized conventional CMOS logic. Since VHDL can describe conventional CMOS logic structures, but does not yet have a way to describe logic equations implemented with pass-transistors, the Whitaker algorithm was eliminated from consideration at the time of coding.

When comparing the sets of memory variable equations produced by the One-Hot and Tan algorithms, it was noticed that the One-Hot algorithm usually produced fewer equations, and therefore resulted in a smaller layout. However, since the One-Hot employed a temporary state during state transitions, and the Tan allowed all the memory variables to switch simultaneously, the former was thought to be slower.

Therefore, the One-Hot algorithm was chosen because it seemed to produce the equations that resulted in the smallest layouts, and the Tan algorithm was chosen because it produced the equations that switched states the fastest.

3.0 Syntactic Structure and Semantic Requirements for the DAG Input Language

A directed acyclic graph (DAG) language is used to describe the function of an asynchronous sequential circuit for SASS. The lexical and syntactical details of the language are given in Appendix B. An example of a DAG for an asynchronous circuit is shown in Figure 6, and its corresponding textual description is shown in Figure 7. All fundamental mode input signals are declared as **input** types, and output signals are declared as **output** types. Next, a list of state to state transition statements delimited by semicolons are given to complete the entire functional description. Each statement begins with a numeral that represents the stable state from which transitions are made to other states, and then continues with a list of transition constructs to other states, which are delimited by commas. Each transition construct is composed of an input signal name, followed by a rising (carrot) or falling (backslash) signal transition type, followed by a transition state numeral. After the the list of transition constructs, there is an optional list of output variable names to be asserted while in the stable state.

3.1 Determination of Root State

There are subtle semantics incorporated into the DAG language. For example, the first stable state encountered in the first state to state transition statement is assumed to be the root state from which all other states can be reached. The input signals at the root state are all assumed to be unasserted; therefore, transitions from the root state to other states can only be caused by input signals that become asserted. This assumption greatly simplifies the DAG language without limiting its descriptive power, because most asynchronous functions encounter a condition where all its input signals are unasserted (usually during initialization). For a function that begins with asserted input signals, such signals can be inverted so that they seem to be unasserted from the perspective of the asynchronous circuit.

3.2 Acyclic Description Requirement

Another semantic subtlety in the DAG language is that in the state to state transition statement description, the transitions from the stable state must be described in a strictly acyclic manner. A statement such as:

```
3, a^6, b^3, c^3, z;
```

is illegal. Although the transition is semantically precise, the semantic of output signal assertion for signal z is vague. There is insufficient information in the statement to decide whether or not to assert z when a, b and c are low, when a and b are low and c is high, when a and c are low and b is high, or when any combination of the previous three situations occur. If z is to be asserted only when both a and b are unasserted and c is asserted, then additional transition statements should be added and the first one revised in the following manner:

```
3, a^6, b^4, c^5;
4, b\3;
5, c\3, z;
```

Thus, the meaning is clarified as to the assertion of the output z.

3.3 Implicit Fundamental Mode Operation

By the nature of syntactical structure of the DAG language, fundamental mode operation is guaranteed. A transition such as:

```
3, a^b\4;
```

states that both a and b must change simultaneously in order to cause a transition to state **4**. This statement is syntactically incorrect, as well as being unrealizable in a circuit.

Another subtlety is implied in the transitions between stable states, which cannot occur unless an input signal changes value. The following state transition statements are illegal:

```
3, a^4, b^5;
4, a^6;
```

The previous statements imply that the sequence of transitions will somehow proceed from state **3** to **4**, and then from **4** to **6**, once signal a becomes asserted and remains asserted. This description would be acceptable in a synchronous sequential environment where a periodic clock is implicit. However, in an asynchronous environment, there is no period clock to force a change in state.

3.4 Redundant Functionality

Another expectation in the functional description is that it does not contain duplicate states that represent the same sequential condition. In Figure 8, a flow table is shown containing a state **4**, which is redundant, and should be merged with state **1**. State **4** is redundant because for the same input stimuli, the same state to state transitions are made as from state **1**; and the output variable, Z, is the same for both states. Although there are methods for checking and eliminating redundant states [(1)], none are currently implemented in SASS. The designer is expected to efficiently describe the sequential function in the DAG language.

3.5 Disjoint DAG

A final semantic subtlety in the DAG language is that the DAG itself not be disjoint. Every state in the state description should be reachable from the root state. A disjoint DAG essentially represents two distinct sequential functions, and would result in a minimized flow table description in which both functions were concatenated; but neither one able to reach the other at any point. Any circuit derived from such a table would be only partly functional.

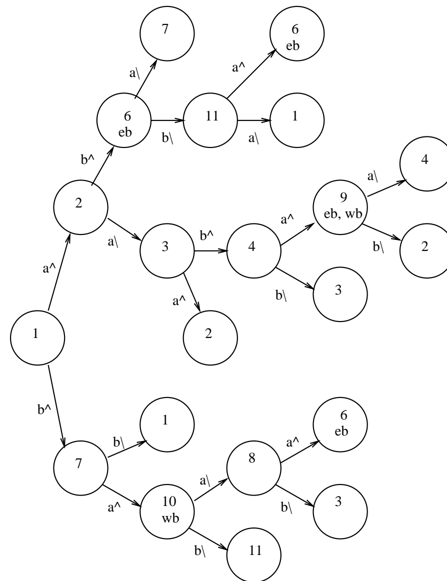


Figure 6 Directed Acyclic Graph For A Sequential Function

```
/* Primitive flow table description for an
   asynchronous sequential function in [Tan71] */
flowtable tan_pg387;
input a, b;
output eb, wb;
  1, a^2, b^7;
  2, b^6, a\3;
  3, b^4, a^2;
  4, a^9, b\3;
  5, a^11, b^8;
  6, a\7, b\11, eb;
  7, a^10, b\1;
  8, a^6, b\3;
  9, a\4, b\2, eb, wb;
 10, a\8, b\11, wb;
 11, b^6, a\1;
endtable
```

Figure 7 Corresponding DAG Language Description

I_1	I_2	I_3	I_4	Z
1	2		3	0
4		6	**3**	0
1	**2**	5		0
4	2		3	0
	2	**6**	3	1
	2	**5**	3	0

Note: states in boldface are stable states

Figure 8 Flow Table with Redundant Functionality

4.0 General Synthesis Algorithm For SASS

4.1 Front-End Routines

The initial steps toward synthesizing asynchronous sequential circuits shown in Figure 9 are the same for either the spatially or temporally optimized synthesis. A DAG language description, such as the one in Figure 7, is parsed into syntax tree form, semantically analyzed for correctness, then converted into a *primitive flow table* format [2]. The One-Hot and Tan equation generation algorithms can utilize a DAG in a tabular format more naturally than in a multi-partite tree format. A sequence of clique partitioning algorithms are then applied to the primitive flow table to condense the quantity of rows. The result is an equivalent, yet minimized flow table that exhibits the same sequential functionality as the primitive flow table. The details of the clique partitioning algorithms are contained in Section 4.1.1. Since it is possible that more than one uniquely minimized flow table may exist for a particular DAG description, a unique architecture can be synthesized from each unique minimized flow table for either spatially or temporally tuned circuits.

4.1.1 Clique Partition Algorithms

Three clique partitioning algorithms have been implemented in SASS in order to determine the optimum row reduction pattern for a primitive flow table. The algorithms are as follows:

Pseudo Fiduccia and Mattheyses Clustering: The heuristic behind this algorithm is that the most strongly connected rows are probably connected to one another, and that if they are compared first, then they can be clustered into larger groups so that the minimal number of groups can be derived [12]. The clustering algorithm is given as follows:

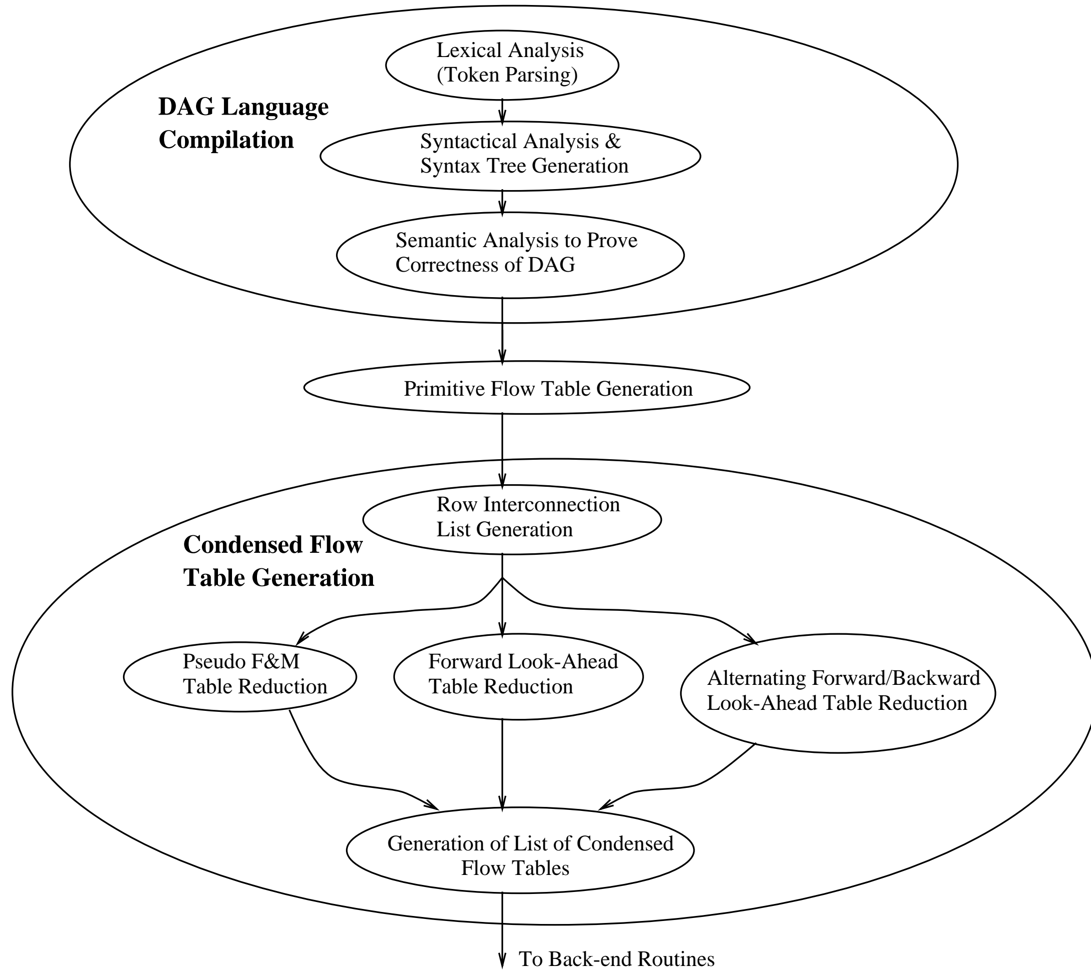


Figure 9 Common Front-End Routines

1. Determine interconnection strength for each row, and place each row in a ranked list called the strength list. The strength of a given row increases with its quantity of interconnections to other rows.
2. Find the most strongly connect row or rows in the strength list
3. Form a new group using the strongest interconnected row.
4. Check all other rows of the same strength and lesser strengths in descending order of strength. If a row is interconnected to every member of the group, then add it to the group.
5. Delete all members of the group from the strength list.
6. If the strength list is not empty, go to Step 2.

The maximum cost of this algorithm is estimated O(n!/n), where n is the number of rows in the primitive flow table. Typically, the cost will be much less than the maximum since many rows in the primitive flow table can be combined during each pass the table; thus, eliminating the need for more iterations through the strength list.

Iterative Look-Ahead Clustering: The heuristic behind this algorithm is that given particular row in the flow table, if every row immediately ahead of it is checked and, if possible, merged into a group, then the optimum clustering might be found because rows that are near one another have a tendency to merge together. The algorithm is iterative because each row in the primitive flow table is taken as the initial row for the algorithm; thus, the maximum number of reduced flow tables generated is equal to the number of rows originally in the primitive flow table.

1. Initialize the list of rows that are not yet in groups. Essentially, declare every row to be ungrouped.
2. Choose a first row from the primitive flow table that has not been used as a seed row and form a new group using it as the seed member.
3. Check all other succeeding rows, $n+1, n+2, \cdots$ If a row is interconnected to every member of the group, then add it to the group. When the all succeeding rows have been checked, go on to next step.
4. Delete all members of the group from the list of rows that are not yet in groups.
5. Find a row, n, that is ahead of the the seed row, and not a member of any group, and form a new group using it as the seed member. Then go to Step 3.
6. If a row cannot be found, thus meaning that the list of rows that are not yet in groups is empty, then store the set of groups that has been formed. Then go to Step 1.

The estimated cost of this algorithm is O(n!), where n is the number of rows in the primitive flow table.

Iterative Window Clustering: This algorithm is an extension of the previous Look-Ahead Clustering algorithm. In the previous algorithm, given a row, the other rows that are immediately ahead might be clustered to it; however, sometimes the optimum clustering is overlooked because the closest rows are grouped together when it would would have been better to cluster rows that were either slightly farther ahead or behind a given row.

To avoid the problem of accidentally making many small clusters, the Iterative Window Clustering algorithm alternates between searching ahead and forward after forming each new cluster of rows. It also iteratively chooses each row in the primitive flow table as the initial seed row of the first set of groups.

1. Initialize the list of rows that are not yet in groups. Essentially, declare every row to be ungrouped.
2. Choose a first row from the primitive flow table that has not been used as a seed row and form a new group using it as the seed member.
3. If toggle = FORWARD, Check all other succeeding rows, $n+1, n+2, \cdots$, else check all previous rows. If a row is interconnected to every member of the group, then add it to the group. When the all succeeding or previous rows have been checked, go on to next step.
4. Delete all member of the group from the list of rows that are not yet in groups.
5. If toggle = FORWARD, find a row, n, that is behind the seed row, and not a member of any group, and form a new group using it as the seed member. Then set toggle = BEHIND.

   If toggle = BEHIND, find a row, n, that is ahead of the seed row, and not a member of any group, and form a new group using it as the seed member. Then set toggle = FORWARD.

   Then go to Step 3.
6. If a row cannot be found, thus meaning that the list of rows that are not yet in groups is empty, then store the set of groups that has been formed. Then go to Step 1.

The estimated cost of this algorithm is O(n!), where n is the number of rows in the primitive flow table.

For the DAG language description in Figure 7, SASS will generate a primitive flow table. A row to row interconnection list will then be formed from the primitive flow table. This list tells which rows in the primitive flow table that can be connected to one another. The three table reduction algorithms use the interconnection list to determine the optimum row condensation for the primitive flow table. The program output of SASS for the row merging routines is contained in Figure 10.

All of the clique partitioning algorithms yield the same clustering method for the primitive flow table in Figure 10. Since all the row groupings are equivalent, only the first grouping from the **Pseudo Fiduccia and Mattheyses Clustering** algorithm is used to generate the reduced flow table, as shown in Figure 11. Had more than one equally minimal, but unique, clustering method been found for the primitive flow table, it would have been saved and used to generate a unique reduced flow table.

These clustering algorithms were derived on the basis of the author's empirical experience in reducing primitive flow tables. An algorithm developed for optimal PLA folding [13] was considered for use in SASS; however, it was found to require that the rows of a given table be combined in a certain order, and placed in a certain location, so as to insure that the PLA could be physically realized. For the primitive flow table of a sequential function, the order of the rows in the flow table, or their order of combination, does not affect the function of the circuit, for two equivalent reduced flow tables. For example, all three reduced flow tables that would be produced from the different clustering methods in Figure 10 would be produce circuits that are functionally equivalent.

4.2 Back-End Routines

In Figure 12, the back-end sequence of events executed by SASS will produce either spatially or temporally tuned asynchronous circuits. If a spatially optimal circuit is desired for a DAG language description in Figure 7, SASS will synthesize the VHDL architecture using the One-Hot algorithm. From the flow table in Figure 11,

```
*********************Primitive Flow Table*********************
ab=
          00   10   01   11

Row   1:  1~S    2    7    -
Row   2:    3  2~S    -    6
Row   3:  3~S    2    4    -
Row   4:    3    -  4~S    9
Row   5:  5~S   11    8    -
Row   6:    -   11    7  6~S
Row   7:    1    -  7~S   10
Row   8:    3    -  8~S    6
Row   9:    -    2    4  9~S
Row  10:    -   11    8 10~S
Row  11:    1 11~S    -    6

*********************Interconnection List*********************
Row   1: 7,
Row   2: 8, 3,
Row   3: 9, 4, 2,
Row   4: 9, 3,
Row   5: 10,
Row   6: 11,
Row   7: 1,
Row   8: 2,
Row   9: 4, 3,
Row  10: 5,
Row  11: 6,

*** PseudoFM ***
*********************Set 1*********************
Group 1: 4, 9, 3,
Group 2: 8, 2,
Group 3: 6, 11,
Group 4: 5, 10,
Group 5: 1, 7,

*** IterativeLookahead ***
*********************Set 1*********************
Group 1: 9, 4, 3,
Group 2: 10, 5,
Group 3: 11, 6,
Group 4: 1, 7,
Group 5: 2, 8,

*** WindowCluster ***
*********************Set 1*********************
Group 1: 9, 4, 3,
Group 2: 8, 2,
Group 3: 10, 5,
Group 4: 7, 1,
Group 5: 11, 6,
```

Figure 10 Results of Flow Table Merging Routines

	$a' * b'$	$a * b'$	$a' * b$	$a * b$
Row 1	**3**	2	**4**	**9**
Row 2	3	**2**	**8**	6
Row 3	1	**11**	7	**6**
Row 4	**5**	11	8	**10**
Row 5	**1**	2	**7**	10

Note: states in boldface are stable

Figure 11 Condensed Flow Table For DAG

the One-Hot algorithm will generate the VHDL architecture contained in Figure 13. The initialization logic is incorporated into the memory variable equations when the VHDL architecture description is formulated. The output variable equations are derived by summing the minterms that are found wherever a particular output signal is asserted for a stable state in the minimized flow table. A minterm is composed of input signals and next-state memory variables. A the end of the VHDL architecture description, a set of worst-case statistics are computed to give an estimate of the layout complexity.

If a temporally tuned circuit is desired, SASS will utilize the Tan algorithm to synthesize the VHDL architecture. The first part of the Tan algorithm will generate type A and type B partitions as shown in Figure 14 from the minimized flow table shown in Figure 11. The definitions and rules for deriving type A and partitions, Wp, Np, and Mp are given in Section 2.2.2

A small set of initial partitions is formed from the aggregate set of type A and type B partitions. It is possible to formulate more than one set of valid initial partitions, as shown in Figure 15. From each partition in the initial set, a next-state memory variable equation is derived. From these seed equations, the complete set of next-state equations necessary to implement the flow table function can be extrapolated. All sets of seed equations are extrapolated in order to find the solution that leads to the least number of total next-state memory equations. For the first initial set in Figure 15, SASS generated the architecture in Figure 16.

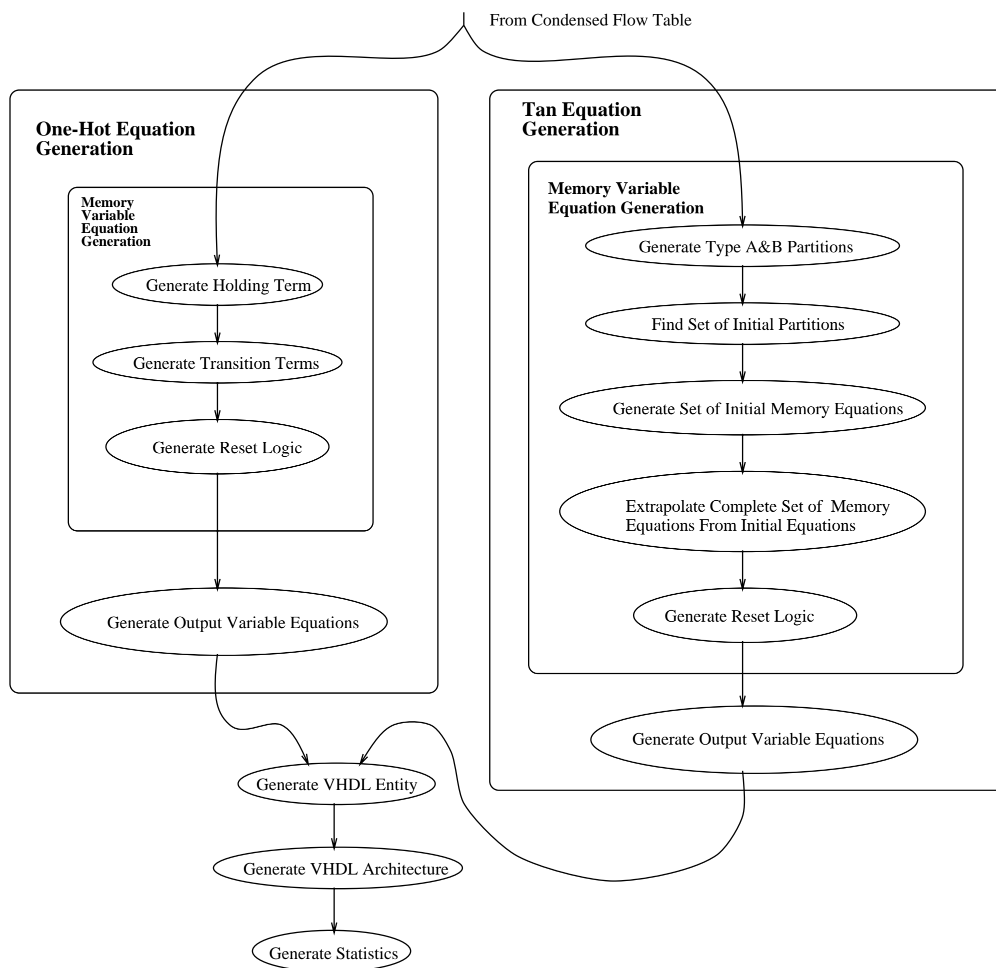


Figure 12 Back-End Equation Synthesis Routines

```
architecture tan_pg387_1Hot_1 of tan_pg387 is
 signal q1, q1b, q2, q2b, q3, q3b, q4, q4b, q5, q5b: bit;
 signal abar, bbar, resetbar: bit;
begin
 abar <= not a;
 bbar <= not b;
 resetbar <= not reset;
 q1b <= not q1;
 q2b <= not q2;
 q3b <= not q3;
 q4b <= not q4;
 q5b <= not q5;
 q5 <= (( abar and b and q3 )or( abar and  bbar and q3 )or
       ( q4b and  q2b and q5 ))or reset;
 q4 <= ((a and b and q5 )or( q2b and  q3b and q4 ))and resetbar;
 q3 <= ((a and b and q2 )or(a and  bbar and q4 )or
       ( q5b and  q5b and q3 ))and resetbar;
 q2 <= (( abar and b and q4 )or(a and  bbar and q5 )or
       (a and  bbar and q1 )or( q3b and  q1b and q2 ))and resetbar;
 q1 <= (( abar and  bbar and q2 )or( q2b and q1 ))and resetbar;
 wb <= (a and b and  q5b and q4 and  q3b and  q2b and  q1b )or
       (a and b and  q5b and  q4b and  q3b and  q2b and q1 );
 eb <= (a and b and  q5b and  q4b and q3 and  q2b and  q1b )or
       (a and b and  q5b and  q4b and  q3b and  q2b and q1 );
end tan_pg387_1Hot_1;
-- Memory + Output Variable Equations = 7
-- Total Inverters = 8
-- Max Transitor Est. (CMOS implementation, using only 2-input logic gates)
--    n-channel transistors = 209
--    p-channel transistors = 209
--        total transistors = 418
```

Figure 13 One-Hot VHDL Synthesized By SASS

```
*****************Type A Tan Partition 1*****************
p6 = [5,4,3,2,]  COLUMNS:4,3, Type A,      Wp = 2, Np = 2, Mp = 0
p5 = [1,]        COLUMNS:4,3, Type A & B,  Wp = 3, Np = 2, Mp = 1
p3 = [4,2,1,]    COLUMNS:3,1, Type A,      Wp = 2, Np = 2, Mp = 0
p2 = [5,3,]      COLUMNS:3,1, Type A,      Wp = 2, Np = 2, Mp = 0

*****************Type B Tan Partition 1*****************
p11 = [1,4,5,]  COLUMNS:4, Type B,  Wp = 1
p10 = [5,4,]    COLUMNS:4, Type B,  Wp = 1
p9 = [3,2,]     COLUMNS:4, Type B,  Wp = 1
p8 = [1,]       COLUMNS:4, Type B,  Wp = 1
p7 = [1,]       COLUMNS:3, Type B,  Wp = 1
p4 = [4,3,]     COLUMNS:2, Type B,  Wp = 1
p1 = [4,]       COLUMNS:1, Type B,  Wp = 2
```

Figure 14 Type A and B Partitions Generated by SASS

```
*****************Tan Seed Partitions For Flow Table 1*****************
Set 1:
p5 = [1,]      COLUMNS:4,3, Type A & B,  Wp = 3, Np = 2, Mp = 1
p4 = [4,3,]    COLUMNS:2,   Type B,      Wp = 1
p3 = [4,2,1,]  COLUMNS:3,1, Type A,      Wp = 2, Np = 2, Mp = 0
Set 2:
p5 = [1,]    COLUMNS:4,3, Type A & B,  Wp = 3, Np = 2, Mp = 1
p4 = [4,3,]  COLUMNS:2,   Type B,      Wp = 1
p2 = [5,3,]  COLUMNS:3,1, Type A,      Wp = 2, Np = 2, Mp = 0
Set 3:
p5 = [1,]    COLUMNS:4,3, Type A & B,  Wp = 3, Np = 2, Mp = 1
p4 = [4,3,]  COLUMNS:2,   Type B,      Wp = 1
p1 = [4,]    COLUMNS:1,   Type B,      Wp = 2
```

Figure 15 Initial Partitions Generated by SASS

```
architecture tan_pg387_Tan_1 of tan_pg387 is
 signal q17, q17b, q14, q14b, q12, q12b, q15, q15b, q13, q13b, q3, q3b,
         q4, q4b, q5, q5b: bit;
 signal abar, bbar, resetbar: bit;
 signal I1, I2, I3, I4: bit;
begin
 abar <= not a;
 bbar <= not b;
 resetbar <= not reset;
 I1 <=  abar and  bbar ;
 I2 <= a and  bbar ;
 I3 <=  abar and b ;
 I4 <= a and b ;
 q17b <= not q17;
 q14b <= not q14;
 q12b <= not q12;
 q15b <= not q15;
 q13b <= not q13;
 q3b <= not q3;
 q4b <= not q4;
 q5b <= not q5;
 q17 <= ((I4 and  q15b )or(I3 and q17 )or
       (I2 and q4 )or(I1 and  q13b ))or reset;
 q14 <= ((I4 and q14 )or(I3 and q17 )or
       (I1 ))or reset;
 q12 <= ((I4 and q5 )or(I3 and q3 )or
       (I2 and  q4b )or(I1 and q12 ))and resetbar;
 q15 <= ((I4 and q15 )or(I3 and  q3b )or
       (I1 and  q12b ))or reset;
 q13 <= ((I4 and q15 )or(I1 and q13 ))and resetbar;
 q3 <= ((I4 and q14 )or(I3 and q3 )or
       (I2 and  q4b )or(I1 and q3 ))and resetbar;
 q4 <= ((I4 and  q5b )or(I2 and q4 )or
       (I1 and q13 ))and resetbar;
 q5 <= ((I4 and q5 )or(I3 and q5 )or
       (I1 and q12 ))and resetbar;
 wb <= (I4 and  q5b and q4 and q3 and q13 and q15 and  q12b and q14 and
        q17b )or (I4 and q5 and  q4b and q3 and  q13b and  q15b and q12
        and q14 and q17 );
 eb <= (I4 and  q5b and q4 and  q3b and  q13b and  q15b and  q12b and
        q14b and q17 )or (I4 and q5 and  q4b and q3 and  q13b and  q15b
        and q12 and q14 and q17 );
end tan_pg387_Tan_1;
-- Memory + Output Variable Equations = 10
-- Total Inverters = 11
-- Max Transitor Est. (CMOS implementation, using only 2-input logic gates)
--    n-channel transistors = 278
--    p-channel transistors = 278
--        total transistors = 556
```

Figure 16 Tan VHDL Synthesized by SASS

5.0 Experimental Results

The temporally and spatially tuned circuits synthesized by SASS were compared to their synchronous counterparts in terms of layout area. The purpose of this comparison was to determine how much better or worse the physical realization of asynchronous circuits were than synchronous circuits.

Another comparison was made between the speed of spatially and temporally tuned asynchronous circuits. Since performance depends on the size of circuit layout, it was suspected that spatially tuned architectures synthesized using One-Hot might be faster than the larger Tan architectures. Although One-Hot equations employ a temporary buffer state between transitions, they could be faster than the Tan equations, which were thought to be faster because they have simultaneously switching memory variables during state transitions. The purpose of this comparison was to determine the effect of layout area on the performance of tuned asynchronous circuits.

5.1 Layout Area Comparison with Synchronous Equivalents

The DAG language description of Figure 7, which was found in: [8], was synthesized into spatially and temporally optimal VHDL architectures by SASS, and layout statistics are found in the second row of Table 1. Additional asynchronous sequential circuit descriptions found in related technical papers were converted into spatially and temporally tuned circuits, and then their synthesized CMOS layouts were compared to each other, and to the layout of a functionally equivalent synchronous circuit. The results in Table 1 show that the spatially optimal asynchronous circuits have layout areas that range between 5% to 46% less than the layout areas of their synchronous counterparts. The temporally tuned asynchronous circuits have layout areas that range between 4% and 61% greater than their synchronous counterparts. These results show that asynchronous circuits are competitive in terms of layout area to functionally equivalent synchronous circuits when using KEYSTONE layout synthesis.

			Asynchronous					Synchronous		
Sequential Function (1)	DAG states	min FT rows (2)	1Hot CMOS trans (3)	1Hot Area (4)	Tan mem state vars	Tan CMOS trans (3)	Tan Area (4)	Mealy state vars (5)	Mealy CMOS trans (3)	Mealy Area (4)
example[14]	11	5	210	193K	9	352	380K	3	366	365K
example[8]	11	5	220	191K	8	320	376K	3	326	353K
example[15]	8	4	176	149K	4	194	177K	2	184	156K
example[9]	14	10	414	573K	19	588	805K	4	530	650K
example[10]	16	12	548	819K	24	810	1482K	4	642	920K
Edge (6)	16	16	1880	4873K	16	1894	5729K	4	454	504K

Notes: (1) the first sequential function is a variant of a traffic light controller FSM, the remainder are various sequential function that were used as examples in the cited references
(2) quantity of rows in minimized flow table is also equivalent to the quantity of next-state memory variables in One-Hot synthesis
(3) transistor count is derived from the KEYSTONE layout tool for CMOS
(4) area is derived from the KEYSTONE layout tool in λ^2 , where $\lambda = 2\mu$
(5) equations were derived from the MEG finite state machine description of the minimized flow table. MISII [3] used to optimized the equations, then a translator was used to generate the guarded VHDL description for layout synthesis by KEYSTONE
(6) 16-bit edge counter

Table 1 Comparison of KEYSTONE Layout Area Statistics

Both the spatially and temporally tuned asynchronous circuits are layout area-competitive because KEYSTONE uses a general purpose layout synthesis approach that does not favor particular design or layout styles. KEYSTONE will recognize gate level structure within a VHDL architecture, then try to recognize hierarchy within the entire description, and then produce a CMOS layout accordingly. Since KEYSTONE does not synthesize pass-transistor logic, the circuits that can be implement with the fewest gates, which are those that are spatially tuned asynchronous circuits, will naturally have the smallest layout area. Once KEYSTONE is able to recognize and synthesize register structures using pass-transistors, the area advantage that spatially optimal asynchronous circuits currently possess will diminish; but it may not be entirely forfeit.

By applying logic optimization techniques to the next-state and output variable equation, both spatially and temporally optimal circuits may be further reduced in layout area. Conventional minimization techniques aimed at reducing the quantity of next-state memory equations might introduce critical race conditions into the circuit. Therefore, the minimizations on a next-state equation must be restricted

to combinational procedures aimed at combining product terms solely within the equation. These techniques are not likely to be very effective on the One-Hot and Tan next-state equations because they are relatively small. However, the output variable equations are particularly susceptible to further minimization because they are currently composed of the sum of minterms derived from the minimized flow table representation. In SASS, the output equation generation method, which is relatively the same for either spatial or temporal optimization, could produce large unoptimized output equations, which would constitute the greatest potential for conventional logic minimization techniques to reduce layout area.

5.2 Performance Comparison Between One-Hot and Tan Circuits

To test the functionality of the synthesized asynchronous circuits, and to compared their performance, an identical sets of test patterns were generated for the next-state memory and output variable signals of some of the functions in Table 1. A linear transistor model simulation was performed using IRSIM. The results in Table 2 show that most of the One-Hot circuits are actually faster than the Tan circuits. This difference in the performance can be attributed to the layout.

Although the Tan equations should be faster because they have STT state assignments, the One-Hot equations are generally better because their layout areas are much smaller. The One-Hot circuits are between 51% and 85% of the area of the Tan circuits, and their corresponding performance ranges from 23% better to 12% worse. Given that other factor such as process parameters, transistor size, and design styles are equivalent, a more compact circuit will generally have better performance because its signal wires are shorter and have lower impedance and capacitance.

The last sequential function in Table 2 had a Tan implementation that was clearly faster by about 12% for both the memory and output variable transitions than its One-Hot counterpart. Incidentally, this function also had the smallest difference between Tan and One-Hot layout area. This result indicates that performance differences between One-Hot and Tan directly depend on the differences in layout area.

Sequential Function (1)(2)	One-Hot Ave. mem var latency	Ave. out var latency	1Hot Area (3)	Tan Ave. mem var latency	Ave. out var latency	Tan Area (3)	% Layout Area Diff(4)	% Speed Diff (5)
example(14)	12.6	3.4	193K	3.0	4.3	380K	-49	-13, -21
example(8)	2.5	3.8	191K	2.9	3.4	376K	-49	-14, +12
example(9)	3.1	3.2	573K	3.9	4.2	805K	-29	-21, -24
example(10)	4.2	4.9	819K	4.5	5.9	1482K	-45	-7, -17
Edge	6.1	11.8	4873K	5.4	10.6	5729K	-15	+12, +11

Notes:
(1) the first sequential function is a variant of a traffic light controller FSM, the remainder are various sequential function that were used as examples in the cited references
(2) The latency time is computed by averaging the propagation delay through the circuit for both type of transitions, for both memory and output signals, and for an identical set of test patterns on both types of circuits. IRSIM was used to perform the simulation
(3) area is derived from the KEYSTONE layout tool in λ^2 , where $\lambda = 2\mu$
(4) the % layout differences area calculated from the perspective of the Tan circuit
(5) the % performance differences are calculated from the perspective of the Tan circuit for both the memory variables and output variables, respectively

Table 2 Comparison of Performance Statistics

6.0 Future Work

Additional work on the subject of asynchronous sequential circuit design can be divided into two topics: *Enhancements to SASS* and *Asynchronous Circuit Characteristics.*

The following enhancements to SASS will change it from a research tool into a useful commercial-type product:

- Implementation of semantic analysis routines to detect sequential functions that cannot be physically realized.
- Implementation of routines to check the functionality of a correct DAG language description, and remove redundant states so as to make the resulting flow table more efficient.
- Implementation of output variable equation optimization using MISII. Currently, the output variable equations are derived by forming the sum of product terms from the reduced flow table.
- Implementation of routines to interlock the product terms of individual output variable equations. By interlocking the product terms of an output variable equation, hazards can be eliminated on the output variable whenever a change in state occurs in the circuit. The following algorithm will interlock an output variable equations, and can be repeated for each output variable:

  1. Minimize each output variable equation, preferably with MISII, so that the equation contains the fewest $tie - set$ [15] terms.
  2. For each output variable equation, check each cut-set term against all the others. If a pair of terms contain a true and complement version of a particular signal or group of signals, then add another term to the equation that is the product of all the signals in the two original cut-set terms, except for the duplicated signal and its complement. For example,

the equation $BC' + A'C$ is inherently hazardous. By adding the product term $A'B$, the equation $B'C + A'C + A'B$ will not contain a hazard when C becomes unasserted. Another equation: $A'B'C' + BD + AD'$, can be made hazard-free by the addition of the following terms: $A'C'D + B'C'D + AB$.

- Detection of condensed flow table that derive One-Hot memory variable equations with inherent race conditions. This topic was discussed in detail in Section 2.3.

- Implementation of a more efficient data structure to hold the primitive flow table, and condensed flow table representations. Currently, the flow tables are implemented as fixed size array-type structures. As the number of inputs, n, increases into the circuit, the width of the table array will grow by 2^n. Ideally, the flow table could be implemented as a set of 2-dimensionally linked structures, which would be more efficient.

With regard to circuit characteristics, a topic for further study is to determine if equally minimal, though unique flow tables for a given DAG description yield circuits with significantly different layout areas. The sequential function found in the Huffman paper seems to indicate that substantial differences exist in the circuit implementations of equally minimal flow tables, as shown in Table 3.

The flow tables that produced each of the four architectures in Table 3 were all different, however, they all had the same minimal number of rows. The first architecture, and the fourth architecture respectively produce the best One-Hot and Tan circuits. Since these architecture have the lowest transistor count in their respective groups, they naturally lead to circuits with the least area. Notably, the second and third One-Hot, and the first, second and third Tan architectures have the same number of transistors, but they all have different layout areas. The most likely explanation for this occurrence is that the KEYSTONE tools are able to optimize some circuits more naturally than others. A further research topic would be to analyze and determine how to synthesize logic that is easier for KEYSTONE use.

Architecture (1)	SASS 1Hot CMOS trans (2)	KEYS 1Hot CMOS trans (3)	1Hot Area (4)	SASS Tan CMOS trans (2)	KEYS Tan CMOS trans (3)	Tan Area (4)
1	320	176	149K	302	176	161K
2	344	190	168K	302	176	167K
3	344	190	178K	302	174	167K
4	368	202	174K	222	122	114K

Notes: (1) the sequential function from which the four architectures were synthesized is found in Huffman [15]
(2) n and p transistor count given by SASS worst case CMOS statistics generator
(3) n and p transistor count given by KEYSTONE layout tool for CMOS
(4) area is derived from the KEYSTONE layout tool in λ^2 , where $\lambda = 2\mu$

Table 3 Statistics for Circuits from Equally Minimal Flow Tables

6.1 Summary of Contributions

The spatially or temporally efficient asynchronous sequential circuits produced by SASS, and physically constructed by KEYSTONE, are viable alternatives to their equivalent synchronous counterparts. The One-Hot and Tan next-state equation generation algorithms will produce equations that are free of critical races, and that are either area efficient or fast. The development of SASS has contributed to the task of logic synthesis for asynchronous sequential functions in the following ways:

- Development of a user-friendly language to describe asynchronous sequential functions
- Automated synthesis of asynchronous sequential logic, that is tuned for either reduced size or more speed, which can then be simulated or fed to layout tools
- Analysis of the robustness of circuits generated from One-Hot and Tan, which led to the discovery of algorithmic flaws
- Showed that asynchronous circuits are layout-area competitive with synchronous counterparts
- Introduced a potential future research topic as to why equivalent architectures result in different layouts

The results of the this development have shown that SASS produced spatially tuned circuits that are often smaller than their synchronous counterparts. Also, the spatially tuned circuits, which are produced using the One-Hot state assignment algorithm, are often faster than the much larger circuits produced by the Tan algorithm, which was thought to be faster because of the simultaneously switching memory variables during state transitions (STT). Further research into the asynchronous sequential logic characteristics that enhance the KEYSTONE layout synthesis was also shown to be feasible. With some minor enhancements, SASS can be converted from a research tool into a commercial-type product for circuit synthesis.

APPENDIX A

APPENDIX A

In the paper by Tracey, an excellent explanation is presented as to the requirements for STT state assignment. Several definitions from Tracey are paraphrased as follows:

Definition 5: a direct transition between two states, S_i and S_j, written $[S_i, S_j]$, occurs when all the next-state memory variables are permitted to change simultaneously.

Definition 6: In a particular column of a flow table, a direct transition $[S_i, S_j]$ races critically with the direct transition $[S_k, S_l]$ if the possibility exists that unequal propagation delays through the circuit might cause the two transitions to momentarily share a common intermediate state.

To prevent critical races from occurring, a Theorem is presented that defines how the state assignment should be derived:

Theorem 1: A state assignment alloting one memory state variable per row can facilitate direct state to state transitions without critical races, if, and only if for every transition $[S_i, S_j]$,

a) if $[S_m, S_n]$ is another transition in the same column, then at least one memory variable partitions the pair $\{S_j, S_j\}$ and the pair $\{S_m, S_n\}$ into separate blocks

b) if S_k is a stable state in the same column, then at least one memory variable partitions

the pair $\{S_i, S_j\}$ and S_k into separate blocks

c) for $i \neq j$, there is at least one memory variable that partitions S_i and S_j into separate blocks.

To prove this theorem, consider a flow table where there is a column with the following transitions, $[S_i, S_j]$ and $[S_m, S_n]$, and the rows of the flow table can be partitioned as follows:

$$\pi_1 = \{S_i, S_j, S_m, S_n\}$$

$$\pi_2 = \{S_i; S_j, S_m, S_n\}$$

$$\pi_3 = \{S_i, S_m, S_n; S_j\}$$

$$\pi_4 = \{S_i, S_j, S_n; S_m\}$$

$$\pi_5 = \{S_i, S_j, S_m; S_n\}$$

$$\pi_6 = \{S_i, S_j; S_m, S_n\}$$

$$\pi_7 = \{S_i, S_m; S_n, S_j\}$$

$$\pi_8 = \{S_i, S_n; S_j, S_m\}$$

This set of partitions leads to the following state assignment:

	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	Y_8
S_i	0	0	0	0	0	0	0	0
S_j	0	1	1	0	0	0	1	1
S_m	0	1	0	1	0	1	0	1
S_n	0	1	0	0	1	1	1	0

In the state to state transition of $[S_i, S_j]$ the memory variables Y_1 to Y_8 could assume any of the following intermediate states: $0 - -000 - -$. During the $[S_m, S_n]$ transition, the memory variables could assume any of the following intermediate

states: $010--1--$, as a result of unequal propagation delays through the circuit. The memory variable Y_6, which is a requirement from part a) of Theorem 1, insures that none of the intermediate states will be common for the two transitions. If Y_6 were removed, $[S_i, S_j]$ would have the intermediate states: $0--00--$, and $[S_m, S_n]$ would have the intermediate states $010----$. Thus, a critical race condition might result because the two transitions would share $01000--$ common intermediate states.

APPENDIX B

APPENDIX B

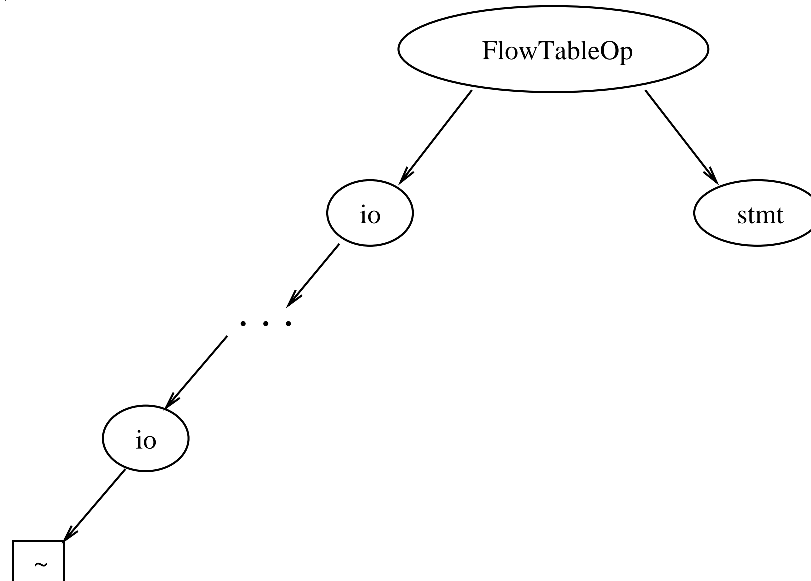
The LEX and YACC compiler tools were used to generate the front-end lexical and syntactical analysis for the DAG language. In Table 4 the token definitions are given for the keywords and symbols of the language. These definitions are understood by LEX. Figures 17 and 18 shows the syntactical structure of the language that was defined for YACC. A DAG language parser was generated from these descriptions. The parser will generate a syntax tree that semantic analysis routines, symbol table generation, and primitive flow table generation routines will use to understand the DAG language description of an asynchronous sequential function.

DAG Language Token	Symbolic Name in LEX
;	SEMI
,	COMMA
(carrot)	UP
(backslash)	DOWN
integer	ICONST
identifier	ID
flowtable	FLOWTABLE
input	INPUTSIG
output	OUTPUTSIG
endtable	ENDTABLE

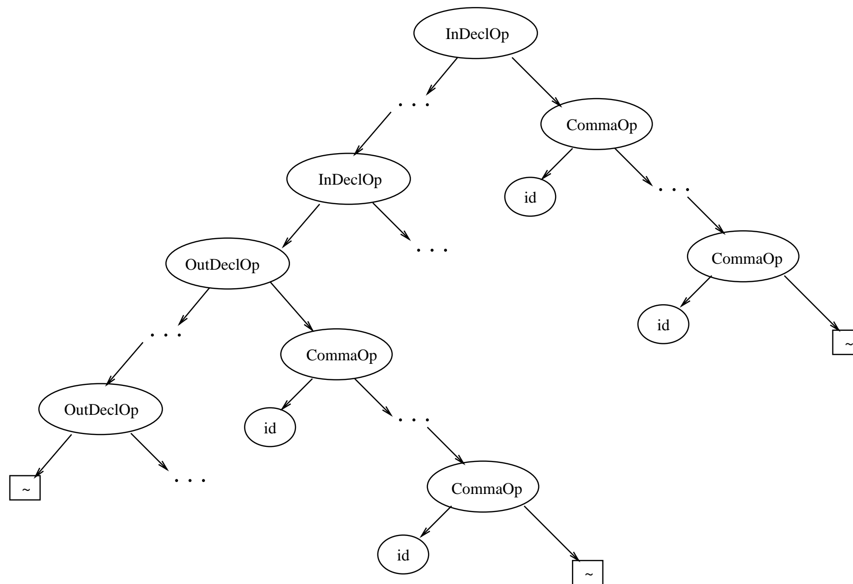
Table 4 Symbol Table for LEX Token Definition

The syntax tree descriptions are given as follows:

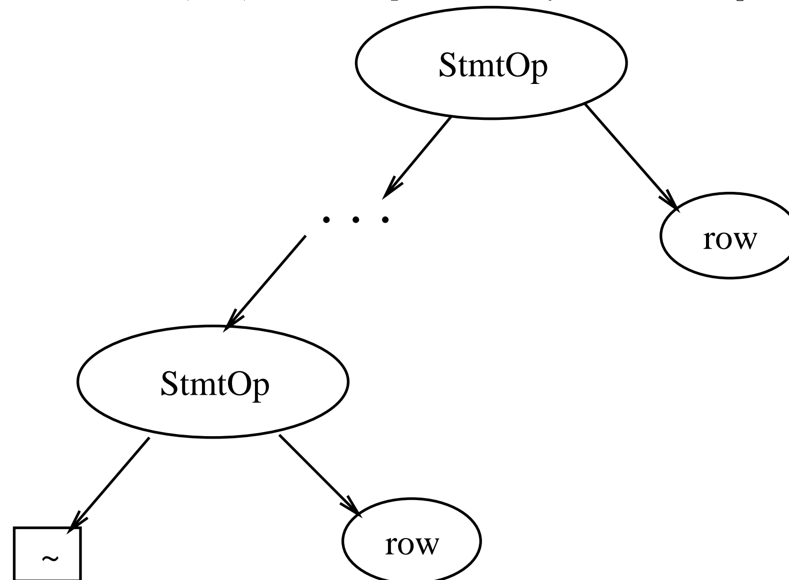
1. A flow table is represented by the following subtree, where *io* is an *InDeclOp* or an *OutDeclOp*, and stmt is a list of rows in the flow table:



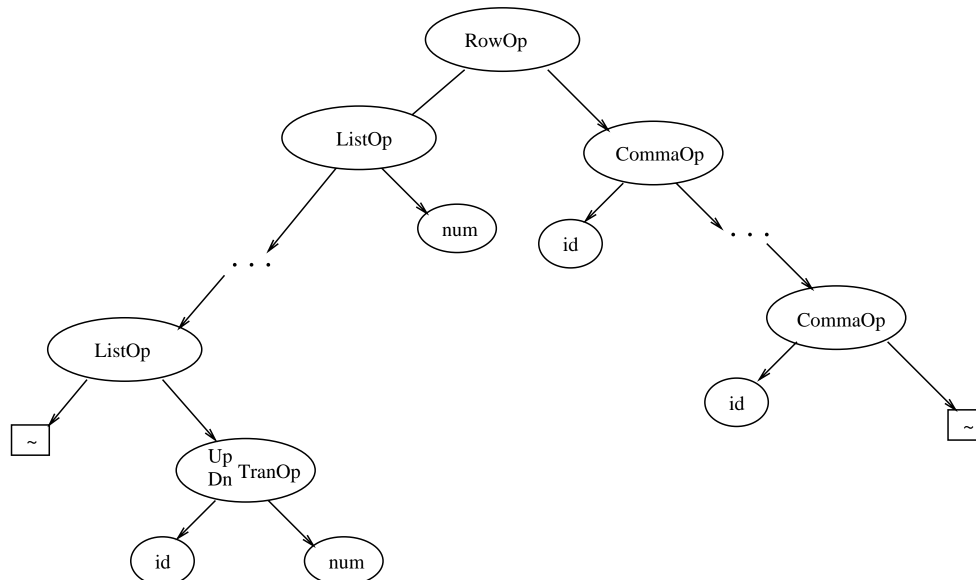
2. An io declaration is of the form: T: $id_1, id_2, \cdots, id_n$, where id is an identifier, and T is the type (either *input* or *output*). The following subtree represents I/O descriptions:



3. A sequence of rows: $row_1, \cdots, row_n$ is represented by the following subtree:



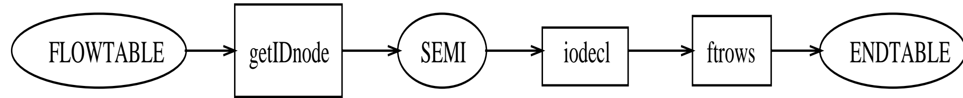
4. A row has a list of transitions, where each transition is caused by a type of change on an input signal, thus resulting in a transition to a state. Also, a list of asserted output variables are included:



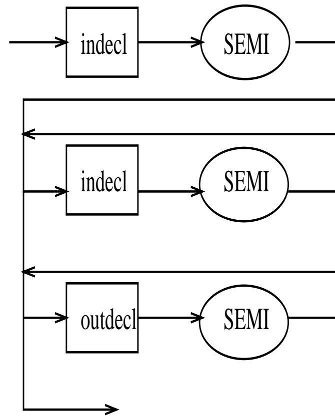
An example of a syntax tree is shown in Figure 19, for the following DAG language description:

```
/* Primitive flow table description for an
   asynchronous direction finder */

   flowtable example_dag;
    input a, b;
    output eb, wb;

     1, a^2, b^3;
     2, b^3, a\1, eb;
     3, a\2, b\1, wb;
   endtable

/* done */
```

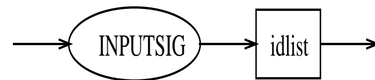
DAG FlowTable



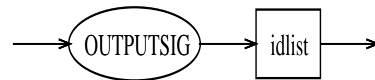
iodecl (I/0 Declarations)



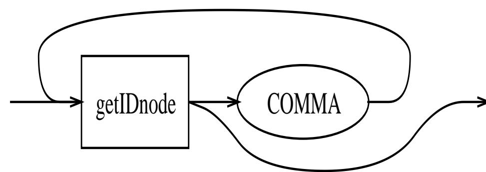
indecl (Input Declaration)



outdecl (Output Declaration)



idlist (List of IDs)



ftrows (flow table rows)

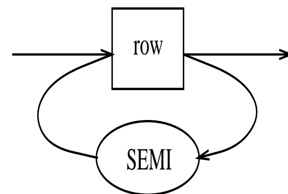
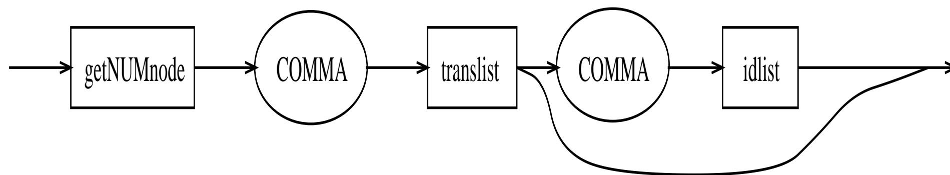
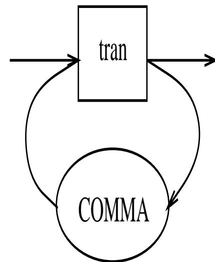


Figure 17 Syntax Diagrams for DAG Language

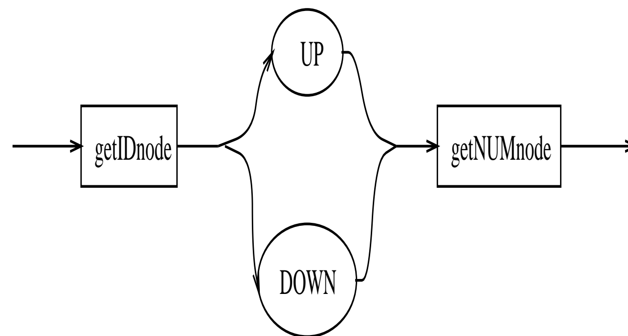
row (Row Statement)



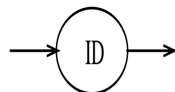
tranlist (Transition List)



tran (Transition)



getIDnode (identifier)



getNUMnode (integer constant)

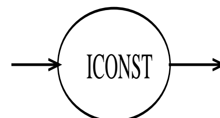


Figure 18 Syntax Diagrams for DAG Language (continued)

```
        +-[DUMMYnode]
      +-[CommaOp]
      | +-[IDNode,0,"wb"]
    +-[RowOp]
    | | +-[NUMNode,3]
    | +-[ListOp]
    |   |   +-[NUMNode,1]
    |   | +-[DnTranOp]
    |   | | +-[IDNode,0,"b"]
    |   +-[ListOp]
    |     |   +-[NUMNode,2]
    |     | +-[DnTranOp]
    |     | | +-[IDNode,0,"a"]
    |     +-[ListOp]
    |       +-[DUMMYnode]
  +-[StmtOp]
  | |     +-[DUMMYnode]
  | |   +-[CommaOp]
  | |   | +-[IDNode,0,"eb"]
  | | +-[RowOp]
  | | | | +-[NUMNode,2]
  | | | +-[ListOp]
  | | |   |   +-[NUMNode,1]
  | | |   | +-[DnTranOp]
  | | |   | | +-[IDNode,0,"a"]
  | | |   +-[ListOp]
  | | |     |   +-[NUMNode,3]
  | | |     | +-[UpTranOp]
  | | |     | | +-[IDNode,0,"b"]
  | | |     +-[ListOp]
  | | |       +-[DUMMYnode]
  | +-[StmtOp]
  |   |   +-[DUMMYnode]
  |   | +-[RowOp]
  |   | | | +-[NUMNode,1]
  |   | | +-[ListOp]
  |   | |   |   +-[NUMNode,3]
  |   | |   | +-[UpTranOp]
  |   | |   | | +-[IDNode,0,"b"]
  |   | |   +-[ListOp]
  |   | |     |   +-[NUMNode,2]
  |   | |     | +-[UpTranOp]
  |   | |     | | +-[IDNode,0,"a"]
  |   | |     +-[ListOp]
  |   | |       +-[DUMMYnode]
  |   +-[StmtOp]
  |     +-[DUMMYnode]
+-[FlowTableOp]
  |     +-[DUMMYnode]
  |   +-[CommaOp]
  |   | +-[IDNode,0,"b"]
  | +-[CommaOp]
  | | +-[IDNode,0,"a"]
  +-[InDeclOp]
    |     +-[DUMMYnode]
    |   +-[CommaOp]
    |   | +-[IDNode,0,"wb"]
    | +-[CommaOp]
    | | +-[IDNode,0,"eb"]
    +-[OutDeclOp]
      +-[DUMMYnode]
```

Figure 19 Syntax Tree Example

APPENDIX C

APPENDIX C

- Analysis of One-Hot Equations Generated By Berkeley MUSTANG. The One-Hot memory variable equations generated by MUSTANG are not sufficient to implement an asynchronous sequential function. The MUSTANG equations have the following shortcomings:

  1. they contain only the next-state transition terms that are needed to implement a synchronous state machine. Only the transition terms are necessary in a synchronous sequential circuit because the register elements hold the current state of the machine. The transition terms indicate the next machine state after a change in the clock. For an asynchronous circuit, a holding term must be derived that will force the memory variable to remain in a particular state until a change in the input signals induces a transition to another state. The holding term for each particular memory variable must be composed of that variable and the complemented product of all the other memory variables to which transitions are made, as found in the flow table (see Section 2.1.2).
  2. they do not have initialization logic incorporated into the equation. In a synchronous state machine, the initialization logic can be incorporated into the register elements, and does not need to be included in the circuits description. Since asynchronous sequential circuits do not use conventional register elements with built-in initialization logic, then the initialization logic must be customized for different sequential functions.

- The synchronous sequential design using the Berkeley tools proceeded as follows:

  1. Formulate MEG description for the sequential function as in Figure 20.
  2. Compile the MEG file with the following options:

```
% meg -i car_director.meg <CR>
```

MEG will produce a file called meg.imp.

3. Modify the file meg.imp as shown in Figure 21.
4. Generate One-Hot assignment using MUSTANG:

```
% mustang -l meg.imp > car_director.pla <CR>
```

5. Modify .pla file as shown in Figure 22.
6. Optimize using misII:

```
% misII <CR>

  misII> read_pla -s car_director.pla  <CR>
  misII> simplify -d  <CR>
  misII> write_eqn car_director.eqn  <CR>
  misII> exit <CR>
```

7. The .eqn file contains the One-Hot transition terms for each equation, however, none of the equations contain a holding term as shown in Figure 23.

- SASS will produce the a set of VHDL equations that can be translated into the following eqn format for comparison, as shown in Figure 24. These equations are slightly more complex because of the holding term.

```
- Example car_director

INPUTS: a b rst;
OUTPUTS: eb wb;

ONE       :  CASE (a b rst)
               0 0 0 => FIVE;
               1 0 0 => ONE;
               0 1 0 => ONE(eb);
               1 1 0 => ONE(wb);
               ? ? 1 => FIVE;
             ENDCASE => FIVE;

TWO       :  CASE (a b rst)
               0 0 0 => TWO;
               1 0 0 => FOUR;
               0 1 0 => TWO;
               1 1 0 => ONE;
               ? ? 1 => FIVE;
             ENDCASE => FIVE;

THREE     :  CASE (a b rst)
               0 0 0 => THREE;
               1 0 0 => THREE;
               0 1 0 => ONE;
               1 1 0 => FOUR;
               ? ? 1 => FIVE;
             ENDCASE => FIVE;

FOUR      :  CASE (a b rst)
               0 0 0 => FIVE;
               1 0 0 => FOUR(wb);
               0 1 0 => FOUR;
               1 1 0 => FOUR(eb);
               ? ? 1 => FIVE;
             ENDCASE => FIVE;

FIVE      :  CASE (a b rst)
               0 0 0 => FIVE;
               1 0 0 => THREE;
               0 1 0 => TWO;
               1 1 0 => FIVE;
               ? ? 1 => FIVE;
             ENDCASE => FIVE;
```

Figure 20 MEG File Input

```
# .ilb a b rst StBit0* StBit1* StBit2*   -- DELETE this line, mustang ignores sy
mbolic I/O names
# .ob StBit2* StBit1* StBit0* eb wb      -- DELETE this line
.i 3      # ADD line
.o 2      # ADD line
.s 5      # ADD line
000  ONE                 FIVE                00
100  ONE                 ONE                 00
010  ONE                 ONE                 10
110  ONE                 ONE                 01
001  ONE                 FIVE                00
101  ONE                 FIVE                00
011  ONE                 FIVE                00
111  ONE                 FIVE                00
000  TWO                 TWO                 00
100  TWO                 FOUR                00
010  TWO                 TWO                 00
110  TWO                 ONE                 00
001  TWO                 FIVE                00
101  TWO                 FIVE                00
011  TWO                 FIVE                00
111  TWO                 FIVE                00
000  THREE               THREE               00
100  THREE               THREE               00
010  THREE               ONE                 00
110  THREE               FOUR                00
001  THREE               FIVE                00
101  THREE               FIVE                00
011  THREE               FIVE                00
111  THREE               FIVE                00
000  FOUR                FIVE                00
100  FOUR                FOUR                01
010  FOUR                FOUR                00
110  FOUR                FOUR                10
001  FOUR                FIVE                00
101  FOUR                FIVE                00
011  FOUR                FIVE                00
111  FOUR                FIVE                00
000  FIVE                FIVE                00
100  FIVE                THREE               00
010  FIVE                TWO                 00
110  FIVE                FIVE                00
001  FIVE                FIVE                00
101  FIVE                FIVE                00
011  FIVE                FIVE                00
111  FIVE                FIVE                00
```

Figure 21 MUSTANG File Input

```
.i 8
.o 7
.ilb a b rst InStBit0 InStBit1 InStBit2 InStBit3 InStBit4    # ADD this line to
assign symbolic input names
.ob OutStBit0 OutStBit1 OutStBit2 OutStBit3 OutStBit4 eb wb  # ADD this line to
assign symbolic output names
000 1---- 01000 00
100 1---- 10000 00
010 1---- 10000 10
110 1---- 10000 01
001 1---- 01000 00
101 1---- 01000 00
011 1---- 01000 00
111 1---- 01000 00
000 --1-- 00100 00
100 --1-- 00010 00
010 --1-- 00100 00
110 --1-- 10000 00
001 --1-- 01000 00
101 --1-- 01000 00
011 --1-- 01000 00
111 --1-- 01000 00
000 ----1 00001 00
100 ----1 00001 00
010 ----1 10000 00
110 ----1 00010 00
001 ----1 01000 00
101 ----1 01000 00
011 ----1 01000 00
111 ----1 01000 00
000 ---1- 01000 00
100 ---1- 00010 01
010 ---1- 00010 00
110 ---1- 00010 10
001 ---1- 01000 00
101 ---1- 01000 00
011 ---1- 01000 00
111 ---1- 01000 00
000 -1--- 01000 00
100 -1--- 00001 00
010 -1--- 00100 00
110 -1--- 01000 00
001 -1--- 01000 00
101 -1--- 01000 00
011 -1--- 01000 00
111 -1--- 01000 00
```

Figure 22 MISII File Input

```
INORDER = a b rst InStBit0 InStBit1 InStBit2 InStBit3 InStBit4;
OUTORDER = OutStBit0 OutStBit1 OutStBit2 OutStBit3 OutStBit4 eb wb;

OutStBit0 = !a*b*!rst*InStBit4 + a*b*!rst*InStBit2 + b*!rst*InStBit0 + a*!rst*In
StBit0;

OutStBit1 = !a*!b*InStBit3 + !a*!b*InStBit1 + a*b*InStBit1 + !a*!b*InStBit0 +
            rst*InStBit4 + rst*InStBit3 + rst*InStBit2 + rst*InStBit1 + rst*InSt
Bit0;

OutStBit2 = !a*b*!rst*InStBit1 + !a*!rst*InStBit2;

OutStBit3 = a*b*!rst*InStBit4 + a*!b*!rst*InStBit2 + b*!rst*InStBit3 + a*!rst*In
StBit3;

OutStBit4 = a*!b*!rst*InStBit1 + !b*!rst*InStBit4;

eb = a*b*!rst*InStBit3 + !a*b*!rst*InStBit0;
wb = a*!b*!rst*InStBit3 + a*b*!rst*InStBit0;
```

Figure 23 EQN File Produced by MISII

```
INORDER = a b reset StBit0 StBit1 StBit2 StBit3 StBit4;
OUTORDER = StBit0 StBit1 StBit2 StBit3 StBit4 eb wb;

StBit4 = (( !a *  !b * StBit3 ) + ( !a *  !b * StBit0 ) +
          ( !StBit1 *  !StBit2 * StBit4 )) +  reset;

StBit3 = ((a * b * StBit2 ) + (a *  !b * StBit1 ) +
          ( !StBit4 * StBit3 ))* !reset;

StBit2 = ((a *  !b * StBit4 ) + ( !StBit3*  !StBit0 * StBit2 ))* !reset;

StBit1 = (( !a * b * StBit4 ) + ( !StBit0 *  !StBit3* StBit1 ))* !reset;

StBit0 = ((a * b * StBit1 ) + ( !a * b * StBit2 ) +
          ( !StBit4 * StBit0 ))* !reset;

wb = (a *  !b *  !StBit4 * StBit3 *  !StBit2 *  !StBit1 *  !StBit0 ) +
     (a * b *  !StBit4 *  !StBit3*  !StBit2 *  !StBit1 * StBit0 );
eb = (a * b *  !StBit4 * StBit3 *  !StBit2 *  !StBit1 *  !StBit0 ) +
     ( !a * b *  !StBit4 *  !StBit3*  !StBit2 *  !StBit1 * StBit0 );
```

Figure 24 Corresponding EQN File Produced by SASS

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Roth, Charles. *Fundamental of Logic Design, Second Edition.* West, 1979.

2. Huffman, D. A. "The Synthesis of Sequential Switching Circuits". *Journal of the Franklin Institute*, Vol. 257 No. 3/4, pp. 161–190,275–303, March/April 1957.

3. Devadas, S., Ma, H.-K., Newton, A. R., and Sangiovanni-Vincentelli, A. "MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations". *IEEE Transactions on Computer-Aided Design*, Vol. 7 No. 12, pp. 1290–1299, December 1988.

4. Brodersen, R. W., Newton, A. R., Sangiovanni-Vincentelli, A., Rabaey, J. M., Brayton, R. K., Lee, E. A., and Messerschmitt, D. G. "Design and Prototyping of Hard Real Time Systems". Technical report, Department of Electrical Engineering and Computer Science, and Electronics Research Laboratory, University of California, Berkeley, May-September 1990.

5. IEEE, 345 East 47th Street, New York, NY. *IEEE Standard VHDL Language Reference Manual*, 1988.

6. Levitan, S.P., Martello, A.R., Owens, R.M., and Irwin, M.J. *Using VHDL as a Language for Synthesis of CMOS VLSI Circuits.* Elsevier, Amsterdam, 1989, Washington D.C., June 19-21 1989.

7. Whitaker, S. R. and Maki, G. K. "Pass-Transistor Asynchronous Seqential Circuits". *IEEE Journal of Solid-State Circuits*, Vol. 24 No. 1, pp. 71–78, February 1989.

8. Tan, C.-J. "State Assignments for Asynchronous Seqential Machines". *IEEE Transactions on Computers*, Vol. C-20 No. 4, pp. 382–391, April 1971.

9. Lui, C. N. "A State Variable Assignment Method for Asynchronous Sequential Switching Circuits". *Journal of the ACM*, 10, pp. 209–216, April 1963.

10. Tracey, J. "Internal State Assignment for Asynchronous Sequential Machines". *IEEE Transactions on Electronic Computers*, Vol. EC-15 No. 4, pp. 551–560, August 1966.

11. Meng, T.H.-Y., Brodersen, R. W., and Messerschmitt, D. G. "Automatic Synthesis of Asynchronous Circuits From High-Level Specifications". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 8 No. 11, pp. 1185–1205, November 1989.

12. Fiduccia, C.M. and Mattheyses, R.M. "A Linear-Time Heuristic for Improving Network Partitions". In *Proceedings of the 19th Design Automation Conference*, pp. 175–181, Las Vegas, June 1982.

13. Hachtel, G. D., Newton, A. R., and Sangiovanni-Vincentelli, A. "An Algorithm for Optimal PLA Folding". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-1 No. 2, pp. 63–76, April 1982.

14. Mead, Carver and Conway, Lynn. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

15. Huffman, D. A. "The Design and Use of Hazard-Free Switching Networks". *Journal of the ACM*, 4, pp. 47–62, January 1957.