Optimizations of Cisco's Embedded Logic Analyzer Module

by

Fangjin Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2009

Abstract

Cisco's *embedded logic analyzer module* (ELAM) is a debugging device used for many of Cisco's *application specific integrated chips* (ASICs). The ELAM is used to capture data of interest to the user and stored for analysis purposes. The user enters a trigger expression containing data fields of interest in the form of a logical equation. The data fields associated with the trigger expression are stored in a set of *Match and Mask* (MM) registers. Incoming data packets are matched against these registers, and if the user-specified data pattern is detected, the ELAM triggers and begins a countdown sequence to stop data capture. The current ELAM implementation is restricted in the form of trigger expressions that are allowed and in the allocation of resources. Currently, data fields in the trigger expression can only be logically ANDed together, Match and Mask registers are inefficiently utilized, and a static state machine exists in the ELAM trigger logic. To optimize the usage of the ELAM, a trigger expression is first treated as a Boolean expression so that minimization algorithms can be run. Next, the data stored in the Match and Mask registers is analyzed for redundancies. Finally, a dynamic state machine is programmed with a distinct set of states generated from the trigger expression. This set of states is further minimized. A feasibility study is done to analyze the validity of the results.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Gordon Agnew, for his valuable guidance, encouragement and support throughout my graduate studies. It is an honour to be chosen for the Cisco Graduate Internship Program. I would also like to give a special thanks to my manager at Cisco, Matthias Loeser, and my mentor, Sifang Li, for their invaluable assistance during my internship there.

Several individuals helped me greatly throughout my graduate studies and my time at Cisco. I would like to thank my thesis readers Dr. Ian Munro and Dr. Paul Ward for their valuable input and suggestions. I also want to thank Liang Yuan and Yangqi Li in helping me prepare this thesis. Lastly, I would like to express my sincere gratitude to the administrative support staff, Wendy Boles and Karen Schooley.

Dedication

This thesis is dedicated to my parents.

Table of Contents

List of Figures

List of Tables

Chapter 1
Background

1.1 Introduction to ELAM

The Cisco Catalyst 6500 is a module chassis switch with a range of integrated services modules, including multi-gigabit network security, content switching, telephony, and network analysis modules. Mainly targeted towards enterprises and service providers, the Cisco Catalyst 6500 is employed for deep packet inspection, security, application awareness and manageability [1]. The main components of the 6500 include chassis, power supplies, supervisor cards, line cards, and service modules [3]. The 6500 Series, shown in Figure 1, uses a common set of modules and OS software across all Cisco chassis. The devices incorporate 11 *application-specific integrated circuits* (ASICs) [6]. Due to the wide ranging functionality of the various ASICs, Cisco developed an *embedded logic analyzer module* (ELAM) that is capable of capturing packet data and assisting developers with debugging faults across ASICs [2].

Figure 1. Cisco Catalyst 6500 Series [1]

Commercial logic analyzers have been crucial debug and diagnostic tools for years. However, as board density increases, it is becoming progressively more difficult to find space for logic analyzer connectors. Further, as clock frequencies increase and new inter-chip protocols (such as double data rate random access memory) are used, commercial logic analyzers are having trouble keeping pace [2].

One solution to this problem is to embed logic analysis functions into ASICs. The benefits of this approach include: diagnostic hardware keeps pace with the system clock frequencies, new inter-chip protocols can be instrumented, no PCB real-estate is used for diagnostic hardware, internal signals can also be instrumented, and configuring the diagnostic hardware is a quick and easy process – there are no cables to hook up [2].

The ELAM is a synthesizable Verilog module which is instantiated at one or more places in an ASIC. It requires an external slave for CPU control. A high level overview of the ELAM is shown in Figure 2 (shown below).

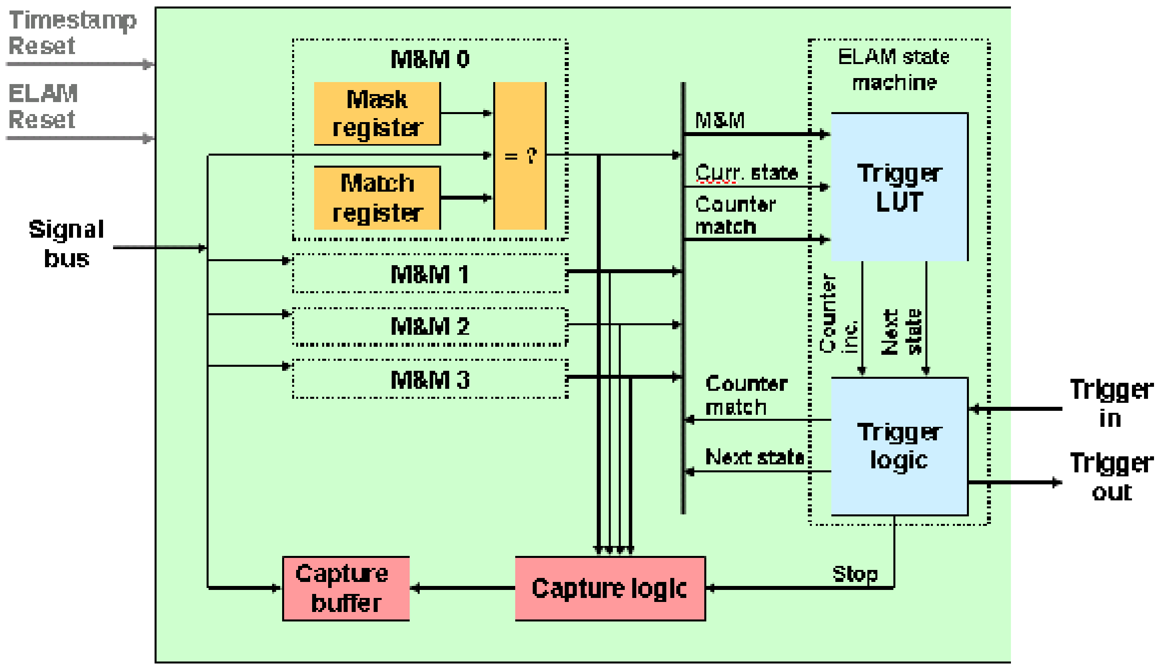

Figure 2. ELAM block diagram [2]

Each of the components shown in Figure 2 will be discussed in more detail in later sections. For ELAM operation, the user selects which signals to capture and routes them into the ELAM instance. The

ELAM can be used for a variety of purposes and as such, the module is parameterized. The user may customize certain aspects of the ELAM through parameters specified at each instantiation. The most important parameters that can be configured are shown in Table 1 below.

Table 1. Important ELAM parameters to capture [4]

Number of bits to capture per clock cycle
Number of bits in the input data bus
Depth of the capture buffer
Width of the CPU address and data buses
Width of the timestamp counter
Number of mask-and-match registers in the trigger logic
Number of state bits in the trigger sequencer

1.2 ELAM Functionality

1.2.1 Trigger Logic

The ELAM captures data for future analysis by utilizing several key components. The first of these components is a powerful and flexible trigger mechanism. The ELAM can detect when a series of pre-defined conditions have been met, and if a user-specified data pattern has been seen. If so, the ELAM will trigger and begin a countdown sequence which quantifies how much data will be captured. The trigger logic consists of a 16-state finite state machine. Transitions between states are a function of the current state, a user-programmable counter, and a trigger statement consisting of one or more trigger fields in product form indicating the data fields of interest to the user. For example, a typical trigger statement entered by the user is `VLAN == 0x3ee`. In this case, all incoming packets will have their VLAN fields inspected to see if the data value in the packet matches the user specified value. If so, the ELAM will enter a trigger state. The user in this example is looking for a valid virtual local area network [4]. This particular example is a common trigger statement.

Besides the current state and the user-programmable counter comparison result, up to 8 *Match and Mask* (MM) register results drive the address inputs to the trigger look up table. The Match and Mask registers are used to store the various fields that compose a trigger statement for data comparison purposes. The trigger state machine uses this look up table to determine the next state, so next-state equations may be arbitrarily complex. Each of these Match and Mask registers have access to all data bits driven into the ELAM, and is the full width of the input bus. The Match and Mask registers are programmed and instantiated before ELAM execution, and cannot be reprogrammed once the ELAM has started its operations. Each bit in the Mask register determines if the corresponding incoming data bit will be compared against the respective bit stored in the Match register. If a bit is set to 1 in the Mask register, the corresponding data bit will be evaluated against the bit in the Match register. For each Match and Mask register pair, the result of all individual bit comparisons are ANDed together - if all are true then the output of that Mask-and-Match comparator is 1, otherwise the result is 0. Unused registers can be disabled by setting all mask bits to 0, and the output of the comparator will be forced to equal 1. The Match and Mask registers are programmed before incoming data is received. The user specified values are entered into the ELAM, where they are processed and stored in a set of virtual Match and Mask registers in software. The ELAM then runs through a sequence of initialization code before programming the Match and Mask registers in hardware. After the registers are programmed in hardware, the ELAM is then ready to receive incoming data.

The ELAM trigger logic also includes a 16-bit user-programmable counter. The counter is cleared when the state machine enters a specified state. The value of the User Counter is compared with a user-programmable 16-bit register. The result of this comparison is an input to the state machine. This counter will stick at 16’hFFFF when it reaches that value, and can be read from CPU [4]. The specifics concerning usage of the counter are beyond the scope of this thesis.

The trigger state machine next-state logic is implemented with a synchronous *random access memory* (RAM) look-up table which runs at the same frequency as the data being captured. The memory used must have exactly one cycle read latency. For the current implementation of the ELAM, relevant user data only occurs across 4 cycles. The organization of data fields in each cycle is distinct and the Match and Mask registers are programmed in advance to prepare for data in a certain cycle. The inputs to the look-up table are the user counter comparison result, the current trigger sequencer state, and the results of the Match and Mask register comparisons. The outputs of the *look-up table* (LUT) are flags to increment the user counter and the next state. The width of the LUT is $2*n+4$, where n is the number of state bits. The depth of the LUT is $2n+m$, where n is the number of state bits and m is the number of Mask and Match comparators.

When the trigger sequencer jumps into the ‘trigger’ state, a certain sequence of actions is run. One cycle after the trigger state occurs, an internal counter is loaded with the number of remaining samples to capture and a countdown is initiated. The number of remaining samples to capture is pre-defined by the user. After this counter has reached zero, capturing is complete and a flag bit is set to notify the user.

To obtain a better understanding of the trigger logic, refer to Figure 3, which summarizes the aforementioned concepts. In Figure 3, the user is looking for a data pattern that occurs across 3 cycles. The first data packet arrives in cycle 0, the next in cycle 1 and the final data packet arrives in cycle 2.

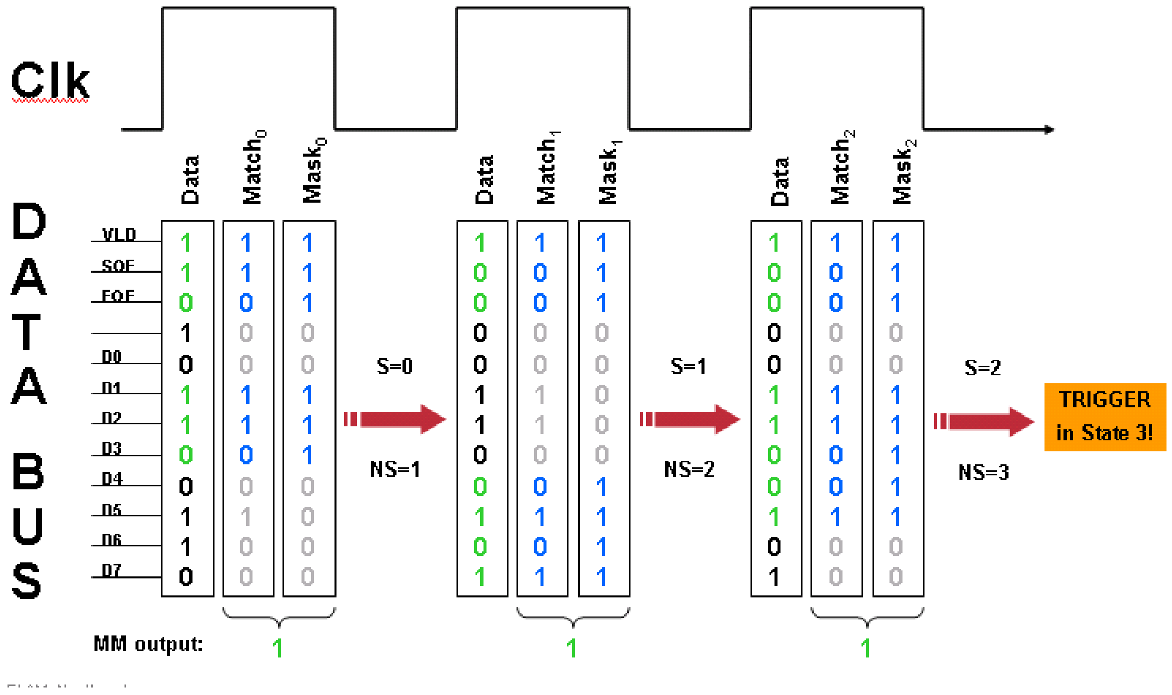

Figure 3. Sample data capture using Match and Mask registers [2]

In each state of the trigger sequencer, different Match and Mask registers are programmed and utilized in validating the incoming data. In Figure 3, the first MM register is programmed to look for a certain pattern. Those bits highlighted in blue indicate the bits of interest, that is, those bits where the Mask register values are set to 1. The bits of interest in the incoming data are highlighted with green font. When the arriving data matches the pattern stored in the Match register, a state transition will occur to progress to state 1. In state 1, the next MM register will be used to compare against incoming data and if this data matches, a transition occurs to state 2. In state 2, the next MM register is utilized and if incoming data is matched, the trigger flag will be set.

1.2.2 Data Capture Logic

The width and depth of the capture buffer are specified by the user for each ELAM instance, and the capture buffer memory is implemented externally to the ELAM. Data is sampled synchronously and stored into the capture buffer along with a timestamp. When the user arms the ELAM, i.e., the user sets a flag indicating the ELAM is primed for data capture, data capture begins. On each clock edge, a word of data is stored at the capture buffer location. This continues until the ELAM is triggered and the Trigger Position counter expires. Once this happens, a done bit is set [4].

1.3 Limitations of Existing Functionality

The basic blocks involved in ELAM functionality are shown in Figure 4 below. The user enters a trigger statement which is programmed into a set of Match and Mask registers. The incoming data is sent to be compared with these registers and comparison results are generated. These results are fed into an incremental state machine which generates a series of states specifying when a trigger state should be reached. When the trigger state is reached, the ELAM completes data capture and ends execution.

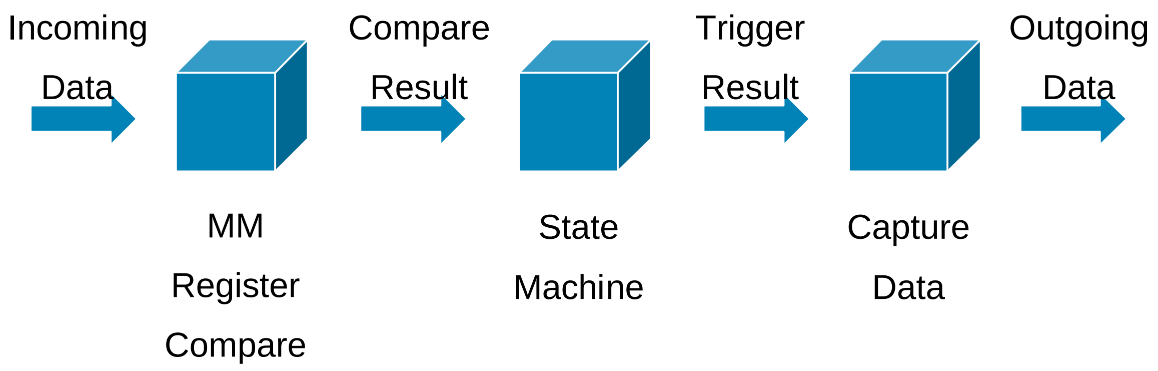

Figure 4. The basic blocks in ELAM

In the existing ELAM architecture, only certain trigger statements are supported. In the event that the user wants to trigger on multiple data fields, the user must first manually enter the trigger values for his desired data fields. Each data field is looked up according to its cycle in a pre-defined database table. Data values in the same cycle are grouped together and Match and Mask registers are programmed to only store the data values for a certain cycle. For example, Match and Mask register 0 would be used to process packets which come in cycle 0. Match and Mask registers are programmed to mimic the incoming data, thus each bit positions in a register are the same as those in an incoming data packet. The individual bit comparisons in a MM register are ANDed together to produce a single bit indicating if all data matched or not. Given this particular method of storing fields, it is quite obvious to see that the only logical expressions that are permissible between different trigger fields are the AND and NAND operations. Since each Match and Mask register generates only a single bit at its output, it is logically impossible to store other forms of Boolean expressions together in one register. Thus, in the current implementation of the ELAM, each MM register by itself can only be used for expressions in the following form:

`field_1` AND `field _2` AND … AND `field _n`…

Now, in the case that `field_n` and `field_n+m` are in adjacent cycles, the state machine is set up so that it first waits for `field_n` to occur before transitioning to the next state. It then waits for `field_n+m` and if it does see `field_n+m` in the next cycle, it will return to the original state and wait for `field_1` again.

Since the ELAM is used primarily as a device for testing and debugging, it is necessary that more complex trigger statements are able to be supported. However, to be able to support more complex trigger equations, several changes are required.

1.4 Proposed Solution

The basic forms of the expressions that need to be supported can be summarized as follows:

1. `field`
2. `!field`
3. `field_1` AND `field_2`

4. `field_1` OR `field_2`

Along with the four basic expressions listed above, the ELAM must be able to support variations and combinations of these fundamental cases, thus providing support for a dynamic range of possible trigger expressions. The current ELAM implementation only supports terms which are logically ANDed together, a small subset of the total range of expressions that are possible. To better understand the limitations of the current ELAM, consider the following example:

Let's assume an ELAM implementation with 4 MM registers and 3 state bits. Only expressions of type 1 from above are used. We further assume the counter is not used and the counter comparison result is always 0. The initial state of the SM is 0.

The programming of the MM registers would be straight forward: each MM register holds the match criteria for the data in one clock cycle. Three MM registers are used in this example. Table 2 below illustrates the state information for this example.

Table 2. State table for example ELAM implementation

	tc	S2	S1	S0	MM3	MM2	MM1	MM0	IncCnt	NS2	NS1	NS0	
1.	0	0	0	0	X	X	X	0	0	0	0	0	
2.	0	0	0	0	X	X	X	1	0	0	0	1	
3.	0	0	0	1	X	X	0	X	0	0	0	0	
4.	0	0	0	1	X	X	1	X	0	0	1	0	
5.	0	0	1	0	X	0	X	X	0	0	0	0	
6.	0	0	1	0	X	1	X	X	0	0	1	1	
7.	0	0	1	0	X	X	X	X	0	0	1	1	Trigger!

This state machine of this example is represented by Figure 5, shown on the next page:

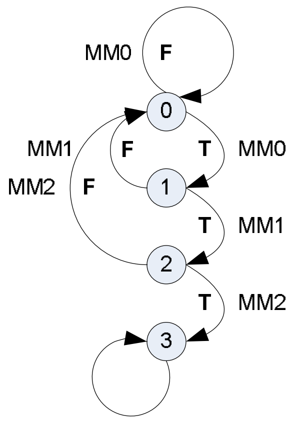

Figure 5. Example state machine

Hence, in this example, the state machine increments the state one by one, depending on the results of MM0-2. The current ELAM is designed in such a fashion. To better understand the restrictions of the current ELAM, consider the following problem:

Let's use the same ELAM parameters as the previous example and also specify the MM width as 8 bits. In each clock cycle the following 3 fields are presented to the ELAM:

{`Seq-#` [7:5], `Type` [4:3] `Cmd` [2:0]}

The goal is to trigger once all of the following conditions are met:

1. Cycle *n*: `Seq-#` == 0 && `Type` == 1
2. Cycle *n*+1: `Seq-#` == 1 && (`Type` ==1 || `Type` == 2)
3. Cycle *n*+*m*: `Type` == 1 && `Cmd` != 0

In this example, the ELAM expects more complex trigger equations across separate cycles. With the inclusion of the OR and NEGATION operators, a separate MM cannot be allowed to each cycle. The second and third conditions must be stored in different MM registers. Any example of how the MM registers could be programmed is shown in Table 3:

Table 3. MM registers setup for more complex example

Mask0	1	1	1	1	1	0	0	0	`Seq-#` == 0 && `Type` == 1
Match0	0	0	0	0	1	0	0	0	
Mask1	1	1	1	0	0	0	0	0	`Seq-#` == 1
Match1	0	0	1	0	0	0	0	0	
Mask2	0	0	0	1	1	0	0	0	`Type` == 1
Match2	0	0	0	0	1	0	0	0	
Mask3	0	0	0	1	1	0	0	0	`Type` == 2
Match3	0	0	0	1	0	0	0	0	
Mask4	0	0	0	0	0	1	1	1	`Cmd` == 0
Match4	0	0	0	0	0	0	0	0	

In this example, we can note that Cycle *n* uses MM0, Cycle *n*+1 uses MM1, MM2, MM3 and Cycle *n*+2 uses MM2, MM4. The state machine for this example would be programmed according to Table 4.

Table 4. State table for more complex example

	tc	**S2**	**S1**	**S0**	**MM4**	**MM3**	**MM2**	**MM1**	**MM0**	**IncCnt**	**NS2**	**NS1**	**NS0**	
1.	0	0	0	0	X	X	X	X	0	0	0	0	0	
2.	0	0	0	0	X	X	X	X	1	0	0	0	1	
3.	0	0	0	1	X	X	X	0	X	0	0	0	0	
4.	0	0	0	1	X	0	0	1	X	0	0	0	0	
5.	0	0	0	1	X	1	X	1	X	0	0	1	0	
6.	0	0	0	1	X	X	1	1	X	0	0	1	0	
7.	0	0	1	0	X	X	0	X	X	0	0	1	0	
8.	0	0	1	0	1	X	X	X	X	0	0	1	0	
9.	0	0	1	0	0	X	1	X	X	0	0	1	1	
10.	0	0	1	1	X	X	X	X	X	0	0	1	1	Trigger!

Hence, the current ELAM implementation is unable to deal with this problem, given the static nature of the state machine. In this case, a dynamic state machine is required. In a practical sense, the scenarios may be even more complex – expressions are not necessarily specific to a single clock cycle.

There are cases where part of a signal is valid in clock cycle *n* and the other part of the signal is valid in signal *m* (usually *m* = *n*+1). Users want to trigger based on expressions that span signals in different bus cycles, as well as signals that span one or multiple bus cycles. An example of such a scenario is shown below in Figure 6:

Figure 6. Multi-cycle example scenario

The example shows 5 different fields: `a`[15:13], `b`[12:11], `c`[10:6] and `d`[5:0]. The frame in this example would span both cycles in the form [`a,b,c,d`], or [MSB, LSB]. A user could i.e. enter the following expression (braces added for clarity):

trigger `if (a == 5 and c == 2)` OR `(b != 3 and d == 0x2d)`

Clearly with the current implementation of the ELAM, such trigger conditions cannot be supported. This example provides further evidence that the state machine would need to become dynamic. The ELAM must be updated to support such trigger statements and this is the main point focus of this thesis. An algorithm is required to determine the optimal state machine based on a given trigger expression and a set of constraints imposed by ELAM parameters (number of states and Match and Mask registers). The maximum amount of MM registers is 8 (in some Cisco ASICs fewer MMs are present), which makes them very valuable resources. Thus, it is also vital to minimize the number of Match and Mask registers required. The algorithm should also take into account that certain data fields in the trigger expression may come in different cycles of a frame. As a final consideration, note the following example:

ELAM taps into the data bus of an ASIC that carries Ethernet frames. The MM register format could look as follows:

{`SOP` [18:18], `EOP` [17:17], `VLD` [16:16], `Data` [15:0]}

The Ethernet Type II frame's format is {`DA`[48], `SA`[48], `ET`[16]} followed by the payload, which could be an IPv4 packet. If the user wants to trigger on the IP address, it would require more than 8 MM registers to look that deep into the frame, since it's necessary to start from *start of packet* (SOP) equal to 1. By possibly employing the user counter, portions of the packet could be skipped without utilizing MM registers.

1.5 Thesis Outline

This thesis consists of 6 chapters. Chapter 2 gives the background information and general areas of research related to the problems discussed in this thesis, and will provide a literature survey of previous work in academics pertaining to the various problems described. The actual algorithms used to increase the flexibility of the ELAM will be discussed from Chapter 3 to Chapter 4. Chapter 5 evaluates the results found for this project and Chapter 6 concludes the work.

Chapter 2: Related Work. This chapter gives a brief introduction to the various areas of research required to increase ELAM flexibility. Topics of discussion will touch upon areas of Boolean expression minimization, NP-complete problems, register allocation algorithms, and finite state machine minimization. Some of the limitations of the current ELAM implementation will be discussed. The current work being done in each of the areas will be summarized.

Chapter 3: **Register Minimization.** Algorithms pertaining to how to minimize the set of Match and Mask registers used will be discussed in this section. An evaluation for several methods to solve some of the main problems will be studied in regards to their applicability to the ELAM problem. Boolean minimization algorithms, including ELAM specific algorithms will be discussed. Further ELAM limitations will be highlighted.

Chapter 4: **State Minimization.** Algorithms pertaining to generating a deterministic set of states will be discussed. How the state machine will be used to support complex trigger expressions will be clarified. A study on finite state machine optimization and its applicability to minimizing the deterministic and dynamic state machines will be shown.

Chapter 5: **Evaluation of Results.** The solution implementation will be analyzed and some limitations of it will be discussed. The main algorithms utilized will be measured in terms of efficiency, and limitations of the solution will be discussed. A usability study is conducted to determine the viability of the proposed solutions.

Chapter 6: **Conclusion**. The work will be summarized and future work on this project will be highlighted.

Chapter 2
Related Work

The problem of optimizing Match and Mask register usage in the ELAM and the ability to support a wide variety of trigger expressions can be broken down to several distinct problems. Although it is difficult to explore every aspect of the different problems in precise detail, several different methods of solving each problem will be discussed. The first problem in this project concerns how a trigger expression from the user can be simplified and compacted so that the least number of Match and Mask registers are required to store it. The trigger expression from the user is really a collection of fields equating to some value. The results of different field evaluations to specified values are further evaluated with respect to each other using a set of logic operators. Thus, the user-inputted trigger expression can be thought of as a Boolean expression and hence, minimization techniques can be applied to simplify it. Once the trigger statement is simplified, it then needs to be optimally stored in the Match and Mask registers. In this case, a register allocation problem of sorts exists. The next part of the project is to dynamically generate a set of states based on the trigger statement. The set of states generated should be deterministic and redundant states should be removed. It is through the dynamic state machine that support for logical NOT, OR, and AND in the ELAM is achieved. Deterministic finite state machine optimization techniques need to be considered for this section.

It is important to note that certain hardware limitations exist in the ELAM, for instance, Match and Mask registers are programmed in a certain way and generate a specific output, and for the purposes of the project, it is impossible to change any hardware implementations. Thus, some of the developed algorithms are ELAM specific and are less efficient than the optimal case to account for these constraints. These cases will be discussed in more detail in later sections, and hardware changes to the ELAM will be proposed in order to achieve more optimal cases.

2.1 Boolean Expression Minimization

Normal numeric algebra generally deals with real numbers. In the case of Boolean algebra however, values of true (0) and false (1) are considered. The values in Boolean algebra are commonly referred to as bits or binary digits, in contrast to standard decimal digits from 0 to 9. Similar to elementary algebra, Boolean algebra possesses a set of operations based upon multiplication, addition and negation. Logical AND is usually represents multiplication, OR represents addition, and NOT represents negation. Other Boolean operations are derivable from these fundamental cases. Boolean algebra possesses its own set of laws and axioms which are built from Boolean operations. The details concerning the axioms of Boolean algebra will be omitted as they do not play an important role in this project [10].

A Boolean expression is an expression that results in a Boolean value, namely, true or false. Boolean expressions are often expressed in the form of an equation, and thus, Boolean algebraic manipulations can be performed on them to simplify their logic. By simplifying Boolean algebraic expressions, the need for extensive calculations are reduced, thus saving both space and time.

2.1.1 Boolean Expression Minimization Literature Survey

Logic minimization techniques have traditionally been used in logic synthesis but in recent years have found applications in areas including logic synthesis [5], routing table reduction [11], and hardware/software portioning [14]. To tackle the problem of logic minimization, exact algorithms are unsuitable or practical size tables, hence approximate algorithms are utilized. In terms of minimizing Boolean expressions, there are three classic methods. These methods are Karnaugh maps [7], the Quine-*McCluskey* (QM) algorithm [8] and the Espresso heuristic logic minimizer [9]. Each of the methods has their own distinct advantages and disadvantages. Asides from the classical methods of logic reduction, more complex but efficient methods also exist.

When utilizing Karnaugh maps, Boolean variables from an expression or truth table are transferred to a 2–dimensional grid. Gray code principles are applied in which only one variable changes between the squares. The output possibilities are transcribed and like terms are grouped together and minimized according to the laws and axioms of Boolean algebra. Karnaugh maps are generally visually based and groups of 1s are encircled in the grid. Overlaps in logic and hence redundancy is detected in this manner [18].

The Quine–McCluskey algorithm (or the method of prime implicants) is another method for the minimization of Boolean functions. In terms of functionality, it is the same as using Karnaugh mapping, but the tabular nature of the algorithm makes it more efficient for use in computer algorithms. Quine-McCluskey also gives a deterministic method of determining if the minimal form of a Boolean function has been reached [17].

The Espresso logic minimizer is a computer program that combines heuristics and specific algorithms for logic minimization. Instead of representing the equation using minterms, the program utilizes "cubes", which represent the product terms in the ON-, DC- and OFF-covers [19]. The Espresso logic minimizer is highly efficient in terms of memory and resource usage. The input is a function table describing the desired functionality, and the output is a result table. Espresso is a two-level logic minimization scheme like the Quine-McCluskey algorithm.

Asides from the classical forms of logic minimization, several more complex schemes have been developed. The Espresso-II logic minimizer in [22] improves upon the original design of the Espresso logic minimizer. A newer logic minimizer, the *Riverside On-Chip Minimizer* (ROCM) is described in [27]. S. Ahmad and R. Mahapatra [5] discussed on-chip logic minimization using *m-tries*. The approximate logic minimization algorithm is based on a ternary trie and marks binary 0, 1 and don't care conditions as a leaves of a tree. Logical patterns are mapped to this structure and branches are collapsed where possible. R. McCalla used a minterm-ring algorithm for simplifying Boolean expressions [24]. The algorithm determines prime implicants and essential prime implicants by counting the number of links of each minterm to logically adjacent minterms.

2.2 Optimizing Register Usage

A large class of computational problems involves properties of graphs, digraphs, integers, Boolean formulas and elements of other countable domains. The primary element of interest in these problems concerns their computational complexity [29]. R. M. Karp [12] produced a famous list of then known problems of this form. These problems can be satisfactory solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. However, a large number of unsolved problems do not possess a polynomial –bounded algorithm and are similar in nature. These problems are found in the areas of covering, matching, packing, routing, assignment and sequencing. The fields in a user-inputted trigger expression to the ELAM can be considered as individual blocks which need to be stored in the set of Match and Mask registers, as shown in Figure 7.

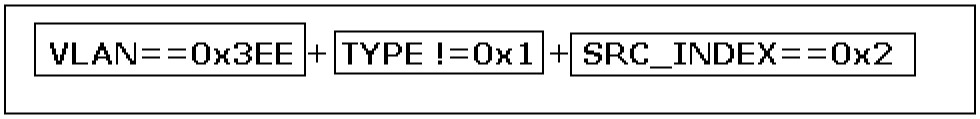

Figure 7. Storing fields in the ELAM

The set of Match and Mask registers have finite storage capacity. Thus, the problem has similarities to packing problems such as the Knapsack problem [20]. The set of combinatorial problems that are related in such a manner are special in that, if a solution exists for one variation of the problem, then the same solution can be applied to all of them.

2.2.1 NP-Complete Problems

The amount of time required for a program to run to completion is vital in most programming applications. For a program *P*, the running time of the program can be defined as the shortest computation time required for an input *x*. A program *P* runs in time bounded by *t(n)* where *n* is the length of input *x* if for every input *x*, the running time of *P* is less than or equal to $t(|x|)$. A program *P* runs in polynomial time if there exists a positive integer *k* such that *P* runs in time $O(n^k)$. Similarly, a non-deterministic program can be defined as programs that may have zero, one, or more than one computation with the same input [29]. The formal definitions are as follows [21]:

- P is the class of languages that are recognized by a deterministic Turing machine programs running in polynomial time
- NP is the class of languages that are accepted by non-deterministic turning machine programs running in polynomial time

The class of decision problems where a solution can be verified efficiently but no efficient way exists of determining the solution is defined as NP-complete problems. These types of decision problems belong to NP but no one knows if they are in P [13]. There are thousands of variations of NP complete problems, but if a solution exists for one of them, a solution exists for all of them.

2.2.2 Optimizing Register Usage Literature Survey

The problem concerning minimizing the MM registers used to store trigger expressions falls under the category of NP-complete problems in a general sense. More specifically, it can be considered as a variation of a logic minimization and register allocation problem. It is important to note here that slight changes to NP-complete problems may change the efficiency of the problem completely. The hardware restrictions in the ELAM serve as an additional requirement to the classic register allocation problem and function to simplify the problem. Regardless, logic minimization and register allocation problems are well researched and several algorithms have been proposed. T. Wu and Y. Lin [23] discussed register minimization using lifetime-analysis methods. A software program called VReg was developed which stores certain variables in state registers, others in signal nets, and some in unclocked sequential networks. F. Pereira and J. Palsberg [16] conceptualized register allocation as a puzzle solving problem. Program variables are modeled as puzzle pieces and the register file is modeled as a puzzle board. In this case, different architectures and requirements yield different puzzle and board variations. S. Liu and W. Zhao [30] further discussed various register allocation algorithms including graph coloring algorithms, MCNF based register allocation algorithms, and ILP based register allocation algorithms. Further work has been done to model register allocation including graph coloring [33], partitioned Boolean quadratic optimization [15] and multi-commodity network flow [31].

Because of the nature of the project, certain aspects of the problems faced are not exactly applicable to the general nature of NP-complete problems. Due to the design of the ELAM, hardware limitations exist concerning how the Match and Mask registers can be used, and because of the limitations, many algorithms associated with the register allocation are not entirely applicable. The way the Match and Mask registers are designed makes it impossible for logically incompatible trigger conditions to be grouped together and furthermore, trigger fields occurring in different clock cycles cannot be stored in the same match and mask register. Because of this strong limitation concerning how storage of trigger fields can occur, the storage problem faced here can be considered are no longer NP-complete, and as shown later, can be solved in polynomial time.

2.3 State Minimization

Finite state machines (FSMs) have been utilized to for a diverse set of fields, ranging from communication protocols to sequential logic circuits [25]. In sequential circuit analysis, reducing the number of states of an FSM is a well-known and highly important problem. Often, the state table of an

FSM contains redundant states that may have been invariably introduced as part of the design. Removing these redundant states reduces the logic required to synthesize and verify hardware [32]. A deterministic finite state machine is defined as a quintuple (S,S,s_0,d,F), where:

- S is a finite, non-empty set of symbols, as known as the input alphabet.
- S is a non-empty and finite set of states.
- s_0 is the initial state.
- d is the state-transition function: $\delta : S \times \Sigma \rightarrow S$
- F is the final set of states.

For most practical cases, the *Mealy* model [37] is used, where the output function is a function of the state and input. In these cases, a Mealy machine is used to represent the set of states. If the output function is merely a function of state, this is known as the *Moore* model [26], and a Moore machine is used for representation. In the case that no output function exists for a finite state machine, this is known as a semiautomation or transition system. The synthesis of finite state machines can be stated in four stages:

1. Representation of system behaviour using state transition tables
2. Reducing the number of states
3. Assigning a code to represent the states (usually binary)
4. Optimize the combinatorial logic in the next-state and output functions

The FSM logic for a Mealy machine is shown in Figure 8.

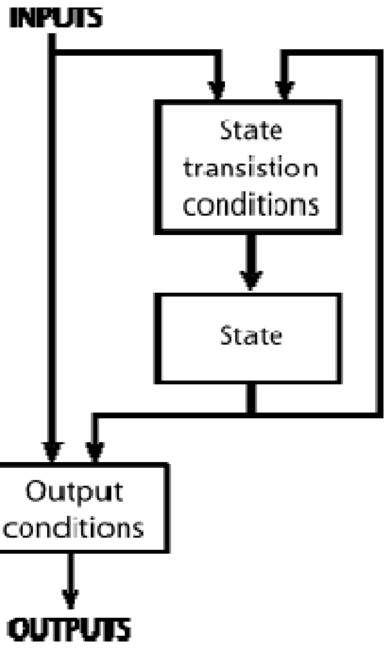

Figure 8. FSM Logic for Mealy Machine

In the case of the ELAM, the current implementation has a set of states that are hard-coded in and inflexible. The new implementation of the ELAM seeks for state transition tables to be generated dynamically for every trigger condition that the user enters. A finite-sized state transition table will be utilized where the full size of the table may or may not be required. It is important to note that there is an upper limit in the number of states that can be generated and if a trigger condition exceeds this limit, it will be deemed invalid. Since for the purposes of this project, the hardware in the ELAM cannot be changed, these restrictions cannot be removed in a practical sense. However, a more theoretically optimal model will still be discussed in the event hardware changes occur in the future.

2.3.1 State Minimization Literature Survey

Minimizing the set of states required for a finite state machine that is completely specified can be solved in polynomial time. Several classical methods exist including the Hopcroft minimization algorithm [32]. Kohavi also minimized FSMs in polynomial time in his some of his classic work [33]. Other techniques include using an Implication chart [36], the row matching method [28], or the Moore reduction procedure [40]. Additionally, cyclic FSMs can be optimized using a simple bottom up algorithm. In addition to the classic algorithms, several additional methods have been developed my researchers, some to tackle the case of an *incompletely specified finite state machine* (IEFSM). An incompletely specified finite state machine is one where, for some combinations of present states and inputs, there exist no specified next-states or outputs. These kinds of machines do not have the next-states and output functions defined over all domains.

Paul and Unger [38] devised a framework and proposed methods for creating maximum compatibles and obtaining the minimal closed cover. Yang et. al [43] proposed new assignment algorithms using look ahead. Methods including look ahead for states, look ahead for codes, and look ahead for states and codes are used. Garnica et al. [41] proposed genetic algorithms to optimized finite state machines. Classical genetic algorithms, as well as ones with new types of operators as utilized. The algorithms developed are applicable to incompletely specified state machines as well. Luccio [45] proposed prime classes and utilized the binate covering problem to devise a minimization problem. Rho et al. [42] developed a program called stamina that uses exact and heuristic modes using explicit enumeration for the state minimization problem. Kannan and Sarma [39] devised fast heuristic algorithms for both completely and incompletely specified finite state machines. The algorithms, NOVA and MUSTANG [39], are more effective for finding the minimal cover and the optimal closed cover.

2.4 Chapter Summary

This chapter highlighted the important fields in academics that pertain to the ELAM project and the related work to the ELAM project in those fields. By treating the trigger expression as a logical equation composed of several variables, Boolean minimization techniques can be performed to reduce the complexity of the trigger expression. Minimization techniques include Karnaugh maps, the Quine-McCluskey algorithm, and Espresso minimization. Following a reduction in the logical complexity of the trigger expression, register allocation techniques need to be considered. Several methods of register minimization and allocation already exist, however, their applicability considering ELAM limitations must be analyzed in greater detail. State minimization techniques are also employed for this project; hence a brief introduction to finite state machines was presented. Several well-established finite state machine optimization techniques were discussed.

Chapter 3
Register Minimization

3.1 Introduction to Register Minimization

The first step in minimizing the number of Match and Mask registers used to store a trigger expression is in simplifying the complexity of the trigger expression. Given a trigger expression *F*, there are two main forms of simplification that can be performed on *F*. In the first block, the trigger expression is treated as a Boolean logic expression. We can classify *F* as a Boolean expression because the trigger expression will always evaluate to true (the state machine will transition to a trigger state) or false (the state machine transitions to an intermediate state or fails). Each field is considered as a distinct variable and to reduce storage space, it seems natural that the trigger expression first be reduced to find the simplest equivalent logic expression. For example, if the trigger equation were "`VLAN==0x3EE TYPE==0x1 + SRC_INDEX = 0x3FF`", the expression would be treated as a Boolean equation of the form "`AB + C`" where `A`, `B`, and `C` denote 3 unique variables, "+" denotes logical OR, and multiplication denotes logical AND. By converting the trigger statement into a Boolean expression, well-documented minimization schemes can be employed. It seems intuitive that in order to reduce MM register storage space, redundancies in the user expression are first removed.

Following the logic minimization component, the actual data bits that will be stored in each MM register are analyzed to find any overlaps and redundancies. Where possible, the redundancies are removed. This binary minimization algorithm is performed before the actual Match and Mask registers are programmed in hardware.

In order to support logical AND, OR and NOT, the current static state machine in the ELAM must be configured so that it can become dynamic. For example, trigger expression "`A + B`" will have 2 entries in the state machine (2 states) that will cause the ELAM to trigger. Thus, after register minimization has completed, a dynamic state machine (that is, a state machine whose states are dynamically generated depending on the trigger statement) will be generated and optimized. This is further discussed in Chapter 4. Figure 9 (below) summarizes the steps required to simplify and store *F*.

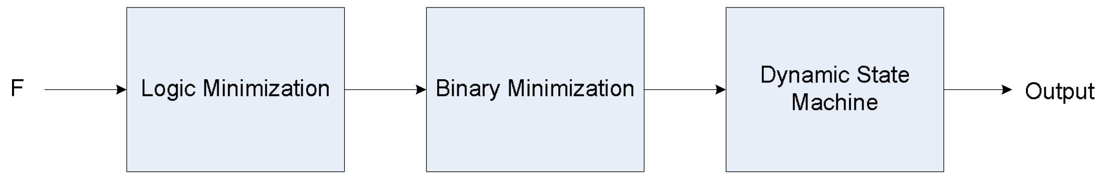

Figure 9. Overall algorithm high-level overview

The output of the dynamic state machine is a set of states that are unique to every trigger expression. The three components shown in Figure 9 are the essential components to increase ELAM flexibility and optimize MM usage.

Most of the code pertaining to the ELAM is contained with a single file, cap_elam.c. As part of this project, all updated code will also be stored in cap_elam.c. Please refer to Appendix A for examples of source code. Because the ELAM is already a well-developed project, much of the work for this thesis had to be integrated with existing code. Figure 10 demonstrates the various blocks of code that were inserted and how they relate to the other blocks in ELAM.

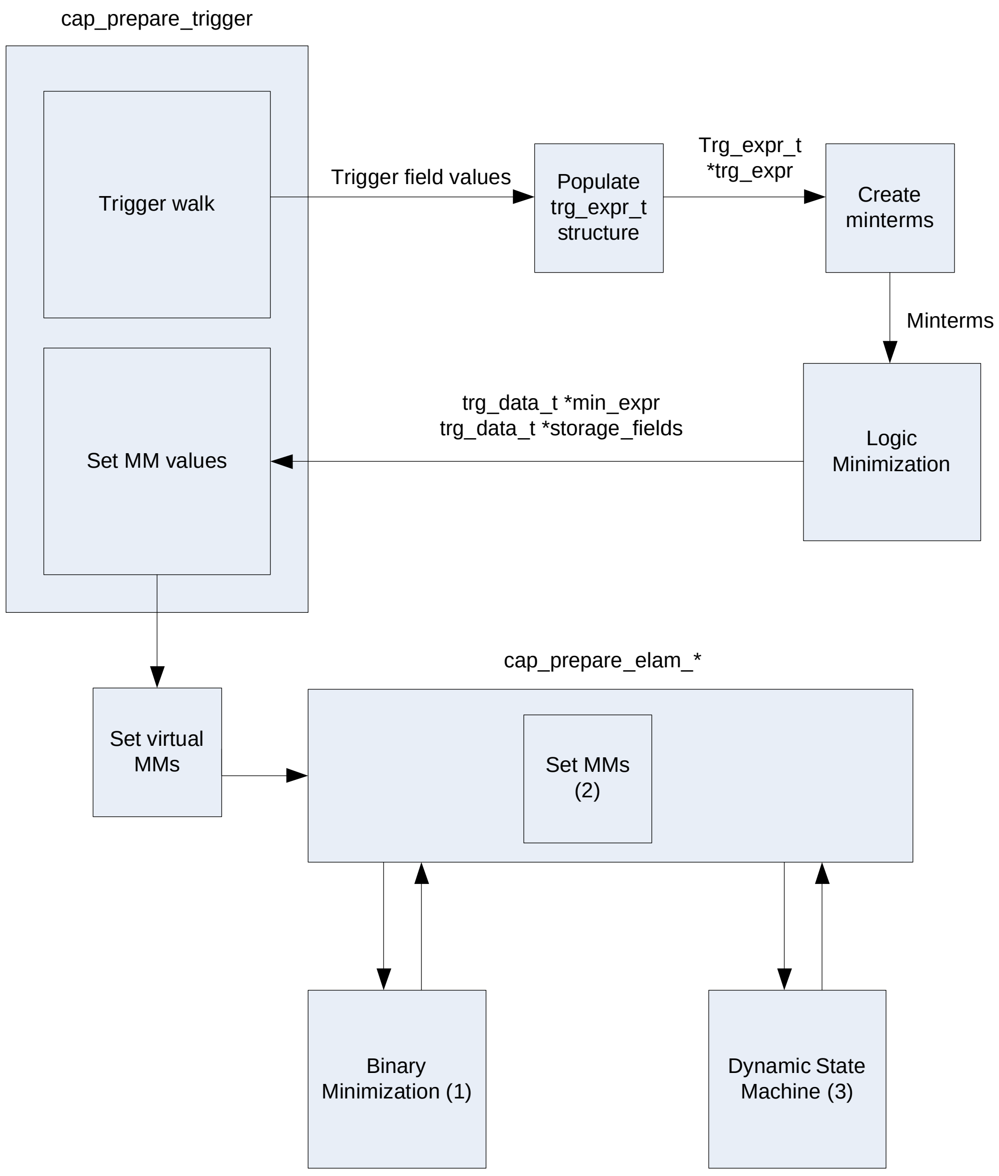

Figure 10. The blocks added/modified in cap_elam.c

It should also be mentioned that the minimization schemes employ certain heuristics that may result in a less than optimal solution when the trigger expression becomes incredibly complex. For example, greedy algorithms are used as part of the Quine-McCluskey algorithm. However, the current implementation should sufficient enough in efficiency for nearly all practical scenarios. More in-depth detail about each component is provided in the following sections.

3.2 Trigger Expression Minimization

3.2.1 Overview

There are several methods to minimize the number of Match and Mask registers used to store the user-inputted trigger statement. These blocks are shown in Figure 11 (below).

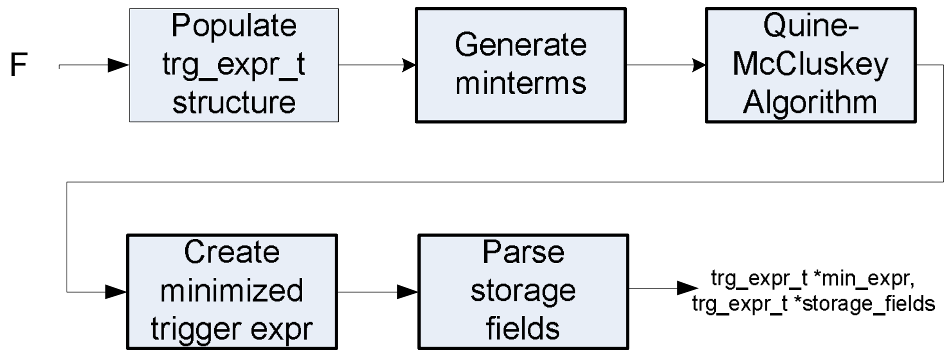
F
Populate trg_expr_t structure
Generate minterms
Quine-McCluskey Algorithm
Create minimized trigger expr
Parse storage fields
trg_expr_t *min_expr,
trg_expr_t *storage_fields

Figure 11. Overview of Components in Logic Minimization Block

Trigger expression minimization or logic minimization is responsible for looking at a trigger expression on a field by field basis. Each field is considered as a variable and the trigger expression is treated as a Boolean equation. Logic Minimization attempts to find the simplest logically equivalent (to the original expression) Boolean expression by first mapping the trigger equation to a set of *minterms*, reducing those minterms, and parsing the result into how different fields can be stored in MM registers.

3.2.2 Input Trigger Statement

In the current design of the ELAM, the trigger statement must be entered in a certain form or else it cannot be processed. Currently, the ELAM only accepts trigger statements in product form; however, this project requires that expressions in a sum of products form be also accepted. For example, expressions can be as follows: "`<FIELD_NAME>==VALUE`" or "`<FIELD_NAME>!=VALUE`". Logical AND is expressed by using a space, such as "`FIELD1==VALUE FIELD2!=VALUE`" and logical OR is expressed with a single plus sign, for example "`FIELD1==VALUE + FIELD2==VALUE`". The equivalent Boolean expression using variables would be `AB + C`.

For future clarification, several definitions are required. A trigger 'field' or 'variable' indicates a single trigger condition. The trigger expression `ABCD + AB'C` is considered as having four fields corresponding to the variables `A`, `B`, `C`, `D`. A trigger 'term' is used to indicate one or more fields joined by logical AND in a product of sums forms. For example, the trigger expression `ABCD + AB'C` is considered as having two terms. A field may span multiple cycles, for example, in the case of source or destination IP headers, the field data may arrive in 4 different cycles. This is because the high bits of the IP address may be stored in one cycle and the low bits stored in another cycle. Concatenating these bits together is not considered in this project as the functionality for that is already in place.

When a trigger expression is entered by the user, it will be parsed and mapped to a set of variables. The uniqueness of a field depends on the name of the field, the data value, the mask value, and the cycle that the field is stored in. For example, the trigger statement `VLAN == 3EE VLAN == 3EF` contains two unique fields and the expression `VLAN == 3EE VLAN != 3EE` contains only 1 unique field (one field is the complement of the other). As soon as a user enters a trigger expression, a mapping algorithm is run to identify unique variables and to generate a mapped expression. Once a set of fields is mapped to a set of variables, the next step in the algorithm can be carried out.

3.2.3 Boolean Expression Reduction Algorithm

Several different algorithms were considered to minimize the trigger expression. From a practical perspective, only the three most popular methods of logic minimization were researched in detail. This was due to time constraints placed on the project. There was insufficient time to implement some of the newest proposed methods. There was also a lack of documentation beyond that of research papers for the newest methods. For Karnaugh maps, the Quine-McCluskey algorithm and the Espresso heuristic logic minimizer, plentiful resources were available. For each of the algorithms, there are advantages and disadvantages to each method. Table 5 below summarizes the main advantages and disadvantages of each method.

Table 5. Boolean minimization comparison

Method	Advantages	Disadvantages
Karnaugh maps	Easy visualization	Inefficient for greater than 5 variables
Quine-McCluskey algorithm	Systematic and well-suited for computer programs	Inefficient for large number of variables
Espresso Heuristic logic minimizer	No restrictions in terms of variables or complexity of expressions; fast and efficient	Not guaranteed to be global minimum

It was decided that the Quine-McCluskey algorithm would be used for the Boolean logic minimization component of the project. This is because, from a programming perspective, the Quine-McCluskey algorithm is simple to code, and its non-complex nature allows an ease of understanding for future coworkers to continue the project. Although the Quine-McCluskey algorithm becomes inefficient and grows in polynomial time given a large number of variables, for the purposes of the ELAM, that limit will never be reached. The user, for almost every practical application, will only enter an expression of under 32 variables. Thus, the Quine-McCluskey algorithm is just as efficient as the Espresso algorithm and much more so than Karnaugh maps in these cases.

3.2.4 Generate Minterms

The next step in the trigger expression algorithm is to generate a set of minterms so they can be minimized with QM. A minterm is essentially a binary sequence representing a term in the overall trigger expression. Each bit in the minterm corresponds to a field in a term. For example, the expression `ABCD + B'C'D` generate a set of minterms: `1111`, `x001`. To carry out this mapping, the number of unique fields in the trigger expression is determined and stored in a two dimensional matrix. In the matrix, each row corresponds to a minterm and each column is a 'bit' in that minterm. For example, minterms `0111` and `001x` are stored in this matrix as follows:

```
0   1   1   1

0   0   1   X
```

Once the minterms have been mapped and correctly set, they are in a form that can be applied to logic minimization algorithms.

3.2.5 Quine-McCluskey Algorithm

QM takes a set of inputs in sum of products form and analyzes that set of variables to reduce the redundancies present in the logic of the expression. The logic reduction here can use any algorithm really; however, QM is a good choice because:

1. It is still relatively efficient when the number of input variables is > 4 and not overly long.
2. The algorithm is systematic, which makes it easier to implement as code.

The algorithm is a two-level logic minimization algorithm and relies fundamentally on the resolution rule of propositional logic, which states that:

$$(F \vee A) \wedge (F \vee \bar{A}) \equiv F$$

What is means in terms of code this that two minterms can be combined if they are identical except for one ‘bit’ position, and that one position is 0 in one term and 1 in the other. Figure 12 (below) illustrates this concept:

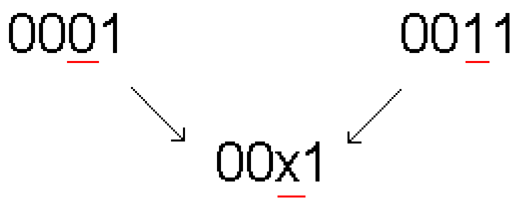

Figure 12. Resolution rule

From a high level perspective, the QM algorithm has three basic steps:

1. Combine terms where possible to produce smaller terms until no more can be produced.

2. Identify those terms that cannot be combined with another term to form smaller terms. These terms are called prime implicants.

3. Find the set of prime implicants that imply the original equation.

Thus, the QM algorithm is composed of two main blocks, as shown in Figure 13 (below):

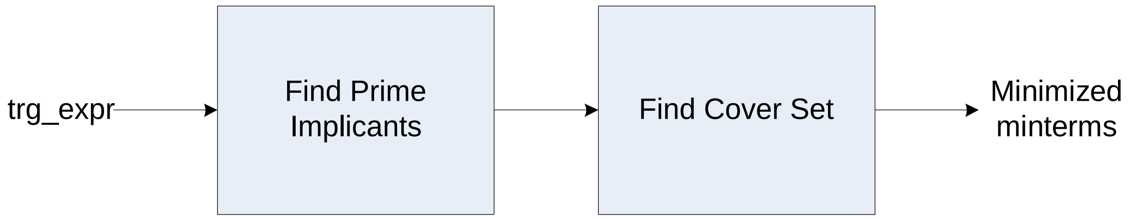

Figure 13. QM algorithm overview

3.2.5.1 Find Prime Implicants

In the example shown in Figure 12, `00x1` is a prime implicant as it implies `0001` and `0011`. To find the set of prime implicants of a more complex set of data, we require an appropriate data structure. The data structure that will be used is a 2D array of queues. Each row in the matrix is the number of ones in a minterm, and each column is the number of don't cares. Minterms from the original equation are placed in this table. For example, minterms `001`, `01x`, `xx0`, `100` and `111` will get placed in the table as follows:

Table 6. QM reduction table example

DC \ Ones	0	1	2	3
0		001, 100		111
1		01x		
2	xx0			
3				

Once all the minterms generated from the original trigger expression have been placed in the appropriate place in the table, a search occurs for any reductions in the terms. Because of the way the table is structured, we only need to attempt to combine terms falling into lists that are next to each other in the table. Although this heuristic reduces the number of comparisons that are required, we still have to try all combinations of these two lists. When a pair of terms is combined, we know where this new combined term will be placed: it will have 1 more don't care than the previous terms and the number of ones in the combined term will be one less than the input term with lesser number of ones. Thus, we can do a single scan through the table to look for terms that can be combined.

Each time two terms are combined, the new combined term is inserted into a prime implicant list. The list is also scanned to see if the two input terms are themselves in the prime implicant list (they were combined from other terms): if they are they are removed from the list. The new combined term is also inserted into the appropriate position in the main table if it does not already exist.

3.2.5.2 Find Cover Set

The next step in the QM algorithm involves finding a subset of prime implicants that imply all of the terms of the original equation. To test if a prime implicant implies a term, we compare the term with the prime implicant. If the bits of the term are the same as the prime implicant or if the prime implicant has a “don’t care” in the same position as a bit in a term, then the term is being implied. An appropriate data structure is required here to check for prime implicants and essential prime implicants. This implies table is a 2D Boolean array. The rows indicate the prime implicants and the columns are for the minterms. Each entry in the table is a Boolean value for whether the prime implicant implies the current term. Table 7 gives an indication of how the implies table would look for the example shown in Figure 12.

Table **7.** Figure 5 example implies table

Num_pi \ num_terms	0001	0011
00x1	1	1

Each time an essential prime implicant is found, it is inserted into a final list which stores the non-redundant terms. We must also take into consideration that a term may be not implied by any of the prime implicants, in which case, we directly insert into the list of final minimized terms. Likewise, if a term is not a prime implicant (not in prime implicant list) but implies another term that is not in the prime implicant list we must also insert it into the list of final minimized terms.

At this point, the cover set problem is NP-hard, and certain heuristics have to be employed. Recall that an essential prime implicant is the only implicant that implies one of the original minterms. Once an essential prime implicant is found, we remove both the row of that essential prime implicant and the columns of all original terms implied by that essential prime implicant.

After finding the essential prime implicants, we need to cover the remaining terms using the other prime implicants. A heuristic selection method is used here. The prime implicants that imply the largest number of remaining original terms are chosen first. The algorithm continues until every one of the original minterms is covered. This heuristic method [44] ensures a more efficient average running time.

For basically every practical case that the ELAM will see, this heuristic selection method will obtain the optimum solution (simplest logical equation). However, it should be noted that if the trigger expression becomes incredibly complex (> 256 fields and terms), there is a chance that a less than optimal solution will be obtained.

When the cover set is found from the Quine-McCluskey algorithm, we should obtain a new list of minimized terms. From this information, a new minimized trigger expression is created. Thus, once redundant terms and/or fields are removed from the original user trigger expression, the next step is to rearrange the new minimized expression for storage in the virtual Match and Mask registers. Here, a parsing algorithm is required to break up fields based on logical compatibility.

3.2.6 Parsing Stored Fields

The first step in the parsing algorithm is to break up terms depending on the line/cycle they are in. The data structure that is used here is an array of queues. Each row in the array represents a different line, and each node in the queue corresponds to a trigger field of that line number. For each line, fields are parsed based on their logical compatibility and if they overlap. Overlap is determined by the start and end bits of a field and the cycle the field is in (for example, `VLAN==0x3EE VLAN==0x3EF` are considered as two different fields, however, they overlap because they are in the same cycle and have the same start/end bits and thus must be grouped as different terms).

The parsing algorithm consists of grouping fields which can and cannot be stored together. Clearly, fields in different cycles are unable to be stored in the same MM register, thus fields are divided into line number first. Next, fields in OR NOT form are grouped. For example, an equation of the form A' + B' + C' can be grouped together to form the equation (ABC)' according to DeMorgan's Theorem [46]. In this case, one Match and Mask register would be required as opposed to 3. Following, single "`!=`" fields that are part of a larger term are extracted. For example, the term AB'C cannot be stored in the same MM register. Instead, two MM registers, one to store AC, and another to store B' are required. Finally, overlapping fields are parsed out.

When the parsing has finished, we now have an idea of how many MM registers are required to store the newly minimized expression. There may be cases where the amount of registers required is greater than the number of registers required to store each field individually (for example, `AB + BC + CD + AD + AC` requires more registers as opposed to storing `A`, `B`, `C`, `D`). If this is the case, it is more efficient to store each field as opposed to each term. The state machine can then be used to create the minimized expression.

After parsing finishes, the next step is to create a link between each field in the minimized expression and the fields that are actually stored. This mapping is required as the state machine may need to reuse MM registers to form the trigger expression and generate a set of states. See Figure 14 (below) for an example of this.

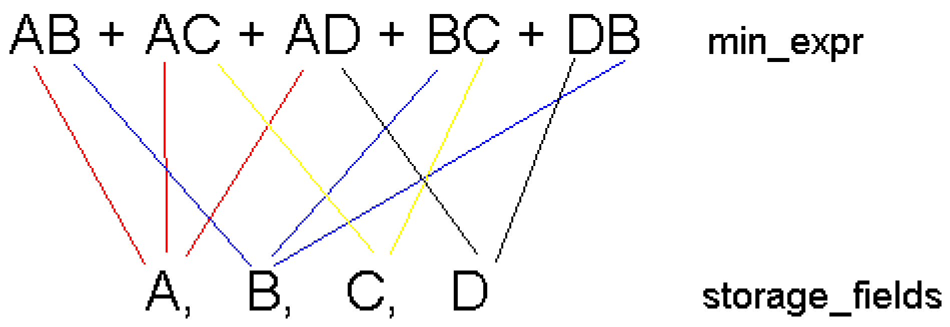

Figure 14. Example of mapping between minimized expression and MM register storage

In the example in Figure 14, only the fields themselves are stored and the minimized expression is created by reusing Match and Mask registers. To accomplish this mapping, each field in the minimized expression contains a storage array that is associated with a corresponding MM register where that field is stored. Note that at this point in the algorithm, the virtual MM registers have not been set up. It is assumed that MM`0` will contain the first field and MM*n* will contain the *n*th field. If this assumption proves incorrect, the mapping will be altered at a later point. For example, in Figure 14, the field `A` in the minimized expression would have a storage array: `[10000000]` with the assumption that field `A` is stored in MM`0`. Please refer to Appendix A for more logic minimization examples.

3.3 Data Compression for Storage

After a user-inputted trigger statement has been simplified, we can often eliminate redundant fields. The next step is to analyze the actual data that will be stored in the Match and Mask registers. Although the Boolean expression from the previous block may be in a minimized form, there exists the possibility of further compressing the stored data.

Since the Match and Mask registers are designed to mimic the structure of the incoming data packets, it is difficult to change their implementation without modifying the ELAM hardware. It is for this reason that several well known compression algorithms will not work in the compressing the data stored in the Match and mask registers. Such methods include the Lempel-Ziv-Welch algorithm is a universal lossless data compression algorithm [50]. Other compression algorithms such as Burrows-Wheeler transform [47], Dynamic Markov compression [48], entropy coding techniques [52], and run-length encoding techniques are all applicable given a different hardware design. Nortel Networks also developed a binary data compression/decompression algorithm [49] that unfortunately cannot be applied in this case as well. With these restrictions in the Match and Mask registers and how data is stored within them, other designs were considered. An ELAM-specific design was developed based on the minimizations that are actually possible.

After a new minimized expression has been outputted from the logic minimization block, the next step is to set the virtual MM registers so that the actual MM registers in hardware can be programmed. Once the virtual MM registers have been set up, we can consider the Match and Mask registers as a binary matrix. The eventual goal then is to remove any data overlaps and redundancies in this matrix using a binary minimization algorithm. Figure 15 (below) shows the essential blocks in binary minimization.

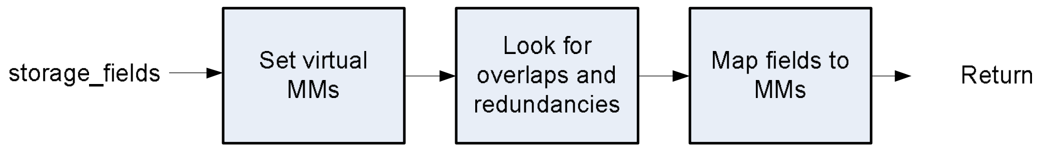

Figure 15. Essential blocks in binary minimization

The binary minimization algorithm is applied to all data stored in the Match and Mask registers. It looks at the bits present in the MM registers and removes redundancy taking several constraint factors into account. The first constraint is that if fields are stored in different Match and Mask registers, but in the same cycle, their start and end bits are already predefined according to a database. The statically defined start and end bits make redundancy removal impossible in this case. However, if two fields were in different cycles (and thus stored in different MMs), there is a chance they may have identical start/stop bits and matching data values. In this case, the Mask and Match registers for the two fields would be identical and it would be redundant to store both. The binary minimization algorithm attempts to locate such redundancies. In the description of the algorithm, the word ‘term’ refers to the data associated with a consecutive sequence of mask bits that are set to ‘1’. These sequences are also referred to as ‘bit sequences’. The binary minimization algorithm is ELAM specific, as it is designed taking the constraints imposed by the Match and Mask registers into account. The algorithm is summarized in the following sections.

3.3.1 Overlap and Redundancy Detection

To determine if the start and end positions of bit sequences of different fields are identical, we need to examine consecutive series of 1s in the Mask register. These bit sequences indicate valid ranges of data where overlap may occur. To determine if one term is overlapping with another term, a comparison is run to see if the non-masked-out bits are equal. This comparison algorithm returns a Boolean result concerning the existence of overlap. Overlap is determined if one term has a start bit less than or equal to the start bit of another term and an end bit greater than or equal to the end of bit another term. The smaller is term is said to overlap with the larger term. Bits here are in most significant bit (MSB) to least significant bit (LSB) form. The end bit refers to the larger bit value and the start bit refers to the smaller bit value. For example, consider the following 2 terms:

```
        1   1   1   0

1   0   1   1   1   0
```

In the example above, the 0th bit, the start bit (LSB) is left aligned and the 5th bit, the end bit (MSB) is right aligned. In this case the start and end bits of `1110` overlap with `101110` as the `1110` sequence has a start bit equal to `101110` and an end bit that is less. The term `1110` is marked as the smaller term, and the term `101110` is marked as the larger term.

If there exists overlapping terms, the overlapping bits of the two terms need to be compared to see if they are equal. In the previous example, for the range of overlap between the smaller and larger term, there exists identical data, thus the smaller term is said to overlap fully with the larger term. If the smaller term has the same bit sub-sequence as the larger term, the smaller term is stored in an array associated with the larger term. The MM register that the bit sequence is stored in is also stored in a separate array.

Each bit sequence is compared with every other bit sequence to determine overlaps. For a given term, all other overlapping terms are inserted into the array associated with the term. These additional terms are 'combined' with the previous value in the array. The combination logic here merely means extending the storage array associated with a term every time an overlap is detected. For example, for the term `101110` shown in the example before, if an additional term `101`, with appropriate start and end bits, were compared against `101110`, an overlap and match would be detected. In this case, the array associated with `101110` already contains the term `1110` so `101` and `1110` would have to be combined. This is shown below:

```
1   0   1

        1   1   1   0

1   0   1   1   1   0
```

Here, `101` represents the new term, `1110` the old term, and `101110` the new combined term. In this case, the 'combined' storage array associated with `101110` is in fact equal to the term itself. In this case the sequence `101110` is marked as redundant. Let us assume that the sequence `101` is stored in MM0, `101110` in MM3 and `1110` in MM4. Each term stores this information in a separate array. In the case of the sequence `101110`, this array would look as follow: [10001000]. Imagine if this bit sequence represented the data for a field. Previously, the data associated with the field is located in MM3, but now, we can use the data in MM0 and MM4, freeing up MM3.

To clarify and summarize, the binary minimization algorithm iterates through all bit sequences and sees if the combined array associated with each term is in fact equal to the term itself. If it is, this means that the term is redundant. We can look at the array which stores the location of the overlapping bit sub-sequences that formed the redundant term to know where these terms are stored. Given the constraints imposed by the structure of the Match and Mask registers, minimizations of this form are the only ones that appear possible.

Now, there is the possibility that a single MM may have multiple bit sequences. For example, the following binary sequence contains 2 distinct terms. This is a possibility when there are multiple fields stored in a single MM register, and the start/end bits of the fields do not line up. In this case, we must first expand the total number of Match and Mask registers so that each bit sequence is allocated its own MM. For instance, consider the following MM:

```
1   0       1   1   0
```

In this case, the sequences 10 and 100 broken up into two separate MMs. Since we have an unlimited number of virtual MM registers in software, this can be easily done. In the event that a term is found to be redundant, it is removed from the virtual MM register (by setting the mask bits to 0 and clearing the Match register). If all terms in a MM register are redundant, that MM register is no longer used. If a term was originally stored in a MM, and is allocated a new register (expanding multiple bit sequences in one MM register), and is now used to form redundant term, it cannot be recombined back to its original MM register. Of course, this implies that bit sequences that were split from another MM register but not used to form any redundant terms can be placed back into their original MM register. The examples in the following sections may serve to clarify these concepts.

After the set of minimization operations are complete, the next step is to check if a reduction in the number of MMs actually occurred. If some bit sequences are allocated new registers during the expansion process and used to form new terms, the total number of MM registers required may be greater than if this operation were not performed. The new number of MM registers required is compared to the amount before binary minimization. If it is found to be greater, and no reduction results, the algorithm does not update the virtual MM registers. Otherwise, redundant MM registers are removed, and each field before binary minimization adjusts its storage array that holds the information containing which MM registers to use. For binary minimization examples, please refer to Appendix A.

3.4 Chapter Summary

Register minimization techniques were the focus of this chapter. Comparing the methods of Karnaugh maps, the Quine-McCluskey algorithm, and Espresso minimization, it was decided that the Quine-McCluskey algorithm is the most suitable algorithm for this particular problem due to its systematic approach to solving minimization problems. Such an approach has the advantage that it is easy to implement using in software, and given the relatively compact data sets that trigger expressions are limited to, it is efficient enough for the problem purposes. The logic minimization component is composed of several steps, including deriving a set of minterms for input to the Quine-McCluskey algorithm, obtaining the prime implicants, and obtaining the essential prime implicants of a given set of minterms.

Following Boolean expression minimization, binary minimization is run on the actual data to be stored within the set of MM registers. Due to the particular structure and implementation of the MM registers, an ELAM specific algorithm was developed. This algorithm breaks user data in the MM registers into a series of binary sequences. The binary sequences are examined to detect overlap. If certain sequences are found to be redundant, they are removed from the MM registers. If the final number of registers is found to be less after these sequences are removed, the reduced set of MM registers is used for further ELAM execution.

Chapter 4
Dynamic State Machine

After binary minimization has completed, the actual MM registers in hardware will be set based on the information contained within the virtual MM registers. It is at this point that some minor rearrangements of MM registers must occur in order to correctly receive incoming data. The first MM (MM0) is used to check the valid bit of an incoming data packet. This bit indicates if a data packet is actually valid. The next MM (MM1) is used to check the start of packet (SOP) bit. The valid bit and the start of packet bit occur in different cycles, and must be checked before the payload arrives. Unfortunately, this means that two MM registers must be set aside to accommodate these conditions. Following this initial setup, the other MM registers are then set based on the virtual MM register data. After the MM registers are set, the dynamic state machine can then be programmed. The main components of the dynamic state machine are shown in Figure 16 (below).

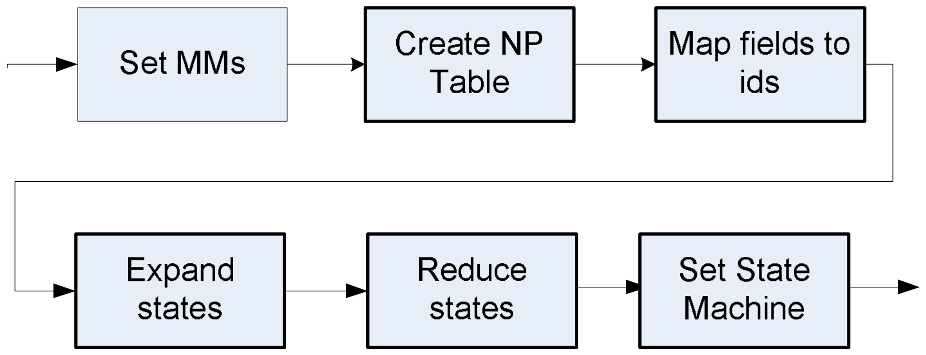

Figure 16. Main blocks in the dynamic state machine

Each of the blocks involved in programming the dynamic state machine will be discussed in greater detail in the following sections.

4.1 Overview

The creation of a set of states from the trigger condition consists of several steps and it may be more worthwhile to refer to the examples section to help clarify the steps. The programming of the dynamic state machine, or rather, generating a set of states based on a user-inputted trigger condition is an ELAM-specific algorithm. The design of the ELAM state machine utilizes information stored in the MM registers for state transition and trigger conditions. The algorithm consists of generating a list of all permutations of possible trigger statements and removing any redundancies in this list. In other words, the minimized trigger expression is converted to a set of states and this set of states is subsequently reduced.

In the first step of creating a set of dynamic states, a *necessary permutations* (NCP) table is created. This table contains a mapping between each field that is stored in the MM registers to a numerical value or ID and will be used in generating all possible state transitions depending on the trigger expression. The NCP table will be sorted in terms of cycle and each ID of the fields in this table increase numerically. We map the minimized trigger expression to each of the ID values given in the NCP table. After mapping of the expression has occurred, we sort the numerical trigger expression for ease of handling later on. The sort algorithm used is merge sort. The merge sort algorithm works as follows [51]:

- If the list length == 0 or 1, it is already sorted. Else:
- Divide unsorted list into two sub-lists of approximately half size of total list length
- Merge sort each sub-list recursively
- Merge two sub-lists, forms one sorted list

An illustrated example of the merge sort algorithm can be found in Figure 17.

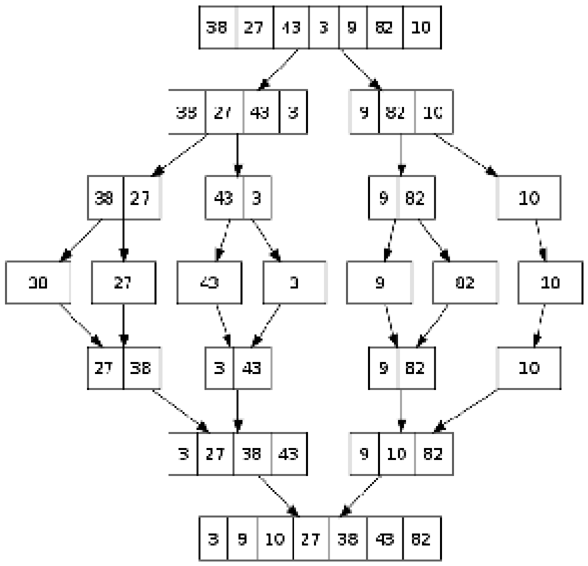

Figure 17. Merge Sort Algorithm

Following, we expand the minimized trigger expression based on the NCP table. This expansion needs to be performed because if trigger fields are in different cycles, a field in the later cycle must be correctly reached regardless of what occurred in earlier cycles. For example, given the equation A + B where A is in cycle 1 and B is in cycle 3, if the correct information for A is seen in cycle 1, we can trigger without having to consider cycle 3. However, if this information is not seen, we need to create state transitions that correctly take us to cycle 3. In this case, the ELAM will still receive cycle 2 data, however, this data will not be used. More specifically, if an expression does not contain terms of a certain cycle, we need to create ‘dummy’ terms which serve as placeholders. When the state machine sees that it should be expecting a dummy variable in a certain cycle, it will instead loop in that cycle and continue to next cycle when next cycle data becomes available.

As a further example, consider the following: If the user wants to trigger on field A AND B OR C, and field A occurs in cycle 0, field B in cycle 1, and field C in cycle 2. In this case, the state machine must note that even if it sees A’ in cycle 0 and B’ in cycle 1, C may still occur in cycle 2 and thus all variations of occurrences in the first 2 cycles must be taken into account.

After we obtain an expanded equation that may also contain dummy variables, we then try and remove redundancy in the set of expanded states. This is done by pattern matching, for example if the numerical trigger expression were `12 + 123 + 1234` where `1 (cycle 0)`, `2 (cycle 1)`, `3 (cycle 2)`, `4 (cycle 3)` are the IDs of fields in the expression, then we know that as soon as the `12` condition is triggered, the `123` and `1234` branches of this expression will never be reached. Such redundancies are removed from the expanded set of terms.

Finally, the deterministic set of states is programmed from this expanded set of terms. When the state machine reaches a trigger condition, it will jump to the trigger state. State 999 is currently set as the trigger state. After the trigger and transition states are created, the conditions which cause “back to state 0” (fail state transitions) are programmed.

4.2 Generating Dynamic States

The necessary permutations table is represented as an array. The numerical minimized expression is represented as a 2D array of integers. There are two such matrices defined, one to represent the original minimized expression and one to represent the expanded version. Negation is represented by a negative sign. For example, the trigger expression `ABC′ + CD + ABE` is mapped to the numerical expression `12-3 + 34 + 125` and stored as:

```
1    2    -3   0    0
3    4    0    0    0
1    2    5    0    0
0    0    0    0    0
```

Thus, each row in this matrix represents a different term and each column represents a different field. Next, we populate the necessary permutations table.

After the minimized expression is mapped into numerical form, we next need to look for redundancies and sort the statement. For example, if the trigger expression were in the form `BCA + B’D`, and `A` was in cycle 0 (ID = 1), whereas `B`, `C`, `D` were in cycle 1 (ID = 2, 3, 4 respectively), and `B` and `C` are stored in the same MM (and thus can be considered as one term). They would be mapped to the equation `221 + 2` and stored as:

```
2    2    1
-2   3    0
```

After sorting the expression and removing identical IDs each row, we obtain:

```
1    2    0
-2   3    0
```

Next, we must expand the minimized numerical expression to derive a deterministic set of states. This is a somewhat complex process, but hopefully the examples make it clearer. Consider the following example:

Trigger expression = `AD + BC`

Table 8. Table of Necessary Permutations for simple example

Field	**ID**	**Cycle**
A	1	0
B	2	1
C	3	2
D	4	3

Numerical trigger expression = `14 + 23`

Expanded equation:

```
1234
12-34
```

```
1-234
1-2-34
123
-123
```

Looking at the first numerical term `14`, it can be seen that in the expanded set of states, the trigger machine will transition in state depending on different permutations of what can occur in cycle 1 and 2. Hence, from the expanded equation, we can derive the set of trigger states. This is done by traversing each term in the expanded set of terms. If we know that field ID 1 is stored in MM 3 and our current state, we transition to the next state if the MM3 state bit is true. Once the trigger states have been obtained, we need to derive the set of states that send the ELAM back to state 0. These fail states can be determined from looking at the transition and trigger states. If an entry in the table does not contain a complement state with the same current state, we need to create fail states for it. For example, if the current state is 2, the MM values are [`111xxxxx`], we need to look for another value where the current state is 2, and the MM values are [`110xxxxx`]. (Recall the first 2 bits are for valid and start of packet). If such a value does not exist, we need to create a state transition that goes from the current state back to the initial state (state 0).

4.3 State Minimization

Several state minimization techniques were examined. Almeida et. al. [55] found that the Hopcroft-Karp algorithm [35] achieves the best performance in terms of state minimization. It is important to remember that because the set of states generated by the finite state machine is deemed to be deterministic, methods of deterministic finite state machine minimization or *deterministic finite automaton* (DFA) minimization can be applied.

Deterministic finite automata minimization is very well researched and established area in the Theory of Computation [54]. Recall that a DFA can be defined as a 5-tuple (S, S, d, s_0, F).

The DFA minimization algorithm uses a table-filling algorithm to determine which states are considered distinct. States that are not marked as distinct are able to be merged. Two states *p* and *q* are defined as distinct when:

1) $p \in F$ and $q \notin F$, or vice versa, or
2) for some a, a ∈ S, *d(p, a)* and *d(q ,a)* are distinct

A table is created for each pair of states and all table cells are initially blank. For clarity purposes, the table will be named DISTINCT. An iterative algorithm is run as follows:

```
1) For every pair of states (p, q)
  If p and is a final state and q is not or vice versa, DISTINCT(p,q) is marked
2) Loop until table does not update
  For each (p, q) and each character a in S
          If DISTINCT(p, q) != empty and DISTINCT(d(p, a), d(q ,a)) != empty
                  Mark Distinct (p, q)
3) Two states p and q are distinct if and only if DISTINCT(p ,q) is not empty
```

By running this algorithm, all equivalent states will be found, and ultimately a simpler DFA will result. Now, in the case of the ELAM, *S* will be the set of states derived from the trigger statement and includes the trigger states and the fail states. The alphabet will consist of all possible inputs that cause the state machine to transition, which will be the bit results of the total number of match and mask registers. The start state will be state 0, which is when the ELAM is expecting a valid bit. The next state will be state 1, which is when the ELAM is expecting a start of packet bit. From there on out, the state machine will be dynamically generated depending on the trigger condition. However, for all cases that the ELAM will see, as long as a deterministic set of states is produced, the DFA minimization algorithm described will be able to determine the simplest set of states.

As a final note, there needs to be some of a limit to the number of states that are generated. The size of the state look up table in the ELAM is finite sized, and the current restriction on the number of generated states is 64. This poses an interesting concern in that, given a more complex trigger statement, this threshold can easily be violated. At the current time, there are changes being proposed to increase the size of the look-up table. The original state look-up table was designed with a static state machine in mind. For state minimization examples, please refer to Appendix A.

4.4 Chapter Summary

This chapter primarily focused on the dynamic state machine that is created by a trigger expression. The algorithm used to generate a series of states from a trigger expression is an ELAM specific algorithm. The simplified trigger expression from Boolean minimization is traversed to generate a set of unique states for that expression. Several default and fail states are also added. Once a series of states is generated for a trigger expression, it is then minimized to reduce redundant states. In this case, DFA minimization techniques are employed. The size of the state machine table for the ELAM is limited, and thus, if a trigger expression is too complex and generates too many states, and an error must be thrown.

Chapter 5
Evaluation of Results

5.1 Logic Minimization Results

5.1.1 Quine-McCluskey Algorithm Results

After implementing the discussed solutions for the logic minimization portion of the project, a general idea of how effective the proposed solutions are was obtained. Several different test scenarios were run with common trigger statements and corner cases. These results, along with the time of execution of the Quine-McCluskey algorithm are shown in Table 9 below. The code was executed on a Windows XP SP3 machine with Intel's T2050 1.6 GHz processor.

Table 9. Quine-McCluskey Algorithm Running Times

Variables	Terms	Total Variables	Execution Time (ms)
1	1	1	0
3	6	18	15
4	11	44	15
4	15	60	32
25	6	150	78
25	25	625	125
10	64	640	78

In Table 9, the number of variables, terms, and the total variables are listed, along with the total execution time of the QM algorithm in (ms). It is worthwhile to note that the Quine-McCluskey algorithm is the limiting factor in the execution time of the logic minimization block. This is because the rest of the functionality in the block is relatively trivial. There are no complex logic calculations and the mapping of variables to functions can be run in O(*n*) time. The proof of this problem is trivial as we merely need to traverse through all the trigger fields. Each unique trigger field maps to a unique variable. Parsing through the trigger statement to generate a new minimized trigger statement is likewise O(*n*). In this case, we traverse through the set minterms and map them back to the original trigger fields. Figure 18 below graphically illustrates the execution time of the Quine-McCluskey algorithm for the various test cases.

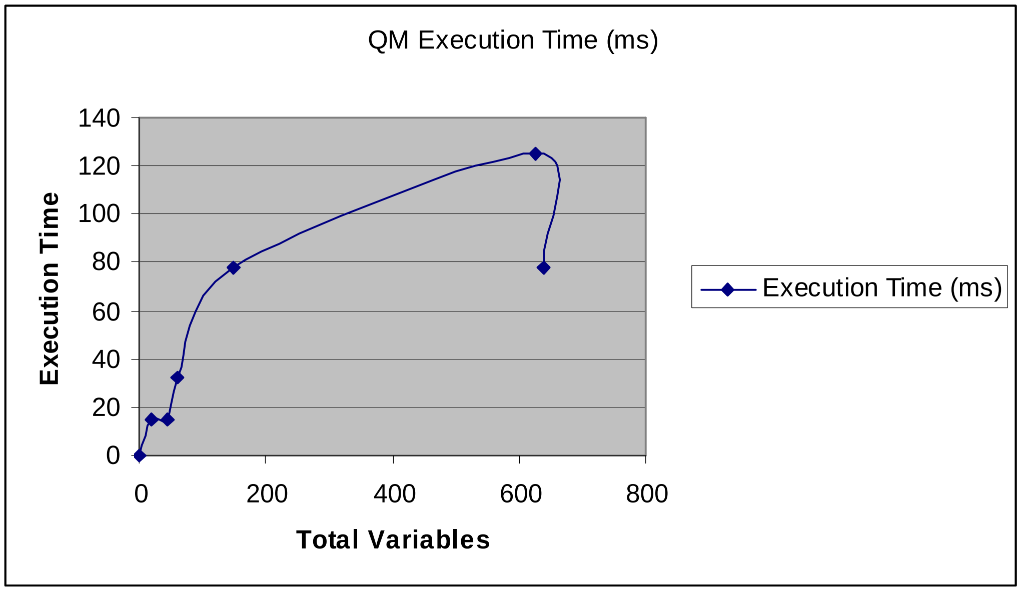

Figure 18. QM execution time

We can note from the results of Figure 18 that the QM algorithm does not show a distinct pattern in growth given a realistic set of input data. There are instances where the total number of variables increases and the execution time decreases. This is because the actual minimizations that can be performed are dependent on the trigger statement. In some of the larger test cases, random variables and terms were generated, and in many of these cases, simplifications do not occur. It is only when minimizations are possible that the algorithm continues to perform its task. The threshold level in ELAM was set to be 64 variables and 64 terms, resulting in a total number of 4096 total variables. Given the results shown in the table, this seems like a reasonable limit. By interpolation, the limit imposed will result in the algorithm still running in polynomial time. In the algorithm, certain heuristics were used, thus it is possible that the QM running time will grow without bounds if a reasonable limit is not imposed. Hence, error checks are performed in the code to limit the number of variables to 64 and the number of terms to 64 as well. Thus, we can ensure that the algorithm will run with little to no effect on the total execution time of the ELAM.

These results have been mirrored in other sources. In [44], it was found that 64 terms and 64 variables require an execution time of 0.159 seconds. The code was executed on Sun's Java 1.5.0 under Gentoo Linux on a 2.8 GHz Pentium 4 machine.

5.1.2 Binary Minimization Results

For the binary minimization component of the project, we can conduct a statistical analysis to determine just how often the code is run. Let us first examine the table of data fields to gain some preliminary understanding of how often overlaps will occur. Table 10 shows the structure of the backend database indicating the possible trigger fields.

Table 10. Database Structure for Trigger Fields

Field Name	Field Type	Trigger State	Start Bit	End Bit	Value Type	Line Type	Cycle
SEQ_NUM	DBUS_SEQ_NUM	A_ALL	255	251	T_HEX	SNGL	1
QOS	DBUS_QOS	A_ALL	250	248	T_DEC	SNGL	0
QOS_TYPE	DBUS_QOS_TYPE	A_ALL	247	247	T_DEC	SNGL	0
TYPE	DBUS_CMP_TYPE	A_ALL	246	243	T_DEC	SNGL	0
…	…	…	…	…	…	…	…

In Table 10 above, the start and end bits are of the most concern to us. Now, the only time that bits can overlaps are if the user wants to trigger on fields that are in multiple cycles. Fields in the same cycle already have their positions pre-defined, thus the only way that two fields can overlap is if they are in different cycles, or if one Match and Mask register contains the negation of a field stored in another match and mask register. Recall in the case of negation that the non-negated field is stored and it is up to the state machine to account for the correct logic.

With the old implementation of the ELAM, only limited statistics were available on what trigger statements users typically input. With the previous implementation of the ELAM, users typically triggered on only single fields as opposed to long expressions. Table 11 lists the top 5 trigger fields of interest to ELAM users as determined by a survey of 27 users.

Table 11. Most popular trigger fields

Field Name	Number of Users	Start Bit	End Bit	Cycle
VLAN	18	215	204	0
SEQ_NUM	4	255	251	1
PACKET_TYPE	2	158	156	0
CARD_TYPE	2	115	112	0
DMAC	1	95	48	0

There are several flaws with this statistical study. First of all, it was difficult to locate individual users who used the ELAM. Sometimes, certain individuals may use it every so often and it was difficult to locate all these users across a large organization like Cisco. Another problem is that many of the users are in the same groups at Cisco, thus the main data they require from using the ELAM are very similar. Likewise, the current ELAM is severely restricted in what the form of the trigger expression is. With the new changes proposed in this thesis, it is highly plausible that new popular trigger statements will arise to replace the current ones being used.

This simple study can still give us some insight into the applicability of binary minimization. We can note from the results obtained that no two fields of the most popular fields actually overlap. In fact, 4 out of the 5 fields occur in the same cycle. Even if they were to be placed in different Match and Mask registers, they would never overlap, unless the user specifies a complex equation involving both the field and its conjugate which then get placed in separate registers.

Although such occurrences are rare, it is not impossible to imagine that they will indeed occur, especially when more complex trigger statements are allowed. Hence, binary minimization still plays a role in the overall process. It is impossible at this time to determine any accurate statistical information how often it will actually be employed as the new ELAM implementation is not yet widespread in its deployment.

Binary minimization run in O(n^2) time [53]. Every binary sequence in the match and mask registers must be compared with every other binary sequence to determine if overlaps occur. Hence, the equation that here is:

$$k + \sum_{i=1}^{n} i \le k + \sum_{i=1}^{n} n = k + n^2 = O(n^2)$$

Therefore, although binary minimization is somewhat slow for conventional software programs, it is still bounded within polynomial time, and thus reasonable to use given the expected limited data set.

5.1.3 Dynamic State Machine Results

There are two main areas of efficiency measurements in the dynamic state machine. The first constraint that could potentially affect the execution time of the dynamic state machine is in creating the table of necessary permutations. Recall in creating the table of necessary permutations, the first step is to assign a unique numerical id to each trigger field in the minimized expression. After an id has been assigned to each field, a numerical trigger expression is created, sorted and then expanded. In creating a unique id for each trigger field, the total runtime would be O(*n*), the proof of which is trivial. Similarly, O(n) time is required to generate the numerical trigger expression. In order to sort the numerical trigger expression, merge sort was used. Merge sort runs in O(*nlogn*) time in the worst case. It should be noted here that the sorting algorithm chosen to sort the numerical trigger expression is not of great concern, as the number of fields and terms is restricted to be 64. Due to this relatively small number of items, sort algorithms run almost identically.

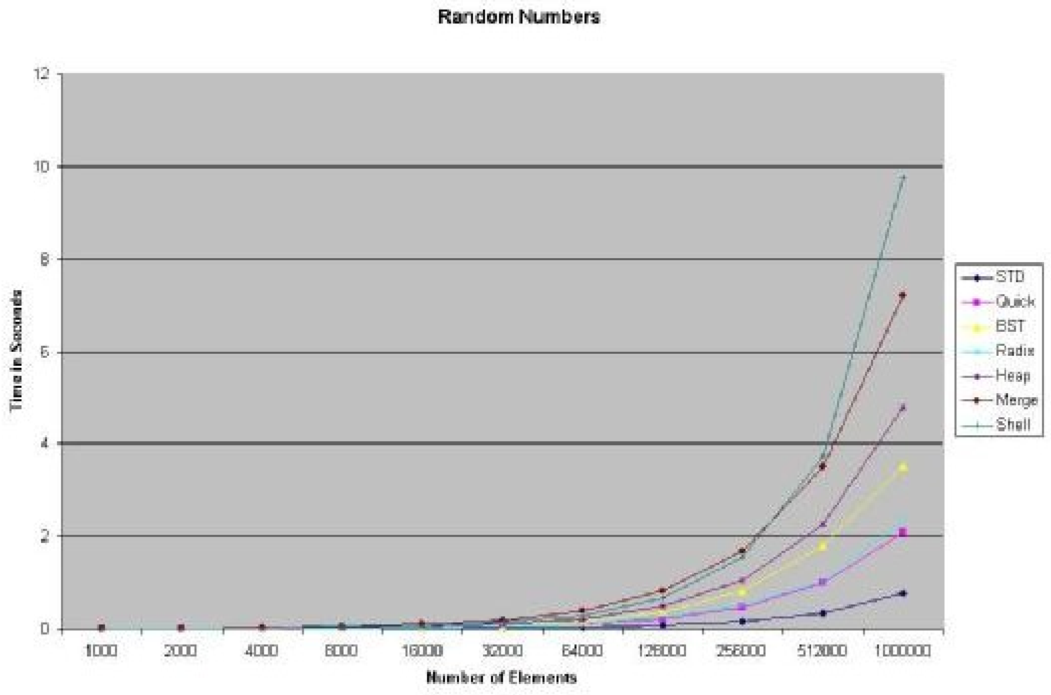

Figure 19. Algorithm efficiency comparisons

From the figure above, it can be seen that for a very small number of values, there is little to no difference in the running time between various sorting algorithms. Thus, insertion sort could have been picked in this case for ease of implementation. It is the next step in the algorithm that proves to be vital in the total running time of the dynamic state machine algorithm.

Depending on the user trigger expression, a variable number of states can be generated. Let us first ignore the states associated the valid bit and the start of packet bit. For example, if the user enters a trigger expression with a single data field in the first cycle, there will be a total of 3 states. A trigger state, a fail state (in the event the expected data is not seen in the first cycle), and a transition state where the data is seen the next state is set to the trigger state. Now, if there were instead 2 data fields in the first and second cycle, a whole new set of states is created. If the two fields are stored in the same MM register, we would have an identical case to the previous example. However, if the two fields are stored in separate MM registers, we would require an additional transition state (from cycle 1, where the first data field is seen, to cycle 2) and an additional fail state. Thus, for every new trigger field that is a part of the expression, in the worse case, 2 additional states are required. Thus, a simple equation can be derived to represent the maximum total number of states generated.

$$\text{Maximum number of states} = 2F + k$$

F – number of unique trigger fields in trigger expression

k – a constant representing fixed states that are part of every expression

For the new ELAM implementation, k consists of the valid bit valid state, valid bit invalid state, start of packet invalid fail state, and trigger state. Keep in mind in that a fair percentage of the cases, DFA minimization will reduce the number of states required.

In the worst case scenario, given the constraints mentioned previously, there will be 64 fields and 64 terms, evenly distributed across 4 cycles. Thus, the total number of states required is:

$$\text{Maximum number of states} = 2(64) + 5 = 133$$

This number is quite reasonable for any computation, and can be easily handled by the state generation algorithm. The only concern here is the memory allocated by hardware for storage of states. The maximum number of states that can be stored in ELAM hardware is currently only 128 states. Thus, if none of the states in the worse case scenario are found to be redundant, then the particular expression cannot be stored. In this case, an error message should be displayed to the user.

DFA minimization occurs in O(*nlogn*) time. Even using parallel implementations [56], significant gains to this efficiency are not obtained. Consider the figure below, which plots the runtime efficiency of the algorithm.

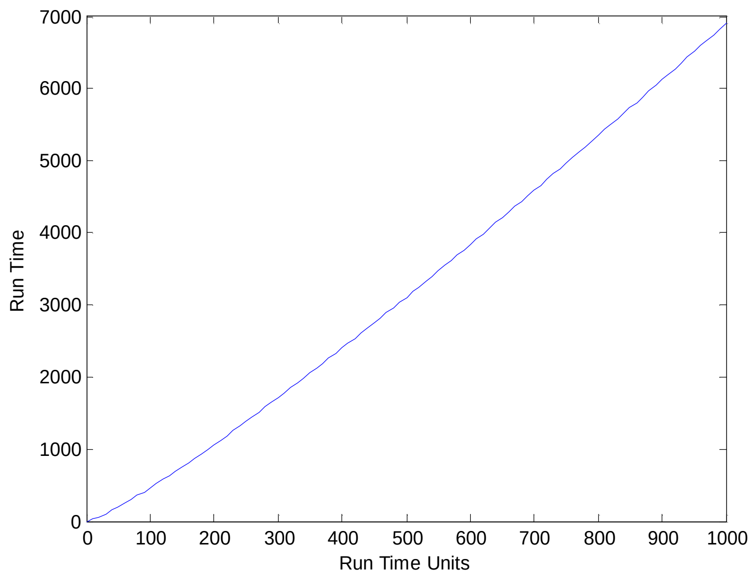

Figure 20. Runtime efficiency for DFA minimization

Looking at the first 1000 values (which is significantly less than the maximum upper bound found previously), it can be seen that the runtime is roughly linear. Hence, the DFA algorithm is suitable for minimizing the expanded set of states. Judging from the results generated, the runtime efficiency prior using a defined threshold is more than adequate for this project.

Chapter 6
Conclusions, Recommendations and Future Work

6.1 Summary

There are several major contributions to the ELAM that were presented in this thesis. As we saw, the original implementation of the ELAM greatly limited the ELAM's functionality and potential to only be able to trigger on a select few trigger statements. More specifically, the trigger statement had to be in product form. With the new changes proposed to the ELAM, the ELAM is now able to support a wide ranging series of trigger statements, allowing for logical operations AND, OR and NOT to be used in whatever way the user desires to form flexible trigger statements. Furthermore, the match and mask registers, which are vital resources, are minimized as much as possible.

Several different algorithms were presented to optimize the match and mask registers and have the ELAM support flexible trigger statements. In the first step, the trigger statement is treated as a Boolean logic expression. This expression is then minimized using the Quine-McCluskey algorithm, which is both systematic and fast. By setting an upper limit to the maximum number of trigger conditions, the Quine-McCluskey algorithm runs without disrupting the total ELAM execution time. Because users enter trigger conditions by hand, the upper limit of the total number of inputs for the Boolean expression minimization component is 64 variables and 64 terms. This value, for all practical cases, is far greater than any user requires. The resulting minimized expression is then used to set the match and mask registers required. After the match and mask registers have been properly set, we examine if there are any overlapping patterns in binary data and remove redundancies where possible. Finally, we generate a deterministic set of states from the new minimized expression. This set of states is then reduced using the methods of state reduction for DFAs. For each trigger expression that is entered, this entire process is run.

6.2 Future Work

There are several ways that the existing design can be improved to increase efficiency in handling flexible ELAM trigger statements and provide additional functionality. Some of these changes require hardware updates and reconfigurations.

The first area for future development for the ELAM is the ability to incorporate other logical operators such as XOR, XNOR, etc. The inclusion of this additional functionality should not be too difficult, as expressions containing other logic operators can always be represented using the 3 fundamental operators. Similarly, the ELAM should be able to support brackets. If the user wishes to enter an expression such as `(A + B)C! + D(EF + G)`, this expression should be expanded into a sum of products form and the current functionality can be used. Of course, to support brackets, the ELAM would require some type of parser. The parser would take trigger statements in any form, with or without brackets and convert the equation to a product of sums form. Such a parser would not be overly difficult, and would allow for support of equations in product of sums form.

Currently, the ELAM is only capable of supporting NOT conditions on variables. The ELAM currently does not provide any functionality to allow multiple variables and sub-sections of an equation to be negated. This leads us to another constraint of the ELAM design. If each trigger field is treated as a separate variable, there is a restraint on how flexible each trigger field expression can be. For example, the user is unable to enter a command where they want to trigger on a trigger field depending on the logic of the bits of that field. The user may want to only see that bits 0-10 are true, AND/OR/ bits 15-23 are false, etc. To be able to support these types of even more complex trigger statements, the trigger fields themselves would need to be broken down to a different format. In this case, the binary minimization algorithm may play in a greater factor in minimizing the match and mask registers used. However, given the current hardware implementation of the ELAM, such statements are not possible.

If the hardware on the ELAM were reconfigured, a lot of additional flexibility can be introduced. The biggest and possibly easiest change to introduce would concern the output of the match and mask registers. If each match and mask register were to return a vector indicating which bits in the registers matched and which ones did not, and if the input to the state machine were instead a matrix representing the information from all match and mask registers, the ELAM would be able to store fields regardless of the equality operator. This would greatly conserve register space. In the current design, it is possible that entire registers are allocated to store only one field. Related to this line of thought, the configuration of the match and mask registers should be evaluated as well. Instead of modeling each register as the information stored in a data packet, it may be possible instead to utilize compression techniques for storage. In this matter, both the data and match and mask registers would use the same compression and decompression techniques so that comparisons can still be made. These compression techniques may likewise conserve register space significantly.

Appendix A
Algorithm Examples

Logic Minimization Examples

A Simple Case

Let’s assume after mapping, the following logic expression is obtained:

`F = A′BC′D′ + AB′C′D′ + AB′C′D + AB′CD′ + AB′CD + ABC′D′ + ABCD′ + ABCD`

After logic minimization, the following statement (in sum of products form) is produced:

`F = AC + AB′ + BC′D′`

This expression is parsed based on terms that can potentially be grouped together and stored in 1 MM register:

`AC`, `A`, `B′`, `B`, `C′`, `D′`

Here there are 6 terms here, but only 4 unique fields, thus it is better to store each field individually in a register.

Terms that cannot be grouped

Let’s assume after logic minimization, the following expression is obtained:

`F = ABC′ + DEF + G`

Parsing yields:

`AB`, `C′`, `DEF`, `G`

No redundancy can be removed; there are 4 terms and 7 fields, so the terms are stored.

Binary Minimization Examples

Example 1

Note: Only data registers are shown. Mask registers to have corresponding bits set to 1.

Table 12. Binary minimization Example 1 initial problem

1	0	1						
				1	0	0	0	1
		1	1	1	0			
1	0	1	1	1	0			
1	0	1	1	1	0	0	0	1

Iterate and look for overlap:

`101` – start/end overlap: `101110` (bits match), `101110001` (bits match)

Update combine arrays: `101110 [101xxxxx]`

`101110001 [101xxxxx]`

`10001` – start/end overlap: `101110001` (bits match)

Update combine arrays: `101110001 [101x10001]`

`1110` – start/end overlap: `101110` (bits match), `101110001` (bits match)

Update combine arrays: `101110 [101110xx]`

`101110001 [101110001]`

`101110` – start/end overlap: `101110001` (bits match)

Update combine arrays: `101110001 [101110001]`

`101110001`– start/end overlap: none

Find redundant terms:

`101110`, `101110001`

Update MM register storage array for each bit sequence:

```
101 [10000000]

10001 [01000000]

1110 [00100000]

101110 [10100000]

101110001 [11100000]
```

Final data to store:

Table 13. Binary minimization Example 1 final results

1	0	1						
				1	0	0	0	1
		1	1	1	0			

Example 2

Table 14. Binary minimization Example 2 initial problem

1	0	1	1	1	0	0
1	0					
1	0	1	1	1	0	
		1	1	1	0	0

Iterate and look for overlap:

`1011100` – start/end overlap: `10` (bits match), `101110` (bits match), `11100` (bits match)

Update combine arrays: `1011100 [10xxxxx]`

`1011100 [101110xx]`

`1011100 [1011100x]`

`10` – start/end overlap: `101110` (bits match)

Update combine arrays: `101110 [10xxxxxx]`

`101110` – start/end overlap: none

`101110001 [101110001]`

`11100` – start/end overlap: none

Find redundant terms:

`1011100`

Update mm_list for each bit sequence:

`1011100 [11100000]`

`10 [10000000]`

`101110 [01000000]`

`11100 [00100000]`

Final terms to store:

Table 15. Binary minimization Example 2 final results

1	0					
1	0	1	1	1	0	0
		1	1	1	0	

Example 3

Table 16. Binary minimization Example 3 initial problem

1	0	1						
				1	0	0	0	1
		1	1	1	0			
1	0	0	1	1	0			
1	0	1	1	0	0	0	0	1

Iterate and look for overlap:

`101` – start/end overlap: `100110` (bits don't match), `101100001` (bits don't match)

`10001` – start/end overlap: `101100001` (bits don't match)

`1110` – start/end overlap: `100110` (bits don't match), `101100001` (bits don't match)

`100110` – start/end overlap: `101100001` (bits don't match)

`101100001`– start/end overlap: none

Find redundant terms:

None

Update mm_list for each bit sequence:

`101 [10000000]`

`10001 [01000000]`

`1110 [00100000]`

`100110 [00010000]`

`101100001 [00001000]`

Final terms to store:

Table 17. Binary minimization Example 3 final results

1	0	1						
				1	0	0	0	1
		1	1	1	0			
1	0	0	1	1	0			
1	0	1	1	0	0	0	0	1

Example 4

Table 18. Binary minimization Example 4 initial problem

1	0	1		1	0	0	0	1
1	0	0						
		1	1	1	0			
1	0	1	1	1	0			
1	0	1	1	1	0	0	0	1
1	0	0	1	1	0	0	0	1

In this case, we need to first expand the matrix to split up `101` and `10001`. After this, we proceed with the algorithm as usual.

Table 19. Binary minimization Example 4 intermediate results

1	0	1						
				1	0	0	0	1
1	0	0						
		1	1	1	0			
1	0	1	1	1	0			
1	0	1	1	1	0	0	0	1
1	0	0	1	1	0	0	0	1

Iterate and look for overlap:

`101` – start/end overlap: `101110` (bits match), `101110001` (bits match), `100110001` (bit don't match)

Update combine arrays: `101110 [101xxxxx]`

101110001 [101xxxxx]

10001 – start/end overlap: 101110001 (bits match)

Update combine arrays: 101110001 [101x10001]

100 – start/end overlap: 101110 (bits don't match), 101110001 (bits don't match), 100110001 (bits match)

Update combine arrays: 100110001 [100xxxxx]

1110 – start/end overlap: 101110 (bits match), 101110001 (bits match). 100110001 (bits don't match)

Update combine arrays: 101110 [101110xx]

101110001 [101110001]

101110 – start/end overlap: 101110001 (bits match), 100110001 (bits don't match)

Update combine arrays: 101110001 [101110001]

101110001– start/end overlap: 100110001 (bits don't match)

100110001– start/end overlap: none

Find redundant terms:

101110, 101110001

Update mm_list for each bit sequence:

101 [10000000]

10001 [01000000]

100 [00100000]

```
1110 [00010000]

101110 [10010000]

101110001 [11010000]

100110001 [00001000]
```

Thus, we finally store:

Table 20. Binary minimization Example 4 final results

1	0	1						
				1	0	0	0	1
1	0	0						
		1	1	1	0			
1	0	0	1	1	0	0	0	1

In this case, `101` and `10001` are split into 2 different registers. Since by doing this, we eliminate 2 registers, this is found to be more efficient than if `101` and `10001` were stored in the same register.

State Minimization Examples

Example 1

Suppose we have trigger expression `ABC′ + CD + ABE`, where `A`, `B` are in line 0, `C` is in line 1, `D` is in line 2, and `E` is in line 3.

```
ABC′ + CD + ABE
```

Now we construct a table of Necessary Permutation as follows:

Table 21. NCP table for Example 1

Field	**ID**	**MM**
A	1	2
B	1	2
C	3	3

D	6	4
E	10	5

The numerical minimized trigger expression is:

```
(1)(1)(-3) + (3)(6) + (1)(10)
```

After sorting:

```
(1)(1)(-3) + (3)(6) + (1)(10)
```

After removal of same ids:

```
(1)(-3) + (3)(6) + (1)(10)
```

Expanding the formula, we get terms:

```
(1)(-3)

(1)(3)(6)

(-1)(3)(6)

(1)(3)(6)(10)

(1)(3)(-6)(10)

(1)(-3)(6)(10)

(1)(-3)(-6)(10)
```

After reducing the terms we get:

```
(1)(-3)

(1)(3)(6)

(-1)(3)(6)

(1)(3)(-6)(10)
```

Which leads to the following transition and trigger states (X indicates that fail states are required):

Table 22. State Table for Example 1

Current State	**Match and Mask Registers**	**Next state**
0	0XXXXXXX	0
0	1XXXXXXX	1
1	10XXXXXX	0
1	111XXXXX	2 (X)
2	1XX0XXXX	99 (Trigger)
2	1XX1XXXX	3
3	1XXX1XXX	99 (Trigger)
1	110XXXXX	4
4	1XX1XXXX	5 (X)
5	1XXX1XXX	99 (Trigger) (X)
3	1XXX0XXX	6
6	1XXXX1XX	99 (Trigger) (X)

State minimization will follow. Given the complexity of the algorithm, it is difficult to write the entire sequence out in full. However, the once the algorithm is applied we should obtain a minimized set of states.

Example 2

Suppose we have trigger expression `ABCDEF + G'H'` where `A` is in line 0, `B`, `C`, `D`, `E`, `G` are in line 1, and `F` and `H` are in line 2.

Now we construct a table of Necessary Permutation as follows:

Table 23. NCP Table for Example 2

Field	**ID**	**MM**
A	1	2
B	3	3

C	3	3
D	3	3
E	3	3
F	4	4
G	8	5
H	9	6

The numerical minimized trigger expression is:

```
(1)(2)(2)(2)(2)(4) + (-3)(-5)
```

After sorting:

```
(1)(2)(2)(2)(2)(4) + (-3)(-5)
```

After removal of same ids:

```
(1)(2)(4) + (-3)(-5)
```

Expanding the formula, we get terms:

```
(1)(3)(4)(8)

(1)(3)(-4)(8)

(1)(3)(-4)(8)(-9)

(1)(3)(-4)(-8)(-9)

(1)(-3)(-4)(8)(-9)

(1)(-3)(-4)(-8)(-9)

(-1)(3)(-4)(8)(-9)

(-1)(3)(-4)(-8)(-9)

(-1)(-3)(-4)(8)(-9)

(-1)(-3)(-4)(-8)(-9)
```

After reducing the terms we get:

```
(1)(3)(4)(8)

(1)(3)(-4)(8)

(1)(3)(-4)(-8)(-9)

(1)(-3)(-4)(8)(-9)

(1)(-3)(-4)(-8)(-9)

(-1)(3)(-4)(8)(-9)

(-1)(3)(-4)(-8)(-9)

(-1)(-3)(-4)(8)(-9)

(-1)(-3)(-4)(-8)(-9)
```

Which leads to the following transition and trigger states (X indicates that fail states are required):

Table 24. State Table for Example 2

Current State	**Match and Mask Registers**	**Next state**
0	0XXXXXXX	0
0	1XXXXXXX	1
1	10XXXXXX	0
1	111XXXXX	2
2	1XX11XXX	3
3	1XXXX1XX	99 (Trigger) (X)
2	1XX10XXX	4 (X)
4	1XXXX1XX	99 (Trigger)
4	1XXXX00X	99 (Trigger) (X)
2	1XX00XXX	5
5	1XXXX10X	99 (Trigger) (X)
5	1XXXX00X	99 (Trigger) (X)
1	110XXXXX	6

6	1XX10XXX	7 (X)
7	1XXXX10X	99 (Trigger) (X)
7	1XXXX00X	99 (Trigger) (X)
6	1XX00XXX	8 (X)
8	1XXXX10X	99 (Trigger) (X)
9	1XXXX00X	99 (Trigger) (X)

State minimization will follow. Given the complexity of the algorithm, it is difficult to write the entire sequence out in full. However, the once the algorithm is applied we should obtain a minimized set of states.

Example 3

Suppose we have trigger expression `B + A`, where `A` is in line 1 and `B` is in line 3.

Now we construct a table of Necessary Permutation as follows:

Table 25. NCP Table for Example 3

Field	**ID**	**MM**
X	1	None
A	2	2
X	3	None
B	5	3

The numerical minimized trigger expression is:

```
(5) + (2)
```

After sorting:

```
(5) + (2)
```

After removal of same ids:

```
(5) + (2)
```

Expanding the formula, we get terms:

```
(1)(2)(3)(5)

(1)(-2)(3)(5)

(1)(2)
```

After reducing the terms we get:

```
(1)(-2)(3)(5)

(1)(2)
```

Which leads to the following transition and trigger states (X indicates that fail states are required):

Table 26. State Table for Example 3

Current State	**Match and Mask Registers**	**Next state**
0	0XXXXXXX	0
0	1XXXXXXX	1
1	10XXXXXX	0
1	11XXXXXX	2
2	1X0XXXXX	3
3	1XXXXXXX	4
4	1XX1XXXX	99 (Trigger) (X)
2	1X1XXXXX	99 (Trigger)

State minimization will follow. Given the complexity of the algorithm, it is difficult to write the entire sequence out in full. However, the once the algorithm is applied we should obtain a minimized set of states.

Appendix B
Source Code

The majority of the work done for this project is contained within a single file cap_elam.c. The file however, encompasses the majority of the functionality required for the ELAM. The three functions that serve as the containing body of the three major discussed sections are shown. Each function calls a variety of sub-functions and routines.

```
boolean
cap_minimize_trigger (trg_expr_t *trg_expr, trg_expr_t *min_expr,
                       trg_expr_t *storage_fields, table_t *table)
{
    char      **minterms;
    int         i, nrows, ncols;

    printf("\n*** Logic Minimization ***\n");
    /*
     * We need to first traverse the expr and set positions
     * in order to obtain a set of minterms
     */
    if (!cap_set_term_pos(trg_expr)) {
        cap_error_message("\n%s: cap_set_term_pos failed",
                             __FUNCTION__);
       return (FALSE);
    }

    nrows  = trg_expr->terms;
    ncols  = cap_get_num_uniq_terms(trg_expr);

    /*
     * Allocate 2D array to store minterms
     * Memory allocated here is cleaned in cap_free_trg_expr()
     */
    minterms = (char **)malloc(nrows * sizeof(char *));
    if (!minterms) {
        cap_error_message("\n%s: Memory not allocated for minterms",
                             __FUNCTION__);
        return (FALSE);
    }
    for (i = 0; i < nrows; i++) {
        minterms[i] = (char *)malloc(ncols * sizeof(char));
       if (!minterms[i]) {
           cap_error_message("\n%s: Memory not allocated for minterms[%d]",
                              __FUNCTION__, i);
           return (FALSE);
        }
    }
    /*
     * Map trigger expression to a set of minterms
     */
    if (!cap_get_minterms(minterms, trg_expr, nrows, ncols)) {
        cap_error_message("\n%s: cap_get_minterms failed",
                             __FUNCTION__);
       return (FALSE);
    }
    /*
     * Minimize minterms using Quine-McCluskey Algorithm
     */
    if (!cap_qm_minimization(minterms, trg_expr, min_expr, nrows, ncols)){
        cap_error_message("\n%s: cap_qm_minimization failed",
                             __FUNCTION__);
       return (FALSE);
    }
    /*
     * Group into storable terms
     */
    if (!cap_parse_storage_terms(min_expr, storage_fields)) {
        cap_error_message("\n%s: cap_parse_storage_terms failed",
                             __FUNCTION__);
       return (FALSE);
    }
    /* Check minterms and storage_fields both valid */
    if (!min_expr->terms || !storage_fields->terms) {
        cap_error_message("\n%s: expression length fault",
                             __FUNCTION__);
       return (FALSE);

    }

    cap_free_trg_expr(trg_expr, minterms, nrows);

    return (TRUE);
}
boolean
cap_bm_algorithm (elam_instance_t *elam,  void *data, void *mask,
                   trg_expr_t *storage_fields)
{
    int       i, j, k;
    int       index = 0, num_mm_used = 0, count = 0;
    bm_data_t bit_seq_list[MAX_EXP_MM_REGS];
    uchar    *d_addr, *m_addr;
    uchar     one_bit_mask[] =
              {BIT0, BIT1, BIT2, BIT3, BIT4, BIT5, BIT6, BIT7};
boolean   d_bit, m_bit, prev_m_bit = FALSE;
uchar     dchar = 0, mchar = 0;
int       num_mm_used_min;
int       mm_bit_width;

printf("\n\n*** Binary Minimization ***");
printf("\nregs: %d, width: %d\n",elam->mm_regs, elam->mm_reg_width);
mm_bit_width = elam->mm_reg_width * UCHAR_BIT_WIDTH;
/*
 * Initialize data structure
 */
for (i = 0; i < MAX_EXP_MM_REGS; i++) {
    for (j = 0; j < mm_bit_width; j++) {
       bit_seq_list[i].data[j] = FALSE;
       bit_seq_list[i].mask[j] = FALSE;
       bit_seq_list[i].combined_data[j] = FALSE;
       bit_seq_list[i].combined_mask[j] = FALSE;
   }
   for (j = 0; j < MAX_EXP_MM_REGS; j++) {
       bit_seq_list[i].mm_list[j] = FALSE;
   }
   bit_seq_list[i].redundant = TRUE;
}

/*
 * Set data structure to start binary minimization
 * We look for consecutive strings of enabled mask bits
 * and take the corresponding data values and fill the
 * data structure bm_data_t
 */
for(i = 0; i < storage_fields->terms; i++) {
    prev_m_bit = FALSE;
    for (j = 0; j < elam->mm_reg_width; j++) {
       d_addr = (uchar *)data + i*elam->mm_reg_width + j;
       m_addr = (uchar *)mask + i*elam->mm_reg_width + j;
       for (k = UCHAR_BIT_WIDTH - 1; k >= 0; k--) {
           d_bit = (*d_addr & one_bit_mask[k]) ? TRUE : FALSE;
             m_bit = (*m_addr & one_bit_mask[k]) ? TRUE : FALSE;
             index = j*UCHAR_BIT_WIDTH + (UCHAR_BIT_WIDTH - k - 1);

             bit_seq_list[num_mm_used].mask[index] = m_bit;

             if (!prev_m_bit && m_bit) {
                 bit_seq_list[num_mm_used].start = index;
                 bit_seq_list[num_mm_used].line = i;
             }
             if (m_bit) {
                 bit_seq_list[num_mm_used].data[index] = d_bit;
             }
             if (!m_bit && prev_m_bit) {
                 bit_seq_list[num_mm_used].stop = index - 1;
                 num_mm_used++;
             } else if (m_bit && (index == mm_bit_width - 1)) {
                 bit_seq_list[num_mm_used].stop = index;
                 num_mm_used++;
             }
             prev_m_bit = m_bit;
       }
   }
}

num_mm_used_min = num_mm_used;
cap_bm_remove_redundancy(bit_seq_list, num_mm_used, mm_bit_width);
for (i = 0; i < num_mm_used; i++) {
    printf("\nRedundant: %d\n", bit_seq_list[i].redundant);
   if (bit_seq_list[i].redundant) {
       num_mm_used_min--;
   }
   for (j = 0; j < MAX_EXP_MM_REGS; j++) {
       printf("%d", bit_seq_list[i].mm_list[j]);
   }
   printf("\n");
   for (j = 0; j < mm_bit_width; j++) {
       printf("%d", bit_seq_list[i].data[j]);
   }
   printf("\n\n");
   for (j = 0; j < mm_bit_width; j++) {
       printf("%d", bit_seq_list[i].mask[j]);
   }
   printf("\n");
}

if (num_mm_used_min >= storage_fields->terms) {
    printf("\nMMs required: %d", storage_fields->terms);
   if (storage_fields->terms > elam->mm_regs - 2) {
        cap_error_message("\n%s: max number of MMs exceeded",
                          __FUNCTION__);
       return (FALSE);
   }
    /* Update storage_fields mapping to mms */
    cap_bm_map_to_storage_fields(storage_fields, bit_seq_list,
                                      mm_bit_width, TRUE);
   return (TRUE);
}
printf("\nMMs required: %d", num_mm_used_min);
```

```
if (num_mm_used_min > elam->mm_regs - VLD_AND_SOP_MM) {
    cap_error_message("\n%s: max number of MMs exceeded",
                         __FUNCTION__);
   return (FALSE);
}
/*
 * Update data and mask pointers
 */
index = 0;
for (i = 0; i < num_mm_used; i++) {
    if (!bit_seq_list[i].redundant) {
       for (j = 0; j < mm_bit_width; j++) {
           if ((j != 0) && ((j % 8) == 0)) {
                 d_addr = (uchar *)data +
                     index*elam->mm_reg_width + count;
                 m_addr = (uchar *)mask +
                     index*elam->mm_reg_width + count;
                 count++;
                 *d_addr = dchar;
                 *m_addr = mchar;
                 dchar = 0;
                 mchar = 0;
             }
             dchar = (dchar << 1) | bit_seq_list[i].data[j];
             mchar = (mchar << 1) | bit_seq_list[i].mask[j];
       }
       if ((mm_bit_width % 8) != 0) {
           for (k = count*8; k < mm_bit_width; k++) {
                 dchar = dchar << 1;
                 mchar = mchar << 1;
             }
             d_addr = (uchar *)data + index*elam->mm_regs +
                 count;
             *d_addr = dchar;
             m_addr = (uchar *)mask + index*elam->mm_regs +
                 count;
             *m_addr = mchar;
       }
       count = 0;
       index++;
   }
}
/*
 * Update storage fields mapping
 */
cap_bm_map_to_storage_fields(storage_fields, bit_seq_list, mm_bit_width,
                                  FALSE);
    return (TRUE);
}
boolean
cap_sm_algorithm (trg_expr_t *min_expr, trg_expr_t *storage_fields,
                   asic_t *asic, elam_instance_t *elam, mm_reg_t *mm)
{
    np_elem_t   np_table[MAX_NUM_TERMS];
    int         i, j, last_id = 0, ins_row = 0;
    int         num_min_expr[MAX_NUM_TERMS][MAX_NUM_TERMS];
    int         num_min_expr_exp[MAX_NUM_TERMS][MAX_NUM_TERMS];

    /* Initialize necessary permutations table */
    for (i = 0; i < MAX_NUM_TERMS; i++) {
        np_table[i].id = 0;
       np_table[i].is_dummy = FALSE;
       np_table[i].trg_field = NULL;
    }
    /* Initialize numerical minimized expression matrix */
    for (i = 0; i < MAX_NUM_TERMS; i++) {
        for (j = 0; j < MAX_NUM_TERMS; j++) {
           num_min_expr[i][j] = 0;
           num_min_expr_exp[i][j]= 0;
       }
    }
    /* Create necessary permutations table */
    cap_sm_create_np_table(storage_fields, np_table);
    for (i = 0; np_table[i].trg_field; i++) {
        printf("\nid: %d, trg_field: %s", np_table[i].id,
              np_table[i].trg_field->field.name);
    }

    /* Create numerical minimized trigger expression*/
    cap_sm_create_num_expr(min_expr, np_table, num_min_expr);
    /* Sort the newly created expression */
    cap_sm_sort_num_expr(num_min_expr);
    /* Remove redundancies in the expression */
    cap_sm_remove_same_id(num_min_expr);

    printf("\n");
    for (i = 0; i < MAX_NUM_TERMS; i++) {
        for (j = 0;  j < MAX_NUM_TERMS; j++) {
           if (num_min_expr[i][j] != 0)
               printf("%d", num_min_expr[i][j]);
       }
       if (num_min_expr[i][0] != 0)
           printf("\n");
    }
```

```
/* Expand numerical expression to derive deterministic set of states */
for (i = 0; num_min_expr[i][0] != 0; i++) {
    last_id = 0;
    for (j = 0; num_min_expr[i][j] !=0; j++) {
       if (abs(num_min_expr[i][j]) > last_id) {
           last_id = abs(num_min_expr[i][j]);
       }
   }
    cap_sm_expand_expr(num_min_expr, num_min_expr_exp, np_table,
                          i, 0, 0, &ins_row, 0, last_id);
   ins_row++;
}

printf("\n");
for (i = 0; i < MAX_NUM_TERMS; i++) {
    for (j = 0;  j < MAX_NUM_TERMS; j++) {
       if (num_min_expr_exp[i][j] != 0)
           printf("%d", num_min_expr_exp[i][j]);
   }
   if (num_min_expr_exp[i][0] != 0)
       printf("\n");
}

/* Find overlaps in simplified equation */
cap_sm_find_overlap(num_min_expr_exp);

printf("\n");
for (i = 0; i < MAX_NUM_TERMS; i++) {
    for (j = 0;  j < MAX_NUM_TERMS; j++) {
       if (num_min_expr_exp[i][j] != 0)
           printf("%d", num_min_expr_exp[i][j]);
   }
   if (num_min_expr_exp[i][0] != 0)
       printf("\n");
}

/* Generate state machine */
if (!cap_sm_program_state_machine(num_min_expr_exp,
                                       np_table, asic, elam, mm)) {
    cap_error_message("\n%s: cap_sm_program_state_machine failed",
                         __FUNCTION__);
   return (FALSE);
}

/* Clean up allocated memory */
 for (i = 0; i < MAX_NUM_TERMS; i++) {
    if ((np_table[i].trg_field) && (np_table[i].is_dummy)) {
       free(np_table[i].trg_field);
```

```
           np_table[i].trg_field = NULL;
       }
     }

    return (TRUE);
}
```

Bibliography

[1] “Cisco Catalyst 6500 Series Switches Introduction,” Oct. 3, 2003. [Online]. Available: http://www.cisco.com/en/US/products/hw/switches/ps708/. [Accessed July 10, 2008].

[2] M. Loeser, “Nerdlunch: ELAM,” Cisco Systems Inc., San Jose, USA, 2008.

[3] “Cisco 6500,” Jul. 13, 2007. [Online]. Available: http://en.wikipedia.org/wiki/Catalyst_6500. [Accessed Aug. 1, 2008].

[4] Cisco Systems, ELAM Functional SPEC, San Jose, CA: Cisco Systems, 2003.

[5] S. Ahmad, R. N. Mahapatra, “An Efficient Approach to On-Chip Logic Minimization,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 15, No. 9, Sept. 2007.

[6] Cisco Systems, Catalyst 6500 Architecture White Paper, San Jose, CA: Cisco Systems, 2005.

[7] M. Karnaugh, "The Map Method for Synthesis of Combinational Logic Circuits," Transactions of American Institute of Electrical Engineers part I , Vol. 72, No. 9, pp. 593–599, 1953.

[8] V. P. Nelson, H. T. Nagle, B. D. Carroll, D. Irwin, *Digital Circuit Analysis and Design*, Boston, MA: Prentice Hall, 1995, pp. 234.

[9] R. L. Rudell, “Multiple-Valued Logic Minimization for PLA Synthesis,” EECS Department, University of California, 1986.

[10] “Boolean Algebra,” Oct. 3, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Boolean_algebra_(introduction). [Accessed Dec. 12, 2008].

[11] H. Liu, “Routing table compaction in ternary-CAM,” IEEE Micro, Vol. 15, No. 5, pp. 58-64, Jan/Feb. 2002.

[12] R. M. Karp, “Reducibility Among Combinatorial Problems,” in *Complexity of Computer Computations*. New York: Plenum, 1972, pp. 85–103.

[13] E. W. Weisstein, "NP-Complete Problem," *mathword.wolfram.com,* July 2002. [Online]. Available: http://mathworld.wolfram.com/NP-CompleteProblem.html. [Accessed: Sept. 10, 2008].

[14] G. Stitt, R. Lysecky, F. Vahid, “Dynamic hardware/software portioning: A first approach,” Proc. 40th Conf. Des. Autom., 2003, pp.250-255.

[15] L. Hames, B. Scholz, “Nearly optimal register allocation with PBQP,” in *JMLC,* 2006, pp. 346-361.

[16] F. M. Q. Pereira, J. Palsberg, “Register Allocation by Puzzle Solving,” in *PLDI’08*, 2008.

[17] “Quine-McCluskey Algorithm,” Nov. 11, 2008. [Online]. Avaiable: http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm. [Accessed Dec.12, 2008].

[18] “Karnaugh Map,” Sept. 14, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Karnaugh_map. [Accessed Dec.12, 2008].

[19] “Espresso Heuristic Logic Minimizer,” Dec.1, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer. [Accessed Dec. 12, 2008].

[20] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman, 1979, pp.247.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* 2nd ed., Boston, MA: McGraw, 2002.

[22] R. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, G. D. Hachtel, *Logic Minimization Algorithm for VLSI Synthesis*, Boston, MA: Luwer, 1984.

[23] T. Y. Wu, Y. L. Lin, “Register Minimization beyond Sharing among Variables,” in *32nd ACM/IEEE Design Automation Conference*, 1995.

[24] T. R. McCalla, “A Minterm-Ring Algorithm for Simplifying Boolean Expressions,” 1991.

[25] S. Devadas, “Optimizing Interacting Finite State Machines Using Sequential Don’t Cares,” in *IEEE Transactions on Computer-Aided Design*, Vol. 10, No.12, Dec. 1991.

[26] E. F. Moore, "Gedanken-experiments on Sequential Machines," *Automata Studies, Annals of Mathematical Studies*, Princeton, N.J.: Princeton University Press, 1956, pp. 129–153.

[27] R. Lysecky and F. Vahid, “On-chip Logic Minimization,” in *Proc. 40th Conf. Des. Autom.*, 2003, pp. 334-337.

[28] “Finite State Machine Optimization,” July 15, 1996. [Online]. Available: http://www.elo.utfsm.cl/~lsb/elo211/aplicaciones/katz/chapter9/chapter09.doc2.html. [Accessed Dec. 5, 2008].

[29] “NP-Complete Problems,” Nov. 11, 2008. [Online]. Available: http://en.wikipedia.org/wiki/NP-complete. [Accessed Dec. 14, 2008].

[30] S. Liu, W. Zhao, “Register Allocation Algorithms,” Michigan Technological University, 2004.

[31] D. R. Koes, S. C. Goldstein, “A global progressive register allocator,” in *PLDI*, 2006, pp. 204-215.

[32] “Finite State Machine,” Nov. 10, 2008. [Online]. Available: http://en.wikipedia.org/wiki/Finite_state_machine. [Accessed Dec. 20, 2008].

[33] L. George, A. W. Appel, “Iterated register coalescing,” TOPLAS, Vol. 18, No. 3, 1996, pp. 300-324.

[34] Z. Kohavi, “Switching and Finite Automata Theory,” McGraw-Hill, 1978.

[35] J. Hopcroft, “An nlog n algorithm for minimizing states in a finite automaton,” *Theory of machines and computations*, Orlando, FL: Academic Press, 1971, pp. 189–196.

[36] “Implication Table,” July 13, 2007. [Online]. Available : http://en.wikipedia.org/wiki/Implication_table. [Accessed Dec.19, 2008].

[37] G.H. Mealy, "A Method for Synthesizing Sequential Circuits," Bell System Tech, Vol. 34, pp. 1045–1079, 1995.

[38] M. C. Paul, S. H. Unger, “Minimizing the number of states in incompletely specified sequential switching functions,” in *IRE Transactions on Electronics Computers*, Sept. 1954.

[39] L.N Kannan, D. Sarma, “Fast heuristic algorithms for finite state machine minimization” in *Design Automation. EDAC., Proceedings of the European Conference on*, Feb. 1991, pp. 192 – 196.

[40] “Moore Reduction Procedure.” [Online]. Available: http://en.wikipedia.org/wiki/Moore_reduction_procedure. [Accessed Dec. 14, 2008].

[41] O. Garnica, J. Lanchares, J. M. Sanchez, “Finite State Machine Optimization Using Genetic Algorithms,” in *Genetic Algorithms in Engineering Systems: Innovations and Applications*, 1997.

[42] J. K. Rho; G.D. Hachtel, F. Somenzi, R. M. Jacoby, in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* Vol. 13, No. 2, Feb. 1994. pp. 167 - 177

[43] J. J. Yang, H. Shin, J. W. Chong, “New State Assignment Algorithms for Finite State Machines Using Look Ahead,” in *IEEE Integrated Circuits Conference*, 1991.

[44] “Quine-McCluskey Algorithm Java,” [Online]. Available: http://en.literateprograms.org/Quine-McCluskey_algorithm_(Java). [Accessed Nov. 20, 2008].

[45] A. Graselli, F. Lucio, “A method for minimizing the number of states in incompletely specified sequential networks” *in IEEE Trans. On Electronics Computeres,* EC-14, 1965. pp. 350-359.

[46] “DeMorgan’s Theorem,” [Online]. Available: http://en.wikipedia.org/wiki/De_Morgan's_laws. [Accessed Dec. 30, 2008].

[47] M . Burrows and D. Wheeler, “A block sorting lossless data compression algorithm”, in *Technical Report 124*, Digital Equipment Corporation, 1994.

[48] G. Cormack and N. Horspool, "Data Compression using Dynamic Markov Modeling", in *Computer Journal*, Vol. 30, No. 6, Dec. 1987.

[49] Nortel Networks, “Binary data compression/decompression apparatus and method of operation for use with modem connections,” Nortel Networks, [Online]. Available: http://www.freepatentsonline.com/6191711.html. [Accessed Oct 10, 2008].

[50] T. A. Welch, "A technique for high-performance data compression," in *Computer*, Vol. 17, pp. 8-19, June 1984.

[51] “Merge Sort,” [Online]. Available: http://en.wikipedia.org/wiki/Merge_sort. [Accessed Nov. 25, 2008].

[52] “Entropy Encoding,” [Online]. Available: http://en.wikipedia.org/wiki/Entropy_encoding. [Accessed Dec. 20, 2008].

[53] “Analysis of Algorithms,” [Online’. Available: http://en.wikipedia.org/wiki/Analysis_of_algorithms. [Accessed Dec. 1, 2008].

[54] L. S. Bobrow, A. A. Michael, *Discrete Mathematics: Applied Algebra for Computer and Information Science* 1st ed., Philadelphia: W. B. Saunders Company, Inc., 1974.

[55] M. Almeida, N. Moreira, R. Reis, “On the performance of automata minimization algorithms,” in *Proc. of DCFS’06*, pp. 58–69, 2006.

[56] A. Tewari, U. Srivastava, P. Gupta, *A Parallel DFA Minimization Algorithm*. Berlin: Springer Berlin, 2002.