Implementation trade-offs of Triple DES in the SRC-6e Reconfigurable Computing Environment

Osman Devrim Fidanci [1], Hatim Diab [1], Tarek El-Ghazawi [1], Kris Gaj [2] and Nikitas Alexandridis [1]
[1] George Washington University
[2] George Mason University

Abstract

In this paper, we overview the architecture and programming model of the SRC-6E Reconfigurable Computing Environment, and demonstrate, using Triple-DES cryptographic application, the trade-offs associated with the different possible implementations. In particular, using the SRC-6E high level programming interface we show that the underlying model allows the programmer to easily manage the tradeoffs between chip area and design speed. Impact of this high-level programming environment on the time-to-solution as well as ease of use and level of hardware design knowledge for application developers is assessed.

I. Introduction

The SRC-6E Reconfigurable Computing Environment is one of the first general-purpose reconfigurable machines combining the flexibility of traditional microprocessors with the power of Field Programmable Gate Arrays (FPGAs). In this environment, computations can be divided into those executed in software, using instructions of microprocessors, and those executed in reconfigurable hardware, using capabilities of modern FPGAs. The programming model is aimed at separating programmers from the details of the hardware description, and allowing them to focus on an implemented function. This approach allows the use of software programmers and mathematicians in the development of the code, and substantially decreases the time to the solution.

Despite this approach of shielding a programmer from the details of the hardware description, the SRC environment provides a programmer with the flexibility necessary to exploit various architectures that can be used to implement the same function. The choice among these architectures can be done at the level of a high-level language, such as Fortran, but it affects the way how function is implemented in hardware. In this paper, we investigate the capabilities of the SRC environment to implement the same function in several possible ways, with various tradeoffs among processing time and area of the circuit implemented inside of FPGA.

The function we chose to implement is an encryption algorithm, called Triple DES, one of the three standardized secret-key ciphers recommended for use in the U.S. government and in multiple commercial applications.

II. SRC-6E General Purpose Reconfigurable Computer

A. Hardware Architecture

SRC-6e platform consists of two processor boards and one Multi-Adaptive Processor (MAP™) board. MAP board has two user logic Xilinx® Virtex II XC2V 6000™ FPGAs. This way, SRC-6e system has 1:1 microprocessor – FPGA ratio. Processor boards are connected to MAP board through SNAP cards with 800MB/s transfer data rate. SNAP card plugs into DIMM Slot on microprocessor motherboard and provides interconnect of MAP to microprocessor [1].

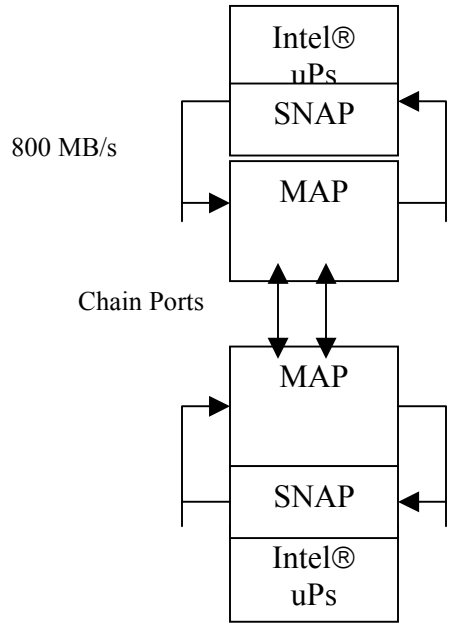

Figure 1: General Hardware Architecture of SRC-6E.

B. Programming Model

Compiling applications for the SRC-6E system involves more steps than the basic compilation model of conventional computer system. The more complicated process provides the user with greater flexibility to control compilation, the ability to stop or interrupt the compilation, and opportunities to add custom-logic or modifications into the compilation process. The compilation process includes compilation of the code that will execute on the MAP, and loading of the separately compiled binary files to create a single executable. The compilation system provides support for executing in the following modes; emulation mode, simulation code, and directly on the MAP hardware.

As indicated in Figure 2, there are two files to be compiled. One is compiled targeting execution on the Intel platform. The other files are compiled targeting execution on the MAP.

The file that contains the program that executes on the Intel processor and invokes routines that run on the MAP is compiled with an Intel target compiler to produce a relocatable object (.o) file. The file containing routines to execute on the MAP is compiled by the MAP Fortran compiler (mftn). mftn executes several distinct steps that result in several relocatable object files. The object files resulting from both the Intel® and MAP compilation steps are then linked with the MAP runtime libraries into a single executable file.

The resulting absolute binary file may then be executed on the Intel and MAP hardware , or run in the emulation mode. Environment variables determine the mode of execution.

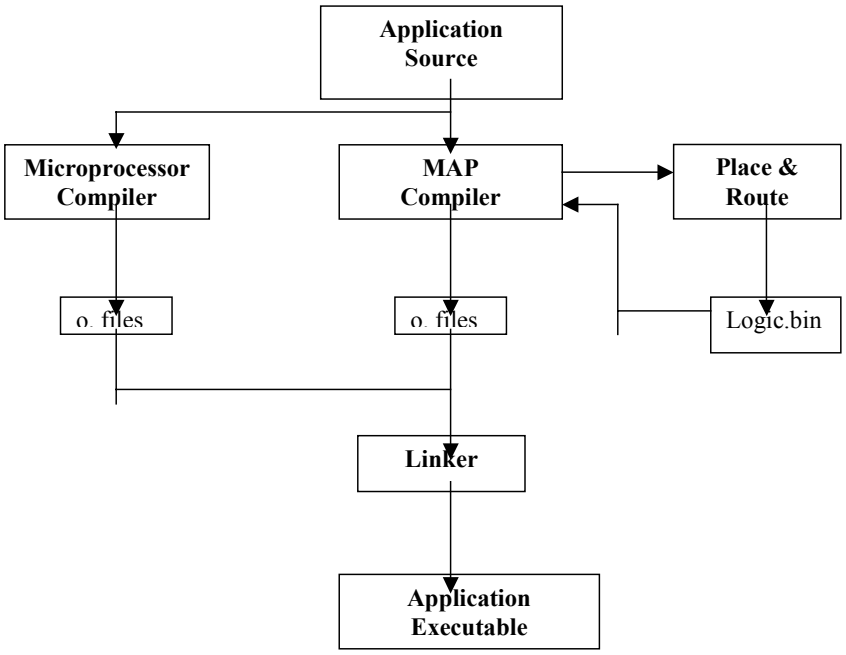

Figure 2: Overview of the SRC-6E Compilation Process

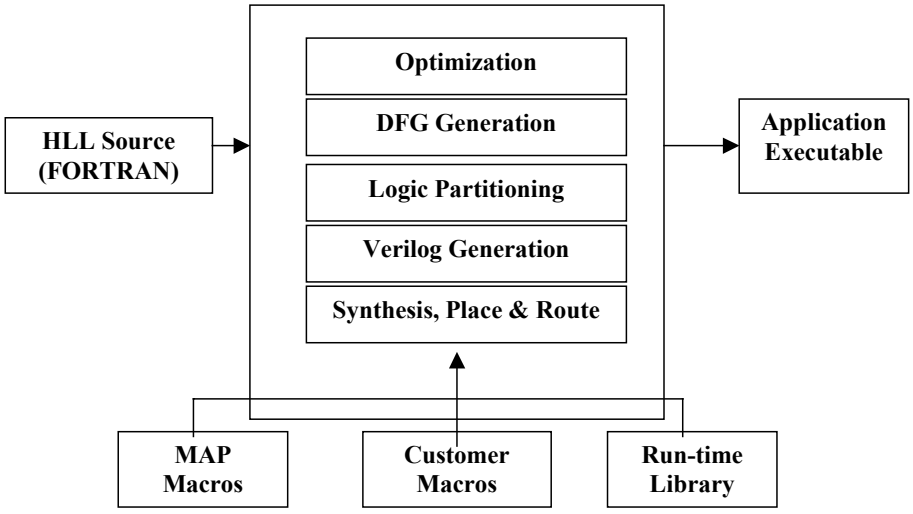

Figure 3: MAP Compilation Process

B.1 MAP Compilation Process

The MAP compiler translates procedures that have been modified for MAP execution into relocatable object files. The translation process has several steps, each performed by a distinct component of the MAP compiler.

The Optimization phase of the compiler performs language syntax and semantic analysis followed by classical scalar and loop optimization. At Design Flow Graph generation point in compilation, additional analysis is performed to verify procedures written for MAP execution. The dataflow graph represents the relationship between basic blocks of the procedure, with basic operations represented as nodes connected by the data that is input to or output from the nodes. Additional nodes are inserted for connecting blocks of graph and communicating data between blocks. Redundant nodes are pruned or optimized away[1].

The ComList Assembler creates a C function that contains the ComList template for the procedure being compiled for the MAP. The template is completed at runtime when the MAP subprogram's dummy arguments become associated with the actual arguments of the calling subprogram. The ComList executes on the MAPs control processor. It controls the execution of the subprogram on the MAP hardware, and controls the transfer of data between the MAP's On-board Memory (OBM) and System Common Memory (SCM). A user may choose to define their own ComList instructions to be assembled and included in the executable program.

The Verilog generator phase of compilation can be regarded as the "code generator" for the MAP. The Verilog generator translates the dataflow graph into its own internal format. After this translation, Verilog generator component produces synthesizable Verilog code.

Synthesis and Place & Route: Synplicity's Synplify Pro™ is the Verilog compiler being utilized in the MAP compilation path. Synplify Pro™ takes as input file synthesizable Verilog code generated by the Verilog generator, and a project file that is used for batch execution mode by mftn. Synplify Pro™ produces the constraint file and .edf file that will be the input for the place and route tools.

The place and route tools (Xilinx® Integrated Software Environment™) complete the bitstream creation process for the MAP. This phase of compilation performs the place and route function on the file output by the synthesis step, .edf file, including the binary files for macros invoked by procedure compiled for MAP execution. SRC-6e supports only one valid FPGA bitstream for any executable program.

Configuration Integrator: The configuration integrator is a small program that takes as input FPGA bitstream files. One file is a valid FPGA bitstream file, .bin file and the other is an indicator that the second bitstream is empty. The bitstreams are loaded into static structures contained in a C function.

B.2 Intel Processor Compilation

After the configuration integrator has completed, the MAP compilation process is finished however at this point there is no executable program generated that will be able to execute on the SRC-6e system. The MAP compilation process has produced C code files that must be compiled into object files and linked with the rest of the application, producing an application executable. Everything described in the previous sections has specifically targeted the code that will eventually execute on the MAP hardware. This next part of the compilation process takes as input the necessary C files from previous steps, which need to be compiled, linked and loaded on the Intel processor so that the output is an Intel executable.

C. User Macro Integration

The MAP compiler translates the source code's various basic operations into macro instantiations. Here, macro can be defined as a piece of hardware logic designed to implement a certain function. Since users often wish to extend the built-in set of operators, the compiler allow users to integrate their own macros into the compilation process. The macro is invoked from within the Fortran subroutine by means of a subroutine call. This call's arguments must be specified so that all incoming values precede all outgoing values [2].

In SRC-6E platform, macros can be categorized by various criteria, and the compiler treats them in different ways based on their characteristics. In the MAP compiler, four characteristics are particularly relevant:

A macro is "stateful" if the results it computes are dependent upon previous data it has computed or seen. In contrast "Non-stateful" macro computes values using only its current inputs; it has no memory of its past values [4]. A macro is "external" if it interacts with parts of the system beyond the code block in which it lives [4].

"Latency" is the number of clock cycles required between the time when a macro is activated with data until valid results appear. Since the pipelined inner loops generated by the MAP compiler use fixed delay queues to balance the paths through the loop, all macros for inner loops must have a fixed latency [4].

A "pipelined" macro is able to accept new data values on its inputs in every clock cycle. Since the MAP compiler produces pipelined inner loops, the macros that will be used in such loops must be capable of pipelined operation [4].

Three types of user macros can be used by MAP compiler: Pure functional, Stateful, and External. The chart below shows their characteristics:

Table 1: User macro characteristics

	Stateful	***External***	***Latency***	***Pipelined***
Pure Functional	No	No	Fixed	Yes
Stateful	Yes	No	Fixed	Yes
External	Yes or No	Yes	Variable	Yes or No

III. TRIPLE DES MACRO IMPLEMENTATION

A. Triple DES Algorithm

In this paper, Triple DES has been chosen as an algorithm to be implemented in the SRC-6E Reconfigurable Computing Environment, using three different implementation approaches. Actually, Triple DES by itself can be defined in a number of ways. In this paper, we will use a Triple DES version proposed by Tuchman that uses only two different keys [3]. This version follows an encryption-decryption-encryption (EDE) sequence:

$$C = E_{K1}[D_{K2}[E_{K1}[P]]]$$

There is no cryptographic benefit to use the decryption stage as the second stage. Nevertheless, it provides users of Triple DES with flexibility of communicating with older single DES user.

$$C = E_{K1}[D_{K1}[E_{K1}[P]]] = E_{K1}[P]$$

Triple DES with two keys is stronger and more reliable alternative to single DES. Triple DES is used in very popular Internet applications like PGP and S/MIME. Triple DES has also been adopted for use in key management standards ANS X9.17 and ISO 8732.

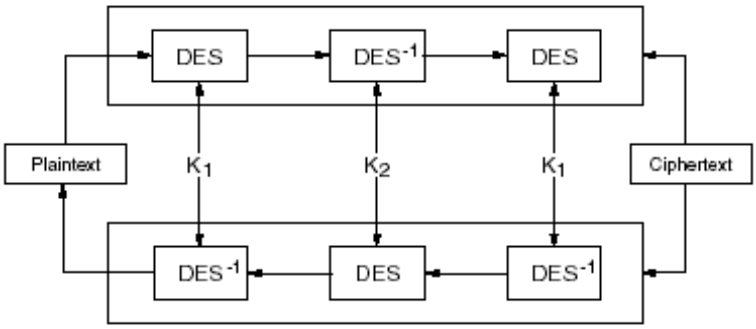

Figure 4: Block diagram for Triple DES algorithm with two keys

B. DES Encryption and Decryption Structure

DES encryption takes 64-bit plaintext (data) and 64-bit key (including 8 bits of parity) as inputs and generates 64-bit ciphertext (encrypted data).

As shown in Figure 5, as a first step, 64-bit plaintext passes through the Initial Permutation block, which re-arranges input bits. Then, data goes down through 16 identical blocks (rounds) with the different sub-keys (round keys) used in each round. Two 32-bit outputs from the sixteenth round are swapped. In the end, the output of the Swap transformation passes through the inverse of Initial permutation.

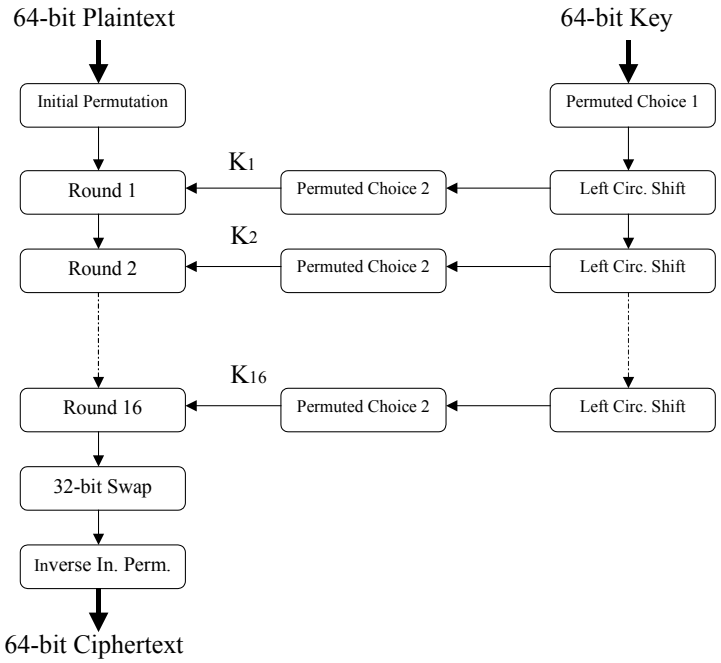

Figure 5 General architecture of DES algorithm

DES Decryption: DES decryption uses the same algorithm as DES encryption except that the round keys are used in the reversed order.

IV. Possible Different Implementations of Triple DES in SRC-6E

In this paper, three different possible implementations of Triple DES, named Case 1, Case 2, and Case 3 are examined.

In Case 1, Main Fortran file calls subroutine1.mf file three times. Subroutine1.mf file calls user-defined macro, which is provided as a DES.v Verilog file, only once.

In Case 2, Main Fortran file calls subroutine2.mf file only once from the main Fortran code. But, Subroutine1.mf file calls user-defined macro (DES.v) three times.

In Case 3, Main Fortran file calls subroutine3.mf file 3 times from the main Fortran code. Subroutine1.mf file calls just once user-defined macro named TriDES.v. that consists of three DES macro instantiated in the top level module called TriDES.v.

As shown in Figure 6, user can call user-defined macro hierarchically. According to these different programming schemes, MAP compiler maps different hardware architectures in the FPGAs and get different job execution durations. The result of these three different implementations is given in the following section considering area, speed, and job execution duration.

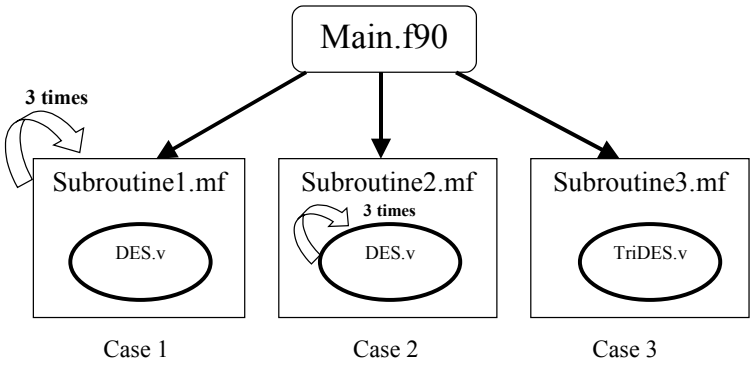

Figure 6: Three different ways of calling DES/Tri-DES macro from Fortran HLL

A. Implied Architectures

At each implementation, we also get different hardware mapped into target FPGAs. As given in the Figure 7, in Case 1; only one DES hardware block are implemented and it communicates with both OBM (On -Board Memory) and CM (Common Memory) for each macro call. In Case 2, entire Triple DES hardware that has consists of three DES sub-blocks implemented into FPGA. In Case 3, user Triple DES macro implemented as single block, which handles entire Triple DES function.

According to these different hardware implementations into FPGAs, user can get quite different area, speed and job execution time values. These different outputs are given in the next section.

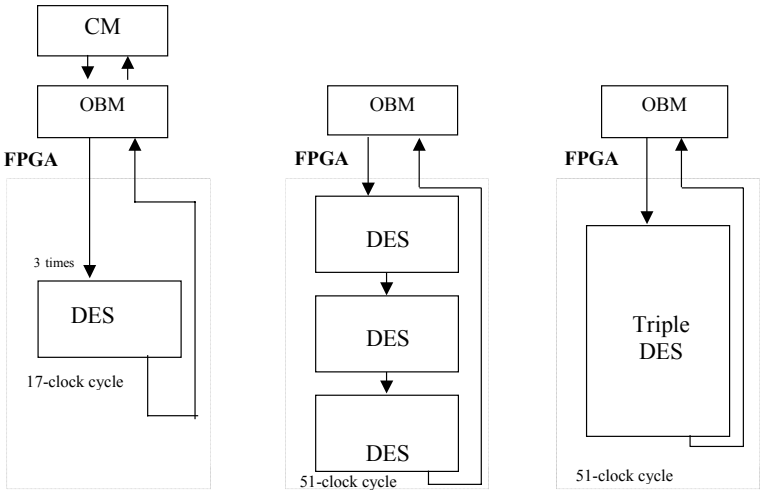

Figure 7: Implied architecture for Triple DES macro

V. IMPLEMENTATION RESULTS

Table 2 indicates maximum clock speed of user macro, CLB slice count (total equivalent gate count) and macro processing (job execution time) of three possible implementation of Triple DES macro.

Although the maximum clock speed for Case 1 is 102.3 MHz, for Case 2 is 100.8 MHz and for Case 3 is 101.8 MHz, the system clock frequency is only 100 MHz and all macros run at 100 MHz. As a result, there is no difference between Case 1, Case 2 and Case 3 from the clock frequency point of view.

The total equivalent gate counts for Case 2 and Case 3 are similar and larger than in Case 1. From the area point of view, Case 2 and Case 3 consume more than double of Case 1 does.

Total processing times (macro execution time) given in the table are calculated for 91 encryption blocks. Case 2 and Case 3 have the same macro processing time. Case 1 has the longest macro processing time as can be seen in Table 2.

Table 2: Implementation results for area, speed, processing time.

	Experiment (1)	**Experiment (2)**	**Experiment (3)**
Maximum Clock Speed (MHz)	102.3	100.8	101.8
CLB Slices (Tot. equiv. Gate count)	6,177 (163,835)	13,269 (382,927)	11,786 (359,635)
Total Processing Time (91 blocks of data)	4440	1820	1820

A. Timing Measurement and Calculation

A.1 Timing Measurement with "Timer Macro"

SRC-6E provides a system macro called "timer macro". Whenever the timer macro is compiled to the MAP, there is some invariant verilog code that is always included along with the code that is specific to the program being compiled. The invariant code contains a free running counter that starts up as soon as the bit stream has been loaded into the FPGA. The counter is being clocked by the same 100 MHz system clock as the rest of the user logic, so each count represents 10 ns of time.

There are two macro calls that access this counter. The first is 'start_timer', which simply resets the count to zero. The second is 'read_timer', which returns the current value of the counter. It is not necessary to call 'start_timer' at all if you use the approach of reading the timer before and after an event of interest, and taking the difference between the two counts.

Timer macro outputs are given in Table 2. Timing estimations/calculations are given in Table 3. As reader can easily see these values perfectly match.

Timing Estimation/Calculation: The macro processing time is estimated as given in Table 3. Pipelines stages, number of data blocks (n), Load/Store time and clock period are given. Using the parametric approach, user can calculate the estimated value of macro processing time for each individual case.

Table 3: Timing calculations for three different applications of Triple DES

Case 1	**Case 2**	**Case 3**
Pipeline stage: 17 Data blocks: 91 Load/Store time: 41 Clock period: 10ns n = 91 Estimated # of clock cycles for the execution: **[(17 + (n - 1)) + 41] x 3 = 444** Estimated total time: **444 x 10 = 4440 ns**	Pipeline stage: 51 Data blocks: 91 Load/Store time: 41 Clock period: 10ns n = 91 Estimated # of clock cycles for the execution: **[(51 + (n - 1)) + 41] = 182** Estimated total time: **182 x 10 = 1820 ns**	Pipeline stage: 51 Data blocks: 91 Load/Store time: 41 Clock period: 10ns n = 91 Estimated # of clock cycles for the execution: **[(51 + (n – 1)) + 41] = 182** Estimated total time: **182 x 10 = 1820 ns**

I/O overhead can be defined the ratio of the total Load/Store time to total macro processing time. Based on this definition, Table 4 indicates how I/O overheads get smaller and almost equal for all three cases when the number of data blocks increases.

Table 4: I/O overhead for three different implementation of Triple DES.

	Case 1	**Case 2**	**Case 3**
91 data blocks	27.7 %	22.5 %	22.5 %
501 data blocks	7.3 %	6.9 %	6.9 %
1001 data blocks	3.8 %	3.7 %	3.7 %
10001 data blocks	0.4 %	0.4 %	0.4 %

VI. CONCLUSIONS

Three different possible implementations of Triple DES on the SRC-6E platform give different results and provide user/programmer with the capability of trading speed for area.

According to the outcomes we obtain from three different implementations of Triple DES macro, Case 2, i.e., calling DES macro three times from the Fortran subroutine, and Case 3, i.e., calling Triple DES macro once from the Fortran subroutine are almost equivalent with the same execution time and 13% larger area for Case 2 because of the default interface between single DES modules.

Case 1, i.e., calling DES macro three times from the Fortran main file, and Cases 2 and 3 are very different with approximately two times smaller area for Case 1 and longer execution time caused by larger I/O overhead (communication between FPGA and on-board memory) and smaller utilization of the pipeline.

As a general conclusion, SRC Programming Model enables flexible choice of the hardware architecture used to implement required function. Implied architecture depends on the function (granularity) of the hardware description language macro and placement of macro calls in a high-level language program. Common features of all implemented architectures are deep pipelining and operational system clock frequency of 100 MHz. Overhead associated with the run-time communication between FPGA (User Chip) and on-board memory can be made negligible for processing of large amounts of data.

REFERENCES

[1] SRC-6E Programming Environment Guide, SRC Computers, Inc. 2002

[2] William Stallings, Cryptography and Network Security, Prentice Hall, 1999

[3] Tuchman, W. “Hellman Presents No Shortcut Solutions to DES.” IEEE Spectrum, July 1979

[4] Macro Integrator’s Manual v1.0, SRC Computers, Inc. 2002