A Reconfigurable Architecture for Searching Optimal Software Code to Implement Block Cipher Permutation Matrices

Elif Bilge Kavun[a], Gregor Leander[a], Tolga Yalçın[b]
[a]*Horst Görtz Institute for IT-Security, Ruhr-Universität Bochum, Germany*
elif.kavun,gregor.leander@rub.de
[b]*University for Information Science and Technology "St. Paul the Apostle", Ohrid, Macedonia*
tolga.yalcin@uist.edu.mk

***Abstract*—Programming in embedded systems has always been a challenge. Highly-constrained nature of embedded devices invalidates conventional coding practices. The whole practice turns into a skill game that heavily depends on the personal skills and experience of the programmer. Embedded security applications are no exceptions. Efficient software implementation of symmetric cryptography primitives such as substitution or permutation layers is a hard task and no systematic approach exists.**

In this study, we propose an efficient reconfigurable hardware architecture to find the most optimal code for the realization of block cipher permutation layers on embedded microcontrollers. The proposed architecture is highly parallel and realized on two Xilinx Virtex-6 XC6VLX240T FPGAs. It operates on a limited set of instructions pertinent to implementation of linear matrices. Predetermined number of instructions is executed in a pipelined manner and the resultant output register contents are checked either for match to a target matrix or for certain cryptographic properties. The realized architecture uses instructions from 8-bit AVR instruction set. However, it can easily be modified to work with instruction sets of different processors.

Using our parallel architecture, we have been able to find several good permutation layer matrices with branch number 4 that can be realized with only 8 instructions. We were able to search up to 11 instructions and cover matrices with branch number 6 as well.

***Keywords*-FPGA; Reconfigurable hardware architecture; Efficient software implementation; Symmetric cryptography; Permutation layer**

I. Introduction

Software coding has been considered by many as an art rather than an engineering practice. The same program written by two different coders almost never result in the same code size upon compilation. The situation gets worse in the coding for embedded devices, where assembly programming is widely preferred to high level languages. The size and cycle count of the program does not depend on the quality of the compiler anymore. It solely relies on the personal skills and experience of the programmer.

However, not every problem can be overcome by a skilled and experienced programmer. Specific applications, such as cryptography, requires also specialized knowledge. It is mostly hard to achieve this perfect combination. Especially, with the significant increase in the utilization of mobile and pervasive devices, the security of communications and its efficient implementations have gained paramount importance. Even more specifically, symmetric cryptography – one of the two main branches of cryptography, the other being asymmetric cryptography – has become more and more used on embedded systems.

The main building stone of symmetric cryptography, block ciphers, generally have a layer to provide non-linearity (substitution) and a layer to permute the bits (permutation). Embedded applications require block ciphers to be efficiently implemented both in hardware or software. Until now, designers have mostly concentrated on design of block ciphers, which are hardware-friendly, resulting in several "excellent" ciphers with "perfect" implementation figures on ASICs and/or FPGAs. It is hard to say the same for software. A minimal area substitution layer may result in several lines of code in software. The situation may get even worse for the permutation layer. A simple *wiring* in hardware corresponds to tediously long codes, and execution times.

In this work, our aim is to come up with high quality (in terms of code length, execution time, and cryptographic strength) software code regardless of the ability and knowledge of the coder – at least for the permutation (linear) layer of the cipher. Such *cheap* linear layers would help researchers in the design of software-oriented block ciphers.

Similar previous attempts targeted to reduce the number of cycles for substitution layer [1]. In their work, authors wanted to come up with the least number of instructions (in turn cycles) for 4-bit Sboxes and find the smallest possible 4-bit Sbox. However, they ran their search on a software platform, which significantly slowed down the search process.

In our work, we focus on the permutation (linear) layer and try to find permutation matrices giving least possible number of instructions. To this end, we make use of a hardware platform, which inherently possesses both parallelism and pipelining properties, that – as explained in the rest of paper – significantly accelerates the search process. We furthermore come up with simple and yet efficient architectures implemented on reconfigurable devices (FPGAs), which gives us the flexibility and ability to easily modify the search parameters and implement architectural modifications, unlike an ASIC platform.

II. Background

Our main target is to come up with the most compact software permutation (linear) layer implementation for a

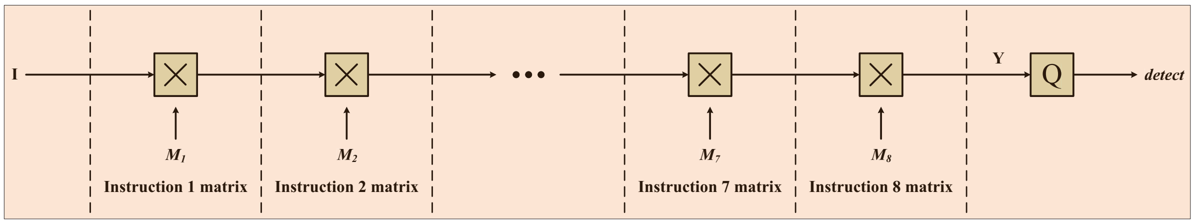

Figure 1. Instruction flow with *quality* check – an example with 8 instructions

Table I
LIMITED INSTRUCTION SET

Instruction	Operation	Description	Input $\rightarrow$ Output
EOR	$D \leftarrow D \oplus S$	Multiply destination matrix with identity matrix (I) and then add (EOR) source matrix.	-
MOV	$D \leftarrow S$	Multiply destination matrix with zero and add source matrix (equivalent to copying source matrix to destination).	-
ANDI	$D \leftarrow D \& S$	Mask certain bits of destination with an immediate mask (an example for mask $K = (11010111)_2$ shown in the next column).	$x_7x_6x_5x_4x_3x_2x_1x_0 \rightarrow x_7x_6 0 x_4 0 x_2x_1x_0$
SWAP	$D \leftarrow D_{3\dots0}, D_{7\dots4}$	Swap upper and lower nibbles of destination.	$x_7x_6x_5x_4x_3x_2x_1x_0 \rightarrow x_3x_2x_1x_0x_7x_6x_5x_4$
ROL	$D \leftarrow D \lll 1$	Rotate destination by 1-bit to left using carry register.	$x_7x_6x_5x_4x_3x_2x_1x_0c \rightarrow x_6x_5x_4x_3x_2x_1x_0cx_7$
ROR	$D \leftarrow D \ggg 1$	Similar to ROL, in right direction instead.	$x_7x_6x_5x_4x_3x_2x_1x_0c \rightarrow cx_7x_6x_5x_4x_3x_2x_1x_0$
LSL	$D \leftarrow D \ll 1$	Logical shift left, 0 is pushed in instead of carry.	$x_7x_6x_5x_4x_3x_2x_1x_0c \rightarrow x_6x_5x_4x_3x_2x_1x_0 0 x_7$
LSR	$D \leftarrow D \gg 1$	Logical shift right, LSL in right direction.	$x_7x_6x_5x_4x_3x_2x_1x_0c \rightarrow 0x_7x_6x_5x_4x_3x_2x_1x_0$
ASR	$D \leftarrow D \div 2$	Arithmetic shift right, divide destination by 2.	$x_7x_6x_5x_4x_3x_2x_1x_0c \rightarrow x_7x_7x_6x_5x_4x_3x_2x_1x_0$
CLC	-	Clear carry. May be needed in order to make sure carry is zero. Especially observed in manual coding attempts.	$x_7x_6x_5x_4x_3x_2x_1x_0c \rightarrow x_7x_6x_5x_4x_3x_2x_1x_0 0$

possible *software-oriented* lightweight cipher. For that purpose, we have analyzed the best figures in the literature, i.e. cycle counts of NSA lightweight ciphers SIMON and SPECK [2]. NSA used 8-bit AVR instruction set [3] in their implementations. In addition, many lightweight ciphers are also implemented using this instruction set [4], [5]. Therefore, we also decided to do the same for a fair comparison.

A naive back-the-envelope calculation of a possible implementation revealed that we have to come up with a linear layer of branch number (BN)[1] [6] 4-6 with a maximum cycle count of 7-9 cycles/byte for a 64-bit block cipher. Our manual attempts provided us matrices with $BN = 4$ and cycle counts of 9-11 cycles/byte. Clearly, in order to reach our targets, these figures have to be reduced.

Since manual coding did not achieve any better results, it was clear that we had search through all code combinations (for a given number of instructions) and see if we end up with a good matrix. This is, in fact, very similar to the *best Sbox (in software) search* implementation conducted by Ullrich et al. [1]. This is not an easy task, considering the tens of possible instructions in the instruction set of the target microcontroller. The search has to be limited only to instructions that can result in a "linear" operation.

In other words, the problem can be summarized as follows: Given N instructions[2] (say $N = 8$) and only linear operations (exclusive OR, rotate, shift, move, and masking) plus 2-3 temporary registers, what are code combinations that will result in a good matrix. More importantly, how can we go through all N-instruction combinations? The starting (initial) value will be the identity matrix, which is the mathematical equivalent of $Y = A$ in software. Then each linear instruction will act as a matrix multiplied with it, which will finally yield a certain value in the output register with the mathematical equivalent of $Y = M \times A$. Finally, the *quality* of this M will be checked. Here, by *quality*, we mean the BN and non-singularity of the matrix. The scheme can be depicted as shown in Figure 1.

Our limited set of instructions are listed in Table I. We also give the corresponding matrices for some of these instructions in Figure 2. Note that instructions ROL, ROR, LSL, LSR, ASR, and CLC introduce carry, therefore they are a bit different than EOR, MOV, ANDI, and SWAP. Here, an 8×8 matrix multiplication will not be enough, so we use

[1]Both linear BN and differential BN

[2]More importantly, clock cycles. It can differ from number of instructions, but each instruction of our limited instruction set costs 1 clock cycle. So, the number of instructions are the same with the clock cycle count in our case.

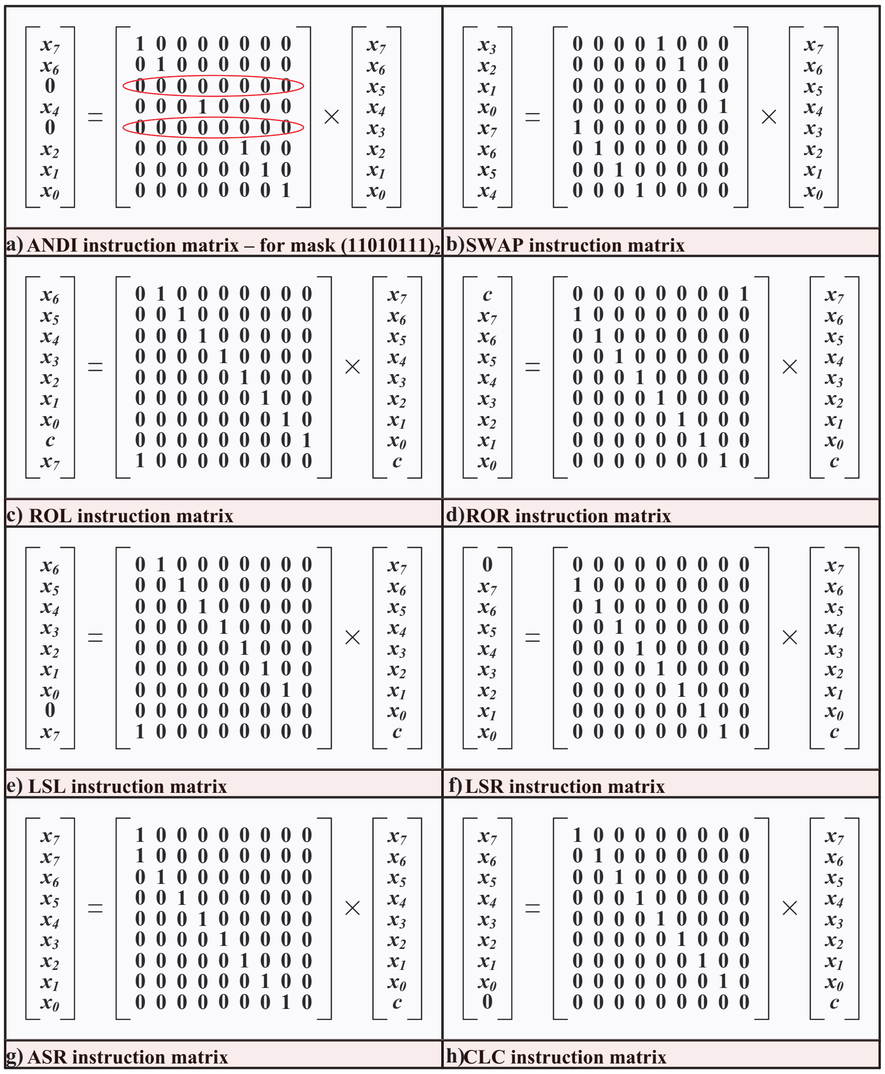
a) ANDI instruction matrix – for mask $(11010111)_2$
b)SWAP instruction matrix
c) ROL instruction matrix
d)ROR instruction matrix
e) LSL instruction matrix
f) LSR instruction matrix
g) ASR instruction matrix
h)CLC instruction matrix

Figure 2. Instruction matrices

a 9×9 matrix. It is also important to mention about ANDI instruction: As can be seen in Figure 2-a, zero rows of the matrix correspond to zero bits in the mask (marked with red circles). Basically, ANDI is equivalent to multiplication with a modified identity matrix whose selected rows are set to all zeros.

III. Proposed Search Model

In the overall structure, we should have a counter that would go through all possible combinations for each one of the targeted N instructions. Depending on these counter value(s), we have to decide on the instruction's equivalent matrix, the source and destination registers; and then apply them. Figure 3 explains this structure in graphical terms.

An important question is the number of source/destination registers. Clearly, one register is needed as the input/output (I/O), and 2-3 registers are needed for temporary storage. Although manual coding attempts revealed that 1-2 registers are sufficient, to be on the safe side, we use 3 temporary registers in our design. The next question is if all 4 registers are needed at every stage. This is not necessarily true. Starting point is the I/O register (which we denote as $R0$). At the end of the first instruction, the result can go to either $R0$ itself, or to the first temporary register, $R1$ (in case of a MOV instruction). In the following instruction, there can be up to two possible sources, $R0$ and $R1$. As a result, there can be at most three destinations, $R0$, $R1$ and $R2$. In the third instruction, which may have three possible sources, there may be up to four destinations, $R0$ to $R3$. Starting with the fourth instruction, we will limit to four sources and four destinations until the last instruction, in which we only the I/O register ($R0$) should remain as the sole destination. The scheme can be seen in Figure 4.

This basically means, we five different types of instruction modules should be defined, one for each of 1^{st}, 2^{nd}, 3^{rd},

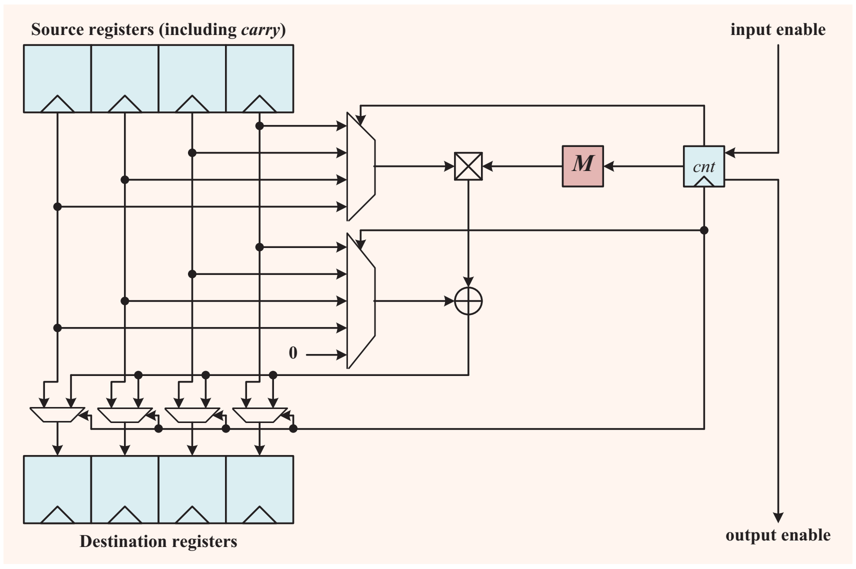

Figure 3. Overall structure

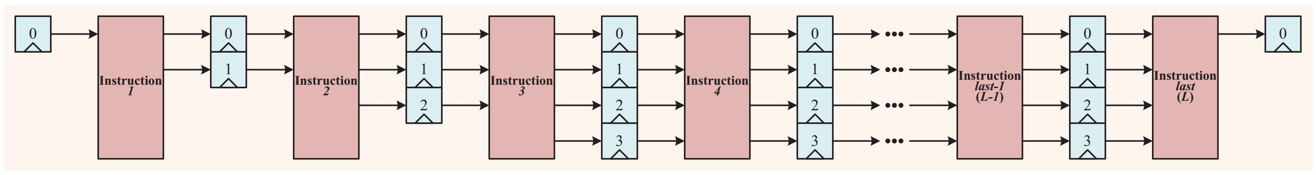

Figure 4. Overall instruction flow – showing sources and destinations at each stage

regular (intermediate), and *last* instructions.

Furthermore, not every instruction is possible at every cycle. For example, the first instruction cannot be ANDI because it destroys input bits. Likewise, it cannot be EOR, since a second operand does not exist yet. Similarly, there shouldn't be any bit *destructing* last instruction, like LSL, LSR, ASR, ANDI.

Among all instructions except for ANDI, EOR has the highest number of combinations. For each 4 possible destinations ($R0, \ldots, R3$), it can have 3 possible sources (e.g. 3 sources for $R0$ are $R1, R2, R3$), resulting in 12 possibilities. On the other hand, ANDI also has 4 destinations and no sources. However, for each destination, it has 254 possible masks from $(00000000)_2$ to $(11111110)_2$, resulting in 1020 different possibilities. This increases the complexity to a huge amount, in fact, to *unimplementable degrees*. In order to deal with this situation, we assume that we can have an instruction chain, which can have up to 3 ANDI instructions in total. In the implementation, these combinations will be hardcoded into each parallel running instruction chain.

In practice, there will be without and with-ANDI versions of each instruction module, excluding first and last instruction modules where ANDI is not permitted. In the following we list all the instructions permitted for each cycle with their possible source/destination (S/D) couples.

- **Instruction 1:**
  6 possibilities (Here we omit ROL & ROR as they are equivalent to LSL & LSR.)

  MOV (1S, 1D)
  SWAP (1S, 1D)
  LSL (1S, 1D)
  LSR (1S, 1D)
  ASR (1S, 1D)

- **Instruction 2:**
  $18 + 1$ (CLC) possibilities for ANDI-less case
  $19 + 2 \times 255$ possibilities for ANDI case

  EOR (2S, 2D)
  MOV (2S, 4D)
  SWAP (2S, 2D)
  ROL (2S, 2D)
  ROR (2S, 2D)
  LSL (2S, 2D)
  LSR (2S, 2D)
  ASR (2S, 2D)

- **Instruction 3:**
  $33 + 1$ possibilities for ANDI-less case

$34 + 3 \times 255$ possibilities for ANDI case

EOR (3S, 6D)
MOV (3S, 9D)
SWAP (3S, 3D)
ROL (3S, 3D)
ROR (3S, 3D)
LSL (3S, 3D)
LSR (3S, 3D)
ASR (3S, 3D)

- **Instruction $L-1$:**
  $48 + 1$ possibilities for ANDI-less case
  $49 + 4 \times 255$ possibilities for ANDI case

  EOR (6S, 12D)
  MOV (4S, 12D)
  SWAP (4S, 4D)
  ROL (4S, 4D)
  ROR (4S, 4D)
  LSL (4S, 4D)
  LSR (4S, 4D)
  ASR (4S, 4D)

- **Instruction L:**
  $6 + 1$ possibilities

  EOR (3S, 1D)
  SWAP (1S, 1D)
  ROL (1S, 1D)
  ROR (1S, 1D)

The corresponding computational complexity figures (in clock cycles) for a given instruction chain can be easily computed using the possibilities given above. For example, the best and worst case complexities and search times for 6-instruction chains with two ANDI instructions can be computed as follows:

- Worst case – I1, I2A, I3A, I4, I5, I6
  $\# \ cyc_w = 6 \times 529 \times 799 \times 49 \times 49 \times 7 = 2^{35}$
  $\rightarrow \ 213 \ sec \ @ \ 200MHz$
- Best case – I1, I2, I3, I4A, I5A, I6
  $\# \ cyc_b = 6 \times 19 \times 34 \times 1069 \times 1069 \times 7 = 2^{34.8}$
  $\rightarrow \ 155 \ sec \ @ \ 200MHz$

Clearly, these timing figures are very short, and will result in extremely fast searches. On the other hand, the same two ANDI cases for 9-instruction chains result in impractically high complexities and execution times:

- Worst case – I1, I2A, I3A, I4, I5, I6, I7, I8, I9
  $\# \ cyc_w = 6 \times 529 \times 799 \times 49 \times 49 \times 49 \times 49 \times 49 \times 7 = 2^{52}$
  $\rightarrow \ 25 \times 10^6 \ sec = 6964 \ h = 290 \ days \ @ \ 200MHz$
- Best case – I1, I2, I3, I4A, I5A, I6, I7, I8, I9
  $\# \ cyc_b = 6 \times 19 \times 34 \times 1069 \times 1069 \times 49 \times 49 \times 49 \times 7 = 2^{51.7}$
  $\rightarrow \ 18 \times 10^6 \ sec = 5066 \ h = 211 \ days \ @ \ 200MHz$

We therefore opt for using a subset of masks, i.e. $(11111110)_2$, $(11111100)_2$, ... , $(00000000)_2$, $(00000001)_2$, ... , $(01111111)_2$, resulting in a total of 15 masks, hence only 60 different combinations for 4 destinations. This certainly reduces the complexity to *computable* degrees.

IV. Register Model

During the execution of the instructions, the carry register should be taken into account in addition to regular register contents. In order to do this, we 8×9-bit states should be stored in the I/O and temporary registers, and 1×9-bit state in the carry register. The initial state of I/O register will be identity matrix concatenated with an all-zero column.

The initial states for the temporary registers should be "X". Likewise, the initial state of the carry register has to be set to all zeros. In the cases of instructions that do not affect carry bit (MOV, EOR, ANDI, SWAP), the last row of the matrix should be $(0 \dots 01)_2$ (no change in carry).

Figure 5 shows how each instruction step should look like with respect to explained register model (note that the figure is sketched only for an intermediate, $4 - to - 4$, instruction, the rest can be produced using it). The red dots represent additional delay elements used for pipelining in order to guarantee high speed operation. Next step is the matrix generator shown in this figure.

V. Matrix Formation

To be implementable in hardware, the instruction matrix should be formalized line-by-line, as shown in Table II.

Table II
Matrix formation

Instruction	Line 1	Line 2	Line 3	Line 4	Line 5	Line 6	Line 7	Line 8	Line 9
SWAP	08/0	04/0	02/0	01/0	80/0	40/0	20/0	10/0	00/1
MOV	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0	00/1
EOR	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0	00/1
ASR	80/0	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0
LSR	00/0	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0
LSL	40/0	20/0	10/0	08/0	04/0	02/0	01/0	00/0	80/0
ROR	00/1	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0
ROL	40/0	20/0	10/0	08/0	04/0	02/0	01/0	00/1	80/0
CLC	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0	00/0
ANDI	80/0	40/0	20/0	10/0	08/0	04/0	02/0	01/0	00/1

In this formulation, the first two digits correspond to hexadecimal representation of the most significant 8 bits of each line, where the last digit corresponds to binary representation of the least significant bit of each line. As can be seen, each instruction is represented in 9 lines, which corresponds to the multiplication with both the 8×9-bit destination register and the 1×9-bit carry register.

VI. Overall Architecture

Our resultant *reconfigurable* architecture is shown in Figure 6. There are two types of blocks to check the properties of the generated linear layer matrices. The first type checks a possible match to a target matrix. This is a simple comparator, which is not shown in the figure. The second type checks both the non-singularity (validity) and linear/differential branch numbers (cryptographic properties) of the matrix.

Non-singularity is checked via the determinant. For binary matrices, a non-zero determinant guarantees non-singularity. Finding the determinant is usually a problem for matrices over 4×4. However, since we are interested in the determinant of a binary matrix, we can simply use U of LU-decomposition.

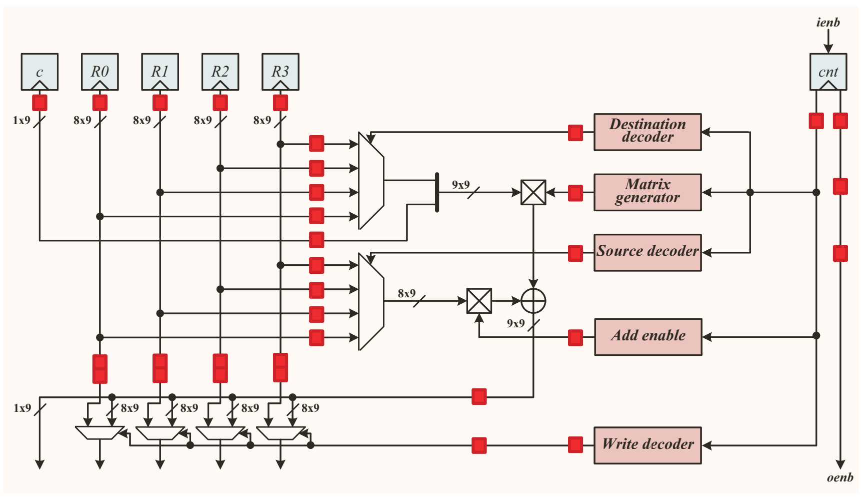

Figure 5. An *intermediate instruction* step

For that, we turn the matrix into an upper triangle. If it can be done, then the matrix is simply non-singular. This operation is realized via line-by-line elimination in a total of 8 steps. On the other hand, branch number can be found by exact realization of "minimum of the sum of input and output Hamming distances" algorithm in hardware, implemented as a $256 - to - 1$ binary tree.

While the determinant module occupies less than 1 % of the resources of Xilinx Virtex-6 FPGA we used, while the branch number calculator module is "huge" (4-5 %), compared to the determinant calculation module. However, on the positive side, we know that only less than 30 % of the binary matrices are non-singular. This means, we can share one branch number calculator between three determinant calculators (hence three instruction paths). This will require three FIFOs (at the output of each determinant calculator) and a simple circular sequencer. This scheme can be seen in Figure 7.

The sequencer will have a *priority counter* which initially checks for *valid* (not empty) from the first FIFO. If it does not exist, it checks the next two FIFOs. Every time a POP (read) is done by a FIFO (0, 1, 2), the priority counter is advanced to +1 of the FIFO index (1, 2, 0).

The "one-branch-number/three-determinant" calculator scheme occupies 6.6 % of all slices on the target FPGA, which corresponds to 2.2 % per instruction path. Considering 8 instructions and a maximum of 2 ANDI, we end up with $\binom{8-2}{2} = 15$ combinations. Having 8 paths per Virtex-6 FPGA makes: $\rightarrow\sim 18\%$ for qualification. As a result, we have to be able to fit the 8 paths into the remaining slices of the FPGA.

VII. Complexity Reduction

Until now, we have explained the overall architecture and implementation together with simple calculations for complexity and execution time. However, as seen in the 9-instruction example, the complexity can be quite high and practically undoable in a reasonable amount of time. Fortunately, total complexity can be reduced via reducing cycles, which can be done by bypassing operations on unused registers. This will also remove any false alarms due to use of *virtually cleared* registers, which is a natural result of connecting the inputs of registers to zero at the first, second, and third instruction blocks.

Since active registers always expand from $R0$ to $R1$ to $R2$, and to $R3$, an "active" counter can be kept at each step of the instruction chain. When the active counter is 0, it means that only $R0$ is active from the previous step. When it is 1, $R0 - R1$ are active, and so on. When 3 (maximum value), all four registers, $R0 - R3$, are active. The instruction module will use only the specified active registers as sources. None of the instructions, except MOV, has the capability to increase the number of active registers (even MOV can do this only in certain cases). For example, when active count input is 2, $R0, R1, R2$ are active. If MOV instruction transfers data from $R0 - R2$ to $R0 - R2$, the number of active registers will remain the same. But it is also possible to move from $R1$ to $R3$ (for example), in which case active register count will increase to 3. This will be the active count into to the next instruction module in the chain.

In this approach, the combination counter for instruction-1 (*cnt1*) is the fastest changing counter while *cntL* is the slowest. In terms of complexity, *cnt1* decides which operations should be executed via *cnt2*, and *cnt2* decides the same for *cnt3*, and so on.

This can be achieved via a simple change of the order of input/output enables (*ienb*/*oenb*), i.e. in the present scheme:

$oenb_1 \rightarrow ienb_2$, $oenb_2 \rightarrow ienb_3$,

and so on, meaning *cnt1* triggers *cnt2* when it reaches its maximum value; *cnt2* triggers *cnt3*, etc. In order to avoid any potential problems, we simply change it to:

$oenb_L \rightarrow ienb_{L-1}$, $oenb_{L-1} \rightarrow ienb_{L-2}$, $\ldots$, $oenb_2 \rightarrow$

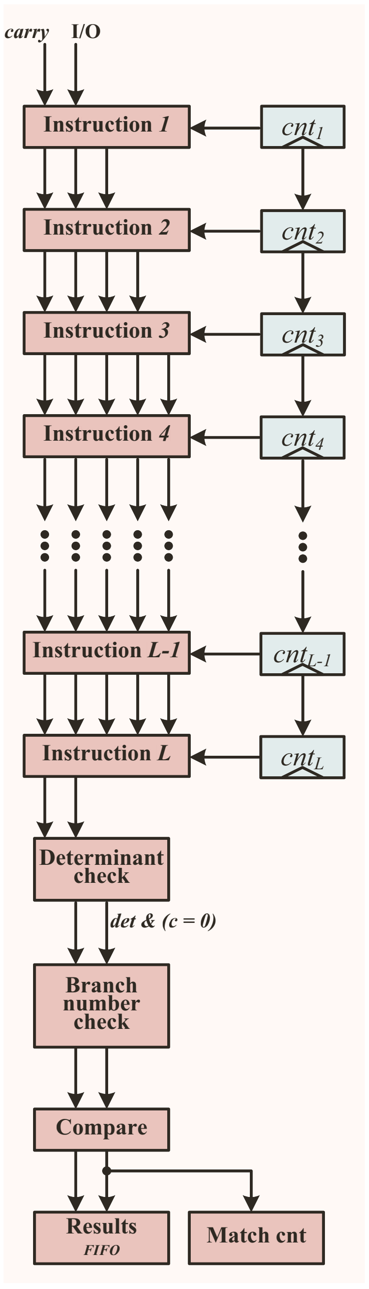

Figure 6. Reconfigurable “optimal linear layer” search architecture

$ienb_1$,

However, we still have one more problem. The sequence of *cnt2* depends on the active output count of *cnt1*. So, there is a trigger (or rather dependency) in the reverse direction as well. In summary, the domino structure of the current scheme cannot be used. Instead of local counters triggering each other, a global counter module, which also decodes active output counts, is needed.

It should work as follows:

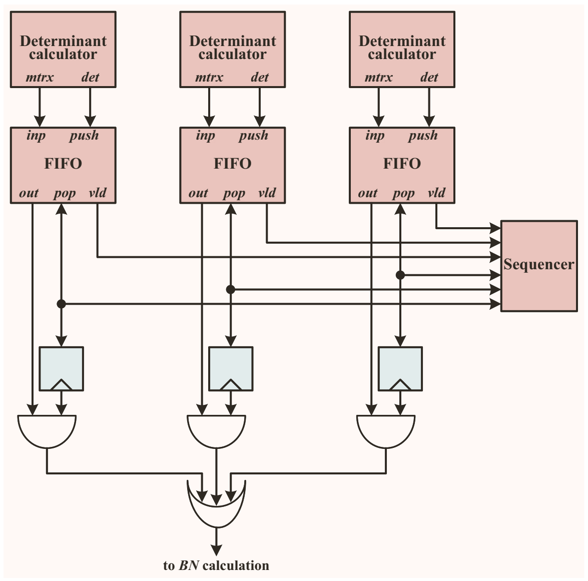

Figure 7. Determinant module sequencer

1) The last counter should be the fastest one. It should trigger the previous *cnt(L-1)*, *cnt(L-1)* should trigger *cnt(L-2)*, . . . , until *cnt2* triggers *cnt1*.
2) Initially all counters should start from 0.
3) Each counter should decide on its own “active output count” ($aout$) depending on the “active output count” from the previous counter and the instruction it currently executes. The only exception is when *cnt* is all zeros, i.e. when $cntN = 0$, $aout_N \leftarrow aout_{N-1}$ independent of Instruction N. This causes a worst case *aout* chain from 1 to L at the very beginning.

There is no clever solution to this worst case chain, except for reducing the overall operating frequency. Fortunately, it is sufficient to reduce the frequency to 100 MHz from the default on-board clock of 200 MHz by a simple frequency divider.

As a result of these optimization, we will end up with an instruction-dependent complexity tree rather than fixed complexity. The complexity tree from Instruction 1 to Instruction L is shown in Figure 8 for a 4 instruction example. Here, each node represents the number of active registers together with the possible number of paths (different instructions) from the previous node.

VIII. Results and Performance Evaluation

We performed our search using two Xilinx Virtex-6 XC6VLX240T [7] FPGAs, where we split the data in half and tried all the possibilities in parallel. Using an FPGA platform was indispensable, as we were trying for different number of instructions each time and we were even changing the definition of “instruction” modules (e.g. the case of *ANDI-less* and *with ANDI* instructions). We tried different combinations of the instructions (as explained in the previous sections)

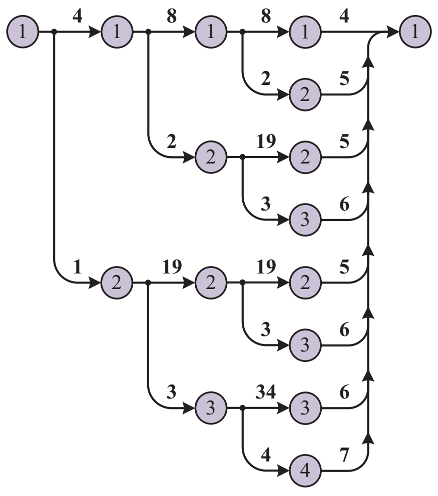

Figure 8. Complexity tree

for 7 to 11 instructions. The resulting matrices are checked either for match to a target matrix or for certain cryptographic properties.

Using our parallel architecture, we have been able to find several good permutation layer matrices with branch number 4, which can be realized with only 8 instructions. We were able to search up to 11 instructions and cover matrices with branch number 6 as well.

The main bottleneck in our search was to overcome with the problem of time complexity. This was a problem mostly for the cases using ANDI instruction, which was using 255 different masks. Trying ANDI for each instruction would end up in years (even worse). A practical solution to this was to limit the number of ANDI instructions used in the instruction list. We went up to 2 ANDI instructions for a reasonable execution time.

The linear layers found in our search results in better overall cycle count in software implementations of block ciphers. For example, considering the linear layer cost of PRESENT [8] (which is about 144 cycles), our matrices helps us to construct *cheaper* and cryptographically better linear layers for lightweight ciphers (as cheap as 36 cycles). These figures are even comparable with the newly proposed NSA ciphers SIMON and SPECKin terms of cycle count and code size. Given a cheap Sbox and cheaper key schedule, a block cipher using this linear layer might come close to or even go below the number of cycles of SIMON and SPECK.

IX. Conclusion and Future Work

In this paper, we proposed a hardware architecture to search for optimal permutation layer software implementations for cryptographic block ciphers. Using our proposed highly parallel and pipelined architecture, we searched for high *quality* matrices with smallest possible cycle counts and code sizes. We were able to try all possible combinations of a certain set of instructions in our architecture.

This work, to the best of our knowledge, is the first attempt to search for efficient software implementations using a hardware architecture. Although in our specific case, we have restricted ourselves to the specific application of search for linear layer codes, the reconfigurable nature of our hardware platform allows it to be modified for other applications as well. Furthermore, our highly parallel architecture has made it possible to search for various parameters in a very short amount of time, which would not be possible on a software search platform.

As a natural result of using FPGAs, in the future, we can extend our work to cover a larger spectrum of microcontroller instruction sets, including the highly popular PIC [9] and ARM [10] instruction sets.

Acknowledgements: The work was done when Tolga Yalçın was at Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany.

References

[1] M. Ullrich, C. D. Cannière, S. Indesteege, Ö. Küçük, N. Mouha, and B. Preneel, "Finding Optimal Bitsliced Implementations of 4×4-Bit S-boxes," in *Symmetric Key Encryption Workshop*, 2011.

[2] Ray Beaulieu and Douglas Shors and Jason Smith and Stefan Treatman-Clark and Bryan Weeks and Louis Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers," IACR ePrint Report 2013/404, Tech. Rep. 404, 2013, http://eprint.iacr.org/2013/404.

[3] "ATmega8 Datasheet," Atmel AVR. [Online]. Available: http://www.atmel.com/images/doc8159.pdf

[4] T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, F.-X. Standaert, and L. van Oldeneel tot Oldenzeel, "Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices," in *AFRICACRYPT*, ser. Lecture Notes in Computer Science, vol. 7374. Springer, 2012, pp. 172–187.

[5] AVRAES: The AES block cipher on AVR controllers, "http://point-at-infinity.org/avraes/."

[6] J. Daemen and V. Rijmen, "The Wide Trail Design Strategy," in *IMA Int. Conf.*, ser. Lecture Notes in Computer Science, B. Honary, Ed., vol. 2260. Springer, 2001, pp. 222–238.

[7] "Xilinx ISE Design Suite," Xilinx.

[8] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsø, "PRESENT: An Ultra-Lightweight Block Cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Springer LNCS, vol. 4727, 2007, pp. 450–466.

[9] "PIC 12-Bit Core Instruction Set," Microchip Technology.

[10] "The ARM Instruction Set," ARM.