SCAN Cryptoprocessor

Raghudeep Kannavara[1], Nikolaos G. Bourbakis[1,2] and Apostolos Dollas[3]
[1]*Wright State University, Joshi Engineering Building, Rm 467, Dayton, OH 45435*
[2] *AIIS Inc. Dayton, OH 45458*
[3]*Technical University of Crete, 73100 Chania, Crete, Greece*
(nikolaos.bourbakis@wright.edu)

Abstract

This paper presents a detailed architecture and instruction set of the SCAN cryptoprocessor. The SCAN cryptoprocessor is a modified SparcV8 processor architecture with a new instruction set to handle image compression, encryption, and information hiding based on the SCAN methodology. The modules for image compression, encryption, and information hiding are synthesized in reconfigurable logic and the results of the FPGA synthesis are presented. We propose to implement the above mentioned modules in an off-chip FPGA.

1. Introduction

Secure computing is gaining importance as computing capability is increasingly becoming distributed. Prevention of piracy and digital rights management has become very important; information security has become mandatory rather than an additional feature. The use of software-based security firewalls and encryption is not completely safe from determined hackers. This problem necessitates the need for information security at the hardware level, where cryptoprocessors assume importance. In this paper, we present a detailed architecture and instruction set of the SCAN cryptoprocessor. The SCAN cryptoprocessor is a modified SparcV8 processor architecture with a new instruction set to handle image compression, encryption and information hiding based on SCAN methodology. The modules for image compression, encryption, and information hiding are synthesized in reconfigurable logic. We propose to implement the above mentioned modules in an off-chip FPGA coprocessor. The IEEE compliant VHDL code is generated using Matlab version 7 Simulink HDL Coder. The synthesis of the reconfigurable architecture is done using the Matlab generated VHDL code and XILINX ISE ver. 9.1i. In this paper, we consider images of size 128x128 pixels with 8 bit depth, unless otherwise mentioned.

This paper is organized as follows. Section 2 gives a brief description of SCAN methodology and its applications. Section 3 gives a brief description of SparcV8 processor architecture and its features. Section 4 describes the SCAN cryptoprocessor architecture. Section 4.1 describes the FPGA coprocessor interface along with the new instruction set to handle the proposed hardware enhancements. Section 4.2, 4.3 and 4.4 describe the FPGA coprocessor for the image compression, encryption, and information hiding modules, respectively. Section 4.5 presents the results of XILINX ISE synthesis of the above mentioned modules. Section 5 provides the conclusion of this work.

2. SCAN Methodology

SCAN is a formal language-based, two-dimensional spatial accessing methodology which can represent and generate a large number of scanning paths easily. The SCAN language is defined by a grammar and has a set of basic scan patterns, a set of transformations, and a set of rules to compose simple scan patterns to obtain complex scan patterns. The rules for building complex scan patterns from simple scan patterns are specified by the production rules of the grammar of the SCAN language. The SCAN language has applications in information compression, information encryption, and information hiding. The basic scan patterns are shown in Fig. 1.

In this paper, FPGA coprocessor architecture capable of implementing the SCAN-based algorithms for image compression, encryption, and information hiding is presented along with new instructions to implement the same.

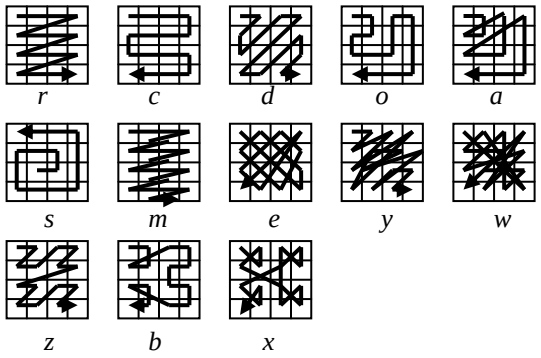

Figure 1. Basic SCAN Patterns

3. SparcV8 Processor

SPARC is a CPU instruction set architecture**,** derived from a reduced instruction set computer lineage. SPARC features a linear, 32 bit address space with a few and simple instructions, 32 bits wide. There are only three basic instruction formats, and they feature uniform placement of op-code and register address fields. Only load and store instructions access memory and I/O. The addressing modes include "register + register" or "register + immediate" and triadic register operations, which operate on two register operands (or one register and a constant), and place the result in a third register. There also exists a large "windowed" register file, which allows a program to see 8 global integer registers plus a 24-register window at any instant. These windowed registers act as a cache for procedure arguments, local values, and return addresses. The SPARC architecture also provides for a separate floating point register file, configurable by software into 32 single precision (32 bit), 16 double precision (64 bit), 8 quad precision registers (128 bit), or a mixture thereof. Trap handling is done through a vectored table that causes allocation of a new register window in the register file. In case of the delayed control transfer instructions, the processor always fetches the next instruction after a delayed control transfer instruction. It either executes the instruction or not depending on the control transfer instruction's "annul" bit. While executing a multiprocessor synchronization instruction, a single instruction performs an atomic read and then set memory operation; another performs an atomic exchange register with memory operation. The SPARC architecture defines a straightforward coprocessor instruction set, in addition to the floating point instruction set. The proposed SCAN cryptoprocessor is a modified SparcV8 architecture.

4. SCAN Cryptoprocessor

In this section, the SCAN cryptoprocessor architecture is described. The FPGA-based coprocessor architecture to implement image compression, encryption, and information hiding is presented along with the new instruction set to handle the off-chip coprocessor.

4.1. FPGA Coprocessor interface

The set of new instructions enable the operation of the FPGA based coprocessor to implement image compression, encryption, and information hiding. The coprocessor instruction set for SparcV8 includes support for a single, implementation-dependent coprocessor. The coprocessor has its own set of registers, the actual configuration of which is implementation-defined. Coprocessor load/store instructions are used to move data between the coprocessor registers and memory. To enable a coprocessor, the enable_coprocessor (EC) bit (Bit 13) in the Program Status Register (PSR) is 1. If a coprocessor is not present, the enable_coprocessor (EC) bit in the PSR is 0, and a coprocessor instruction generates a cp_disabled trap. All of the coprocessor data and control/status registers are optional and implementation-dependent. The coprocessor working registers are accessed via load/store coprocessor and CPop1/CPop2 format instructions. The Sparc architecture also provides instruction support for reading and writing a Coprocessor State Register (CSR) and a coprocessor deferred-trap queue (CQ).

A tightly coupled coprocessor architecture with dedicated data-path and a local memory system is required. The coprocessor interacts with the main processor through asynchronous FIFOs, which enable the coprocessor to be implemented with a clock frequency different from the processor system. The coprocessor is integrated into the processor subsystem using a dedicated coprocessor interface as shown in Fig. 2. A dedicated coprocessor interface provides a higher data rate between the processor cache and the coprocessor, and it provides special purpose instructions to handle communication with the coprocessor.

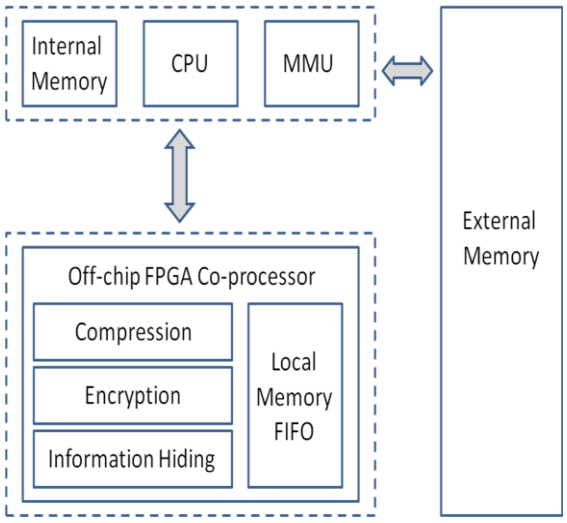

Figure 2. FPGA Coprocessor Interface

The CPop1 and CPop2 instruction format is as shown in Fig. 3, as given in the SparcV8 manual.

Opcode	op3	Operation
CPop1	110110	Coprocessor Operate
CPop2	110111	Coprocessor Operate

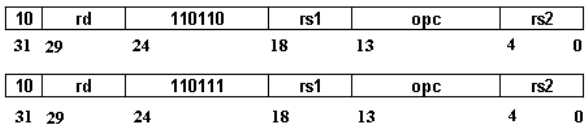

Figure 3. CPop1 & CPop2 Instruction Format

The suggested assembly syntax for the above instructions is as shown in Fig. 4.

Copop1 *opc*, $creg_{rs1}$, $creg_{rs2}$, $creg_{rd}$
Copop2 *opc*, $creg_{rs1}$, $creg_{rs2}$, $creg_{rd}$

Figure 4. Suggested Assembly Syntax

The new coprocessor instructions to enable SCAN based image compression, encryption, and information hiding are explained next. Data transfer across the external memory and the coprocessor memory is implemented by transferring four bytes of data simultaneously on the 32 bit data bus.

LOADIMG – The load image (*LOADIMG*) instruction moves the image file from external memory into the coprocessor memory.

STOREIMG – The store image (*STOREIMG*) instruction removes the image file from the coprocessor memory and stores it in the external memory.

LOADMSG – The load message (*LOADMSG*) instruction moves the message file from external memory into the coprocessor memory.

STOREMSG – The store message (*STOREMSG*) instruction removes the message file from the coprocessor memory and stores it in the external memory.

LOADKEY1 – The load key1 (*LOADKEY1*) instruction moves the first encryption key from the external memory into the coprocessor memory.

LOADKEY2 – The load key2 (*LOADKEY2*) instruction moves the second encryption key from the external memory into the coprocessor memory.

LOADSEED – The load seed (*LOADSEED*) instruction loads the seed needed by the random number generator of the encryption and decryption modules. This instruction is followed immediately by the seed value as an immediate operand.

CMPRS_IMG – The compress image (*CMPRS_IMG*) instruction compresses the image, and the result is available in the coprocessor memory. The image file should be present in the coprocessor memory before this instruction can be issued.

DCMPRS_IMG – The decompress image (*DCMPRS_IMG*) instruction decompresses the compressed image, and the result is available in the coprocessor memory. The compressed image file should be present in the coprocessor memory before this instruction can be issued.

ENCRYPT – The encrypt image (*ENCRYPT*) instruction encrypts the image, and the result is available in the coprocessor memory. The image file should be present in the coprocessor memory before this instruction can be issued.

DECRYPT – The decrypt image (*DECRYPT*) instruction decrypts the image, and the result is available in the coprocessor memory. The encrypted image file should be present in the coprocessor memory before this instruction can be issued.

INFOHIDE – The information hide (*INFOHIDE*) instruction embeds the message into the image file. The result of this operation is available in the coprocessor memory. Both the message file and the image file should be present in the coprocessor memory before this instruction can be issued.

INFOEXTRACT – The information extract (*INFOEXTRACT*) instruction extracts the embedded message from the cover image. The image file needs to be present in the coprocessor memory before this

instruction can be issued. The extracted message is available in the coprocessor memory.

CMPRS_ENCR – The compress and encrypt image (*CMPRS_ENCR*) instruction compresses the image and encrypts the compressed image. The result of this operation is available in the coprocessor memory. The image file should be present in the coprocessor memory before this instruction can be issued.

DECR_DCMPRS – The decrypt and decompress image (*DECR_DCMPRS*) instruction decrypts the input image and further decompresses the result of decryption. The result of this operation is available in the coprocessor memory. The image file should be present in the coprocessor memory before this instruction can be issued.

INFOHIDE_CMPRS – The information hide and compress (*INFOHIDE_CMPRS*) instruction embeds the message into the cover image and compresses the resultant image. Both the message file and the image file should be present in the coprocessor memory before this instruction can be issued.

DCMPRS_INFOEXTRACT – The decompress and information extract (*DCMPRS_INFOEXTRACT*) instruction decompresses the compressed image and extracts the message from the decompressed image. The result of this operation is available in the coprocessor memory. The compressed image file should be present in the coprocessor memory before this instruction can be issued.

INFOHIDE_ENCR – The information hide and encrypt (*INFOHIDE_ ENCR*) instruction embeds the message into the cover image and further encrypts the cover image. The result of this operation is available in the coprocessor memory. Both the message file and the image file should be present in the coprocessor memory before this instruction can be issued.

DECR_INFOEXTRACT – The decrypt and information extract (*DECR_INFOEXTRACT*) instruction decrypts the encrypted image and extracts the embedded message from the decrypted image. The result of this operation is available in the coprocessor memory. The encrypted image file should be present in the coprocessor memory before this instruction can be issued.

INFOHIDE_CMPRS_ENCR – The information hide, compress, and encrypt instruction (*INFOHIDE_CMPRS_ENCR*) embeds the message file into the cover image file and further compresses and encrypts this resultant image. The result of this operation is available in the coprocessor memory. Both the image file and the message file should be present in the coprocessor memory before this instruction can be issued.

DECR_DCMPRS_INFOEXTRACT – The decrypt, decompress and information extract (*DECR_DCMPRS_INFOEXTRACT*) instruction decrypts the encrypted image and further decompresses it and extracts the embedded message. The result of this operation is available in the coprocessor memory. The compressed, encrypted image file should be present in the coprocessor memory before this instruction can be issued.

In the following sections, the modules for image compression, encryption, and information hiding are described. The size of image considered is 128x128 bytes throughout this paper, unless otherwise mentioned.

4.2. Lossless Image Compression

The lossless image compression algorithm consists of four main steps. These steps are (1) scanning and prediction (2) scan path encoding (3) context modeling (4) arithmetic coding. The image is first partitioned into blocks of size $2^k \times 2^k$, $k \geq 2$. Each block is then scanned with various scan paths, and pixel values are predicted using different predictors along the scan paths, i.e., for each scan path, the sum of absolute values of prediction errors (the difference in pixel intensity values) and the number of bits needed to encode the scan path are computed. For each block, the scan path that minimizes the prediction errors and encoding bits is chosen as the best scan path of the block. Recursive hardware to find the best scan path is presented in the work by Kachris, Maniccam, Dollas and Bourbakis [1].

In this paper, a reconfigurable architecture for the *BlockError*() function, for scanning and prediction, and the *Context*() function, for context modeling, is presented. The *BlockError*() function is as shown next.

(*E, L*) = *BlockError*(*I, kt*)
Inputs: Image block *I*, scan path *kt*
Outputs: Sum *E* of absolute values of prediction errors along *kt*, Sequence *L* of prediction errors along *kt*
{
  Let $E = 0$, $L = \phi$
  Scan block *I* using scan path *kt* and at each pixel *p* do
  {
    Determine predictor at *p* and determine predictor neighbors {*q*, *r*}
    Let *s* be the pixel which was scanned before *p*
    If *q* and *r* are already scanned
      $e = p - (q + r)/2$
    Else
      $e = p - s$
    $E = E + Abs(e)$, $L = Append(L, e)$
  }
  Return (*E*, *L*)
}

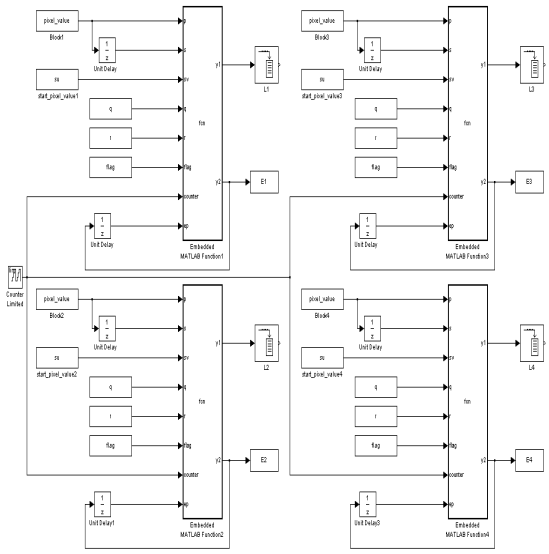

Figure 5. Simulink Model to find Prediction Errors

The 128x128 image is partitioned into four 64x64 blocks, which can be processed in parallel by the *BlockError*() function implemented on the FPGA. Matlab code in the embedded Matlab functions, along with the entire Simulink subsystem that is set up to find the prediction errors, as seen in Fig.5, is used to generate the IEEE-compliant VHDL code that can be used to synthesize the FPGA to implement error prediction.

All the prediction errors are buffered and the absolute sum of the prediction errors is written into a register on the FPGA. This register is continuously updated with the absolute sum of the prediction errors and the final value is obtained in this register at the 4096^{th} clock cycle since one clock cycle is required to find the prediction error at one pixel 'p'. Even though all four blocks fit into a single XCV600 with 2.3% logic utilization, there is a 92% IO utilization, which prevents us from duplicating further blocks on the same FPGA. The best scan path is the path with the least absolute sum of prediction errors. This operation is continuously repeated until all the basic SCAN patterns {C, D, O, S} including {0, 1, 2, 3, 4, 5, 6, 7} are exhausted. Multiple FPGAs with four blocks each to achieve maximum parallelization to exhaust all the basic SCAN patterns in 4096 clock cycles can be used. By making the circuit edge triggered, it is possible to reduce the number of clock cycles by half.

Once the best scan path and the sequence of prediction errors along the scan path are determined for each block, the best scan path is encoded and the sequence of contexts of prediction errors along the best scan path are computed using the *Context*() function, which is described next.

L = Context(I, P)
Inputs: Image block *I*, Scan path *P*
Output: Sequence *L* of contexts along *P*
{
Let $L = \varphi$
Scan *I* using scan path *P* and at each pixel *p* do
{
Determine predictor at *p* and determine context neighbors {*q*,*r*,*s*}
Let *u*, *v* be the first and second pixels which were scanned before *p*
If *q*, *r*, *s* are already scanned
$a = (|q - r| + |r - s|)/2$
Else
$a = |u - v|$
Let $b = 0$ if $0 \le a \le 2$, $b = 1$ if $3 \le a \le 8$, $b = 2$ if $9 \le a \le 15$, $b = 3$ if $a \ge 16$
L = *Append*(*L*, *b*)
}
Return *L*
}

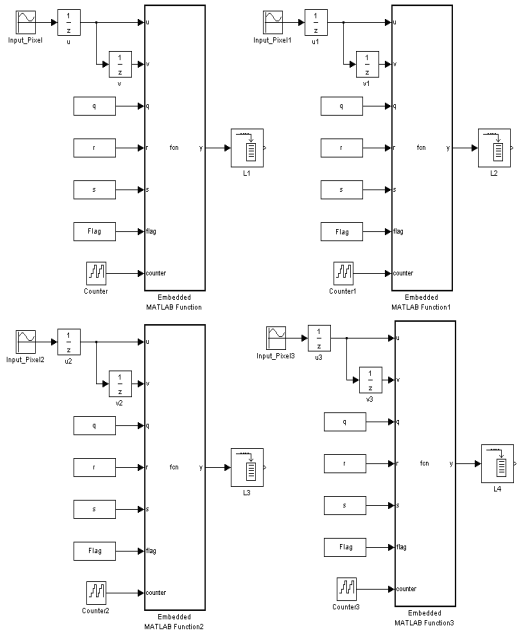

Figure 6. Simulink Model to find the Context

The values of 'q,' 'r,' and's' are previously buffered, and at every clock cycle, the context at pixel 'p' is calculated. FLAG is set to 1 implies that the 'q,' 'r,' and's' are already scanned else not. The context 'b' at 'p' is appended to a buffer. The method to find

predictor values, predictor neighbors and context neighbors (q, r, and s) is explained in [3]. Four blocks can be processed in parallel by parallelizing the *Context*() function. Matlab code to find the context, embedded in the Simulink subsystem, as seen in Fig.6, is used to generate the IEEE compliant VHDL code that can be used to synthesize the FPGA to find the context.

Once the sequence of prediction errors and the corresponding sequence of contexts are determined for each block, the prediction error sequence and the corresponding context sequence of the whole image are obtained by appending the error sequences of blocks together and context sequences of blocks together in the *C*0 order in which the blocks were processed. The prediction errors of the whole image are then encoded with context based adaptive arithmetic coding using *ArithmeticCode*() function which is described in [3]. Arithmetic coding is a method for lossless data compression. It is a form of variable-length entropy encoding that converts a string into another representation that represents frequently used characters using fewer bits and infrequently used characters using more bits, with the goal of using fewer bits in total.

After arithmetic coding is completed, the encoder sends a header information containing number of bits to the decoder and then sends the bits sequentially to the memory. The header of the compressed image provides information on the SCAN path chosen, the size of the image and the values of the first two pixels. This information is used to decompress the compressed image.

4.3. Image Encryption and Decryption

SCAN methodology offers a symmetric private key encryption. Image encryption using the SCAN methodology and its reconfigurable architecture is discussed in this section. The image encryption method is based on permutation of the pixels of the image and replacement of the pixel values. The permutation is done by scan patterns generated by the SCAN methodology. The pixel values are replaced using a simple substitution rule which adds confusion and diffusion properties. The permutation and substitution operations are applied in intertwined manner and iteratively producing an iterated product cipher. The encryption is done by *Encrypt*() function which is described next.

J = Encrypt(I, N, k_1, k_2, p, m)
Inputs: Image *I*, Image size $N \times N$ ($N = 2^n$, $n \geq 2$)
Encryption keys k_1 and k_2, Random seed integer *p*
Number of encryption iterations *m*
Output: Encrypted image *J*
{
Let *A*, *D*, *G* be two dimensional arrays of size $N \times N$ and let *B*, *C*, *E*, *F*, *R* be one
dimensional arrays of length $N \times N$
Generate $N \times N$ random integers between 0 and 255 using random seed *p* and assign to *R*
Copy *I* into *A*
Repeat *m* times
{
Read pixels of *A* using key k_1 and write into *B*
$C[1]=B[1]$, $C[j]=(B[j]+((C[j-1]+1)R[j])\text{mod}256)\text{mod}256$ for $2 \leq j \leq N \times N$
Read pixels of *C* and write into *D* using spiral key *s*0
Read pixels of *D* using diagonal key *d*0 and write into *E*
$F[1]=E[1]$, $F[j]=(E[j]+((F[j-1]+1)R[j])\text{mod}256)\text{mod}256$ for $2 \leq j \leq N \times N$
Read pixels of *F* and write into *G* using key k_2
}
Copy *G* into *J* and return *J*
}

The encryption key actually consists of four components, namely, the two user specified scan keys k_1 and k_2, the random seed integer *p*, and the number of encryption iterations *m*. These four encryption key components are known to both the sender and the receiver before the communication of encrypted image, via an untrusted media. The encryption algorithm uses four scan keys to increase the complexity of pixel rearrangement. The keys k_1 and k_2 are specified by the user as part encryption key. The other two keys spiral *s*0 and diagonal *d*0 are fixed as part of encryption algorithm. These two keys *s*0 and *d*0 are chosen because they have opposite directions of scanning as seen in Fig.1 and hence increase the complexity of pixel rearrangement caused by the user specified keys k_1 and k_2.

A good compromise between block size and SCAN keys is needed in order to achieve a combination of high encryption and high performance. In the architecture presented here, images of size 128x128 are partitioned into blocks of 64x64 for the purpose of encryption. The encryption process is iterative to improve the strength of encryption. Five iterations of the encryption process using different seeds for the random number generator provide a high encryption quality. The reconfigurable architecture of the substitution block of *Encrypt*() function is as shown in Fig.7.

The bitwise '*and*' with 255 is equivalent modulus operation with 256 and is easier to implement in hardware. Though 16 bit multiplier is used in the implementation, only 8 bits of the LSB contain the

needed value after the bit wise 'and' operation. Four such blocks of encryption to enable parallel encryption of the 4 image blocks are used. The reconfigurable architecture of the substitution block of decryption process is as shown in Fig. 8.

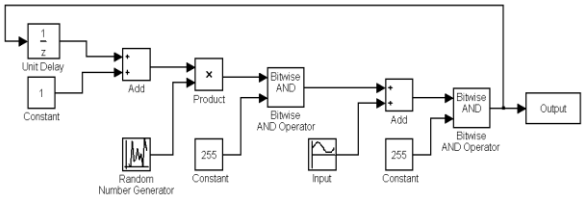

Figure 7. Simulink Model for Encryption (Pixel Value Substitution)

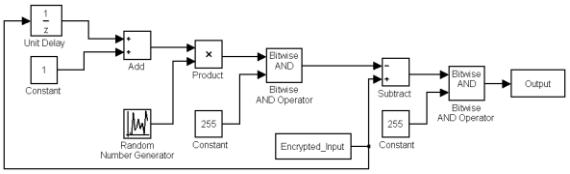

Figure 8. Simulink Model for Decryption (Pixel Value Substitution)

Similarly, four such blocks for decryption are used to enable parallel decryption. A clear description of the address generation units that interface with the encryption and decryption modules is presented by Dollas, Kachris and Bourbakis [6].

4.4. Information Hiding

The main idea of the image information hiding method is to identify the complex regions of the cover image and embed the secret data into those regions. The bits from the secret data are embedded into variable number of least significant bits of the pixels in complex regions depending on their complexity. The bits from the secret data are embedded into complex regions in random order determined by the secret SCAN key chosen by the user [3]. The embedding algorithm consists of four main steps. These steps are (1) complexity identification of cover image, (2) SCAN rearrangement of cover image and complexity matrix, (3) bit embedding of secret data into rearranged cover image, and (4) reverse SCAN rearrangement of rearranged embedded image. The algorithm to identify the complexity of the cover image chosen to embed the message is as described next.

Complexity(I, C, k1, k2, k3, k4)
Inputs: Cover image I
Threshold values 0<k1<k2<k3<k4<255
Output: Complexity matrix C
{
Let C[i][j] = 0 for 1≤i≤height(I), 1≤j≤width(I)

For (i = 2 to height(I)-1, i = i+2, j = 2 to width(I)-1, j = j+2)
{

$$d = \frac{1}{8}\sum_{k=1}^{8} abs(NB_k(I[i][j]) - NB_{(k+1) \bmod 8}(I[i][j]))$$

Let C[i][j] = 0,1,2,3,4 if 0≤d<k1, k1≤d<k2, k2≤d<k3, k3≤d<k4, k4≤d≤255, respectively
}
}

The reconfigurable architecture of the above function is synthesized on the Xilinx XCV600 FPGA. Matlab code in the embedded Matlab functions, along with the entire Simulink subsystem that is set up to identify the complexity of the cover image, is used to generate the IEEE-compliant VHDL code that can be used to synthesize the FPGA for identification of complexity of the cover image.

The embedding capacity i.e. ((embedded data size)/(cover image size))×100% is at most 12.5% because at most 4 least significant bits of at most 1 in 4 pixels of the cover image are used to embed data. In the bit embedding step, the secret data is embedded into the rearranged cover image. The secret data is a bit stream which can come from binary/grayscale/color/compressed image, text, sound, video, copyright information, and etc. The length of the bit stream is less than or equal to the total available embedding space (in bits), which was made known to the user after the complexity identification of the cover image. Bit embedding is done by *EmbedBit*() function which is described next.

EmbedBit(I, C, M)
Inputs: Rearranged cover image I
Rearranged complexity matrix C
Secret data bit stream M
Output: Bits of M are embedded into I
{
Let p = length(M)
For (i = 1 to height(I), j = 1 to width(I))
{
If p equals 0
Stop embedding
Else
k = minimum{C[i][j], p}
Replace k least significant bits of I[i][j] with next k bits from M
p = p - k
}
}

In the reconfigurable architecture the complexity values are initially calculated and stored in a FIFO buffer. Based on the buffered and rearranged complexity values, the bits in the image pixel are replaced with the message bits as described in the *EmbedBit*() function. Bit embedding is implemented using '*bitand*' operation to clear the LS bits of the image byte and '*bitor*' operation to embed the message data in the image byte, based on the calculated '*k*' values. Every cycle, one image pixel is embedded with the message based on the calculated '*k'* values. Matlab code to embed the message in the cover image based on the identified complexity is embedded in the Simulink subsystem, as seen in Fig. 9. This embedded Matlab function along with the Simulink subsystem is used to generate the IEEE-compliant VHDL code that can be used to synthesize the FPGA that implements message embedding.

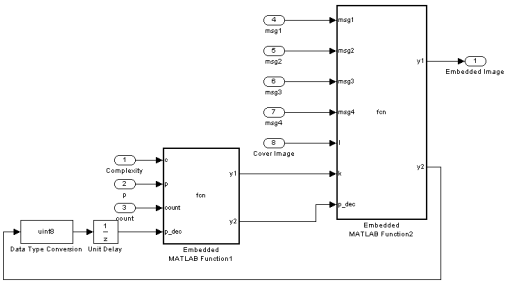

Figure 9. Simulink Model to Embed Message

In the bit extraction step, the bits of the secret data are extracted from the rearranged embedded image using *ExtractBit*() function, which is described next.

ExtractBit(I, C, n, M)
Inputs: Rearranged embedded image I
Rearranged complexity matrix C
Length n of secret data in bits
Output: Secret data M
{
For (i = 1 to height(I), j = 1 to width(I))
{
If n equals 0
Stop extracting
Else
k = minimum{C[i][j], n}
Extract k least significant bits of I[i][j] and append to M
n = n - k
}
}

A '*bitand*' operation is used on the image bytes to extract the message bits from the cover image. The complexity identification step is the same as used before in the *EmbedBit*() function. The complexity matrices are the same because the 8-neighbors of each embedding candidate pixel are left unchanged. Note that the length (in bits) of the secret data is needed by *ExtractBit*() function, and this length is sent to the receiver in the header of the embedded image. Matlab code to extract the message in the cover image based on the identified complexity is embedded in the Simulink subsystem, as seen in Fig. 10. This embedded Matlab function along with the Simulink subsystem is used to generate the IEEE-compliant VHDL code that can be used to synthesize the FPGA that implements message extraction.

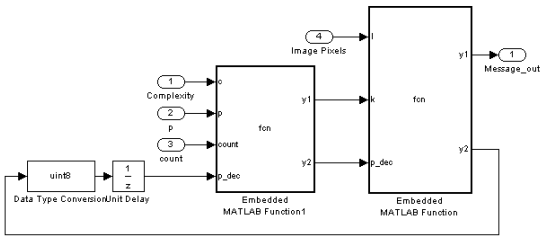

Figure 10. Simulink Model to Extract Message

If the presence of secret data is detected, the data cannot be read or understood without knowing the embedding SCAN key, because the data is embedded in a random order specified by the SCAN key. Therefore, the method is robust against passive attacks, which are aimed at only detecting and reading the secret data. But the method is not robust against active, attacks which are aimed at destroying the secret data without reading it. Since an attacker knows possible pixel locations where secret data can be embedded, the attacker can change the four least significant bits of all those pixels, thereby destroying the secret data.

4.5. Results of FPGA Synthesis

The results of XILINX ISE synthesis of FPGA modules to support the image compression, encryption, and information hiding are tabulated in this section. Table. 1 presents the synthesis results of compression module. Table. 2 presents the synthesis results of the encryption and decryption modules. Table.3 presents the synthesis results of the modules to embed and extract a message in the cover image. XCV600 from

the Virtex family was selected for the purpose of FPGA synthesis.

Table. 1

Module	Slices of Virtex XCV600	Percent occupied	Estimated maximum operating frequency
Block Error Module (four blocks)	164	2.3	62 MHz
Context Module (four blocks)	148	2.14	349 MHz

Table. 2

Module	Slices of Virtex XCV600	Percent occupied	Estimated maximum operating frequency
Encrypt Module (one block)	33	0.47	58 MHz
Decrypt Module (one block)	33	0.47	58 MHz

Table. 3

Module	Slices of Virtex XCV600	Percent occupied	Estimated maximum operating frequency
Embed Module	40	0.57	75 MHz
Extract Module	26	0.37	78 MHz
Complexity Module	80	1.15	N/A

Table. 4 provides the number of clock cycles required by the compression modules, i.e. the Block Error module and the Context module. Each block requires 4096 clock cycles to find the prediction error for one scan path. The basic scan paths {C, D, O, S} including {0, 1, 2, 3, 4, 5, 6, 7} are exhausted to find the minimum prediction error, processing 4 blocks in parallel. For the Context module, 4 bytes are transferred in parallel on the 32 bit data path to achieve parallelism in processing.

Table. 4

Module	Number of clock cycles
Block Error Module	131072
Context Module	4096

Table. 5. provides the number of clock cycles required for the Encryption module and Decryption module. The size of the images considered is 128x128 bytes of 8 bit depth. These images are partitioned into four 64x64 blocks, which are encrypted in parallel using the same encryption key. Further iterations will require a multiple of the number of clock cycles required for one round of encryption. Four image bytes are encrypted in parallel in one clock cycle.

Table. 5

Module	Number of clock cycles
Encrypt Module	4096
Decrypt Module	4096

The number of clock cycles required for information hiding depends on the size of the message and the complexity matrix of the cover image. The maximum coprocessor memory to accommodate all the results of the intermediate operations is 1024KB, which includes the intermediate results of compression, encryption, and information hiding. A tightly coupled hardware and software architecture is essential for the operation of the FPGA-based coprocessor presented in this paper. Recursive and time consuming operations are implemented in reconfigurable logic, while software and control signals play an important role in synchronizing the FPGA coprocessor with the processor system. One time operations, such as Arithmetic coding during image compression, are implemented in software as they do not present a huge computational overhead.

5. Conclusion

In the previous sections we presented the architecture of the SCAN cryptoprocessor, which is a modified SparcV8 architecture. We presented an instruction set to implement SCAN methodology-based algorithms for image compression, encryption, and information hiding, implemented on an off-chip FPGA coprocessor. We presented the XILINX ISE synthesis results of the FPGA coprocessor for the SCAN-based compression, encryption, and information hiding.

We see that we require a seamless integration of hardware and software for optimum performance of the processor. The throughput of the system can further be increased by transferring 4 bytes at a time on the 32 bit wide bus along with using an edge triggered RAM for storing and retrieving data. We further note that we are not only limited to processing images using the SCAN-based methodology, but we can also extend the methodology to process voice, text, and other types of data by converting these types of data to 2-D and treating them as images. We can use zero padding on the data to achieve a $2^n x 2^n$ size in case the data cannot

be converted into a square matrix before processing it with the SCAN algorithms. Further work in this direction will be the VLSI implementation of the coprocessor subsystem that will have increased throughput and a higher clock rate.

6. References

[1] C. Kachris, S. Maniccam, A. Dollas, N. Bourbakis, "A reconfigurable Logic-Based Processor for the SCAN Image and Video Algorithm", *1st Workshop on Application Specific Processors, International Workshop on Application Specific Processors*, WASPI, November 19, 2002, Istambul, Turkey.

[2] N. Bourbakis, C. Alexopoulos, "A fractal-based Image Processing Language: Formal Modeling", *Patten Recognition*, 32 (1999) 317-338.

[3] S.S. Maniccam, *A Lossles Compression, Encryption and Information Hiding Methodology for Images and Video*, PhD thesis, Dept. ECE, Binghampton University, 2000.

[4] S.S. Maniccam and N.Bourbakis, "Lossless Compression and Information Hiding in Images", *Pattern Recognition Journal*, vol.36, 2004.

[5] SPARC International Inc., *The SPARC Architecture Manual, Version 8*, 535 Middlefield Road, Suite 210, Menlo Park, CA 94025.

[6] Apostolos Dollas, Christopher Kachris, Nikolaos Bourbakis, "Performance Analysis of Fixed, Reconfigurable, and Custom Architectures for the SCAN Image and Video Encryption Algorithm", *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[7] The ArchC Team, *The ArchC Assembler Generator-v1.5 Reference Manual*, Av. Albert Einstein, 1251 13084-971, PO Box 6176 - Campinas/SP – Brazil.

[8] Open SystemC Initiative, *SystemC 2.0.1 Language Reference Manual, Revision 1.0*, 1177, Braham Lane #302, San Jose, CA 95118-3799.

[9] Svensson, H. Lenart, T. Owall, V., "Accelerating Vector Operations by Utilizing Reconfigurable Coprocessor Architectures", 1177, *IEEE International Symposium on Circuits and Systems*, 2007, ISCAS 2007.

[10] Helger Lipmaa, Phillip Rogaway, David Wagner, Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption, September 2000. http://citeseer.ist.psu.edu/cache/papers/cs/17187/http:zSzzSzwww.tml.hut.fizSz~helgerzSzpaperszSzlrw00zSzctr.pdf/lipmaa00comments.pdf

[11] David A. McGrew, Counter Mode Security: Analysis and Recommendations, November 15, 2002, Cisco Systems, Inc. http://www.mindspring.com/~dmcgrew/ctr-security.pdf