# Encryption algorithm based on Rubik's cube principle

-An implementation on AD BF533 DSP-

Technical University of Cluj Napoca, Faculty of Electronics, Telecommunications and Information Technology

*Author: Bogdan Mesesan*

*Coordinators: Prof. Dr. Ing. Romulus Terebes, Drd. Ing. Gabriel Lazar*

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA ROMANIA

## Contents

1. Introduction........................................................................................3
2. Rubik's Cube image encryption algorithm..................................................3
2.1. Encryption........................................................................................3
2.2. Decryption........................................................................................5
3. Implementation in C and results ......................................................................6
4. Implementation in ASM and optimization .....................................................9
5.Conclusions........................................................................................13
6.References........................................................................................13

## 1. Introduction

The following paper presents an image encryption algorithm based on Rubik's cube principle as part of the project work for the Information Processing Technologies university course. The work is based entirely on the research published [1]. It is not my objective to present a new algorithm, but rather to implement the mathematical rationale behind it on AD BF533 Digital Signal Processor (DSP). The project firstly presents the logic behind the algorithm and its implementation in the C programming language, using Visual DSP as IDE. Results of both encrypted and decrypted images of different sizes show the diffusion and confusion elements of the algorithm. In addition, an Assembly programming language implementation has been done, in order to optimize the speed and memory characteristics of the algorithm. Last but not least, a speed comparison between the C and Assembly is displayed. The active work done by the student author of the project is creating a version of the Rubik's cube image encryption algorithm optimized for the BF553 DSP.

The paper is organized in the following manner: Section 2 describes the logical and mathematical operations of the algorithm, Section 3 presents the C implementation and results in Visual DSP, Section 4 shows the optimization done in Assembly programming language with the aim to increase performance.

## 2. Rubik's Cube image encryption algorithm

### 2.1. Encryption

1) As it has been stated above, the main source of inspiration for this project is [1]. The implemented code is entirely based on the following sequence of operations. The input is represented by an $\alpha$-bit gray scale image with M rows and N columns called $I_0$. The solution has three encryption keys:

- $K_R$ and $K_C$ randomly generated vectors of length M and N. Both vectors are populated with values ranging from $[0, 1, 2, \ldots, 2^{\alpha} - 1]$, which must have non-constant values.
- The maximum number of iterations which will be done $ITER_{max}$.

After determining the encryption keys, which will be done in a setup phase, the encryption process can begin by initializing a counter ITER = 0.

2) The counter will be incremented ITER = ITER + 1.

3) For each row, an a[i] vector will be calculated, representing the sum of all elements on the row i: $\sum_{j=0}^{N-1} I_0[i,j],\ \ i = 0,\ 1,\ \ 2,\ \ldots\ ,\ \ M-1.$

Compute Ma vector, where Ma[i] = a[i] % 2.

Each row of the input image $I_0$ will be circular shifted by $K_R[i]$ positions according to the following rule:

If Ma[i] = 0 then right circular shift.

If Ma[i] = 1 then left circular shift.

4) For each column, a b[i] vector will be calculated, representing the sum of all elements on the column j: $\sum_{i=0}^{M-1} I_0[i,j],\ \ j = 0,\ 1,\ \ 2,\ \ldots\ ,\ \ N-1.$

Compute Mb vector, where Mb[i] = b[i] % 2.

Each column of the input image $I_0$ will be circular shifted by $K_C[i]$ positions according to the following rule:

If Mb[i] = 0 then up circular shift.

If Mb[i] = 1 then down circular shift.

Steps 4) and 5) will create a scrambled image, representing the diffusion element of the algorithm.

5) Using the $K_C$ vector, the Bit wise operation XOR will be applied to each row of the scrambled image, using the following rule:

$I_1(2i - 1, j) = I_{SCR}(2i - 1, j) \oplus K_C[j]$;

$I_1(2i, j) = I_{SCR}(2i , j) \oplus \mathrm{rot180}K_C[j]$, where rot180 represents the 180 degree flip of vector $K_C$.

6) Using the $K_R$ vector, the Bit wise operation XOR will be applied to each column of the scrambled image, using the following rule:

$I_{ENC}(i, 2j - 1) = I_1(2i - 1, j) \oplus K_R[j]$;

$I_{ENC}(i, j) = I_1(2i , j) \oplus \mathrm{rot180}K_R[j]$, where rot180 represents the 180 degree flip of vector $K_R$.

Steps 5) and 6) change the value of each pixel of the image, representing the confusion element of the algorithm.

7) If ITER = $\text{ITER}_{\text{max}}$, the encryption process is complete. Otherwise, the algorithm jumps back to step 2).

## 2.2. Decryption

The decrypted image is recovered from the encrypted image $I_{ENC}$, using the secret keys: vector $K_R$, vector $K_C$ and $ITER_{max}$.

1) Initiate the counter ITER = 0.

2) Increment the counter ITER = ITER + 1.

3) Using the $K_R$ vector, the Bit wise operation XOR will be applied to each column of the encrypted image, using the following rule:

$I_1(i, 2j - 1) = I_{ENC}(2i – 1, j) \oplus K_R[j]$;

$I_1(i, j) = I_{SCR}(2i , j) \oplus \mathrm{rot180}K_R[j]$, where rot180 represents the 180 degree flip of vector $K_R$.

4) Using the $K_C$ vector, the Bit wise operation XOR will be applied to each row of the image $I_1$, using the following rule:

$I_{SCR}(2i – 1, j) = I_1(2i – 1, j) \oplus K_C[j]$;

$I_{SCR}(2i, j) = I_1(2i , j) \oplus \mathrm{rot180}K_C[j]$, where rot180 represents the 180 degree flip of vector $K_C$.

Steps 3) and 4) will recreate the original scrambled image,

5) For each column, a b vector will be calculated, representing the sum of all elements on the column j: $\sum_{i=0}^{M-1} I_{SCR}[i,j],\ \ j = 0,\ 1,\ 2,\ \dots\ ,\ N-1.$

Compute Mb vector, where Mb[i] = b[i] % 2.

Each column of the scrambled image $I_{SCR}$ will be circular shifted by *$K_C[i]$* positions according to the following rule:

If Mb[i] = 0 then up circular shift.

If Mb[i] = 1 then down circular shift.

6) For each row, an a[i] vector will be calculated, representing the sum of all elements on the row i: $\sum_{j=0}^{N-1} I_{SCR}[i,j],\ \ i = 0,\ 1,\ 2,\ \dots\ ,\ M-1.$

Compute Ma vector, where Ma[i] = a[i] % 2.

Each row of the scrambled image $I_{SCR}$ will be circular shifted by *$K_R[i]$* positions according to the following rule:

If Ma[i] = 0 then right circular shift.

If Ma[i] = 1 then left circular shift.

7) If ITER = ITER$_{max}$, the decryption process is complete. Otherwise, the algorithm jumps back to step 2).

## 3. Implementation in C and results

The first implementation for AD Blackfin 533 has been done using C programming language, in order to test the functionality of the algorithm in a Digital Signal Processor environment.

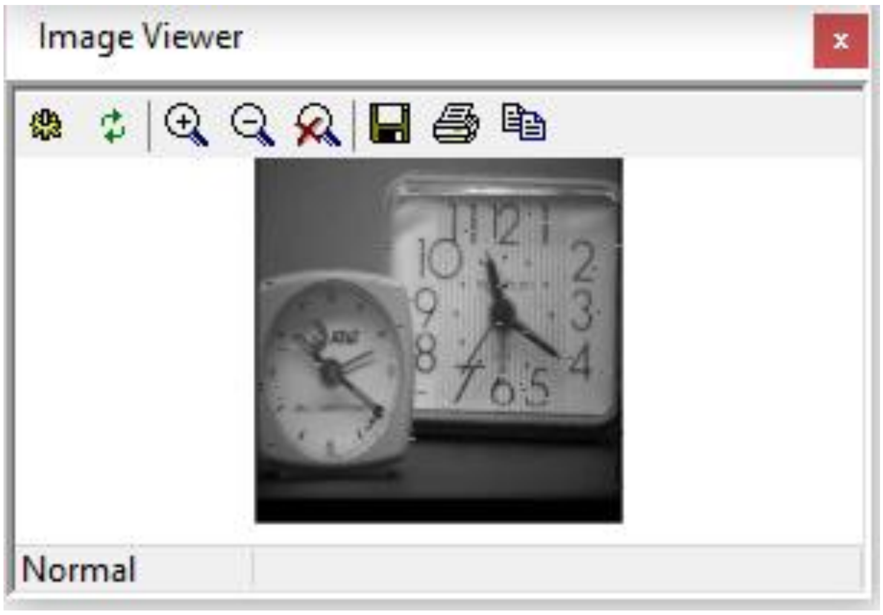
The C algorithm is based on the following key functions, which are based on the mathematical model presented in Section 2. They are used in both encryption and the decryption.

```
void Sum_Rows(unsigned char *input, int *a, unsigned char *Ma);

void Rotate_Row(unsigned char *input, unsigned int pos, unsigned char dir);

void Circular_Shift_Rows(unsigned char *input, unsigned char *Ma);

void Sum_Columns(unsigned char *input, unsigned int *b, unsigned char *Mb);

void Rotate_Column(unsigned char *input, unsigned int pos, unsigned char dir);

void Circular_Shift_Columns(unsigned char *input, unsigned char *Mb);

void Rows_XOR(unsigned char *input);

void Columns_XOR(unsigned char *input);
```

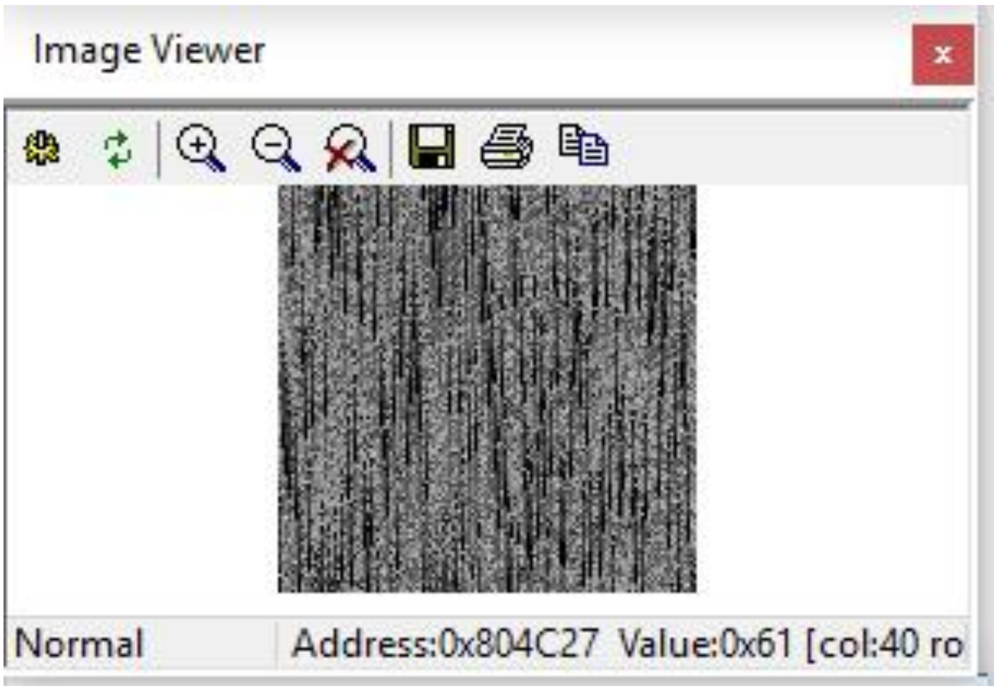
The following example illustrates the encryption of an 128x128 pixel gray scale image.

Example 1:

Original image:

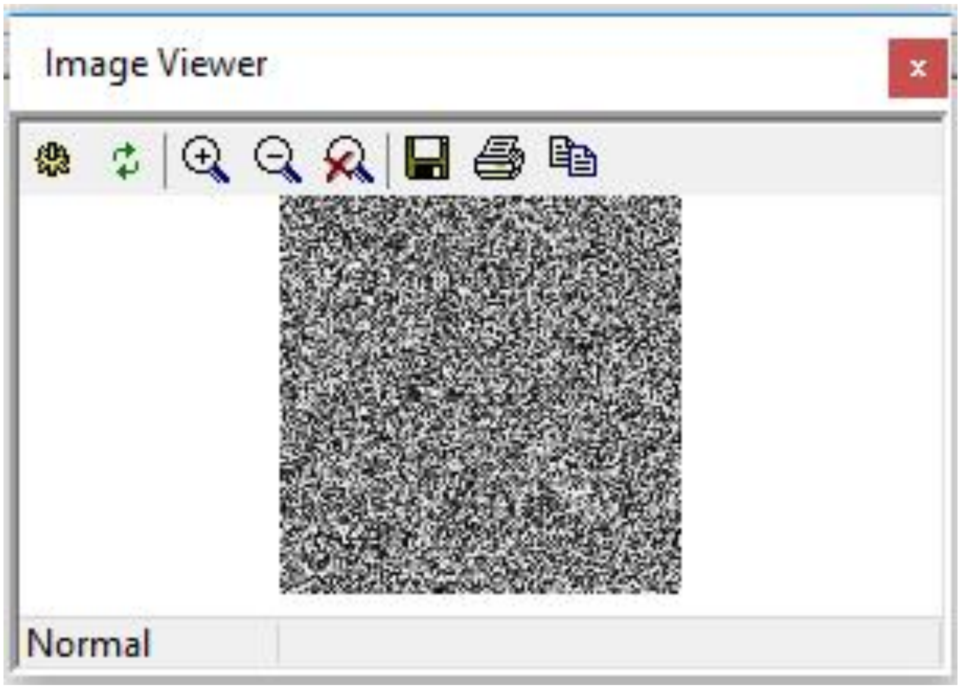


Scrambled image:



Normal Address:0x804C27 Value:0x61 [col:40 ro

Encrypted image:

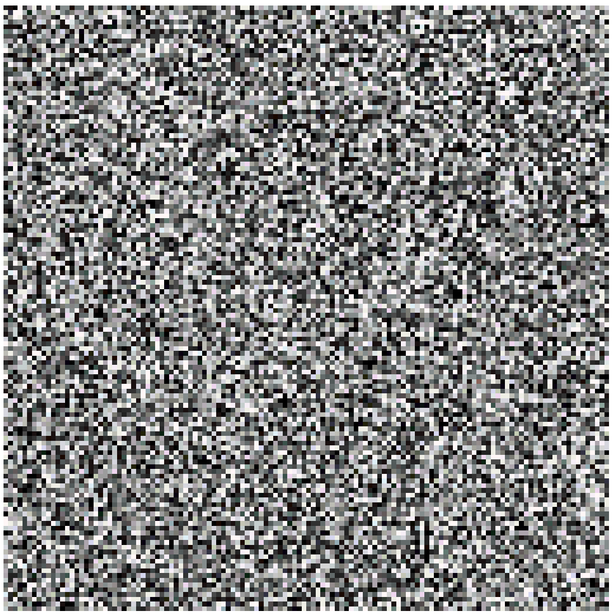


Normal

Example 2 presents the encryption of another 128x128 pixel gray scale image, “Lena”.

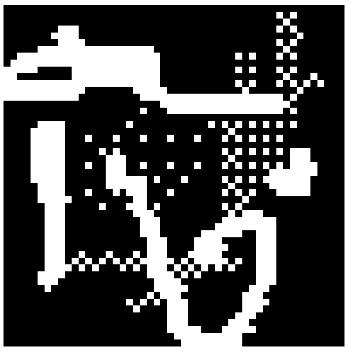
Original image:

Encrypted image:

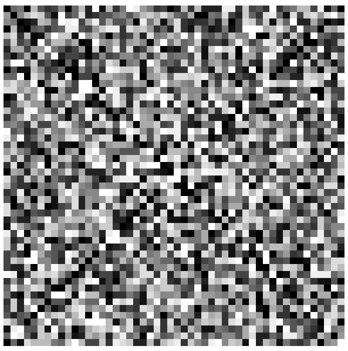


Example 3 presents the encryption of a 50x50 pixel gray scale image.

Original image:



Encrypted image:

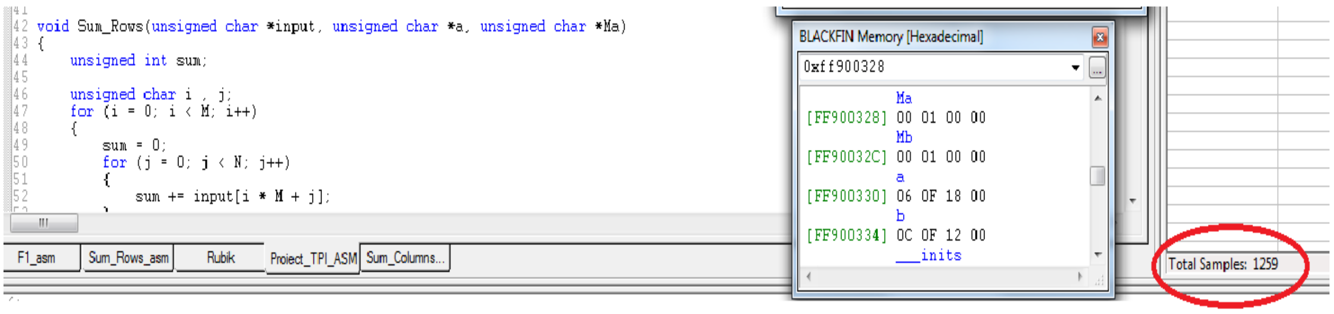


## 4. Implementation in ASM and optimization

I have chosen to use Assembly language to optimize the algorithm for Blackfin 533. The following section firstly presents optimized functions, called in separately from the main project in order to illustrate speed improvement. The second part of the section analyses the global effect of the ASM written code.

The first functions that have been optimized have been **Sum_Rows** and **Sum_Columns.**

Profiling C implementation:

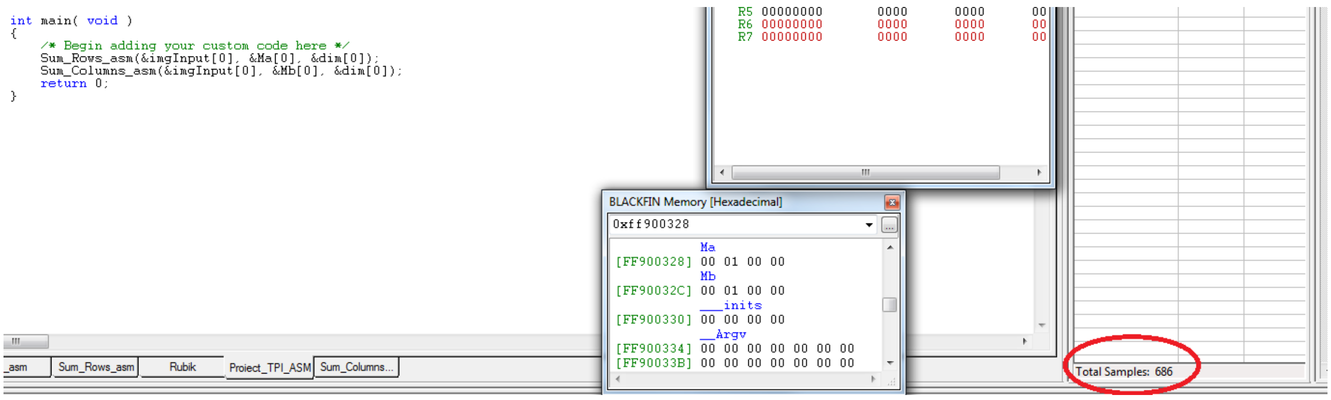


```
void Sum_Rows(unsigned char *input, unsigned char *a, unsigned char *Ma)
{
    unsigned int sum;

    unsigned char i , j;
    for (i = 0; i < M; i++)
    {
        sum = 0;
        for (j = 0; j < N; j++)
        {
            sum += input[i * M + j];
```

```
BLACKFIN Memory [Hexadecimal]
0xff900328
            Ma
[FF900328]  00 01 00 00
            Mb
[FF90032C]  00 01 00 00
            a
[FF900330]  06 0F 18 00
            b
[FF900334]  0C 0F 12 00
            ___inits
```

Total samples 1259 after calling the C functions.

Profiling ASM implementation:



```
int main( void )
{
    /* Begin adding your custom code here */
    Sum_Rows_asm(&imgInput[0], &Ma[0], &dim[0]);
    Sum_Columns_asm(&imgInput[0], &Mb[0], &dim[0]);
    return 0;
}
```

```
R5  00000000      0000      0000      00
R6  00000000      0000      0000      00
R7  00000000      0000      0000      00
```

BLACKFIN Memory [Hexadecimal]

```
0xff900328
           Ma
[FF900328] 00 01 00 00
           Mb
[FF90032C] 00 01 00 00
           ___inits
[FF900330] 00 00 00 00
           __Argv
[FF900334] 00 00 00 00 00 00 00 00
[FF90033B] 00 00 00 00 00 00 00 00
```

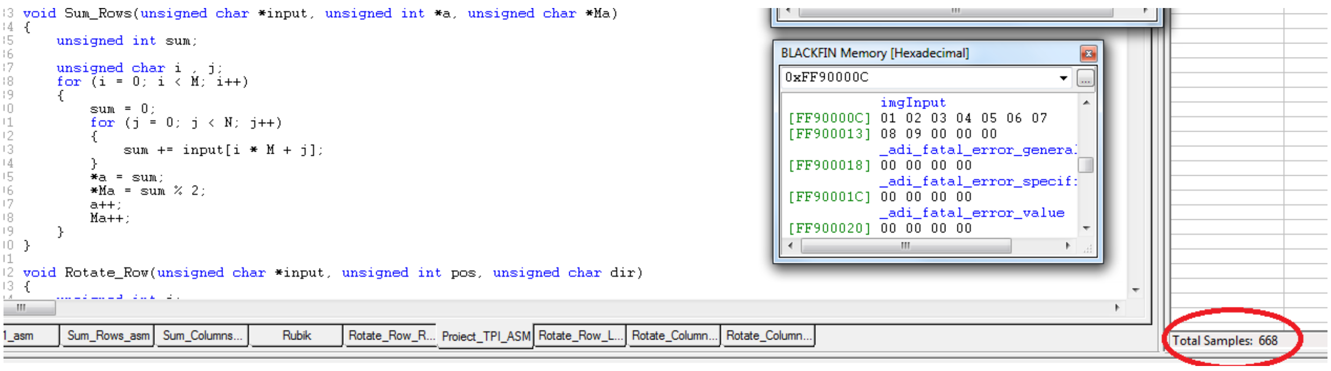
Total Samples: 686

Total samples 686 after calling the ASM functions.

**Rotate_Row()** and **Rotate_Column()** functions have also been optimized for AD BF 533.

**Rotate_Row()** Profiling:

C implementation:



```
void Sum_Rows(unsigned char *input, unsigned int *a, unsigned char *Ma)
{
    unsigned int sum;

    unsigned char i , j;
    for (i = 0; i < M; i++)
    {
        sum = 0;
        for (j = 0; j < N; j++)
        {
            sum += input[i * M + j];
        }
        *a = sum;
        *Ma = sum % 2;
        a++;
        Ma++;
    }
}

void Rotate_Row(unsigned char *input, unsigned int pos, unsigned char dir)
{
```

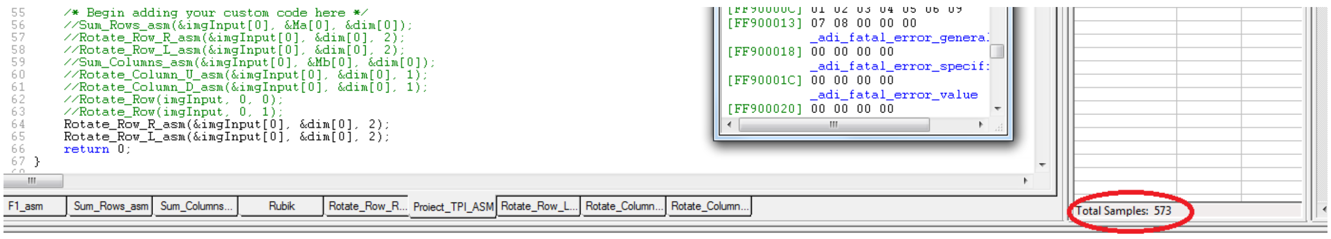
BLACKFIN Memory [Hexadecimal]

```
0xFF90000C
           imgInput
[FF90000C] 01 02 03 04 05 06 07
[FF900013] 08 09 00 00 00
           _adi_fatal_error_genera
[FF900018] 00 00 00 00
           _adi_fatal_error_specif
[FF90001C] 00 00 00 00
           _adi_fatal_error_value
[FF900020] 00 00 00 00
```

Total Samples: 668

Total samples 668 after calling the C function.

ASM implementation;



```
    /* Begin adding your custom code here */
    //Sum_Rows_asm(&imgInput[0], &Ma[0], &dim[0]);
    //Rotate_Row_R_asm(&imgInput[0], &dim[0], 2);
    //Rotate_Row_L_asm(&imgInput[0], &dim[0], 2);
    //Sum_Columns_asm(&imgInput[0], &Mb[0], &dim[0]);
    //Rotate_Column_U_asm(&imgInput[0], &dim[0], 1);
    //Rotate_Column_D_asm(&imgInput[0], &dim[0], 1);
    //Rotate_Row(imgInput, 0, 0);
    //Rotate_Row(imgInput, 0, 1);
    Rotate_Row_R_asm(&imgInput[0], &dim[0], 2);
    Rotate_Row_L_asm(&imgInput[0], &dim[0], 2);
    return 0;
}
```

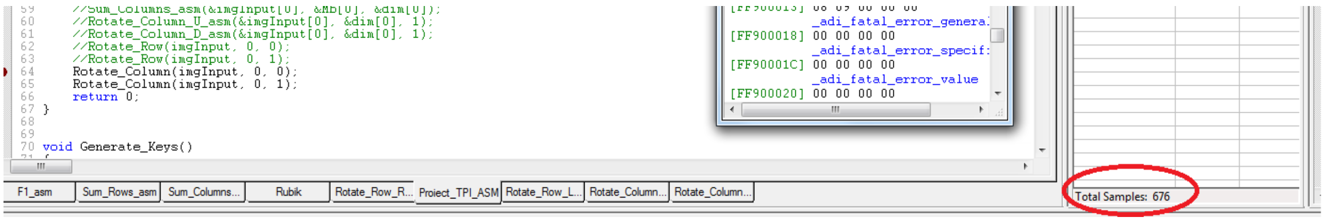
```
[FF90000C] 01 02 03 04 05 06 07
[FF900013] 07 08 00 00 00
           _adi_fatal_error_genera
[FF900018] 00 00 00 00
           _adi_fatal_error_specif
[FF90001C] 00 00 00 00
           _adi_fatal_error_value
[FF900020] 00 00 00 00
```

Total Samples: 573

Total samples 573 after calling the ASM function.

**Rotate_Column()** Profiling:

C implementation:



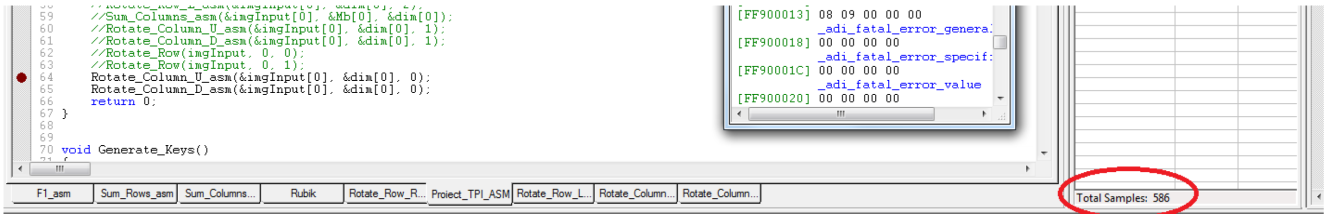
```
    //Sum_Columns_asm(&imgInput[0], &Mb[0], &dim[0]);
    //Rotate_Column_U_asm(&imgInput[0], &dim[0], 1);
    //Rotate_Column_D_asm(&imgInput[0], &dim[0], 1);
    //Rotate_Row(imgInput, 0, 0);
    //Rotate_Row(imgInput, 0, 1);
    Rotate_Column(imgInput, 0, 0);
    Rotate_Column(imgInput, 0, 1);
    return 0;
}


void Generate_Keys()
```

```
[FF900018] 00 00 00 00
           _adi_fatal_error_specif:
[FF90001C] 00 00 00 00
           _adi_fatal_error_value
[FF900020] 00 00 00 00
```

Total Samples: 676

Total samples 676 after calling the C function.

ASM implementation:



```
    //Sum_Columns_asm(&imgInput[0], &Mb[0], &dim[0]);
    //Rotate_Column_U_asm(&imgInput[0], &dim[0], 1);
    //Rotate_Column_D_asm(&imgInput[0], &dim[0], 1);
    //Rotate_Row(imgInput, 0, 0);
    //Rotate_Row(imgInput, 0, 1);
    Rotate_Column_U_asm(&imgInput[0], &dim[0], 0);
    Rotate_Column_D_asm(&imgInput[0], &dim[0], 0);
    return 0;
}


void Generate_Keys()
```

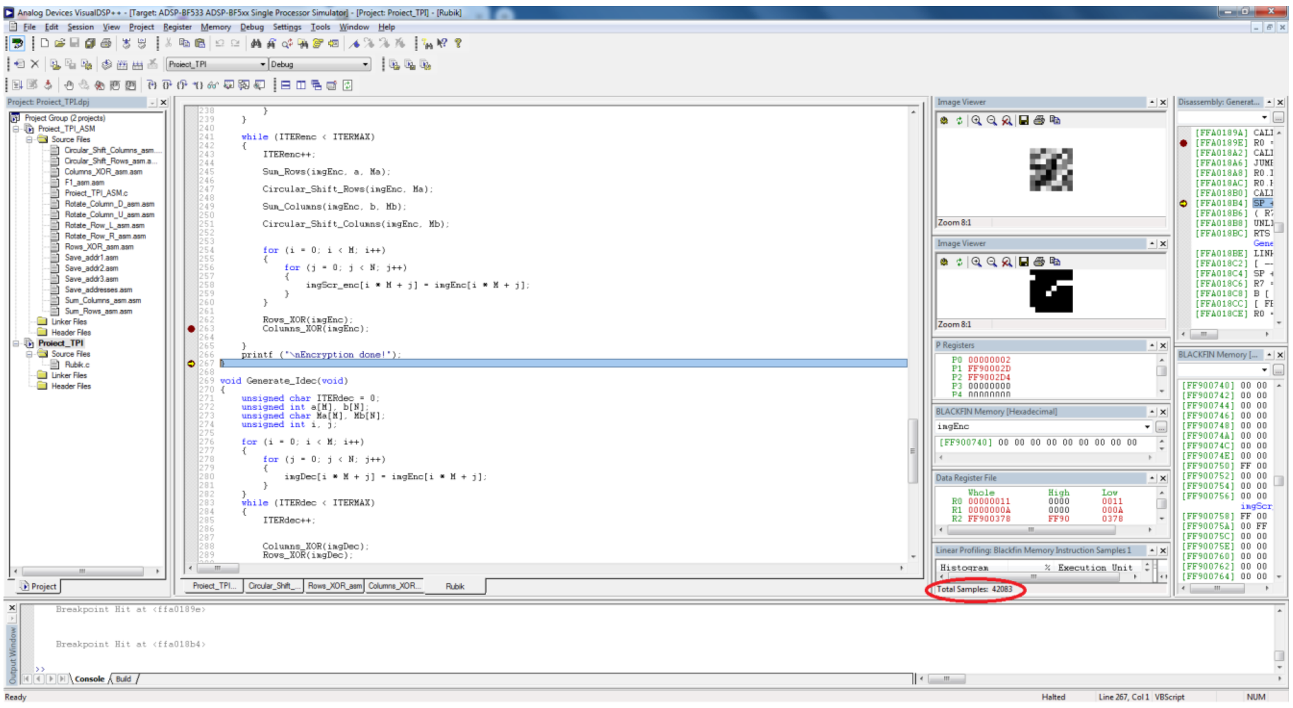
```
[FF900013] 08 09 00 00 00
           _adi_fatal_error_genera:
[FF900018] 00 00 00 00
           _adi_fatal_error_specif:
[FF90001C] 00 00 00 00
           _adi_fatal_error_value
[FF900020] 00 00 00 00
```

Total Samples: 586

Total samples 586 after calling the ASM function.

Additionally, **Rows_XOR()** and **Columns_XOR()** functions have also been implemented in ASM. The following examples present the global effects of the optimization.

Encryption C:



Analog Devices VisualDSP++ - [Target: ADSP-BF533 ADSP-BF5xx Single Processor Simulator] - [Project: Project_TPI] - [Rubik]

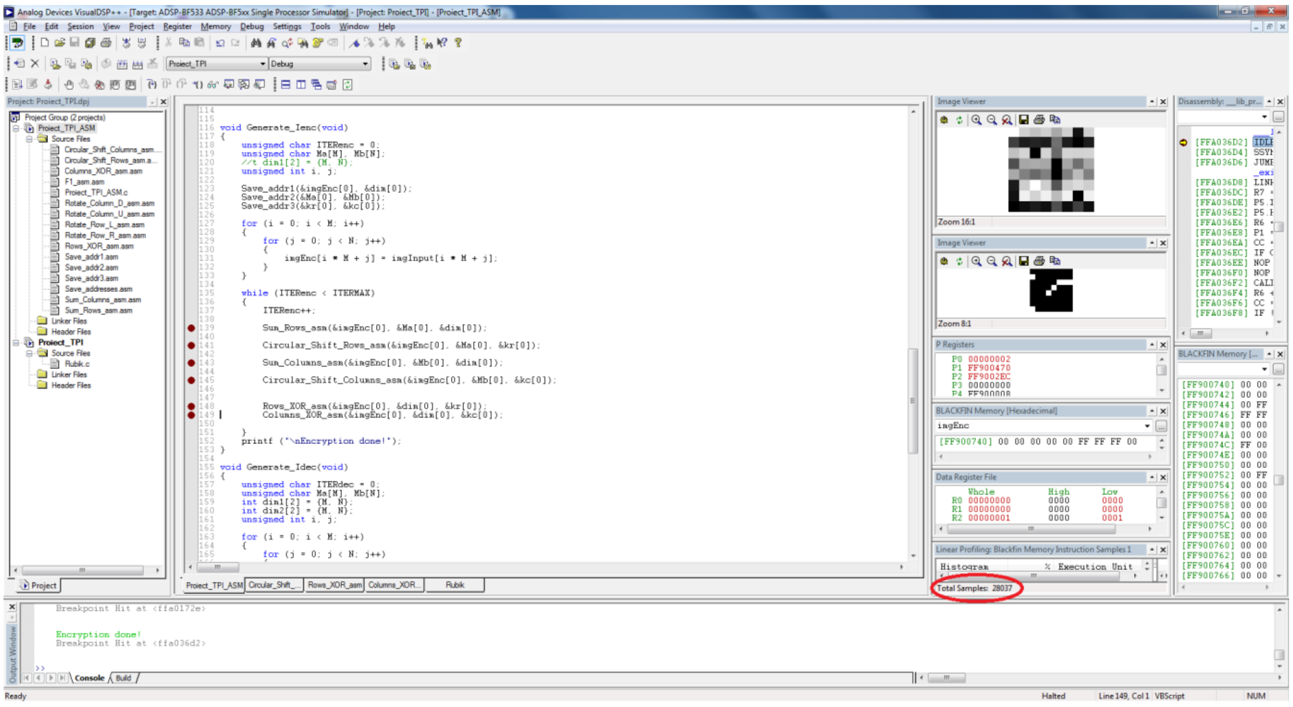
```
        }
    }
    while (ITERenc < ITERMAX)
    {
        ITERenc++;

        Sum_Rows(imgEnc, a, Ma);

        Circular_Shift_Rows(imgEnc, Ma);

        Sum_Columns(imgEnc, b, Mb);

        Circular_Shift_Columns(imgEnc, Mb);


        for (i = 0; i < M; i++)
        {
            for (j = 0; j < N; j++)
            {
                imgScr_enc[i * M + j] = imgEnc[i * M + j];
            }
        }

        Rows_XOR(imgEnc);
        Columns_XOR(imgEnc);

    }
    printf ("\nEncryption done!");
}

void Generate_Idec(void)
{
    unsigned char ITERdec = 0;
    unsigned int a[M], b[N];
    unsigned char Ma[M], Mb[N];
    unsigned int i, j;

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            imgDec[i * M + j] = imgEnc[i * M + j];
        }
    }
    while (ITERdec < ITERMAX)
    {
        ITERdec++;


        Columns_XOR(imgDec);
        Rows_XOR(imgDec);
```

```
Breakpoint Hit at <ffa0189e>

Breakpoint Hit at <ffa018b4>
```

Total Samples: 42083

Total samples: 42083.

Encryption ASM:


Analog Devices VisualDSP++ - [Target: ADSP-BF533 ADSP-BF5xx Single Processor Simulator] - [Project: Proiect_TPI] - [Proiect_TPI_ASM]

```
void Generate_Ienc(void)
{
    unsigned char ITERenc = 0;
    unsigned char Ma[M], Mb[N];
    //t dim1[2] = {M, N};
    unsigned int i, j;

    Save_addr1(&imgEnc[0], &dim[0]);
    Save_addr2(&Ma[0], &Mb[0]);
    Save_addr3(&kr[0], &kc[0]);

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            imgEnc[i * M + j] = imgInput[i * M + j];
        }
    }

    while (ITERenc < ITERMAX)
    {
        ITERenc++;

        Sum_Rows_asm(&imgEnc[0], &Ma[0], &dim[0]);

        Circular_Shift_Rows_asm(&imgEnc[0], &Ma[0], &kr[0]);

        Sum_Columns_asm(&imgEnc[0], &Mb[0], &dim[0]);

        Circular_Shift_Columns_asm(&imgEnc[0], &Mb[0], &kc[0]);


        Rows_XOR_asm(&imgEnc[0], &dim[0], &kr[0]);
        Columns_XOR_asm(&imgEnc[0], &dim[0], &kc[0]);

    }
    printf ("\nEncryption done!");
}

void Generate_Idec(void)
{
    unsigned char ITERdec = 0;
    unsigned char Ma[M], Mb[N];
    int dim1[2] = {M, N};
    int dim2[2] = {M, N};
    unsigned int i, j;

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
```

```
P0 00000002
P1 FF900470
P2 FF9002EC
P3 00000000
P4 FF900008
```

```
imgEnc
[FF900740] 00 00 00 00 00 FF FF FF 00
```

```
     Whole          High      Low
R0   00000000       0000      0000
R1   00000000       0000      0000
R2   00000001       0000      0001
```

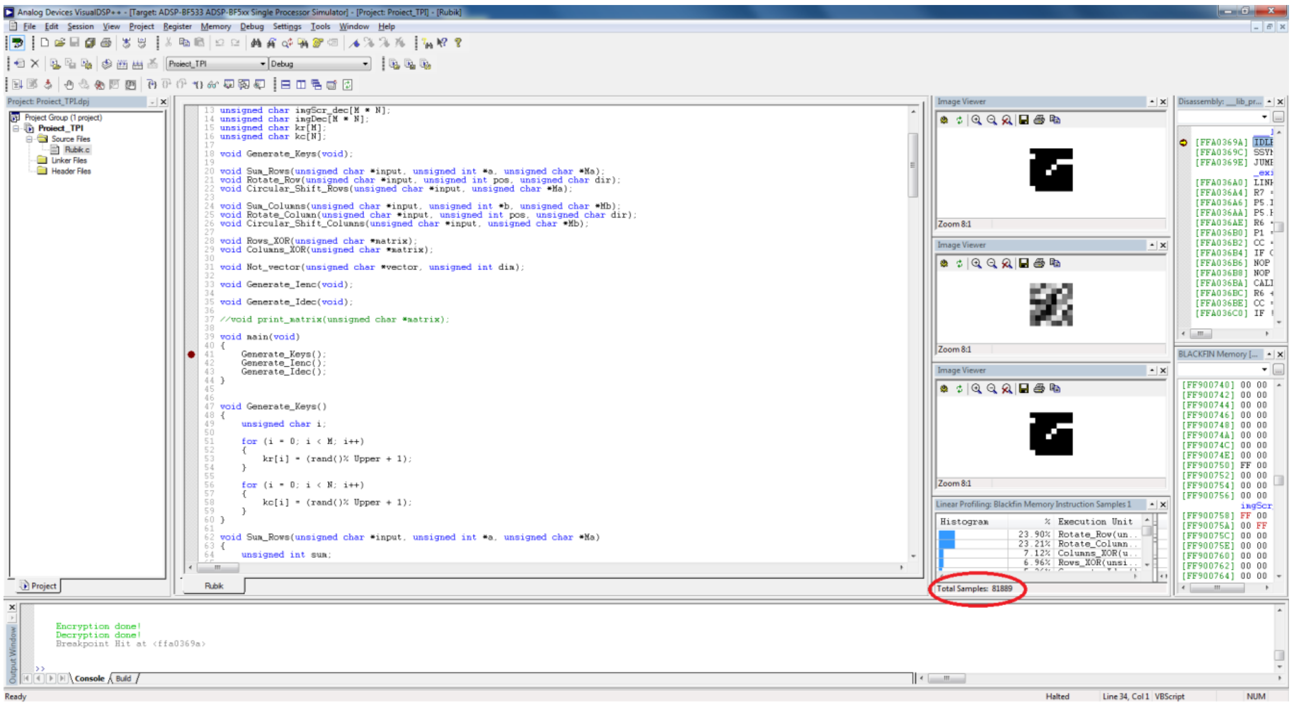
```
Breakpoint Hit at <ffa0172e>

Encryption done!
Breakpoint Hit at <ffa036d2>
```

Total Samples: 28037

Halted    Line 149, Col 1    VBScript

Total samples: 28037.

Full algorithm for an 8x8 gray scale image C:


Analog Devices VisualDSP++ - [Target: ADSP-BF533 ADSP-BF5xx Single Processor Simulator] - [Project: Proiect_TPI] - [Rubik]

```
unsigned char imgScr_dec[M * N];
unsigned char imgDec[M * N];
unsigned char kr[M];
unsigned char kc[N];

void Generate_Keys(void);

void Sum_Rows(unsigned char *input, unsigned int *a, unsigned char *Ma);
void Rotate_Row(unsigned char *input, unsigned int pos, unsigned char dir);
void Circular_Shift_Rows(unsigned char *input, unsigned char *Ma);

void Sum_Columns(unsigned char *input, unsigned int *b, unsigned char *Mb);
void Rotate_Column(unsigned char *input, unsigned int pos, unsigned char dir);
void Circular_Shift_Columns(unsigned char *input, unsigned char *Mb);

void Rows_XOR(unsigned char *matrix);
void Columns_XOR(unsigned char *matrix);

void Not_vector(unsigned char *vector, unsigned int dim);

void Generate_Ienc(void);

void Generate_Idec(void);

//void print_matrix(unsigned char *matrix);

void main(void)
{
    Generate_Keys();
    Generate_Ienc();
    Generate_Idec();
}


void Generate_Keys()
{
    unsigned char i;

    for (i = 0; i < M; i++)
    {
        kr[i] = (rand()% Upper + 1);
    }

    for (i = 0; i < N; i++)
    {
        kc[i] = (rand()% Upper + 1);
    }
}

void Sum_Rows(unsigned char *input, unsigned int *a, unsigned char *Ma)
{
    unsigned int sum;
```

```
Histogram        %   Execution Unit
            23.90%   Rotate_Row(un...
            23.21%   Rotate_Column...
             7.12%   Columns_XOR(u...
             6.96%   Rows_XOR(unsi...
```

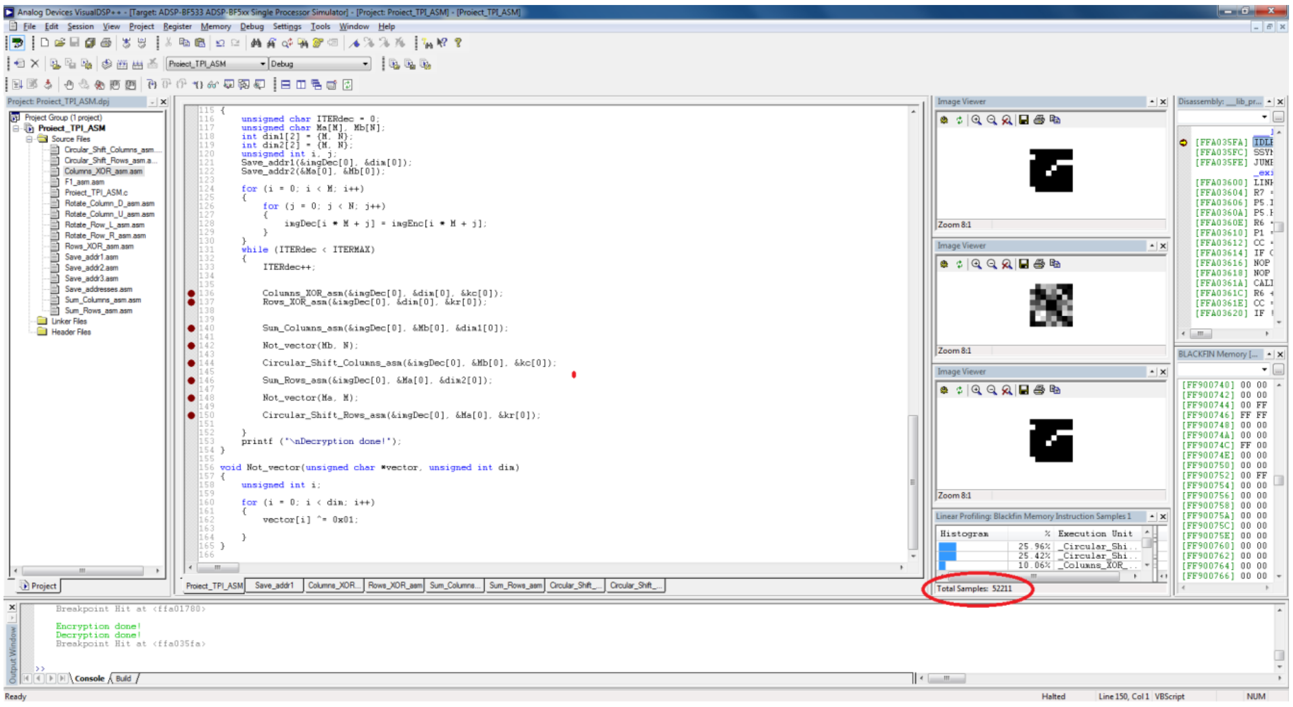
Total Samples: 81889

```
Encryption done!
Decryption done!
Breakpoint Hit at <ffa0369a>
```

Halted    Line 34, Col 1    VBScript

Total samples: 81889.

Full algorithm for an 8x8 gray scale image ASM:



```
{
    unsigned char ITERdec = 0;
    unsigned char Ma[M], Mb[N];
    int dim1[2] = {M, N};
    int dim2[2] = {M, N};
    unsigned int i, j;
    Save_addr1(&imgDec[0], &dim[0]);
    Save_addr2(&Ma[0], &Mb[0]);

    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
        {
            imgDec[i * M + j] = imgEnc[i * M + j];
        }
    }
    while (ITERdec < ITERMAX)
    {
        ITERdec++;


        Columns_XOR_asm(&imgDec[0], &dim[0], &kc[0]);
        Rows_XOR_asm(&imgDec[0], &dim[0], &kr[0]);


        Sum_Columns_asm(&imgDec[0], &Mb[0], &dim1[0]);

        Not_vector(Mb, N);

        Circular_Shift_Columns_asm(&imgDec[0], &Mb[0], &kc[0]);

        Sum_Rows_asm(&imgDec[0], &Ma[0], &dim2[0]);

        Not_vector(Ma, M);

        Circular_Shift_Rows_asm(&imgDec[0], &Ma[0], &kr[0]);

    }
    printf ("\nDecryption done!");
}

void Not_vector(unsigned char *vector, unsigned int dim)
{
    unsigned int i;

    for (i = 0; i < dim; i++)
    {
        vector[i] ^= 0x01;

    }
}
```

```
Histogram    %      Execution Unit
          25.96%  _Circular_Shi...
          25.42%  _Circular_Shi...
          10.06%  _Columns_XOR_...
Total Samples: 52211
```

```
Breakpoint Hit at <ffa01780>

Encryption done!
Decryption done!
Breakpoint Hit at <ffa035fa>
```

Total samples: 52211.

After running the project with the optimizations presented above, the algorithm runs with 36% faster than the C implementation.

## 5.Conclusions

The Rubik's Cube encryption algorithm is a real time image encryption algorithm, which runs even better on AD BF533 after optimization. It is a secure and robust algorithm, resistant to brute force attacks due to its large key space. Key sensibility is also present in our algorithm, a small change in the triplet of the secret keys ($K_{R,}$ $K_C$, $ITER_{max}$) will create a big change in the output. In the same idea, for most images, the algorithm changes the value of 98% of the input image's pixels, making it resistant to statistical analysis attacks.

I would like to thank the researchers from source [1] for their contribution and for making the realization of this project possible. My contribution to this project was the following: I have read and understood the algorithm, I have implemented and tested it in C programming language, I optimized it for AD BF533 having good results overall.

## 6.References

[1] *Journal of Electrical and Computer Engineering Volume 2012, Article ID 173931, 13 page; Authors: Khaled Loukhaoukha, Jean-Yves Chouinard, and Abdellah Berdai*