

ELEN 448: REAL TIME DSP LAB

DESIGN AND IMPLEMENTATION OF IS-95 REVERSE LINK

ON TMS 320C6701/6201 DSP

Rajeshwary Tayade

000 07 1573

Code division multiple Access (CDMA) is a radically new concept in wireless technology. CDMA is a form of *spread-spectrum*, a family of digital communication techniques that have been used in military applications for many years. The core principle of spread spectrum is the use of noise-like carrier waves, and, as the name implies, bandwidths much wider than that required for simple point-to-point communication at the same data rate. It has gained widespread international acceptance by cellular radio system operators as an upgrade that will dramatically increase both their system capacity and the service quality. CDMA changes the nature of the subscriber station from a predominately analog device to a predominately digital device. Old-fashioned radio receivers separate stations or channels by filtering in the frequency domain. CDMA receivers do not eliminate analog processing entirely, but they separate communication channels by means of a pseudo-random modulation that is applied and removed in the digital domain, not on the basis of frequency. Multiple users occupy the same frequency band.

The CDMA Cellular Standard:

With CDMA, unique digital codes, rather than separate RF frequencies or channels, are used to differentiate subscribers. The codes are shared by both the mobile station (cellular phone) and the base station, and are called "pseudo-Random Code Sequences." All users share the same range of radio spectrum. For cellular telephony, CDMA is a digital multiple access technique specified by the Telecommunications Industry Association (TIA) as "IS-95".

IS-95 systems divide the radio spectrum into carriers which are 1,250 kHz (1.25 MHz) wide. One of the unique aspects of CDMA is that while there are certainly limits to the number of phone calls that can be handled by a carrier, this is not a fixed number. Rather, the capacity of the system will be dependent on a number of different factors. IS-95 uses a multiple access spectrum spreading technique called Direct Sequence (DS) CDMA. Each user is assigned a binary, Direct Sequence code during a call. The DS code is a signal generated by linear modulation with wideband Pseudorandom Noise (PN) sequences. As a result, DS CDMA uses much wider signals than those used in other technologies. Wideband signals reduce interference and allow one-cell frequency reuse. There is no time division, and all users use the entire carrier, all of the time.

CDMA Reverse Link:

A reverse Link is defined as the connection path from a Mobile unit to the base-station and it carries the voice and mobile power control information.

CDMA Modulation:

Both the Forward and Reverse Traffic Channels use a similar control structure consisting of 20 millisecond frames. For the system, frames can be sent at either 14400, 9600, 7200, 4800, 3600, 2400, 1800, or 1200 bps. For example, with a Traffic Channel operating at 9600 bps, the rate can vary from frame to frame, and can be 9600, 4800, 2400, or 1200 bps. The receiver detects the rate of the frame and processes it at the correct rate. This technique allows the channel rate to dynamically adapt to the speech or data activity for example, voice is active only 40% of the total call duration, and this can be used to conserve on power. CDMA starts with a basic data rate of 9600 bits per second. This is then spread to a transmitted bit rate, or chip rate (the transmitted bits are called chips), of 1.2288 MHz. The spreading process applies digital codes to the data bits, which increases the data rate while adding redundancy to the system. The chips are transmitted using a form of QPSK (quadrature phase shift keying) modulation which has been filtered to limit the bandwidth of the signal. This is added to the signal of all the other users in that cell. When the signal is received, the coding is removed from the desired signal, returning it to a rate of 9600 bps. When the decoding is applied to the other users' codes, there is no despreading; the signals maintain the 1.2288 MHz bandwidth. The ratio of transmitted bits or chips to data bits is the coding gain. The coding gain for the IS-95 CDMA system is 128, or 21 dB.

The Reverse Link Block Structure: The reverse link block structure is shown in Fig 1. The functionality of each block and the implementation details are explained below. The input data rate to the block is of rate 9600bps. (Only the highest rate has been implemented and hence a symbol repeater and randomizer blocks were not required).

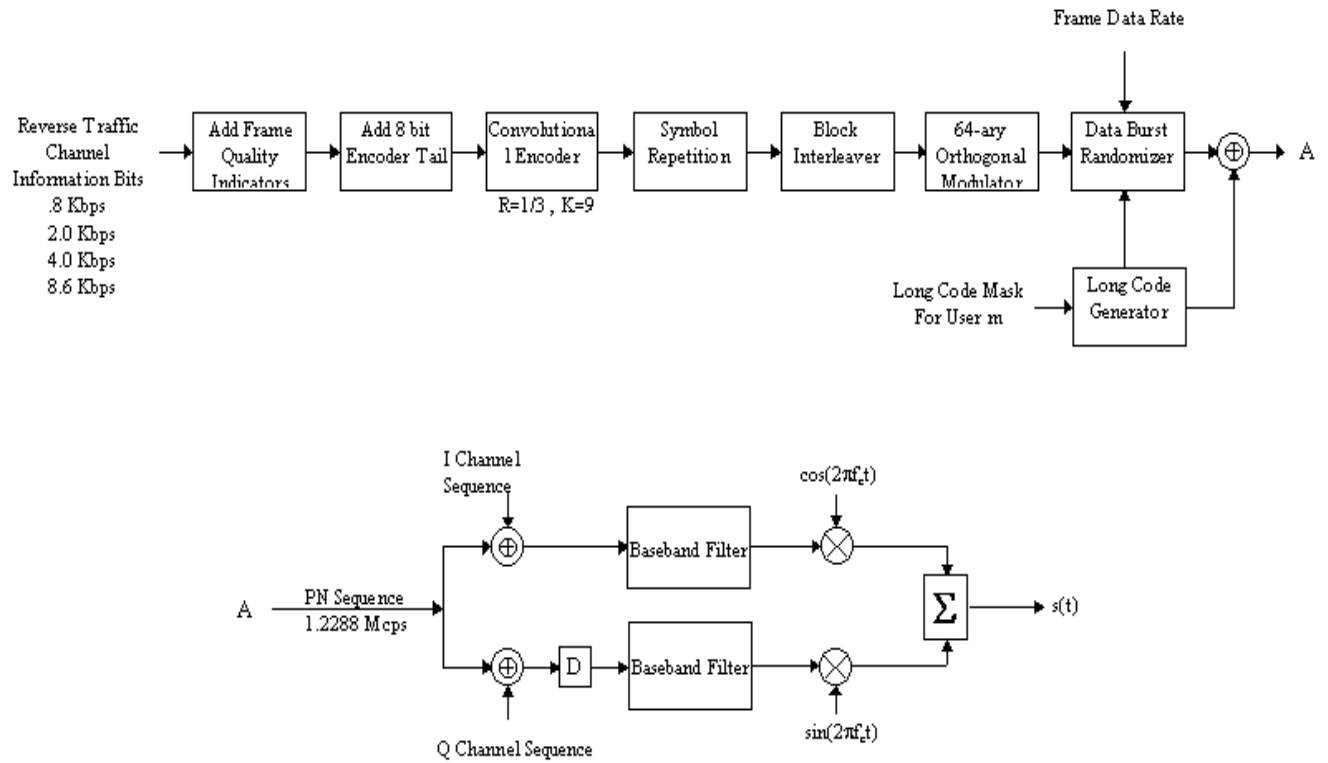


Figure 1: Block Diagram of IS-95 reverse link

Source: The data to be processed is read in frames of 184 bits each. This is for rate 1, or 9600bps. For different bit rates, the frame lengths vary as shown in table 1.

Data Rate	Frame Length
9200bps	184
4400bps	88
2000bps	40
800bps	16

Table 1: Source Data rates

Bit Adder: The bit adder simply adds a string of 8 zeros to each frame. This is to reset the Viterbi decoder used at the decoding unit. The new frame lengths and the corresponding bit rates are shown below:

Data Rate	Frame Length
9600bps	192
4800bps	96
2400bps	48
1200bps	24

Table 2: Data Rates after zero padding

Convolution Encoder: The encoder used is a convolution encoder of constraint length $K=9$ and rate $1/3$. The encoding is important to make the data less susceptible to channel noise, as the redundancy added by coded reduces the probability of error due to non-ideal channel. The convolution encoder is described using a generator matrix that introduces memory in the data stream and hence at the decoding end a sequence detection has to be performed using a Viterbi decoder. In the project the encoder used was with a generator matrix of

$$G1 = [0 \ 0 \ 1] , G2 = [0 \ 1 \ 0]; G3 = [1 \ 0 \ 0];$$

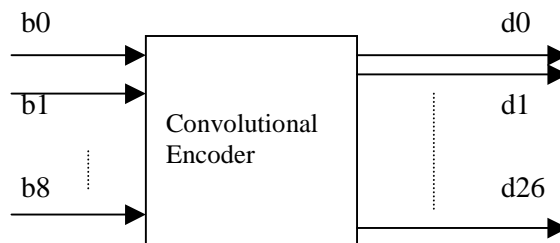


Figure 2: Convolutional Encoder

This becomes a trivial case of just a repetition code for simplicity of implementation. Since there is no memory introduced in this case, the decoding requires a simple un-repeater instead of a Viterbi decoder. The bit rates after encoding are

Data Rate	Frame Length
28800bps	576
14400bps	288
7200bps	144
3600bps	72

Table 3: Data Rates after coding

Block Interleaver: The purpose of the interleaver is to reduce burst errors. The channel coding and error correction capabilities of the convolution encoder is limited to random errors, and does not help much in case of burst errors. The interleaver scrambles the data so that even though the channel introduced burst errors on a sequence of bits, the errors on the actual information bits is random and hence independent. The interleaver used is a block or matrix of size 32×18 , the incoming data is written into the matrix column-wise and read out row wise. Thus any two adjacent code bits are now separated by 18 bits. This operation however introduces delay in the processing, and so the total processing time is 20ms.

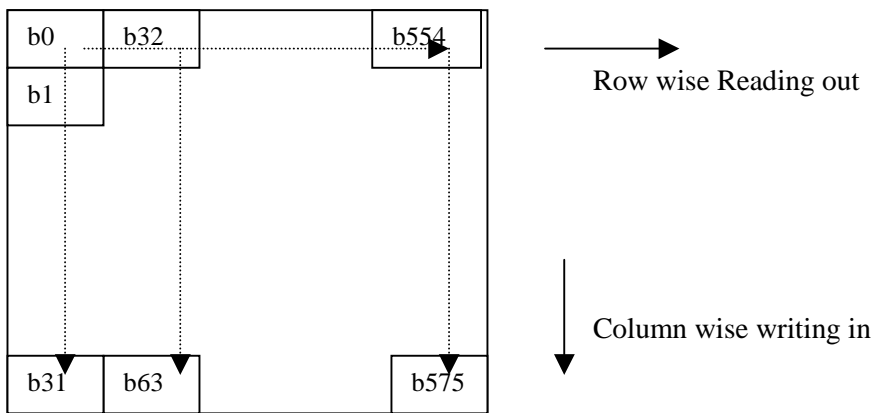


Figure 3: Block Interleaver

Reading from the interleaver: b0,b32,b64...b554,b1...562, b31..b575.

Walsh Encoder: For the reverse link the Walsh codes are used as the main spreading code. Every $\log_2 N$ bits are spread to N bits. The Walsh matrix is generated using the Hadamard matrix, and all the rows are mutually orthogonal. Thus with a Hadamard matrix of order 64 there are 64 mutually orthogonal sequences available of 64 bits each. The Hadamard matrix of order 2 is defined as

$$H_2 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

The matrices for higher order can be generated recursively using the relation

$$H_{2n} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}$$

The Hadamard matrix of order 64 is attached in the appendix. The Walsh matrix is then generated from the Hadamard matrix using the mapping shown below. The Walsh encoder reads data from the interleaver, one row at a time (for the highest rate) and then forms 3 groups of 6 bits each. Each 6 bit group is then used to index to a specific Walsh sequence.

Example : if the 6 bit are 001101 ,then the 13th row of the Walsh matrix is read. The new frame length is now $576 \times 64 / 6 = 6144$ and the rate is 307.1Kbps.

PN-Long Generator: The reverse link is different than the forward link because the signals from each user do not originate from a same source as in the forward link. The transmission from each user will arrive at a different time, due to propagation delay, and synchronization errors. Due to the unavoidable timing errors between the users, Walsh codes cannot be used, as they will no longer be orthogonal. For this reason, simple Pseudo random sequences, which are uncorrelated, but not orthogonal, are used for the PN codes of each user.

The PN-Long sequence is generated using a 42-bit long Linear Feedback Shift Register as shown in Fig below.

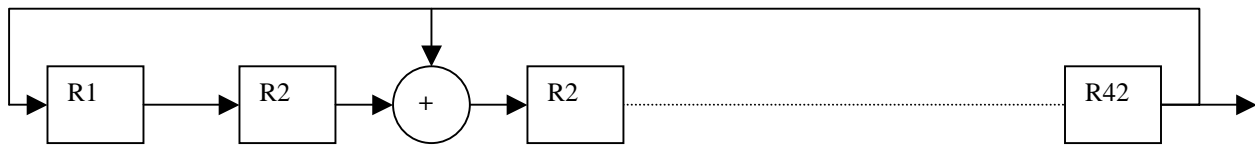


Figure 4: Linear Feedback shift register for generating PN-sequence

The LFSR is first initialized using some seed which can be any string of 42 bits except all

zeros. The feedback polynomial used is

$$f(x) = 1 + x^7 + x^9 + x^{11} + x^{15} + x^{16} + x^{17} + x^{20} + x^{21} + x^{23} + x^{24} + x^{25} + x^{26} + x^{32} + x^{35} + x^{36} + x^{37} + x^{39} + x^{40} + x^{41} + x^{42}$$

```

interleaver1_la(interleaved,encoder_out);

walsh_enc1_la(walsh_out,interleaved,W_Matrix);

framelength = 64*576/6;
repeater1_la(output_data,walsh_out);

pnseq12_la(output_data,output_data);

fp=fopen("testasm.txt","w");

for(i=0;i<768;i++)
{
    fprintf(fp,"%d\n",output_data[i]);
}
fclose(fp);

printf("/**** DONE ****/");

}

```

```

void read_data()
{
    int i,j,cnt=0,k=0;
    unsigned int mask,c,tmp;
    char *str;

    str="This is the Final Test";
    printf("**** READING STRING **\n");
    len =strlen(str);

    len =strlen(str);
    printf("Length :%d\n",len);
    for(i=0;i<len;i++)
    {
        c = *str++;
        for (j=0;j<8;j++)
        {
            mask = (int) pow(2,(7-j));
            input_data[cnt] = (mask & c)>>(7-j);

            cnt++;
        }
    }

    for(j=0;j<6;j++)
    {
        tmp=0;
        for(i=0;i<32;i++)
        {
            k = j*32+i;
            mask=0;
            if(input_data[k] == 1)
            {
                mask = (int)pow(2,(31-i));
            }
            tmp = tmp | mask;
        }
    }
}

```

```

        }
        frame[j] = tmp;
    }
}

```

The bit string is initially stored as an array of short, and then groups of 32 bits are packed together as integers in-order to save on memory requirements and also because bit wise operations are faster as compared to arithmetic operations on integer data-types. Also the number times data is loaded from memory is reduced by a factor of 32 which reduces the IO calls.

The convolutional encoder is a simple bit repeater that repeats each bit 3 times and the code is shown below:

```

global _encoder1_la
    .sect ".encoder1_la"

_encoder1_la: .proc A4,B4,B3
    .reg i,pos1,pos2,cnt1,cnt,tmp,tmp1,ind,tmp2,j
    MVK 6,cnt
    ZERO j
Loop2:    .trip 6
    ZERO tmp
    ZERO A7
    ZERO A8
    ZERO A10
    LDW *B4[j],A9      ; load the passed frame from memory
(source)
    MVK 31,pos1
    MVK 29,pos2
    MVK 10,cnt1
    ZERO mask
    SET mask,31,31,mask

    MVK 31,i

Loop3:    AND mask,A9,tmp
    SHRU tmp,i,tmp
    MV pos1,ind
    MV pos2,tmp1
    SHL tmp1,5,tmp1
    OR tmp1,ind,ind
[tmp] SET A7,ind,A7
    SHRU mask,1,mask
    SUB pos1,3,pos1
    SUB pos2,3,pos2
    SUB i,1,i
[cnt1] SUB cnt1,1,cnt1
[cnt1] B Loop3

    AND mask,A9,tmp

    SHRU tmp,i,tmp
[tmp] SET A7,0,1,A7

```

```

[tmp] SET A8,31,31,A8
      SHRU mask,1,mask
      SUB i,1,i

      MVK 30,pos1
      MVK 28,pos2
      MVK 10,cnt1

Loop4:      AND mask,A9,tmp
            SHRU tmp,i,tmp
            MV pos1,ind
            MV pos2,tmp1
            SHL tmp1,5,tmp1
            OR tmp1,ind,ind
[tmp] SET A8,ind,A8
            SHRU mask,1,mask
            SUB pos1,3,pos1
            SUB pos2,3,pos2
            SUB i,1,i
[cnt1] SUB cnt1,1,cnt1
[cnt1] B Loop4

            AND mask,A9,tmp

            SHRU tmp,i,tmp
[tmp] SET A8,0,0,A8
[tmp] SET A10,30,31,A10
            SHRU mask,1,mask
            SUB i,1,i

            MVK 29,pos1
            MVK 27,pos2
            MVK 10,cnt1

Loop5:      AND mask,A9,tmp
            SHRU tmp,i,tmp
            MV pos1,ind
            MV pos2,tmp1
            SHL tmp1,5,tmp1
            OR tmp1,ind,ind
[tmp] SET A10,ind,A10
            SHRU mask,1,mask
            SUB pos1,3,pos1
            SUB pos2,3,pos2
            SUB i,1,i
[cnt1] SUB cnt1,1,cnt1
[cnt1] B Loop5

            STW A7,*A4++
            STW A8,*A4++
            STW A10,*A4++

            ADD j,1,j
[cnt] SUB cnt,1,cnt
[cnt] B Loop2

            .endproc A4,B3

            B B3
            NOP 5

```

Block Interleaver:

```
.global _interleaver1_la
.sect ".interleaver1_la"

_interleaver1_la: .proc A4,B4,B3
                  .reg dat1,dat2,tmp,row_cnt,col_cnt,mask,pos,shift_cnt,row

                  MVK 18,col_cnt
                  MVK 17,pos

Loop1:            .trip 18
                  LDW *B4++,dat1

                  MVK 31,shift_cnt
                  ZERO mask
                  SET mask,31,31,mask
                  MVK 32,row_cnt
                  ZERO row
                  ZERO tmp

Loop2:            .trip 32
                  AND mask,dat1,tmp
                  SHRU tmp,shift_cnt,tmp
                  LDW *A4[row],dat2
                  SHL tmp,pos,tmp
                  OR tmp,dat2,dat2

                  STW dat2,*A4[row] ;row]
                  ADD row,1,row
                  SHRU mask,1,mask
                  SUB shift_cnt,1,shift_cnt
[row_cnt] SUB row_cnt,1,row_cnt
[row_cnt] B Loop2

                  SUB pos,1,pos

[col_cnt] SUB col_cnt,1,col_cnt
[col_cnt] B Loop1

                  .endproc A4,B3

                  B B3
                  NOP 5
```

Walsh Encoder: The walsh matrix is generated offline using a C program and every 64 bit row of the Walsh matrix is stored as two 32 bit integers. This saves a lot of memory and also makes accessing the Walsh codes very fast

```

.global _walsh_enc1_la
.sect ".walsh_enc1_la"

_walsh_enc1_la:      .proc A4,B4,A6,B3      ;B4 is interleaver out and A4 is the
new frame(pointers),A6 is pointer to the Hadamard Metrix
.reg   index1,index2,index3,i,shift_cnt,row,dat1,mask,dat2,row_cnt

                MVK 32,row_cnt                ; 576/6 = 96
                ZERO row
Loop1:          .trip 32
                LDW *B4[row],dat1
                ZERO mask
                SET mask,12,17,mask

                AND dat1,mask,index1
                SHRU mask,6,mask
                SHRU index1,12,index1

                AND dat1,mask,index2
                SHRU mask,6,mask
                SHRU index2,6,index2

                AND dat1,mask,index3
                SHRU mask,6,mask

                SHL index1,1,index1
                LDW *A6[index1],dat2
                STW dat2,*A4++
                ADD index1,1,index1
                LDW *A6[index1],dat2
                STW dat2,*A4++

                SHL index2,1,index2
                LDW *A6[index2],dat2
                STW dat2,*A4++
                ADD index2,1,index2
                LDW *A6[index2],dat2
                STW dat2,*A4++

                SHL index3,1,index3
                LDW *A6[index3],dat2
                STW dat2,*A4++
                ADD index3,1,index3
                LDW *A6[index3],dat2
                STW dat2,*A4++

                ADD row,1,row
[row_cnt] SUB row_cnt,1,row_cnt
[row_cnt] B Loop1

                .endproc A4,B3

                B B3
                NOP 5

```

PN-Long and PN-short sequence generator: The PN-long and PN-short sequences are generated simultaneously in the module below. The data frame generated by the Walsh encoder is given to a bit repeater that repeats every bit four times so as to form a frame of length 24576. This frame is then EX-Ored with the Pn-sequences

```

.global pnseq12_la
.sect ".pnseq12_la"

_pnseq12_la:          .proc A4,B4,B3          ; A6 =
regh,A8=regl,B6=fdbckh,B8=fdbckl
    .reg
j,mask,tmp1,tmp2,tmp3,tmp4,fdbck,cnt,cnt1,regh,regl,fdbh,fdbl,bit31,fdb31,pnlong,pnshort,
ort,pos,var1,var2,tmps,regs,out_bit,fdbck1
    ;B4 is repeater_out and A4 is output_data,B6 is the feedback
register and A6 is the register

    ZERO cnt
    MVK 768,cnt1
    MVK 512,regh
    MVK 0,regl
    MVK 0xA,fdbh
    MVK 0x8e6f04ef,fdbl
    MVKH 0x8e6f04ef,fdbl

    MVK 0x4000,regs
    MVK 0x23a1,fdb31

Loop5: .trip 768

    MVK 31,pos
    ZERO pnlong
    ZERO pnshort

    ;pn short
    MVK 32,j

Loop: .trip 32

    ; pn long
    ZERO out_bit
    MVK 1,mask          ;Get the last
bit as output
    AND regl,mask,out_bit
    SHL out_bit,pos,out_bit
    OR    out_bit,pnlong,pnlong

    AND regs,mask,out_bit
    SHL out_bit,pos,out_bit
    OR out_bit,pnshort,pnshort

    SUB pos,1,pos

    AND regh,fdbh,tmp2
    AND regl,fdbl,tmp3

```

;get the feedback value for pn-long
ZERO fdbck

```
MVK 0x00ff,mask
AND tmph,mask,var1
SHL mask,8,mask
AND tmph,mask,var2
SHRU var2,8,var2
XOR var1,var2,var1
```

```
MVK 0x00f0,mask
AND var1,mask,var2
SHRU var2,4,var2
SHRU mask,4,mask
AND var1,mask,var1
XOR var1,var2,var1
```

```
MVK 0x000C,mask
AND var1,mask,var2
SHRU var2,2,var2
SHRU mask,2,mask
AND var1,mask,var1
XOR var1,var2,var1
```

```
MVK 0x0002,mask
AND var1,mask,var2
SHRU var2,1,var2
SHRU mask,1,mask
AND var1,mask,var1
XOR var1,var2,var1
```

```
MV var1,fdbck
```

```
MVK 0x0000ffff,mask
MVKH 0x0000ffff,mask
AND tmp1,mask,var1
SHL mask,16,mask
AND tmp1,mask,var2
SHRU var2,16,var2
XOR var1,var2,var1
```

```
MVK 0x0000ff00,mask
MVKH 0x0000ff00,mask
AND var1,mask,var2
SHRU var2,8,var2
SHRU mask,8,mask
AND var1,mask,var1
XOR var1,var2,var1
```

```
MVK 0x00f0,mask
AND var1,mask,var2
SHRU var2,4,var2
SHRU mask,4,mask
AND var1,mask,var1
XOR var1,var2,var1
```

```
MVK 0x000C,mask
AND var1,mask,var2
SHRU var2,2,var2
SHRU mask,2,mask
AND var1,mask,var1
```

```

        XOR var1,var2,var1

MVK 0x0002,mask
    AND var1,mask,var2
    SHRU var2,1,var2
    SHRU mask,1,mask
    AND var1,mask,var1
    XOR var1,var2,var1

    XOR var1,fdbck,fdbck

;shift the register
    ZERO bit31
    MVK 1,mask
    AND regh,mask,bit31
    SHRU regl,1,regl
[bit31] SET regl,31,31,regl
    SHRU regh,1,regh
[fdbck] SET regh,9,9,regh                                ; put the feedback

    ZERO fdbck
    AND regs,fdbs,tmps

MVK 0x00ff,mask
    AND tmps,mask,var1
    SHL mask,8,mask
    AND tmps,mask,var2
    SHRU var2,8,var2
    XOR var1,var2,var1

    MVK 0x00f0,mask
    AND var1,mask,var2
    SHRU var2,4,var2
    SHRU mask,4,mask
    AND var1,mask,var1
    XOR var1,var2,var1

    MVK 0x000C,mask
    AND var1,mask,var2
    SHRU var2,2,var2
    SHRU mask,2,mask
    AND var1,mask,var1
    XOR var1,var2,var1

    MVK 0x0002,mask
    AND var1,mask,var2
    SHRU var2,1,var2
    SHRU mask,1,mask
    AND var1,mask,var1
    XOR var1,var2,var1

    MV var1,fdbck

    SHRU regs,1,regs
[fdbck] SET regs,14,14,regs

[j] SUB j,1,j
[j] B Loop

```

```

; multiply with the data

LDW *B4[cnt],tmp1
XOR tmp1,pnlong,tmp1
XOR tmp1,pnshort,tmp1
STW tmp1,*A4[cnt]
    ADD cnt,1,cnt
[cnt1] SUB cnt1,1,cnt1
[cnt1] B Loop5

.endproc A4,B3

B B3
NOP 5

```

The data obtained after spreading by PN-sequences is stored in a file and then decoded using a C program. This decoder is only for testing purposes and hence was not implemented on the DSP.

The decoder code is shown below.

```

#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<math.h>
#include<time.h>

/* Declare all variables */
/*int *input_data;*/
int framelength;
int *frame;
int *walsh_in;
int interleaver[32][18];
int *w_block6;
int **H_Matrix;
int *Reg;
int *frame1;

int Wlookup[64] =
{0,63,31,32,15,48,16,47,7,56,24,39,8,55,23,40,3,60,28,35,12,51,19,44,4,59,27,36
,11,52,20,43,1,62,30,33,14,49,17,46,6,57,25,38,9,54,22,41,2,61,29,34,13,50,18,4
5,5,58,26,37,10,53,21,42};
int walsh[64][64];
/*int input_data[184];

/* function declaration*/
void Hadamard();
void read_data();
void pn_short();
void pn_long();
void unrepeater();
void walsh_decode();
void interleave();
void decoder();
void getstring();
void show_data(int *data,int length);
void compare_bits();

```

```

void main()
{
    time_t start,stop;
    time(&start);
    /* Read Input data from File*/
    Hadamard();
    framelength = 24576;
    frame = (int *)malloc(sizeof(int) * 24576);
    read_data();
    pn_short();
    pn_long();
    walsh_in = (int *)malloc(sizeof(int)*6144);
    unrepeater();
    free(frame);
    framelength = 6144;
    walsh_decode();
    framelength = 576;
    interleave();
    frame = (int *)malloc(sizeof(int)*576);
    frame1 = (int *)malloc(sizeof(int)*192);
    decoder();
    framelength =192;
    getstring();
    /*compare_bits()*/
    framelength=184;
    time(&stop);
    printf("%3.3f",difftime(stop,start));
}

void Hadamard()
{
    int **Htmp;
    int i, j, n_rows=0,n_cols=0,row,col,size=0,w_row;
    double order;
    Htmp = (int **)malloc(sizeof(int *)*2);
    Htmp[0] = (int *)malloc(sizeof(int)*2);
    Htmp[1] = (int *)malloc(sizeof(int)*2);
    Htmp[0][0] = 0;
    Htmp[0][1] = 0;
    Htmp[1][0] = 0;
    Htmp[1][1] = 1;

    for(i=1;i<6;i++)
    {
        order = i;
        n_rows = (int)pow(2,order);
        n_cols = (int)pow(2,order);
        size = (int)pow(2,(order+1));

        H_Matrix = (int **)malloc(sizeof(int *) *size);
        for(j=0;j<size;j++)
        {
            H_Matrix[j] = (int *)malloc(sizeof(int) *size);
        }

        for(row=0;row<n_rows;row++)
        {
            for(col=0;col<n_cols;col++)
            {
                H_Matrix[row][col] = Htmp[row][col];
            }
        }
    }
}

```

```

        H_Matrix[row][col+n_cols]= Htmp[row][col];
        H_Matrix[row+n_rows][col] =Htmp[row][col];
        H_Matrix[row+n_rows][col+n_cols] = 1-Htmp[row][col];

    }
}
Htmp = (int **)malloc(sizeof(int *) * (int)pow(2,order+1) );
for(j=0;j<pow(2,(order+1));j++)
{
    Htmp[j] = (int *)malloc(sizeof(int) *(int)pow(2,order+1) );
}

for(row=0;row<(pow(2,order+1));row++)
{
    for(col=0;col<(pow(2,order+1));col++)
    {
        Htmp[row][col] = H_Matrix[row][col];
    }
}

for(row=0;row<64;row++)
{
    w_row=Wlookup[row];
    for(col=0;col<64;col++)
    {
        walsh[w_row][col] = H_Matrix[row][col];
    }
}
}

void read_data()
{
    FILE *fp;
    int i,tmp,j,tmp1,n=0,mask;
    fp = fopen("testasml.txt","r");
    for(i=0;i<768;i++)
    {
        fscanf(fp,"%d",&tmp);
        for(j=0;j<32;j++)
        {
            mask = (int)pow(2,31-j);
            tmp1 = (mask & tmp)>> (31-j);
            frame[n++] = tmp1;
        }
    }
}

void pn_short()
{
    FILE *fp;
    int *pn_seq,len=24576;
    int seed[15] = {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int fdbck[15] = {0,1,0,0,0,0,1,1,1,1,0,1,0,0,0,1};
    int feedback=0,i,j,k,n=0;

    pn_seq = (int *)malloc(sizeof(int)*24585);
    Reg = (int *)malloc(sizeof(int)*15);
    printf("/*** PN SHORT **/");
}

```

```

        }
        for(k=42;k>0;k--)
        {
            Reg[k] =Reg[k-1] ;
        }
        Reg[0] = feedback;

    }

    for(i=0;i<len;i++)
    {
        /*      printf("%d*d=",frame[i],pn_seq[i]);*/
        frame[i]=(frame[i])^(pn_seq[i]);
        /*printf(":%d \n",frame[i]);*/
        fprintf(fp1,"%d\n",pn_seq[i]);
        fprintf(fp,"%d\n",frame[i]);
    }
    fclose(fp);
    free(pn_seq);
}

void unrepeater()
{
    FILE *fp;
    int j,k=0;
    int factor=4;
    printf("/**UN-REPEATER **/\n");
    fp = fopen("walshin_de.txt","w");
    for(j=0;j<framelength;j+=factor)
    {
        walsh_in[k] =frame[j];
        k++;
    }
    /*for(j=0;j<6144;j++)
    {
        fprintf(fp,"%d\n",walsh_in[j]);
    }*/
    fclose(fp);
}

void walsh_decode()
{
    FILE *fp;
    int j,cnt=0,i,maxval=0,maxind=0,k,tmp,mask;
    int *walsh_row;
    /* walsh enoder*/
    w_block6 = (int *)malloc(sizeof(int) *576);
    walsh_row = (int *)malloc(sizeof(int)*64);

    fp=fopen("walsh_dec.txt","w");
    printf("/** WALSHEDECODER **/\n");
    for(i=0;i<framelength;i+=64)
    {
        for(j=0;j<64;j++)
        {
            walsh_row[j]=walsh_in[i+j];
        }
        maxval =0;
        for(k=0;k<64;k++)

```

```

        {
            tmp = 0 ;
            for(j=0;j<64;j++)
            {
                tmp = tmp + (2*walsh_row[j]-1) * (2*walsh[k][j]-1);
            }
            if (tmp > maxval)
            {
                maxval = tmp;
                maxind = k;
            }
        }
        printf("%d ",maxind);
        mask =0;
        for (k=5;k>=0;k--)
        {
            mask = (int)pow(2,k);
            tmp = mask & maxind;
            w_block6[cnt] = (tmp>>(k));
            cnt++;
        }
    }
    /*for(i=0;i<576;i++)
    {
        fprintf(fp,"%d\n",w_block6[i]);
    }*/
    printf("finished walsh decoder");
    fclose(fp);
}

```

```

void interleave()
{
    FILE *fp;
    int row,col;
    printf("/** Inside Interleaver **/\n");
    fp = fopen("interleav_de.txt","w");
    for(row = 0;row<32;row++)
    {
        for(col = 0;col<18;col++)
        {
            interleaver[row][col] = w_block6[row*18+col];
        }
    }
    /*for(row=0;row<32;row++)
    {
        for(col=0;col<18;col++)
        {
            fprintf(fp,"%d\n",interleaver[row][col]);
        }
    }*/
    fclose(fp);
}

```

```

void decoder()
{
    int k=0,row,col,n=0,i=0;
    int factor=3;
    printf("/** DECODER **/\n");
    for(col=0;col<18;col++)
    {
        for(row=0;row<32;row++)

```

```

        {
            frame[n] = interleaver[row][col];

            n++;
        }
    }
    for(i=0;i<576;i+=factor)
    {
        frame1[k] =frame[i];
        k++;
    }
    for(i=0;i<184;i++)
        printf("%d ",frame1[i]);
}

```

```

void getstring()
{
    int i,k,tmp1,c,n=0,j,cnt=0;
    char input[23];
    printf("/*** GET STRING **/\n");
    for(i=0;i<23;i++)
    {
        tmp1=0;
        for(j=0;j<8;j++)
        {
            c = frame1[cnt];
            tmp1 =(tmp1 |(c << (7-j)) );
            cnt++;
        }
        input[n] = (char)tmp1;
        printf("%d ",tmp1);
        n++;
    }
    printf("\n***");
    for(k=0;k<23;k++)
    {
        printf("%c",input[k]);
    }
    printf("***\n");
}

```

```

void show_data(int *data,int length)
{
    int i;
    for(i=0;i<length;i++)
    {
        printf("%d ",*data++);
    }
}

```

```

void compare_bits()
{
    FILE *fp1;
    FILE *fp2;
    int arr1[576],i,j,tmp1,tmp2;
    int arr2[576];
    printf("\n***Compare bits***\n");
    fp1=fopen("interleav_de.txt","r");
    fp2=fopen("interleaved.txt","r");
    for(i=0;i<576;i++)

```

```

{
/*   for(j=0;j<18;j++)
    {*/
      if(feof(fp1))
      {
          printf("EOF at %d ",i);
          break;
      }

      fscanf(fp1,"%d",&tmp1);
      arr1[i]=tmp1;
      fscanf(fp2,"%d",&tmp2);
      arr2[i] = tmp2;
      tmp1=(tmp1-tmp2);
      if (tmp1 !=0 )
      {
          printf("i::%d ",i);
          printf("ind 1 =%d, ind2 = %d\n",arr1[i],arr2[i]);
      }

      /*}
      printf("\n");*/
    }
    j=0;
    for(i=0;i<576;i++)
    {
        if((arr1[i] - arr2[i]) !=0)
        {
            j++;
        }
    }
    fclose(fp1);
    fclose(fp2);
    printf("\nError :%d\n",j);
}

```

Results and Performance: The program being in Linear Assembly is highly optimized as the Assembler optimizer does a fairly good job of reducing computation times by optimizing the code. The number of instruction cycles required from the encoder to the final PN-sequence spreading is 2801221.

Processor	Clock	Period	Time required
TMS 3206701	166MHz	6.024ns	16.9ms
TMS 3206201	200MHz	5ns	14ms

Table 4:Results

