Lab 4
Pseudo Random Sequences
EE 445S

Enoc Balderas Daniel Diamont

April 1, 2019

Date Performed: March 25, 2019
Instructor: Professor Evans

1 Introduction

For this lab we explored pseudo random sequences. The main sequence that we observed is the m-sequence (max length). We created the m-sequence using a simple shift register (SSRG). Finally we used the m-sequence at the transmitter to scramble a bit and then used the same sequence at the reciever to descramble the bit.

2 Methods

We started off by creating a $[5, 2]_s$ SSRG to implement the sequence. We tested the sequence by profiling the output with an oscilloscope and making sure that it was periodic. Next we used the m-sequence to scramble a transmit bit, and subsequently descramble the bit at the reciever. This was achieved by starting the SSRG of the transmitter and receiver in the same initial state, and XORing the input bit with the resulting sequence.

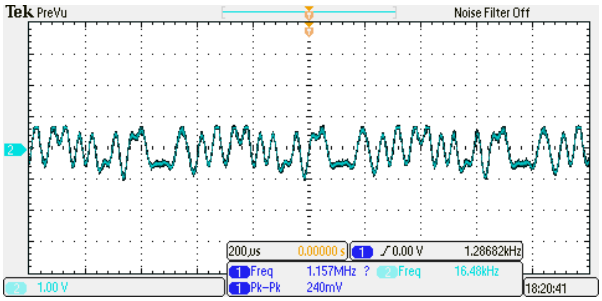

Figure 1: SSRG $[5, 2]_s$.

3 Results

3.1 DSK Implementation of PN Sequence Generation

Count	State
[0]	10000
[1]	01000
[2]	10100
[3]	01010
[4]	10101
[5]	11010
[6]	11101
[7]	01110
[8]	10111
[9]	11011
[10]	01101
[11]	00110
[12]	00011
[13]	10001
[14]	11000
[15]	11100
[16]	11110
[17]	11111
[18]	01111
[19]	00111
[20]	10011
[21]	11001

Count	State
[22]	01100
[23]	10110
[24]	01011
[25]	00101
[26]	10010
[27]	01001
[28]	00100
[29]	00010
[30]	00001
[31]	10000
[32]	01000
[33]	10100
[34]	01010
[35]	10101
[36]	11010
[37]	11101
[38]	01110
[39]	10111
[40]	11011
[41]	01101
[42]	00110
[43]	00011
[44]	10001
[45]	11000
[46]	11100
[47]	11110
[48]	11111
[49]	01111
[50]	00111
[51]	10011
[52]	11001
[53]	01100
[54]	10110
[55]	01011

Count	State
[56]	00101
[57]	10010
[58]	01001
[59]	00100
[60]	00010
[61]	00001
[62]	10000
[63]	01000
[64]	10100
[65]	01010
[66]	10101
[67]	11010
[68]	11101
[69]	01110
[70]	10111
[71]	11011
[72]	01101
[73]	00110
[74]	00011
[75]	10001
[76]	11000
[77]	11100
[78]	11110
[79]	11111
[80]	01111
[81]	00111
[82]	10011
[83]	11001
[84]	01100
[85]	10110
[86]	01011
[87]	00101
[88]	10010
[90]	01001
[91]	00100
[92]	00010
[93]	00001
[94]	10000
[95]	01000
[96]	10100
[97]	01010
[98]	10101
[99]	11010

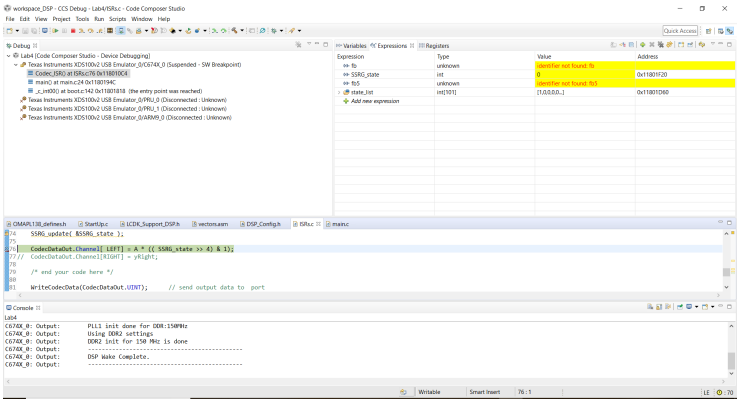

Figure 2: PN Sequence Generation Profiling.

The PN Generation took 70 cycles.

PN Generation C Code:

```
#define LEFT  0
#define RIGHT 1
#define A       32000
#define TAPS    5

int SSRG_state = 1;

int SSRG_update(int * state) {
    // compute feedback
    int fb5 = (*state)&1;
    int fb = ((*state >> 3)&1) ^ fb5;

    // shift state register
    *state = *state >> 1;

    // add feedback
    *state = (fb << 4) | *state;
    return fb5;
}
```

3.2 DSK Implementation of Data Scrambler and Descrambler

Count	Scrambler Output	Descrambler Output
[0]	0	1
[1]	1	1
[2]	1	1
[3]	1	1
[4]	1	1
[5]	0	1
[6]	1	1
[7]	0	1
[8]	1	1
[9]	0	1
[10]	0	1
[11]	0	1
[12]	1	1
[13]	0	1
[14]	0	1
[15]	1	1
[16]	1	1
[17]	1	1
[18]	0	1
[19]	0	1
[20]	0	1
[21]	0	1

Count	Scrambler Output	Descrambler Output
[22]	0	1
[23]	1	1
[24]	1	1
[25]	0	1
[26]	0	1
[27]	1	1
[28]	0	1
[29]	1	1
[30]	1	1
[31]	0	1
[32]	1	1
[33]	1	1
[34]	1	1
[35]	1	1
[36]	0	1
[37]	1	1
[38]	0	1
[39]	1	1
[40]	0	1
[41]	0	1
[42]	0	1
[43]	1	1
[44]	0	1
[45]	0	1
[46]	1	1
[47]	1	1
[48]	1	1
[49]	0	1
[50]	0	1
[51]	0	1
[52]	0	1
[53]	0	1
[54]	1	1
[55]	1	1

Count	Scrambler Output	Descrambler Output
[56]	0	1
[57]	0	1
[58]	1	1
[59]	0	1
[60]	1	1
[61]	1	1
[62]	0	1
[63]	1	1
[64]	1	1
[65]	1	1
[66]	1	1
[67]	0	1
[68]	1	1
[69]	0	1
[70]	1	1
[71]	0	1
[72]	0	1
[73]	0	1
[74]	1	1
[75]	0	1
[76]	0	1
[77]	1	1
[78]	1	1
[79]	1	1
[80]	0	1
[81]	0	1
[82]	0	1
[83]	0	1
[83]	0	1
[84]	1	1
[85]	1	1
[86]	0	1
[87]	0	1
[88]	1	1
[89]	0	1
[90]	1	1
[91]	1	1
[92]	0	1
[93]	1	1
[94]	1	1
[95]	1	1
[96]	1	1
[97]	0	1
[98]	1	1
[99]	0	1

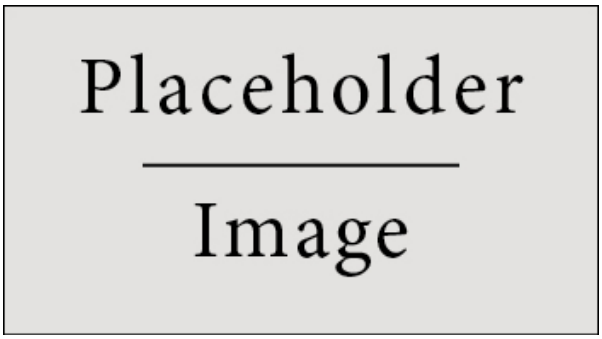

Figure 3: Scrambler and Descrambler Profiling

The Scrambler took () cycles. The Descrambler took () cycles.

Scrambler and Descrambler C Code:

```
#define LEFT  0
#define RIGHT 1
#define A       32000
#define TAPS    5

int DS_state = 1;
int DD_state = 1;

int scramble(int * state, int in) {
    // compute feedback
    int fb5 = (*state)&1;
    int fb = ((*state >> 3)&1) ^ fb5;

    // shift state register
    *state = *state >> 1;

    // add feedback
    *state = (fb << 4) | *state;
    return fb5^in;
}

int descramble(int * state, int in) {
    // compute feedback
    int fb5 = (*state)&1;
    int fb = ((*state >> 3)&1) ^ fb5;

    // shift state register
```

```
    *state = *state >> 1;

    // add feedback
    *state = (fb << 4) | *state;
    return fb5^in;
}
```

3.3 Autocorrelation

Autocorrelation Code:

```
pn = [1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1,
sc = [0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0,

pn(pn == 0) = -1;
sc(sc == 0) = -1;

tmp1 = [pn pn];
tmp2 = [sc sc];

s1 = fft(tmp1);
pn_corr = ifft(s1.*conj(s1))/length(tmp1);

s2 = fft(tmp2);
sc_corr = ifft(s2.*conj(s2)/length(tmp2));

plot(pn_corr);
title('PN circular autocorrelation');
xlabel('chip offset');
ylabel('R');
saveas(gcf, '../../report/lab_report_4/img/pn_corr.png');

figure;
plot(sc_corr);
title('scrambled circular autocorrelation');
xlabel('chip offset');
ylabel('R');
saveas(gcf, '../../report/lab_report_4/img/sc_corr.png');
```

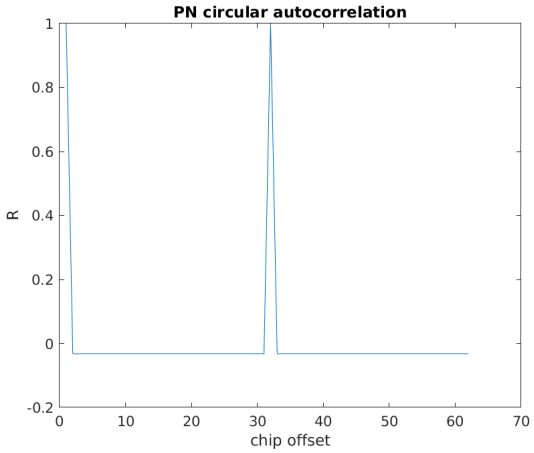

Figure 4: PN Sequence Autocorrelation.

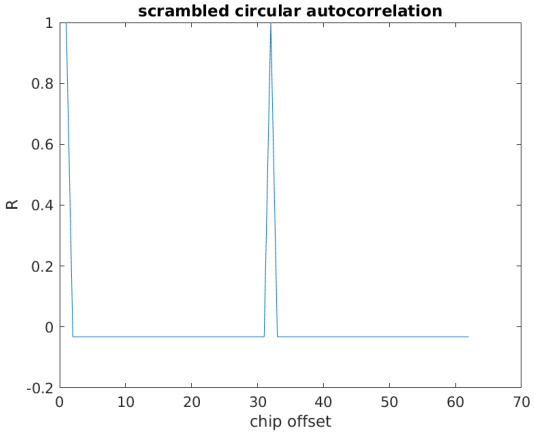

Figure 5: Scrambler Autocorrelation.