

## II. LAB

**Software Required:** NI LabVIEW 2012, NI LabVIEW 4.3 Modulation Toolkit.

**Functions and VI (Virtual Instrument) from the LabVIEW software to be used in this lab:** MT Generate Bits (VI), Tick Count (Function), Multiply (Function), Numeric Constant (Function), Case Structure.

### **PART-1: RANDOM BIT SEQUENCE GENERATION**

**INPUTS:** Packet length (in bits) (long [32-bit integer (-2147483648 to 2147483647)]).

**OUTPUTS:** Output bit stream (as key bit sequence) (1-D array of unsigned byte [8-bit integer (0 to 255)]).

A virtual instrument template (“student\_source.vit”) is provided to you with these inputs and outputs. The template should be populated according to instructions. Then, newly created VIs are plugged in the simulator.

### **PROCEDURE**

Please read *LabVIEW Handbook* (active link to the page) document first, if you’re not familiar with LabVIEW. There, you can find detailed information on palettes, tools, and functions along with their screenshots.

In this part, we develop a virtual instrument to generate a random bit sequence. We start by opening a source template file (“student\_source.vit”) which has previously set modulation parameters in it. The default modulation parameters for this experiment are as shown in Fig. 1 below. We are concerned here only with the number of iterations and sequence length. We mention the role of modulation type and noise power (dB) in Part 3.

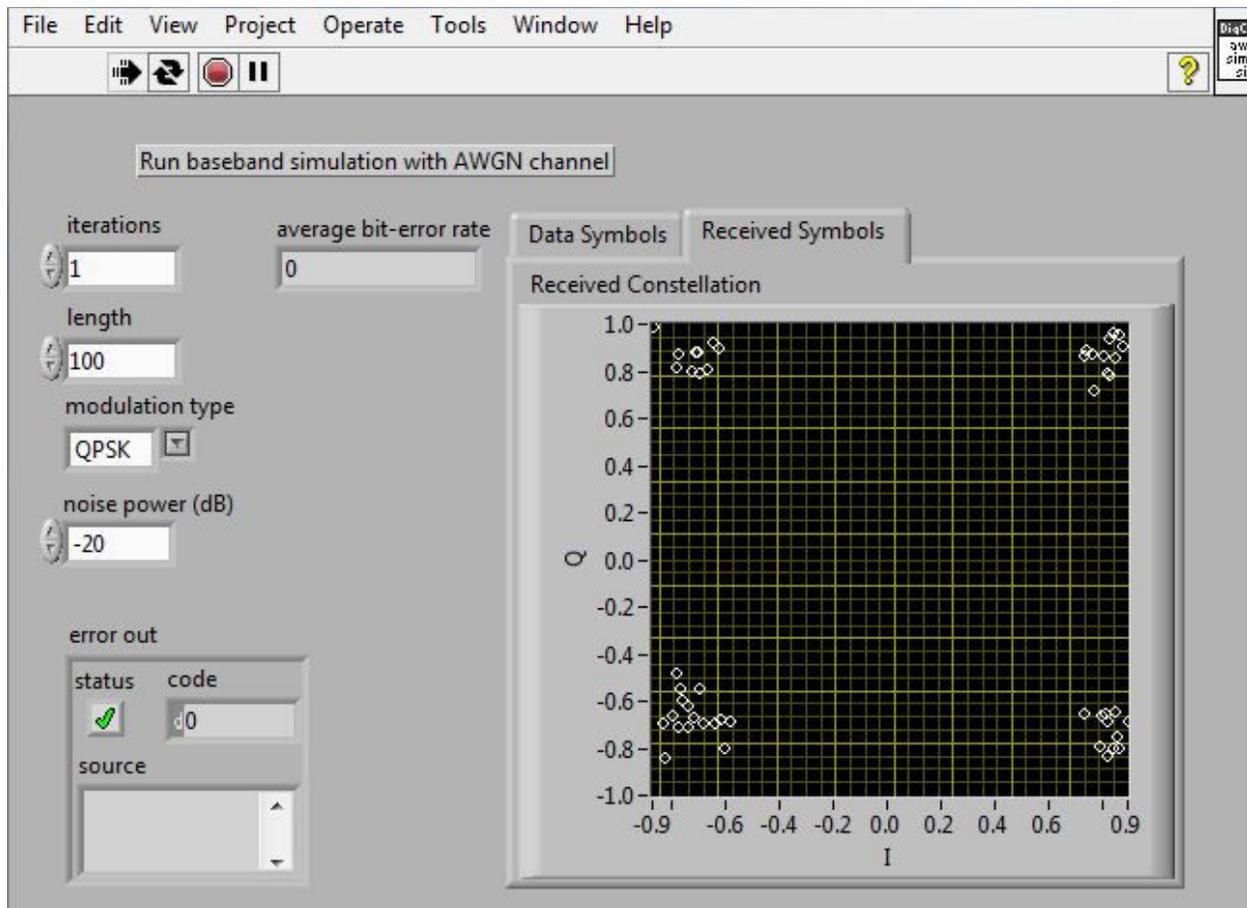


Fig. 1 – Default modulation parameters used to generate random bit sequence

After modulation parameters are set, the pre-defined package length (bits) is used as an input to the MT generate bits function of LabVIEW, which generates a random bit sequence which approximately has the following distribution: The bits are stochastically independent and each is uniformly distributed in the set  $\{0,1\}$ .

MT generate bits function uses the Mersenne-Twister (MT) random bit sequence generation algorithm. This generates a PseudoNoise (PN) sequence of random bits. This sequence, as said, has approximately independent and equally likely bits. This is done as follows.

The MT generate bits function generates an m-sequence, which is a common example of PN sequences. This PN sequence is a periodic sequence of length  $L = 2^m - 1$  bits for some integer  $m$ , and is generated by a Linear Feedback Shift Registers (LFSR) as shown in Fig. 2.

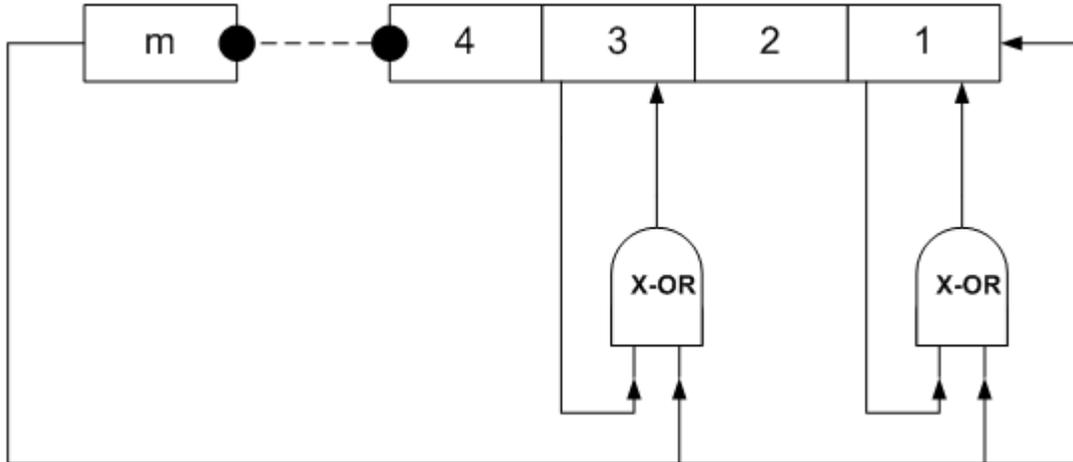


Fig. 2 – Galois implementation of LFSR for PN sequence generation

The LFSR contains  $m$  shift registers, each containing one bit. The shift registers set are initially filled with an  $m$ -bit seed. The initial seed value is selected as a random number in order to ensure randomness of the output. This is explained later.

We now explain how the LFSR works starting from a given initial seed. We recall that the X-OR (Exclusive Or) logic gate sums two 1-bit values and returns the result in mod 2 (i.e., 0 or 1). Specifically, if inputs are  $\{0, 0\}$  or  $\{1, 1\}$  then the output is  $\{0\}$ , and, if the inputs are  $\{0, 1\}$  or  $\{1, 0\}$  then the output is  $\{1\}$ .

The following example demonstrates the operation of the LFSR.

Initial seed: 000000010101001

The circuitry is shown in Fig. 3 below:

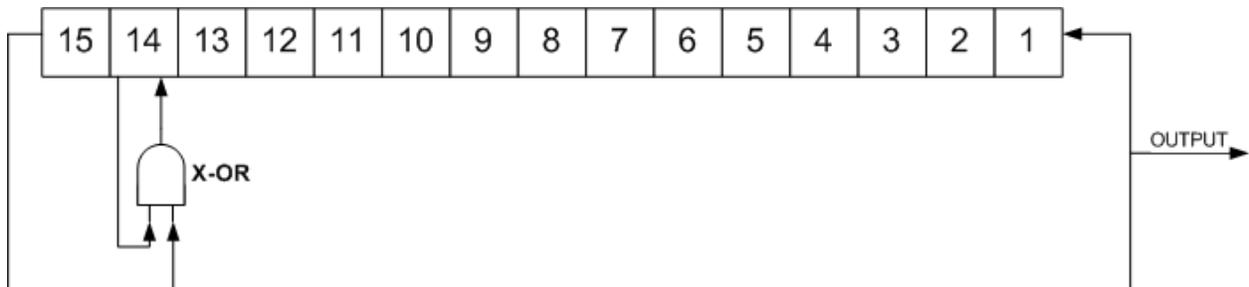


Fig. 3 – LFSR-Galois for PN sequence generation example

Seed	Output
000000010101001	0
000000101010010	0
000001010100100	0
000010101001000	0

000101010010000	0
001010100100000	0
010101001000000	0
101010010000000	1
110100100000001	1
001001000000011	0
010010000000110	0
100100000001100	1
101000000011001	1
110000000110011	1
000000001100111	0
000000011001110	0

Table 1 – Bit generation process

At each iteration, the bit in the last position (15) is output by the LFSR. In order to go from one iteration to the next, bits are shifted to the left (bit-1 becomes bit-2 and bit-15 becomes bit-1). After shifting is done, bit-14 and bit-15 values are summed and bit-15 value is replaced by the result. For instance, in Table-1, 101010010000000 is shifted one position to the left (010100100000001), then bits 14 and 15 are summed (0+1=1), finally this result is placed on bit-15 of the seed after shift (110100100000001). Please note that the rest of the (shifted) seed sequence stays the same. The procedure continues, and the output bits make the PN sequence.

It remains to discuss how to generate the initial seed. In order to generate an approximately random initial seed number, the VI reads the millisecond timer value from the tick count function. The value of the millisecond timer ranges from 0 to  $[(2^{32})-1]$ .

Since the time at which this reading is done is not pre-determined, one can consider this number to be approximately random. The initial seed is generated by adding a tick clock, a multiplier, a constant, and an increment operation as formulated in equation (1.1):

$$\text{Initial Seed} = \text{Tick Count} * \text{Constant} + 1 \quad (1.1)$$

We now detail the construction of the VI at hand in LabVIEW:

#### a) Setup

\* Open the LabVIEW program (Start > All Programs > National Instruments > LabVIEW 2012 > LabVIEW 2012)

\* Open “student\_source.vit” file (Open Existing (OR Ctrl+O) > Select “student\_source.vit” > OK). A basic graphic user interface (GUI) opens. (NOTE: If a warning window pops up, click on “Ignore” option.)

\* Open Block Diagram (Window > Show Block Diagram (OR Ctrl+E)).

\* Enable Functions Palette which contains virtual instruments (VIs), functions and constants needed to create the block diagram (View > Functions Palette).

\* Place a Case Structure box (Functions Palette > Programming > Structures > select and drag Case Structure to cover the area as shown in Fig. 4) in order to tell VI what to do in case of an error. Case structures have two cases, by default, *true (no error)* and *false (error)*. True case is where you operate.

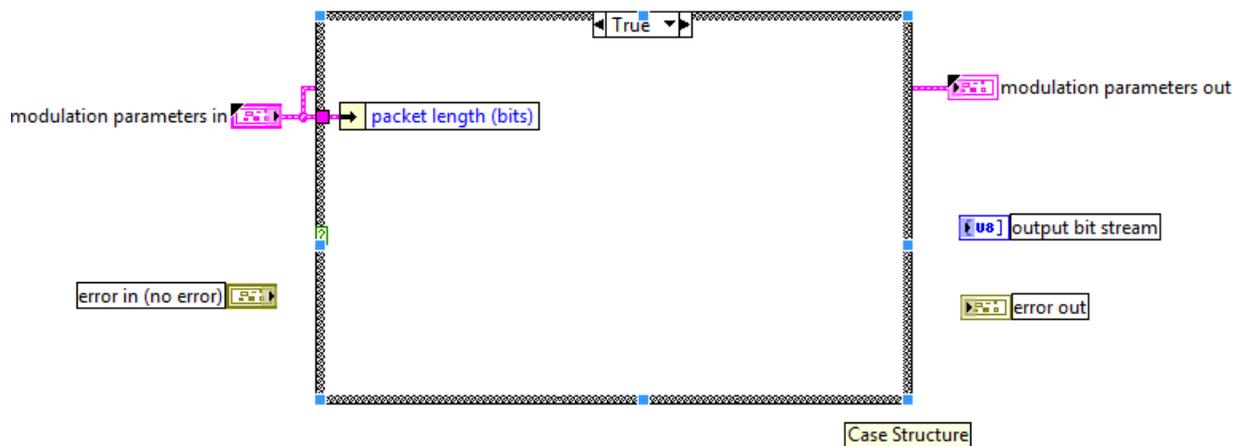


Fig. 4 – Representation of how to place case structure on “student\_source.vi”

### b) Define bit sequence length:

\* Packet length (bits) object is used to define the bit sequence length. It is an input to MT generate bits function.

\* Move “output bit stream” cluster inside the case structure (A cluster is a LabVIEW data type which is used to group different types of elements. For more information please see LabVIEW Handbook document).

\* Connect the output of the “error in (no error)” cluster to the interrogation mark located on the left side border of the Case Structure Box.

### c) Generate random bit sequence:

\* Place the following four components inside the case structure in order to generate the 32-bit initial seed sequence:

Tick Count (Functions Palette > Programming > Timing > Tick Count),

Multiply (Functions Palette > Programming > Numeric > Multiply),

Numeric Constant (Functions Palette > Programming > Numeric > Numeric Constant). Once you place it, click inside the box and type the number “2”,  
Increment Function (Functions Palette > Programming > Numeric > Increment).

\* Place an MT Generate Bits VI (Functions Palette > RF Communications > Modulation > Digital > MT Generate Bits).

\* Connect millisecond timer value terminal of *Tick Clock* to y terminal of *Multiply* function.

\* Connect output terminal of *Numeric Constant* to x terminal of *Multiply* function.

\* Connect x\*y terminal of *Multiply* function to x terminal of *Increment* function.

\* Connect the output of *packet length (bits)* to the total bits (128) terminal of the *MT Generate Bits*.

\* Connect  $x + 1$  terminal of the *Increment* function to *seed in* terminal of the *MT Generate Bits*.

#### **d) Output bits:**

\* Connect *output bit stream* terminal of *MT Generate Bits* to input of *output bit stream* numeric array.

#### **e) Necessary connections**

In this section, necessary error in – error out connections are made in order to finalize VI design.

\* Connect *error in (no error)* to *error out* through corresponding MT generate bits (*error in* and *error out*) terminals and the case structure box. If you fail to do this step, you cannot run your VIs. You need to connect *error in (no error)* to *error out* through case structure box for *False (Error)* case, too.

At the end, your block diagram should look like the one below in Fig. 5:

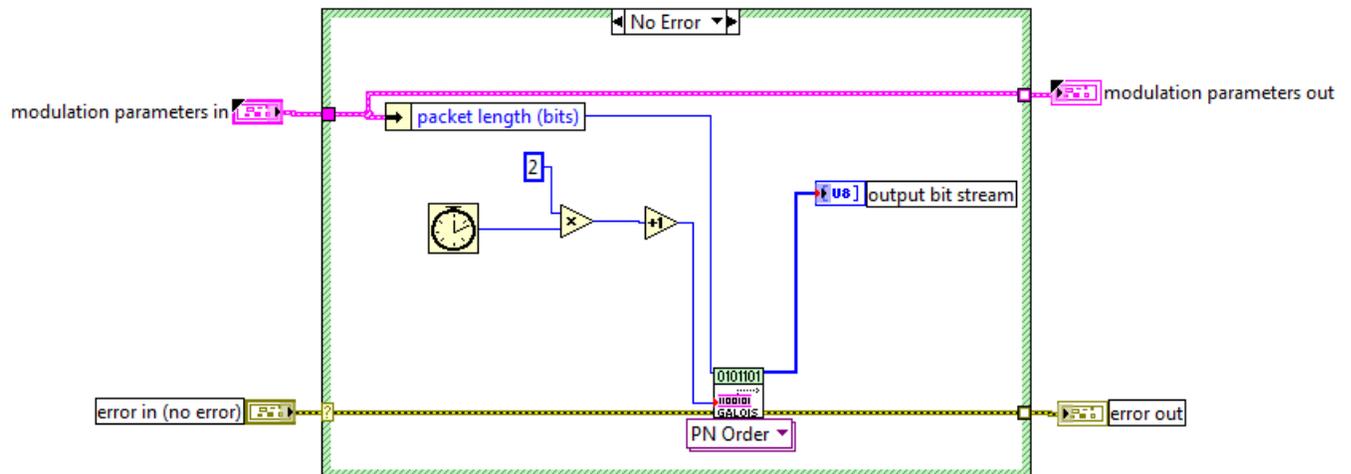


Fig. 5 – Complete block diagram

For visual reference, check this video out: [See the video: How to build the source.vi](#)

\* In order to check if your file has errors click on Operate > Run (OR Ctrl+R). If no error window is generated, then your file is good.

\* SAVE it AS a new file by adding your name at the end (example: if your name is John, then the new file name is “student\_source\_John.vi”).

## **PART-2: ERROR DETECTION**

**INPUTS:** Key bit sequence (as the original bit sequence) (1-D array of unsigned byte [8-bit integer (0 to 255)]) and input bit sequence (as the estimated bit sequence which is the original bit sequence after modulator, AWGN channel, and decoder) (1-D array of unsigned byte [8-bit integer (0 to 255)]).

**OUTPUTS:** Bit-error rate (double [64-bit real ( $\approx 15$  digit precision)]).

## **PROCEDURE**

In this part, we develop a VI that calculates the bit error rate (BER). We start with opening an error detection template file (“student\_error\_detect.vit”). This VI has two inputs: *Input Bit Sequence* and *Key Bit Sequence*. Key bit sequence is the random bit sequence we generated at the output of our source file. This corresponds to the transmitted bit sequence. The input bit sequence is the same source output sequence after going through modulation, Additive White Gaussian Noise (AWGN) channel, and decoder. The input bit sequence is hence the bit sequence detected by the receiver. We discuss modulation, channel and decoder in the following labs. For the time being, it suffices to say that, due to the inevitable noise at the receiver side, there is some probability that the each bit is not decoded correctly.

The BER is the fraction of bits that have been incorrectly decoded. In order to calculate the BER, we first compute the Hamming distance between the transmitted and the received bit sequences. The Hamming distance counts the number of different bits between the two sequences. For example, if the transmitted sequence is {0, 1, 0, 1} and the detected sequence is {1, 0, 1, 1}, then the Hamming distance is 3 (because the first, second, and third digits are different). BER is the ratio of the Hamming distance to total number of bits. Therefore, the BER is given as:

$$BER = \frac{|A_k - \hat{A}_k|}{n} \quad (1.2)$$

where  $|A_k - \hat{A}_k|$  is Hamming Distance between the key bit (transmitted) sequence ( $A_k$ ) and the input bit (detected) sequence ( $\hat{A}_k$ ). and n is total number of bits sent.

We now detail the construction of this VI in LabVIEW:

#### **a) Setup**

\* Open the LabVIEW program (Start > All Programs > National Instruments > LabVIEW 2012 > LabVIEW 2012)

\* Open “student\_error\_detect.vit” file (Open Existing (OR Ctrl+O) > Select “student\_error\_detect.vit” > OK). A basic graphic user interface (GUI) opens. (NOTE: If a warning window pops up, click on “Ignore” option.)

\* Open Block Diagram (Window > Show Block Diagram (OR Ctrl+E)).

\* Enable Functions Palette which contains virtual instruments (VIs), functions and constants needed to create the block diagram (View > Functions Palette).

\* Place a Case Structure box (Functions Palette > Programming > Structures > select and drag Case Structure to cover the area as shown in Fig. 6).

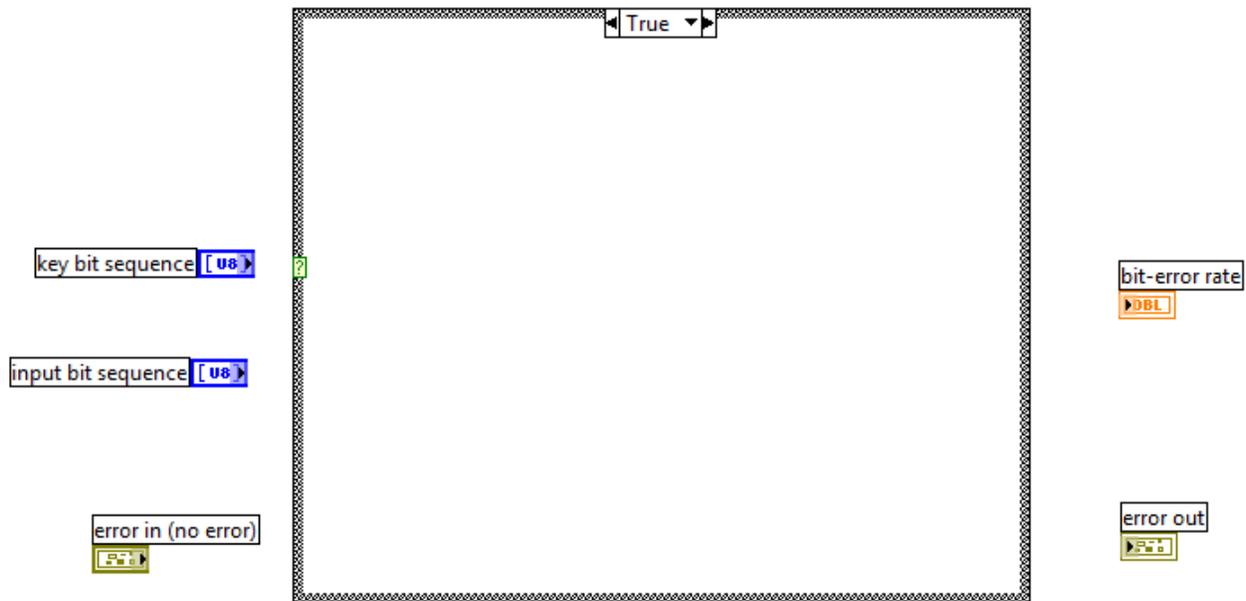


Fig. 6 - Representation of how to place case structure on "student\_error\_detect.vi"

#### b) Input bit sequence and Key bit sequence:

- \* Move "**key bit sequence**" and "**input bit sequence**" inside the Case Structure box by simply selecting, dragging, and dropping.
- \* Move "**bit-error rate**" inside the Case Structure box, this is the output of our error detection vi.
- \* Insert a For Loop (Functions Palette > Programming > Structures > For Loop) inside the case structure box. We use this to reiterate hamming distance and bit sequence length calculations for all incoming bits.

#### c) Calculate total bit sequence length:

- \* Place an Array Size (Functions Palette > Programming > Array > Array Size) which returns the size of the array at its input. Then, connect the key bit sequence output to *array* input of it.
- \* Connect *size (s)* output of Array size to *count (N)* terminal of For Loop.
- \* Place a Max & Min (Functions Palette > Programming > Comparison > Max & Min) function which receives values from its two inputs and gives the maximum and minimum of them as separate outputs. We only use the *max(x,y)* output of this function.
- \* Connect *count (N)* terminal of For Loop to *x* terminal of Max & Min function.

\* Right click on  $y$  terminal of Max & Min function and create a constant (Right click menu > Create > Constant). Change the constant value to 1.

#### **d) Calculate total Hamming Distance:**

\* Place an Index Array (Functions Palette > Programming > Array > Index Array) inside the For Loop which returns the element or sub-array of the input sequence at the current index number of for loop.

\* Place a Not Equal function (Functions Palette > Programming > Comparison > Not Equal) inside the for loop which compares input bit sequence and key bit sequence bit-by-bit and give a Boolean result of True or False.

\* Connect key bit sequence to  $x$  input of not equal function and *element or subarray* output of index array to  $y$  input of it.

\* Place a Boolean To function (Functions Palette > Programming > Boolean > Bool to (0,1)) inside the for loop and connect the output of not equal function to its input. This function converts the Boolean True to 1 and False to 0.

\* Place an Add Array Element (Functions Palette > Programming > Numeric > Add Array Element) function outside for loop but inside the case structure box. This function sums all calculated Hamming distances.

#### **e) Output - Bit Error Rate (BER):**

\* Place two To Double Precision Float functions (Functions Palette > Programming > Numeric > Conversion > To Double Precision Float) one for Add Array Element *sum* output and the other for Min & Max *max(x,y)* output. This function converts the number into double-precision, floating point number. We need this conversion to observe the BER with great precision. DBL allows us to see the exact value without rounding or cutting some digits.

\* Place a Divide function (Functions Palette > Programming > Numeric > Divide).

\* Connect DBL output from *sum* to  $x$  input and DBL output from *max(x,y)* to  $y$  input of Divide Function and  $x/y$  output of Divide to bit-error rate.

#### **f) Necessary connections**

In this section, necessary error in – error out connections are made in order to finalize VI design.

\* Connect the output of the *error in (no error)* to *error out* through the case structure for both cases (True/No Error and False/Error).

At the end, your block diagram should look like the one below in Fig. 7:

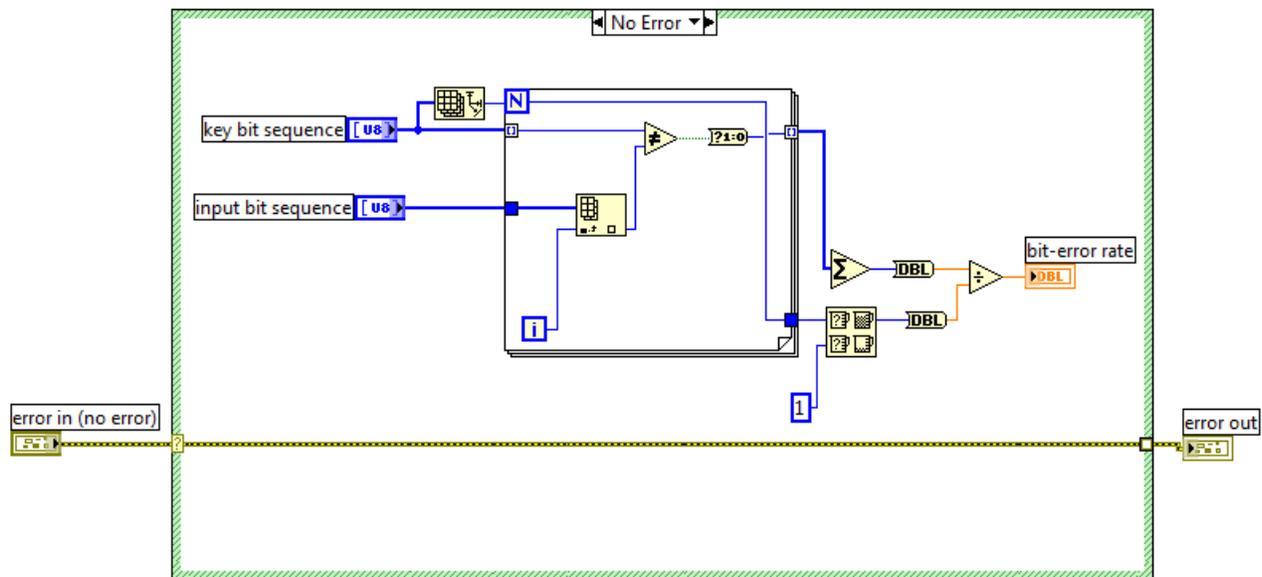


Fig. 7 – Complete block diagram.

For visual reference, check this video out: [See the video: How to build the error\\_detect.vi](#)

\* In order to check if your file has errors click on Operate > Run (OR Ctrl+R). If no error window is generated, then your file is good.

\* SAVE it AS a new file by adding your name at the end (example: if your name is John, then the new file name is “student\_error\_detect\_John.vi”).

### **PART-3: SIMULATION**

Simulation is the technique of representing the real world by a computer program. It is widely used for testing designs before implementing them, in order to fix design flaws, if any, and see the results to expect from the implemented system.

In this part of the laboratory, we substitute the generic source generation and error detection blocks supplied with the simulator with the modules developed previously according to Parts 1 and 2. This enables to test the system operation before implementation on the USRP board.

In order to simulate your source and error detection designs, please follow these steps:

\* Open the LabVIEW program (Start > All Programs > National Instruments > LabVIEW 2012 > LabVIEW 2012).

## a) Simulator setup

- \* Open “awgn\_simple\_sim.vit” file (Open Existing (OR Ctrl+O) > Select “awgn\_simple\_sim.vit” > OK). A basic graphic user interface (GUI) opens. (NOTE: If a warning window pops up, click on “Ignore” option.)
- \* Open Block Diagram (Window > Show Block Diagram (OR Ctrl+E) ).
- \* Right click on the “source” box > Replace > All Palettes > Select a VI...>insert your “student\_source\_<name>.vi” file.
- \* Right click on the “error detect” box > Replace > All Palettes > Select a VI...>insert your “student\_error\_detect\_<name>.vi” file.
- \* In order to check if your file has errors click on Operate > Run (OR Ctrl+R). If no error window is generated, then your file is good.
- \* Go to “avgn\_simple\_sim.vi” front panel and click on “Run Continuously” button (two looping arrows).

For visual reference, check this video out: [See the video: How to build simulate](#)

## b) Data collection

Bring up the simulator front panel and perform the following measurement shown on Fig. 8. Detailed information on front panel objects can be found in Appendix-A of this document.

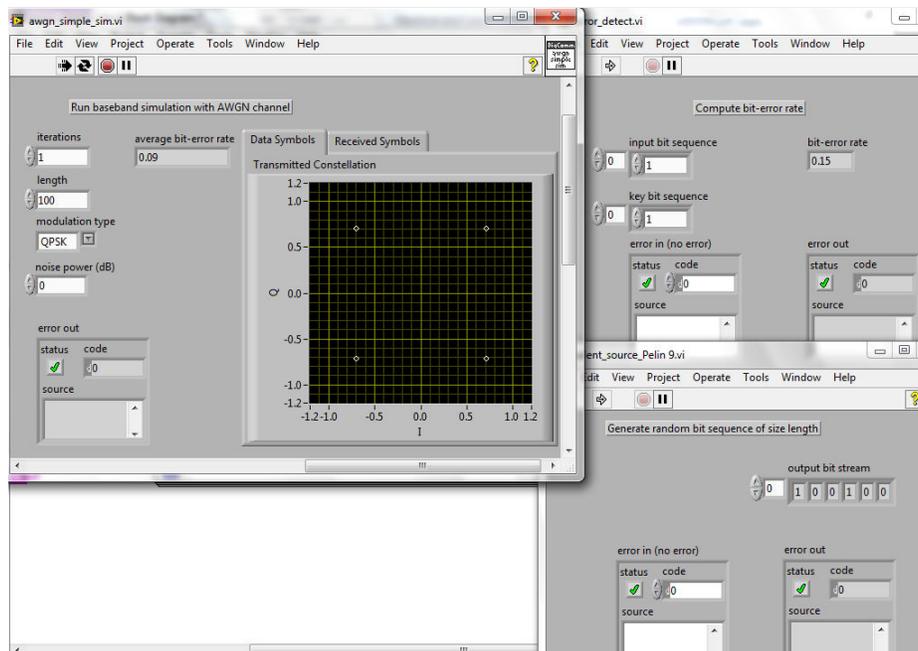


Fig. 8 – Simulation windows

Repeat the simulation based on following instructions: Plot 4 different curves for Average BER vs  $\text{SNR} = 1/N_0$  (in dB) where  $N_0$  is noise power and given in the simulator as dBW. Plot for only BPSK for  $N_0$  from -10 dB to 10 dB in 2 dB steps (**Hint:**  $\text{SNR (linear)} = (1/N_0)$  while  $\text{SNR (dB)} = [10 \cdot \log_{10}(1 - N_0 \text{ (dB)})]$ ). Curve-1 and 2: number of iterations=10. Curve-3 and 4: number of iterations= $10^4$ . These plots should be used for report preparation.

#### **PART-4: HOMEWORK QUESTIONS**

**#1:** Write a lab report describing the experiments and results you observed. Include in the report the images you collected in Part-3.

**#2:** Answer the following questions:

- Explain why the initial seed value cannot be all zeros.
- State which parameters affect the bit-error rate (BER).
- What does BER=1 mean? How can you convert a system with BER=1 into a system with BER=0? What is the worst-case value for BER?

## **APPENDIX A**

#### **PART-1: SIMULATION**

On “awgn\_simple\_sim.vi” simulation window you can manipulate the following values:

- 1) Iterations:** Data type is long [32-bit integer (-2147483648 to 2147483647)]. This controls how many times the simulation is performed. Note that for each simulation run, the BER is calculated as explained above. The final BER shown is the average of the BER calculated in each simulation run. For example, if Iterations is set to 100, the simulation is run 100 times (each time run with the number of bits indicated by the length field), and the corresponding BER is calculated 100 times. The average of the BER is displayed as the final BER result. Clearly, higher iteration numbers increase the statistical significance of the BER results, but decrease the processing speed. For noise power higher than -10 dB one does not need a high number of iterations since the BER is large enough and the required precision is not large, but for noise power lower than -10 dB one needs to set the iteration number high enough ( $>10^3$ ) in order to properly calculate the BER, which is now a small number (See the related homework question#2 in Part-5).

- 2) **Length:** Data type is long [32-bit integer (-2147483648 to 2147483647)]. This changes the length of the key bit sequence and input bit sequence used in each iteration.
- 3) **Modulation type:** Data type is string. There are two modulation options to choose from: (i) Binary Phase Shift Keying (BPSK) and (ii) Quadrature Phase Shift Keying (QPSK). We discuss the properties of these modulations in Lab 3. The visible difference on the simulation window between two modulation schemes is the number of constellation points seen on I-Q plot (2 constellation points for BPSK and 4 constellation points for QPSK).
- 4) **Noise power:** Data type is double [64-bit real ( $\approx 15$  digit precision)]. Here you can define the noise power in Decibel Watts (dBW) ( $\text{dBW} = 10 \cdot \log_{10}(W)$ ).

The data symbols I-Q graph shows I (in-phase or real) and Q (quadrature or imaginary) values of the constellation points, while received symbols I-Q graph shows continuously changing I-Q values of the constellation points as symbols are received. I-Q constellation maps are discussed in Lab 3.

In order to export the values of a graph or array on the simulation window, follow this path and pick the output format you need: Right click on the graph/array > Export > Export (clipboard/excel/simplified image).

In order to copy the current value of an indicator, right click on the indicator while simulation is running and select "Copy Data". This copies the current value as an image.

Also, if you need to run the simulator only for one loop, you can click on the single arrow ("Run").