LAB Assignment #5 for ECE 525

Assigned: Thur., Mar. 2, 2017
Due: Tue., March. 7, 2017

Description: Process HELP Timing Data into PND and PND$_c$

1) This lab requires that you modify the code that I provided to you for enrollment. You should keep a clean copy of the original code (make a subdirectory and call it enrollment). You will always use the original code for enrollment, which measures the path delays (PNs) and then transfers them to your laptop (lab4).

2) For regeneration, you do NOT transfer the PNs off of the token, so you need to eliminate the socket call that transfers the timing values in *main*. Instead, you need to store them in two local arrays. Right now, the PNs are being stored in an array for each vector, which is dimensioned as 16 x 16 (2-D array of PNs and samples). You can continue to use this integer array as is.

3) As I mentioned in class, you need to compute the average value of each PN as a floating point number by summing the rows of the original array (16 x 16) and dividing by 16. Save the averaged PNs in two new arrays that are dimensioned according to the number of rising and falling PNs that you collected. You can determine this by vi'ing the enrollment data file that was transferred to your laptop and looking for a line in the middle of the file of the form "V: 1449 O: 15 C: 10002'. This indicates that the number rising PNs collected by the challenge vectors is 10,003 (C: is a zero based counter). The last line in the file is of the form 'V: 2899 O: 7 C: 20009' so the number of falling PNs is 20009 - 10002 = 10007. Create two 1-D arrays of size 10,000 and store the first 10,000 averaged rising PNs and first 10,000 averaged falling PNs in these arrays.

4) Add the following code to your new C code. It defines two 11-bit LFSRs that count pseudorandomly from 0 to 2047. You will use these below to compute PND by subtracting a falling PN from a rising PN. Only the first 2048 elements from your arrays of 10,000 will be used in the bitstring generation process.

```
//==================================================================================================
//==================================================================================================

uint16_t LFSR_11_A_bits_low(int load_seed, uint16_t seed)
   {
   static uint16_t lfsr;
   uint16_t bit, nor_bit;

/* Load the seed on the first iteration */
   if ( load_seed == 1 )
      lfsr = seed;
   else
      {

/* Allow all zero state. */
      if ( !( (((lfsr >> 9) & 1) == 1) || (((lfsr >> 8) & 1) == 1) ||
              (((lfsr >> 7) & 1) == 1) || (((lfsr >> 6) & 1) == 1) || (((lfsr >> 5) & 1) == 1) ||
              (((lfsr >> 4) & 1) == 1) || (((lfsr >> 3) & 1) == 1) || (((lfsr >> 2) & 1) == 1) ||
              (((lfsr >> 1) & 1) == 1) || (((lfsr >> 0) & 1) == 1) ) )
         nor_bit = 1;
      else
         nor_bit = 0;
      bit  = ((lfsr >> 10) & 1) ^ ((lfsr >> 8) & 1) ^ nor_bit;

/* Change the shift of the bit to match the width of the data type. */
      lfsr =  ((lfsr << 1) | bit) & 2047;
      }

   return lfsr;
   }

//===================================================================================================
//===================================================================================================

uint16_t LFSR_11_A_bits_high(int load_seed, uint16_t seed)
   {
   static uint16_t lfsr;
   uint16_t bit, nor_bit;

/* Load the seed on the first iteration */
   if ( load_seed == 1 )
      lfsr = seed;
   else
      {

/* Allow all zero state. */
      if ( !( (((lfsr >> 9) & 1) == 1) || (((lfsr >> 8) & 1) == 1) ||
              (((lfsr >> 7) & 1) == 1) || (((lfsr >> 6) & 1) == 1) || (((lfsr >> 5) & 1) == 1) ||
              (((lfsr >> 4) & 1) == 1) || (((lfsr >> 3) & 1) == 1) || (((lfsr >> 2) & 1) == 1) ||
              (((lfsr >> 1) & 1) == 1) || (((lfsr >> 0) & 1) == 1) ) )
         nor_bit = 1;
      else
         nor_bit = 0;

      bit  = ((lfsr >> 10) & 1) ^ ((lfsr >> 8) & 1) ^ nor_bit;

      lfsr =  ((lfsr << 1) | bit) & 2047;
      }

   return lfsr;
   }
```

5) Write a routine that computes the PND as follows. Functional definition should look similar to this:

```
float ComputePNDiffsTwoSeeds(int max_PNDiffs, float PNR[max_PNDiffs],
float PNF[max_PNDiffs], float PND[max_PNDiffs], int LFSR_seed_low, int
LFSR_seed_high)
```

'max_PNDiffs' should be set to 2048 when you call this routine from main. The 'PNR' and PNF' arrays are the array names that store the 10,000 rising and falling average PNs, resp. 'PND' is also a new array that you create in main (in addition to PNR and PNF), dimensioned as a 1-D array of 2048 floating point elements. 'LFSR_seed_low' and 'LFSR_seed_high' can be set to any value between 0 and 2047. For the trials, use 0 and 0 for the two seeds when you call this routine from main.

6) You need to create a 'for' loop that computes the PND from the PNR and PNF. Add calls to the LFSRs as follows on the FIRST ITERATION ONLY:

```
lfsr_val_low = LFSR_11_A_bits_low(1, (uint16_t)LFSR_seed_low);
lfsr_val_high = LFSR_11_A_bits_high(1, (uint16_t)LFSR_seed_high);
```

'lfsr_val_low' and 'lfsr_val_high' can be used as indexes into PNR and PNF to select the two values used in the difference and stored in the PND array **at the index given by 'lfsr_val_low'**.

7) On subsequent iterations, use the following calls to obtain the next set of ‘lfsr_val_low’ and ‘lfsr_val_high’ indexes for the remaining 2047 PND:

```
    lfsr_val_low = LFSR_11_A_bits_low(0, (uint16_t)0);
    lfsr_val_high = LFSR_11_A_bits_high(0, (uint16_t)0);
```

8) We also discussed the TVCOMP process in class. You can use the slide set ‘HELP_details’ to determine how to compute the PNDc. This is a 4th new floating point array dimensioned as the PND array. Please be sure to get steps 1-7 done by Tue next week. If you can do the TVCOMP process too, then you’ll get extra points. Otherwise, we’ll make it due for the next lab.