LAB Assignment #3 for ECE 525

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

The HELP algorithm for regeneration consists of a sequence of modules called *PNDiffs*, *TVComp*, *Modulus* and then *BitGen* (see PUFs II(F) through II(J)). In this lab, you will write C code to implement the first two of these modules. The remaining modules will be implemented in subsequent labs. The goal is to build a *codesign* version of HELP regeneration with a portion of it running in the PL-side (hardware), namely the *LaunchCapture* timing engine component, and the remaining portion running on the PS-side (software). The enrollment version that you ran in lab #2 implements the hardware component. You will reuse the PL-side for regeneration and will modify the enrollment C code to implement the regeneration version.
The enrollment version accepts challenge vectors from the server, transfers them to the PL-side through a GPIO register, starts the *LaunchCapture* timing engine, retrieves the digitized timing values from a second GPIO register and transmits the timing values back to the server over the network. You should preserve this operation in the regeneration version except for the last step, i.e., do not transmit the timing data back to the server. Instead, store the timing values into two arrays, PNR and PNF, and process the 2048 PNR and PNF through the HELP modules as described below and in the following labs.

0) I have provided you a starter program token_regeneration.c that you should modify as described below. You also need to run a new version of the verifier that is stored in the PROTOCOL directory. You can compile it on your laptop or UNM server by running ./compile.csh. You run the verifier_regeneration on your laptop or UNM server BEFORE running token_regeneration.elf on the Zybo board (as was true for lab2). See the README.txt file for the command line parameters.

1) The enrollment version stores the PN in an array as each vector is transferred through the GPIO register. The array is dimensioned as a 64 x 16 (2-D array of PN and samples).

Table 1: 2-D array of PN collected per vector

Output	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	Ave
0	307	307	308	308	308	308	308	308	308	308	308	308	308	307	309	308	307.8750
1																	
...																	
63																	

This 2-D array shows the functional unit outputs as a set of 64 rows and the samples in 16 columns. Note that challenge vectors rarely generate timing values for all 64 outputs. Usually, only a subset of this array is filled in with actual timing data, with as few as only 1 row of data for some challenge vectors as shown above. The values read from the hardware through the GPIO register are always integers as shown. You need to first compute the average value for each row as shown by the column on the far right. Only the average value for each row is to be stored in the 1-D PNR and PNF arrays. Round to 4 binary digits of precision when computing the floating point average by using the following formula, where *ave* is the floating point average:

```
    PNR[i] = (float)((int)(ave*16.0))/16.0
```

When printed as base 10 values, you should only see the following fractions, .0000, .0625, .1250, ..., .9375 in the PNR and PNF array.

NOTE: All of the floating point arrays that you will create and store data into, including PND, PNDc, PNDco, modPNDco, should round to 4 binary digits as you do here. This is to ensure that the values you compute are consistent with the server version (I will provide the server version of regeneration in a later lab).

NOTE: You will need to determine if the challenge vector that is being applied is a *rising* or *falling* vector so you know which array, PNR or PNF, to store the average values into. This information is given as the *num_rise_vecs* in the *ReceiveVectors* routine in the enrollment version (which you will reuse in your regeneration version).

2) Once all 2048 PNR and 2048 PNF are collected into the arrays, then you need to implement the *PNDiff* module to compute the differences. Store the differences in a 1-D PND floating point array of 2048 elements as described below.

3) Add the following code to your new token_regeneration.c file. It defines two 11-bit LFSRs that count pseudo-randomly from 0 to 2047. You will use these below to compute PND by subtracting a falling PN from a rising PN.

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

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;
   }
```

4) Write a routine that computes the PND as follows. The function definition should look similar to this:

```
float ComputePNDiffs(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 2048 rising and falling average PNs, resp. ‘PND’ is 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’ are parameters that 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.

5) 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. Always store the difference value in the PND array **at the index given by ‘lfsr_val_low’**.

6) 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);
```

7) The TVCOMP module used within HELP is described in the screencasts. Use the following pseudo-code to develop your C version of this module. *ComputeBoundedRange* creates a histogram distribution of the PND:

```
ComputeBoundedRange(int max_PNs, int num_PNs, float PN_vals[max_PNs], float rangel_low_limit,
   float range_high_limit, int DIST_range)
   {
   Find the smallest value in the array of 2048 PND and store it in smallest_val (note, PND is usually
      a negative number)
   Create and zero out an integer array called PN_bins that is dimensioned of size DIST_range
   For each PND,
      Subtract the smallest_val from the PND
      Round up by adding 0.5
      Truncate to an integer using (int)
      Use this value as an index into the PN_bins array
      Add one to the indexed cell in PN_bins
   Set sum = 0.0, low_done = 0, low_index = 0 and high_index = 0
   For each bin i in PN_bins
      Add PN_bins[i] to sum
      Check if low_done is 0 and sum >= range_low_limit
      If true
         Set low_done to 1
         Save the current bin number into low_index
      Check if sum <= range_high_limit
      If true
         Save the current bin number into high_index
   Return range as high_index - low_index
   }
```

Use the following pseudo-code to develop your C version of this module. *TVCOMP* computes the mean and range of the original PND distribution, applies two transforms and stores the results in a 1-D floating point array PNDc. Note that the constants *range_low_limit*, *range_high_limit* and *DIST_range* are already defined in common.h.

```
void TVComp(int max_vals, int num_vals, float PND[max_vals], float range_low_limit,
   float range_high_limit, int DIST_range, float reference_mean, float reference_range,
   float PNDc[max_vals])
{
Compute the mean of the PND array values and store in orig_mean
Compute the range of the PND (call ComputeBoundedRange) and store in orig_range
For each PND
   Standardize the PND, i.e., subtract orig_mean and divide the difference by orig_range
   Compute the PNDc by applying 2nd transform, i.e., multiply standardized PND by reference_range
      and add in the reference_mean
}
```

8) Create a lab report that includes a description of this lab. Include a copy of your *Compute-BoundedRange* and *TVComp* C code. Run your code on the Zybo board and generate a histogram graph using your favorite software (matlab) of the PND and PNDc arrays. Use 0 for the *reference_mean* and 180 for the *reference_range*. Report the original mean and range of the PND. As noted above, use 0 for both the *LFSR_seed_low* and *LFSR_seed_high* parameters to the *ComputePNDiffs* module.