LAB Assignment #2 for ECE 525

Assigned: Mon., Feb. 24, 2016
Due: Wed., Mar. 2, 2016

Description: Analyze the timing values produced by the HELP

1) Program the Zedboard with your bitstream (or the one that I have provided on my website)

2) Transfer the Linux_GPIO_version.elf to the Zedboard using scp (see '*Zedboard instructions for creating a project, programming the board and configuring linux + network*' link on the course webpage. The elf file is located in the project directory at:
Vivado/WDDL_AES_SBOX_NR/AES_SBOX_WDDL/AES_SBOX_WDDL.sdk/Linux_GPIO_version/Debug

2) Transfer the vector file from the TestVectors directory from the tar.gz file that I supplied to you:
AES_SBOX_456_WDDL_ATPG_risingfalling_HW_vecfile.txt

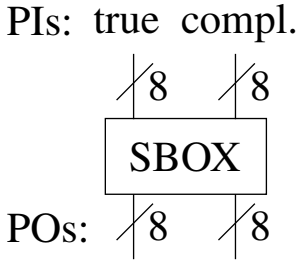
3) Run the program from the / directory as follows ('mv' the two files above from 'root' to '/' if needed):
./Linux_GPIO_version.elf Cx 0 0 AES_SBOX_456_WDDL_ATPG_risingfalling_HW_vecfile 0

Substitute the 'x' in Cx with the number from your board.

4) The PUF timing values are stored in:
/tmp/Cx_25C_1.00V_E_PUFNums.txt

5) Transfer the timing values produced by the Zedboard to your host computer using scp

6) The functional unit is a WDDL version (glitch-free) of the AES SBOX with the following inputs and outputs:



7) The format of the file is as follows:

```
V: x  O:y  C:z  T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12 T13 T14 T15 T16 0
```

'V' is vector number (0 to 11,595)
'O' is output number (0 to 15)
'C' is a counter (0 to 116,287)
'Tx' are 16 samples of the timing values associated with the path.

8) Write a perl program that reads the data into an array and carries out the following process:

a) Compute the average values of the 16 Tx values on each line
b) Compute randomized differences:
In part A, we will work with the first 4,096 timing values (numbered 0 to 4,095) of the 116,288 total number of values.

Divide the first 4,096 average values into two groups. For each sequential value numbered 0 to 2,047 in the first group, pseudo-randomly select a value in the second group (from 2,048 to 4,095) using the following LFSR code. Note: this code returns a value between 0 and 2,047 that must be offset by adding the constant 2,048 to the returned value. Use a 'seed' of 1.

```
sub LFSR_11_A_bits
   {
   my ($load_seed, $seed, $lfsr) = @_;

   my ($bit, $nor_bit);

# Load the seed on the first iteration
   if ( $load_seed == 1 )
      { $lfsr = $seed; }
   else
      {

# Allow all zero state. See my BIST class notes in VLSI Testing. Note, we use low order bits
# here because bit is shifted onto the low side, not high side as in my lecture slides.
      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; }

# xor_out := rand(10) xor rand(8);
      $bit  = ((($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;

#printf "LFSR value %dNOR bit value %dlow order bit %d\n", $lfsr, $nor_bit, $bit;
      }

   return $lfsr;
   }
```

In your loop that creates the differences, the first call to the LFSR should be as follows:

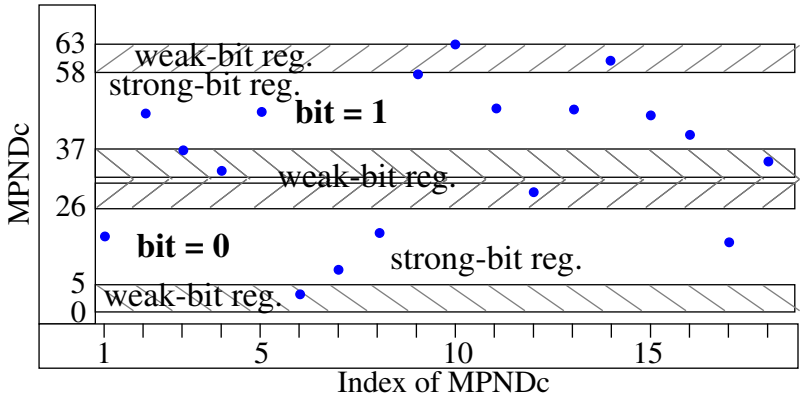
```
$lfsr = 0
($lfsr) = LFSR_11_A_bits(1, $seed, $lfsr);
```

All of the 2,047 subsequent calls should be made as follows:

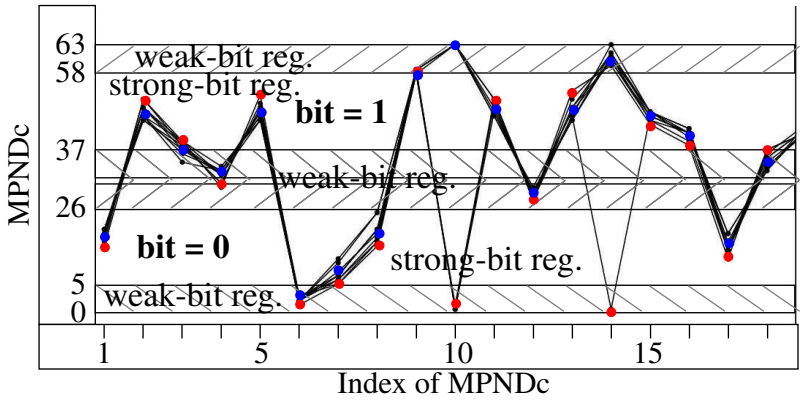
```
($lfsr) = LFSR_11_A_bits(0, $seed, $lfsr);
```

c) For each of the 2,048 difference values, compute the a modulus of the values using a modulus of **64**.
d) Compute a bitstring using the following procedure. The difference values after computing the modulus can be represented as a graph shown below. Here, MPNDc stands for *modulus-PUFNum-difference-compensated*, where refers to the values you computed after completing step *c)* above (you have not *compensated* the values yet -- we'll do that later). The x-axis plots the index of consecutative values in your array while the y-axis represents the computed values). In order to avoid bit flip errors, you need to apply a margin technique that excludes specific MPNDc's from participating in the bitstring generation process. The margin in the

following illustration is set to 6. The margin creates *weak-bit regions* around the bit-flip lines 0, 31 and 63. MPNDc's that fall within these *weak-bit regions* have a higher probability of introducing a bit flip error during regeneration than those in the *strong-bit regions*.



For example, if we were to re-collect the MPNDc's under other environmental conditions, i.e., with different supply voltage values and/or temperature conditions, the values will *shift* as shown by the following set of curves. The red dots represent a 'worst-case' shift that occurs to each MPNDc. The vertical shift in MPNDc's that are close to the bit-flip lines can result in the MPNDc value being re-classified from a '1' to a '0' or vise versa (identify where this occurs in the graph).



Therefore, MPNDc's that fall within the *weak-bit regions* are to be skipped when generating the bitstring. For example, the *strong* bitstring generated by processing the MPNDc's shown the graph is as follows:

0	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**

So only 9 bits of the possible 18 are actually used. A **helper data bitstring** is also generated that identifies which bits are classified as *strong-bits* during the enrollment process (blue points).

1	1	0	0	1	0	1	1	1	0	1	0	1	0	1	1	1	0
1				5					10					15			

The helper data bitstring is used during regeneration to 'select' the MPNDc values to be used for generating the bitstring, i.e., the margins are NOT used during regeneration. The helper data bitstring does NOT reveal any information about the bitstring itself, which is the secret generated by the PUF.

Apply this technique to the subset of 2,048 MPNDc values that you obtain from step *c)* above.