HELP Overview

The HELP PUF produces a bitstring using a challenge-response mechanism. The challenge component for HELP consists of a randomly selected, two-vector test sequence applied to the inputs of the macro-under-test (MUT). The test sequence introduces a set of transitions that propagate through the core logic of the MUT and appear on its outputs. The responses are defined as the measured path delays, represented as 8-bit numbers as explained below, for each of the outputs. The delays on each MUT output are measured one-at-a-time.

The precision of the delay measurement impacts the stability of HELP. We use an embedded test structure called REBEL to obtain high-precision, digitized representations of the path delays. REBEL is integrated into the scan chain logic and uses the on-chip clock tree network for launch-capture (LC) timing events.

Fig. 1 depicts a overview of the REBEL test structure, which consists of two rows of flip-flops (FFs) connected together into a scan chain. Small logic blocks on the left of each row, labeled RCL for Row Control Logic, allow the scan elements on each row to be configured as follows:

- The top row is the launch row, and is configured to operate in functional mode.
- The second row is the capture row, and is configured in 'mixed mode', in which a specific FF, called the **insertion point (IP)**, is chosen. This scan-FF and each scan-FF to the right of it in the row are placed in 'flush delay' mode (**FD mode)** (described below), and form a combinational delay chain, effectively extending the path at the IP.

Flush-delay mode (FD) is a special mode in which a scan chain can be configured as a combinational delay chain. This is depicted in the callout in Fig. 1, which shows two master/slave FFs in which the output of the first master feeds into the scan input of the second FF. Any transition that occurs on the IP propagates through the functional input and into the first master using logic that selects that path (not shown). In contrast, the logic controlling the scan MUX for the second FF (and all FFs to its right) selects the scan input, effectively allowing the transition to propagate unimpeded through the masters of these FFs..

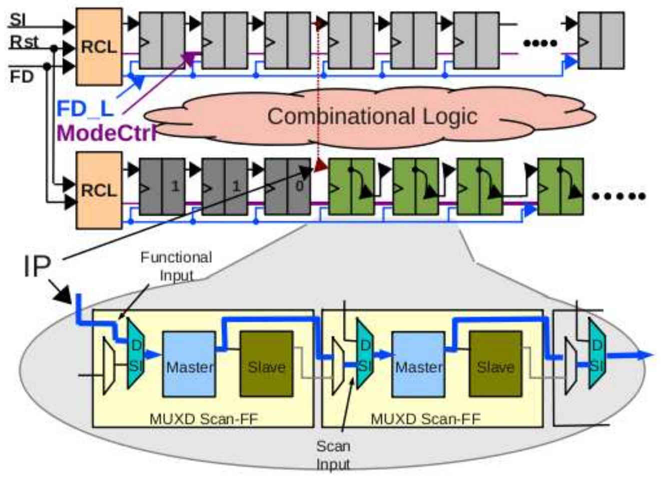


Fig. 1. The REBEL embedded test structure.

A REBEL path delay test is carried out by scanning in configuration information, which selects the IP and configures the delay chain as shown in Fig. 1. A clock transition is then applied to the launch row FFs which generates transitions that propagate into the MUT. Any transition that occurs on the MUT output at the IP will propagate into the delay chain. By asserting the clock input on the capture row FFs, the master latches revert to storage mode and digitize the time behavior of the transition(s) as a sequence of 1's and 0's. The combined delay of the MUT path and the delay chain can be derived by searching, from right to left, in the binary sequence for the FF that contains the first '0'-to-'1' or '1'-to-'0' transition.

0.1 FPGA Implementation

Fig. 2 shows a top-level structural diagram of the HELP implementation. The MUT used in this implementation is the logic defining a single round of a pipelined AES implementation from OpenCores. The block labeled 'Initial Launch Vector (256)' represents the pipeline FFs in the full-blown AES implementation, converted here to MUX-D scan-FFs. A second copy of this block, labeled 'Final Launch Vector (256)', allows two randomly generated vectors that represent the challenge to be scan-loaded into these two rows of scan-FFs.

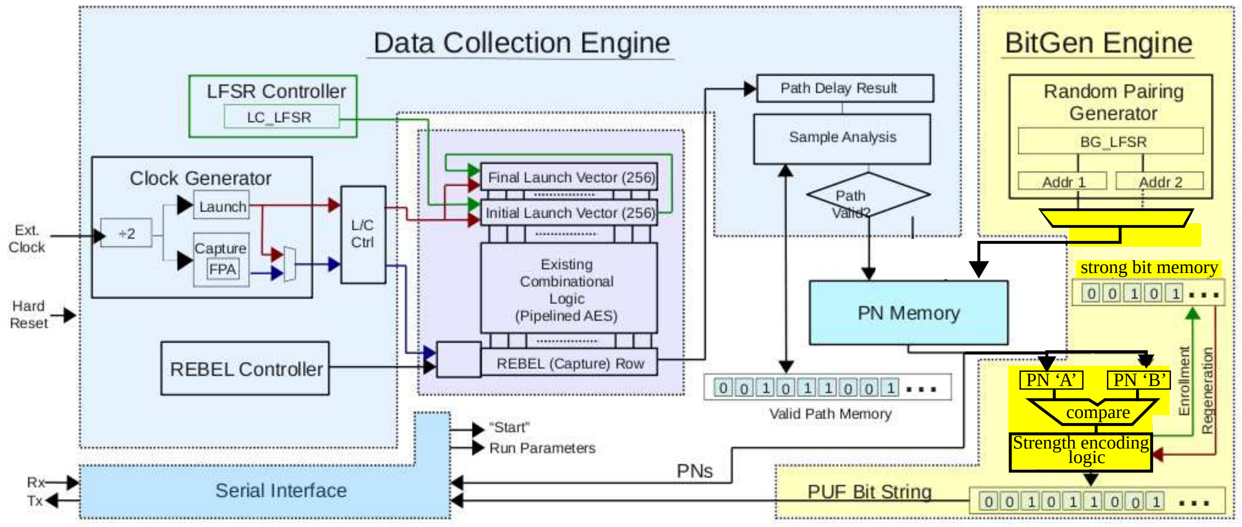


Fig. 2. Top-Level HELP system diagram.

The block labeled 'REBEL (Capture) Row' in Fig. 2 incorporates REBEL, and designed it to implement the delay chain. The number of FFs in this row is expanded from 256 to 264 to extend the delay chain for the IPs on the right end of the MUT.

The remaining components in Fig. 2 define the HELP PUF engine, and can be divided into the **Data Collection Engine** (DCE) and the **BitGen Engine** (BGE). One iteration of the whole process produces the bitstring. The engine behaves differently depending on whether a new bitstring is requested (a process called **enrollment**) or whether the bitstring needs to be reproduced (a process called **regeneration**). We distinguish between these scenarios in the following description as needed.

0.2 HELP Components

The DCE in Fig. 2 carries out a sequence of LC tests, measures the path delays, and records the digitized representation of them, called PUF numbers or PNs, in block RAM on the FPGA. In our current implementation, the DCE runs to completion before the BGE component is started.

Clock Generator: The clock generator module generates two clock signals: a Launch clock and a Capture clock, and is shown on the left side of Fig. 2. In our design, this module contains three digital clock managers, or DCMs. A 'master' DCM is used to reduce the off-chip oscillator-generated 100 MHz clock to 50 MHz. The output of the master DCM drives the Launch and Capture DCMs. We utilize the fine phase adjustment (FPA) feature of the Capture DCM to 'tune' the phase relationship between the Launch and Capture clocks. At 50 MHz, the FPA allows 80 ps increments/decrements in the phase shift of the Capture clock on the Virtex-II Pro chips.

When the DCE is configuring the scan chains in preparation for the LC test, the phase relationship between the Launch and Capture clocks is set to 0. Just prior to the launch event, the controlling state machine selects the 180° phase-shifted output of the Capture DCM, and the FPA feature is used to tune the phase in an iterative process designed to meet a specific goal (to be discussed).

Table 1: Capture clock phase adjustment

Phase Adj.	Phase Shift	LC Interval
0	90^o	5 ns
64	180^o	10 ns
128	270^o	15 ns

Table 1 summarizes the characteristics of the Capture clock, and Fig. 3 illustrates the timing relationship between the Launch and Capture clocks for different values of the 'Phase Adj.' control counter in the DCM. The launch and capture events occur on the rising edge of the corresponding clocks. From the timing diagram, this allows path delays from 5 ns to 15 ns in length to be measured. The 0 to 128 range of values (called PNs) are used as a digital representation of the path delays.

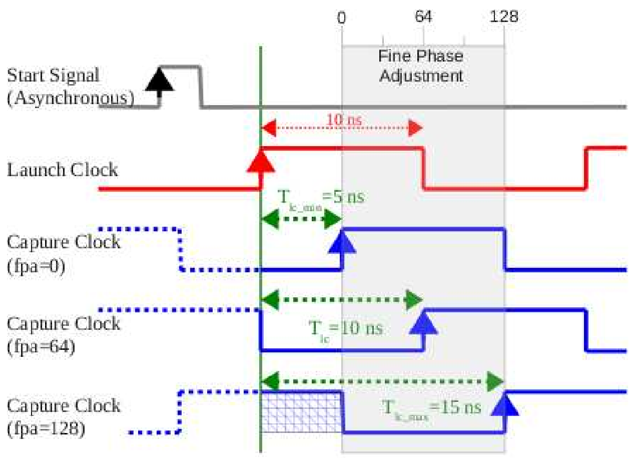


Fig. 3. Launch/Capture timing diagram.

PN Memory: A block RAM used to store the PNs.

LC LFSR Controller: A 32-bit linear feedback shift register (LFSR) used to produce the randomized launch vectors.

REBEL Controller: Configures the IP in the REBEL row attached to the output of the AES logic block.

Sample Analysis Engine (SAE): Analyzes the digitized results in the delay chain after each LC test for a given path and determines whether the path is ’valid’. ***A valid path is defined as one that has a real transition, is glitch-free, and produces consistent results across multiple samples.***

Valid Path Memory: A block RAM used to record a pass/fail flag for each tested path that reflects its validity (as defined under SAE). These values are technically stored during enrollment and then read back in from non-volatile or off-chip memory (public storage) during regeneration, and represents the *helper data* needed in the regeneration process.

Random Pairing Generator: Uses a 28-bit LFSR to generate randomized pairings of PNs for bit generation.

Stop Point Memory/Strong Bit Memory: A block RAM used by the Bit Generation Engine to record ‘stop points’ or ‘strong bits’ (depending on the bit generation method in use) during enrollment. The values stored in this memory, like the Valid Path Memory, are also components of the helper data.

The **Serial Interface** component is used to interact with the HELP engine, and to transfer the results of the path testing and bit generation processes.

0.3 Path Delay Measurement

A sequence of paths are tested by the DCE process to produce the PNs used later in bit generation. The starting point and order in which the paths are tested is determined by the LC LFSR. The DCE process begins by loading the LC LFSR with a seed (provided by the user), and instructs the LC LFSR controller to load a random pair of vectors into the launch rows. Simultaneously, the REBEL controller configures the REBEL row with a specific IP and places the REBEL row in FD mode. The same random vector pair is reloaded to test each of the 256 IPs, one at a time, before the LC LFSR generates and loads the next random vector pair.

A key contribution of our technique is the discovery that **path stability** can be used as the basis for random bitstring generation. Path stability is defined as those paths which have a rising or falling transition, do not have temporary transitions or glitches, and that produce a small range of PNs (ideally only one) over multiple repeated sampling. We found that the paths that pass the stability test are different for each chip in the population.

A state machine within the DCE is responsible for measuring path delays and for determining the stability of the paths. Our algorithm begins testing a path by setting the FPA to 128, which configures the Capture clock phase to 270˚. It then iteratively reduces the phase shift in a series of LC tests, called a sweep. For paths that have transitions, the process of ‘tuning’ the FPA toward smaller values over the sweep effectively ’pushes’ the transition backwards in the delay chain, since each successive iteration reduces the amount of time available for the transition to propagate. When the edge is ’pushed back’ to a point just before a target FF in the delay chain, the process stops (the goal has been achieved). The target FF is an element in the delay chain that is a specific distance (in FFs) from the IP. The value of the FPA at the stop point is saved as the PN for this path. Note that the PN represents the ’response’ to the ’challenge’, where the challenge is defined by the launch vector and IP.

Evaluating path stability is accomplished by counting the number of transitions that occurred in the REBEL row by ’XOR’ing’ neighboring FFs in the delay chain. The path is immediately classified as unstable (and the sweep is halted) if the number of transitions exceeds 1 at any point during the sweep. Once the sweep is complete, the whole process is repeated multiple times. If the range of PNs measured across multiple samples varies by more than a user-specified threshold, the path is classified as unstable and is discarded.

Note that path stability evaluation occurs ONLY during enrollment. In order to make it possible for regeneration to replay the valid path sequence discovered during enrollment, the 'valid path' bitstring is updated after testing each path. For paths considered valid, a '1' is stored and for those classified as unstable (or have no transition), a '0' is stored. During regeneration, the exact same sequence of tests can be carried out by loading the LC LFSR with the same seed and using the 'valid path' bitstring to determine which paths are to be tested (a '1' forces the path to be tested, and a '0' forces the path to be skipped).

0.4 Bit Generation Process and Procedure

The BitGen Engine for UNMD, shown on the right side of Fig. 2, randomly selects two PNs to compare. The Random Pairing Generator produces the two addresses of the PNs to compare and the values are read from on-chip memory into a pair of registers (PN 'A' and PN 'B'). PN 'B' is then subtracted from PN 'A' to produce a PN difference. The magnitude of the difference determines the strength of that pairing, as discussed in the next section. If that difference is sufficiently large, then the sign of the comparison determines the value of the generated bit. A negative sign produces a '0', and a positive sign produces a '1'. The Random Pairing Generator produces all combinations of addresses, e.g., if n PNs are collected and stored by the DC Engine, then $n(n-1)/2$ pairs of addresses are produced during the bit generation process.

0.4.1 Thresholding Technique

A thresholding technique is used to decide if a given comparison generates a strong bit (which is kept) or a weak bit (which is discarded). Thresholding works as follows. During enrollment, a noise threshold is defined using the path distribution histogram for the chip. The histogram is constructed using all n PNs collected by the DC engine. The noise threshold is then computed as a constant that is proportional to the difference between the PNs at the 5 and 95 percentiles of this distribution. Therefore, each chip uses a different threshold that is 'tuned' to chip's overall (chip-to-chip) delay variation profile.

For each comparison, the difference between the two PNs is compared against the noise threshold. A strong bit is generated if the magnitude of the difference exceeds the threshold, otherwise the bit is discarded. Simultaneously, a bit is added to the 'Strong Bit Memory' shown in Fig. 2 that reflects the status of the comparison, with '1' indicating a strong bit and '0' indicating a weak bit. During regeneration, the Strong Bit Memory is consulted to determine which comparisons are used to regenerate the bitstring.

This is illustrated for UNMD in the graphs of Fig. 4. The graphs plot the 'strong bit' number along the x-axis against the PN differences on the y-axis, with the noise thresholds (as described above) set to +/- 77.4 for this chip. The data points from enrollment on the left all fall above or below these thresholds (by definition), but data points from measurements taken at different TV corners in the graph on the right 'infringe' into the space between the thresholds. Most data points remain close to the thresholds, but some move significantly b/c of TV variation as highlighted by as much as 71 PNs.

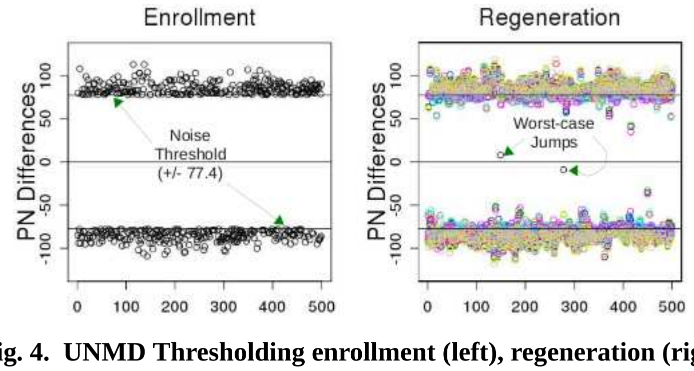


Fig. 4. UNMD Thresholding enrollment (left), regeneration (right).

PROJECT: Part 1:

Before implementing the process described above in VHDL and then testing your algorithm on a set of FPGAs, you need to implement an important component of the above process in software (C, perl, whatever language you prefer).

I have provided the digital snapshots of stable paths from 2 test chip in the data directory (I will update this eventually with data from all 52 chips at a later date). All data for each chip is contained in its own file. I have already selected paths that are stable from the original ‘raw data’ (which I have NOT provided). Bear in mind that you’ll need to implement a stability component of the algorithm (to be discussed in the near future).

The format of a typical file is as follows:

- There are 159 digital snapshots per path delay measurement, with a header before the sequence of each snapshot as follows:

  25C_1.20V   PathID 216   InsPt 0

  The first component of the header gives the temperature/voltage environmental conditions (TV corner) in place when the snapshots were collected. Other possibilities are ‘25C_1.08V’, ‘25C_1.32V’, ‘-40C_1.20V’, ‘-40C_1.08V’, ‘-40C_1.32V’, ‘85C_1.20V’, ‘85C_1.08V’ and ‘85C_1.32V’. **ENROLLMENT data** is the data at ‘25C_1.20V’, the remaining TV corners are regenerations.

  The PathID is a unique identifier for the path that is tested.

  The InsPt is the insertion point of that path

  Data is grouped such that the digital snapshots for all TV corners for each path are consecutive in the file. So you will find a header + 159 digital snapshots repeated 9 times for each path in a sequence of lines.

- Example digital snapshots from CHIP1 for the header give above are as follows:

```
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111000000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
11111100000000000000000000000000000000000000000000000000000000000
```

```
111111100000000000000000000000000000000000000000000000000000000
```
... continues for 159 total lines.

This example data is from a path with the InsPt set to 0. In other words, all ‘0’s and ‘1’s in these snapshots are from the delay chain. Other paths will have the InsPt set to something > 0, e.g.,

```
25C_1.32V       PathID 21586    InsPt 6
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111100000000000000000000000000000000000000000000
010100111111110000000000000000000000000000000000000000000
010100111111110000000000000000000000000000000000000000000
```

In this case, the delay chain data for this path does NOT start until position 6, counting from the left with the leftmost bit indexed at 0. So the binary data string ‘010100’ is NOT part of the path timing data is MUST BE IGNORED.

You will include a parameter called ‘target FF’ in your C/perl/etc implementation. The target FF is a number between 0 and 9 that represents a relative distance in FFs from the insertion point. It is the bit position in the digital snapshots that you will look at to find the transition. For example, if you specify the target FF as 8 using the path above with PathID 21586 with InsPt 6, then your algorthm should find the 12th snapshot (first snapshot is numbered 0) because the 13th snapshot shows the propagating ‘1’ has moved into the target FF -- you want to find the snapshot immediately preceeding the snapshot where the target FF receives the transition value.

IMPORTANT: You should search the snapshots IN REVERSE, starting with the 158th snapshot and working toward the 0th snapshot because of glitching that can occur (see below). Also, there are several ‘anomalies’ that you’ll need to deal with as you parse the snapshots. Some snapshots will be given as -1. This indicates that there is NO transition in the delay chain elements. You should skip these -1 (completely ignore them). Second, for some target FFs, e.g., 8 as given above, the delay chains will not be long enough, e.g.,

```
25C_1.20V       PathID 354      InsPt 10
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
```

```
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
-1
000011001001111111
000011001001111111
```

This path has no transitions until FPA 37. Moreover, if the target FF is 8, then it is not ‘wide’ enough, i.e., this path is NOT ‘untimeable’ and should be ignored. This is true because the insertion point is 10, so the delay chain data begins with the rightmost portion ‘01111111’, which shows a ‘0’ (falling) transition emerging on this path’s output (ignore the dummy bitstring ‘0000110010’ on the left). But the delay chain does NOT include data for a target FF of 8 -- it ends at 7. Remember, the insertion point is numbered as 0. So this path can NOT be timed -- skip it.

Your algorithm should first find the initial value for the path so you know what type of transition to look for. You should use the snapshot closest to FPA 0 (that is not -1) to determine the path’s initial value. The inital value is the value of the bit at the RIGHTMOST position in this snapshot. For example, the initial value from the example directly above is ‘1’.

If the first snapshot that you examine in your backward search does NOT have the initial value, then you can NOT time the path -- skip it.

The most challenging issue to deal with are dynamic and static glitching, e.g., here are the last 5 snapshots from -40C_1.08V PathID 8469 InsPt 13

```
11111000101010000000100111111111111111111111111111111111111111
11111000101010000000100111111111111111111111111111111111111111
11111000101010000000100111111111111111111111111111111111111111
11111000101010000000100111111111111111111111111111111111111111
11111000101010000000100111111111111111111111111111111111111111
```

Note the hazard in the path data ‘0000000100111111111111111111111111111111111111111’

PROJECT: Part 2:

Now that you have the PNs, implement a bit generation technique based on thresholding as follows:

1) Discard all path IDs in all chips if any of the PNs for it at any of the TV corners produced a underflow (0) or overflow (158) value.
2) Compute the number of PNs that remain for each chip.
3) Truncate the list of PNs, i.e., remove PNs from the end of the list, of each chip such that the number of PNs is no longer than the number found in the SHORTEST list for one of the chips.
4) For each chip, randomly (using an ‘random’ function’) choose two PNs from the list and compare them. If the magnitude of the difference is greater than or equal to a user-defined threshold, generate a bit based on the sign of the difference, where negative generates a 0 and positive generates a 1.
5) Truncate the bitstrings to the size of the smallest bitstring for one of the chips.
6) Compute the inter-chip HD.
7) Download the NIST statistical tools (sts-2.1.1) and run the NIST tests on the bitstrings.

PROJECT: Part 3:

I am providing the core components of HELP in VHDL, which includes the following:

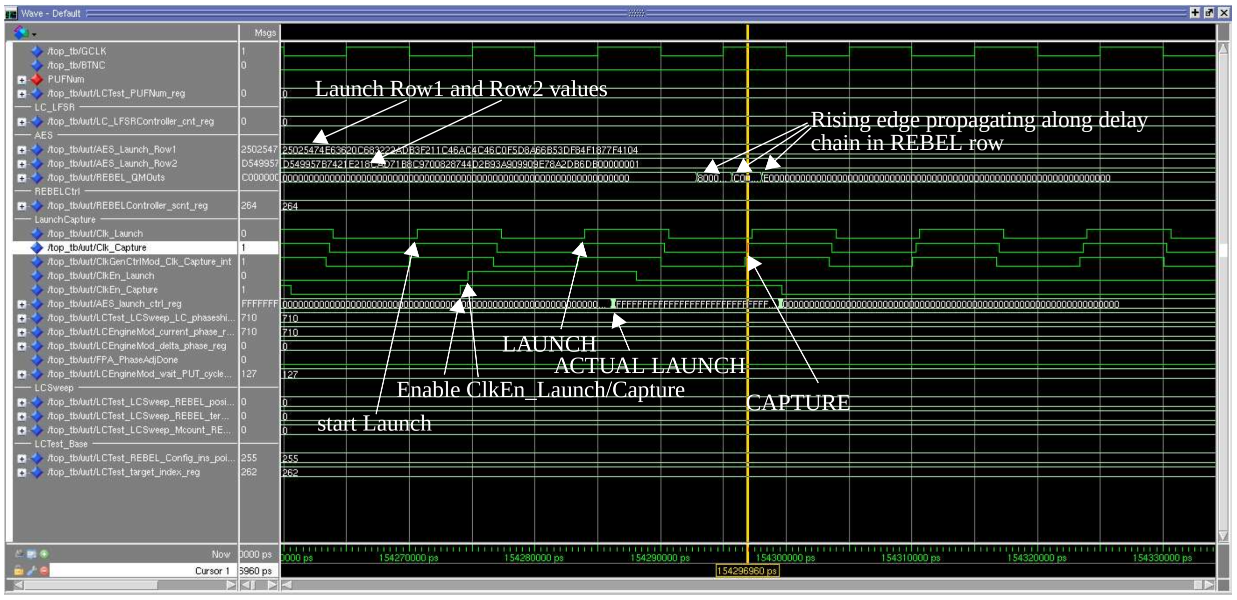
- **launch_capture_clk.vhd and xco:** An MMCM (mixed-mode clock manager), generated by coregen, with one 100 MHz input clock (GCLK for simulations and later this will be FCLK0 once we incorporate into EDK).
  Clk_Launch: Output clock 1, no phase shift
  Clk_Caputre: Output clock 2, fine phase shift (used only in REBEL row latches)
- **ClkGenCtrl.vhd**: A clock control module, which instantiates the launch_capture_clk MMCM and provides BUFGMUX primitives to allow the Clk_Capture output clock to be switched between phase non-phaseshifted Clk_Launch and phaseshifted Clk_Capture during launch capture event. A reset signal is also generated.
- **LaunchCaptureEngine.vhd:** A state machine to carry out the launch-capture event. A sequence of operations are carried out as follows:
  a) Fine phase is adjusted to a requested value (input to the module), beginning with a large phase shift and working toward smaller phase shifts (for subsequent launch-capture events).
  b) Set REBEL row into flush delay (FD) mode and wait for initial path value to propagate all the way along the 264 master latches.
  c) Switches Clk_Capture to phaseshifted version.
  d) Starts up two state machines that control ClkEn_Launch and ClkEn_Capture signals (see timing diagram).
  e) Switch Clk_Capture back to non-phaseshifted version.
  f) Disables FD mode.
- **LCSweep.vhd**: A state machine to carry out a complete sweep of launch-capture events by iterative ‘tuning’ the fine phase shift to smaller values until the propagating edge is pushed back into a target FF. It includes the following functions:
  a) Counts the number of transitions that occur in the digital snapshot (from the launch-capture event).
  b) If > 1, fail, else check if initial path output value is in target register. If it is, return success, else reduce fine phase shift and run launch-capture again (other details omitted)
- **LCTest_Base.vhd/_Driver.vhd:** State machines that carries out a single test. The following operations are carried out:
  a) An LFSR is started to load the two launch row FFs (256 FFs/row)
  b) The insertion point in the REBEL row is selected and a absolute target FF is computed using a user-specified ‘relative target FF’.
  c) A sweep is carried out by starting the LCSweep engine. If successful, the final fine phaseshift selected is returned and saved as the PUFNum.
- **REBEL/REBELController.vhd**: Implement REBEL row and scan chain controller for it.
- **ScanFFs.vhd**: Non-REBEL MUXD FFs for launch rows.
- **LC_LFSRController.vhd**: LFSR and scan chain controller for launch rows.
- **Top.ucf**: UCF constraints that relax timing constraints in the REBEL row. You should add these to the planAhead project under constaints.
- **Top.vhd**: Top level module that instantiates the above components and links them to I/Os. The low-order 8-bit PUFNum are connected to the ZEDBoard LEDs right now but need to be connected to two GPIO registers in an EDK project (see below).

I have simulated them using modelsim on the post-route version and it appears to work. I’ve also tested the code in hardware and it works. Please copy the VHDL down to your local machine and monitor for updates (REVISED: indicators) until class next Monday, at which point it will be removed.

You should create an ISE project, compile and simulate this code, at least at the behavioral level. (Note: Behavioral level simulations will model the phase shift properly and includes some delay modeling but it is crude. For example, the actual path delays within the AES unit will be significantly underestimated, which will result in the edges propagating through MANY FFs (40 or 50) along the delay chain. If you do a post-route simulation, the actual delay will allow the edge to propagate only through about 3 to 5 FFs.)

You should also create a planAhead project and add two GPIO ports, each 32-bits wide. See AVNET lab number 4. Name the first GPIO port CtrlRegA and make it an OUTPUT port and the second DataRegA and make it an INPUT port. I’ve provided a copy of my ‘system_stub.v’ file. (REVISED to include 2 GPIO instances, for 4 ports).

Launch-Capture timing diagram:



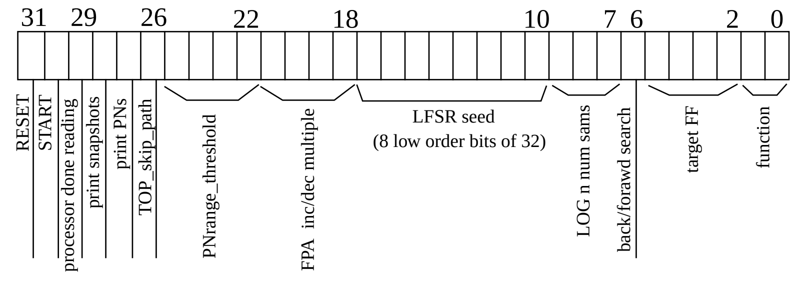
	Msgs
/top_tb/GCLK	1
/top_tb/BTNC	0
PUFNum	
/top_tb/uut/LCTest_PUFNum_reg	0
LC_LFSR	
/top_tb/uut/LC_LFSRController_cnt_reg	0
AES	
/top_tb/uut/AES_Launch_Row1	2502547
/top_tb/uut/AES_Launch_Row2	D549957
/top_tb/uut/REBEL_QMOuts	C000000
REBELCtrl	
/top_tb/uut/REBELController_scnt_reg	264
LaunchCapture	
/top_tb/uut/Clk_Launch	0
/top_tb/uut/Clk_Capture	1
/top_tb/uut/ClkGenCtrlMod_Clk_Capture_int	1
/top_tb/uut/ClkEn_Launch	0
/top_tb/uut/ClkEn_Capture	1
/top_tb/uut/AES_launch_ctrl_reg	FFFFFFF
/top_tb/uut/LCTest_LCSweep_LC_phaseshi...	710
/top_tb/uut/LCEngineMod_current_phase_r...	710
/top_tb/uut/LCEngineMod_delta_phase_reg	0
/top_tb/uut/FPA_PhaseAdjDone	0
/top_tb/uut/LCEngineMod_wait_PUT_cycle...	127
LCSweep	
/top_tb/uut/LCTest_LCSweep_REBEL_posi...	0
/top_tb/uut/LCTest_LCSweep_REBEL_ter...	0
/top_tb/uut/LCTest_LCSweep_Mcount_RE...	0
LCTest_Base	
/top_tb/uut/LCTest_REBEL_Config_ins_poi...	255
/top_tb/uut/LCTest_target_index_reg	262
Now	0000 ps
Cursor 1	6960 ps

PROJECT: Part 4:

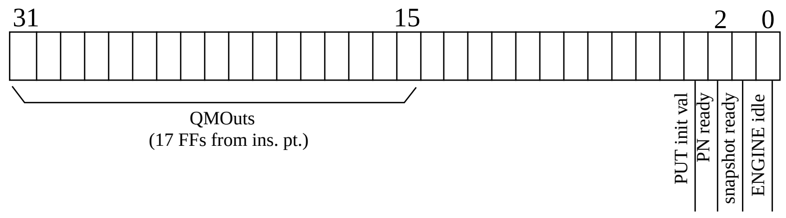
ProcessInputParams.vhd processes the CtrlRegA signals and some of the handshake DataRegA signals.

Signal assignments to the four 32-bit GPIO registers:

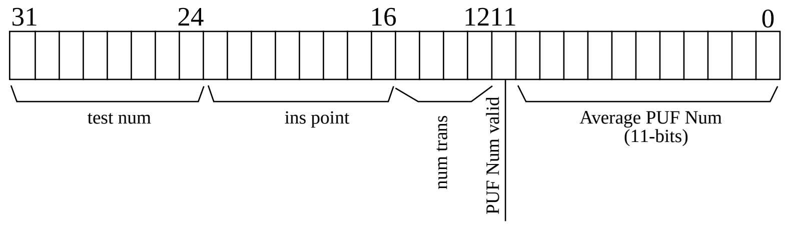
CtrlReg **(Update: 4/28/2014)**



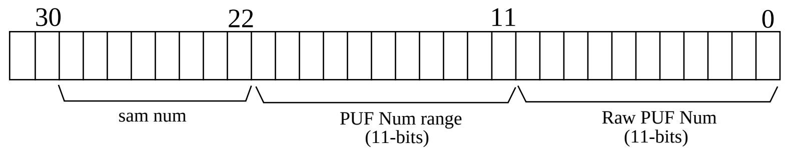
DataRegA (Valid when print snapshots = ‘1’ and snapshot ready = 1)



DataRegB (Valid when print PNs = ‘1’ and PN ready = 1)



DataRegC (Valid when print PNs = ‘1’ and PN ready = 1)



function:
“00”: enrollment
“01”: regeneration
“10”: authentication

FPA inc/dec multiple MUST be at least 1 (is forced to 1 if 0) and represents how many increments of 17.86ps (1 FPA at 50 MHz) are made between each LC test in a sweep.

During regeneration, you can save time and skip testing a path when the public data is ‘0’ by setting TOP_skip_path to ‘1’ and then issuing ‘START’

Added “Engine idle’ recently to make timing issues easier to resolve. You should wait for this bit to become ‘1’ after you issue a ‘START’, i.e., busy wait in your C program until it becomes 1.

Only 4, low order bits, of num trans(itions) are present, which are set to 15 if the actual number is >= 15.

There are two approaches to implementing the processor side control:
1) Standalone C program
2) Linux + C program

For either of these, you should first be sure to get HelloWorld to work. Please download the ‘ISE 14.5 Version’ under Zynq Concepts, Tools and Techniques on ZedBoard at http://www.zedboard.org/design/1521/11. Search for ‘2.1.2 Take a Test Drive! Exporting to SDK’ in the documentation (zedboard_refdoc-14-5_V4.pdf) provided with the download. You can use your existing planAhead project (which incorporates HELP) to do this (no need to generate the planAhead design described in earlier sections unless you want to).

For Standalone, Chapter 3 on page 34 of zedboard_refdoc-14-5_V4.pdf describes a board design that uses the AXI GPIO (similar to what you have for HELP). The helloworld.c program provided in the distribution will be helpful in designing your custom C program to control HELP (Note: if you do this version, most of the C code you write will be useful for the Linux version). This example code uses interrupts, which is not required in your code unless you want to use them.

Your C code should carry out the following functions:

- Initialize the GPIO interface (XGpio_Initialize)
- Set the directions for CTRL_CHANNEL(1) as OUTPUT and DATA_CHANNEL(2) as INPUT (these are set with respect to the processor).
- While(1)
- Read ASCII value, either ‘r’ for reset, ‘s’ for start or ‘q’ for quit
- if (‘r’), write a ‘1’ to CtrlReg ‘RESET’ and then a ‘0’. Follow this with a wait (I loop for 1000000 cycles) before returning to the top of the while loop. Probably not needed.
- if (‘s’), write a ‘1’ to the CtrlReg ‘START’ and then a ‘0’. Busy wait for 1000000 cycles. Enter a nested while(1) loop, read the DataReg and check the low order bit for ‘1’. If it is ‘1’, print the DataReg, and write a ‘1’ to the ‘processor done reading’ CtrlReg bit, busy wait for 1000000 cycles and repeat. When the low order bit of the DataReg becomes ‘0’, exit the nested loop.
- if (‘q’), break out of the outer loop

To operate the interface, press ‘r’, followed by a sequence of ‘s’s. Each time you press ‘s’, you will see the snapshots for one of the insertion points. The insertion points begin on the left and work toward the right. The relevant data in the delay chain changes as a function of the insertion point. DataRegA is coded so that it captures ONLY the relevant data, in particular, it captures 9-bits of the delay chain including the insertion point FF plus 8 FFs to its right.

On one of my ZedBoards, I get the following for the first test (first insertion point)

```
Snapshot: 0   InitVal: 1
Snapshot: 0   InitVal: 1
...
Snapshot: 1   InitVal: 1
...
Snapshot: 1   InitVal: 1
...
SnapShot: 2   InitVal: 1

PUF Data: SamNum 1   InsPt 255   NumTrans 3   Valid 0   PUFNum 705
```

The InitVal indicates that the output of this path under the first pattern is 1. The test data shows 0 for the first 10-20 snapshots, indicating that a 0 has propagated through all 9 FFs from the insertion point. The snapshot showing 1 indicates that a 1 is pushed back into the rightmost FF of the 9 FFs. The 1 disappears (it is a glitch and is between FFs in the delay chain) for a while and then a 2 appears. This “010’ pattern in the rightmost FF indicates a glitch and the engine aborts on an invalid path. Note the number of transitions is indicated as 3 but only 2 occur in “010”. The VHDL code actually inspects more of the delay chain FF than are printed (more FFs to the right of the 9) so there must be another transition that occurred -- you just can’t see it.

For regeneration, I’ve added TOP_skip_path to allow you to quickly skip a path without testing it. Here is my GetPUFNum routine that shows the usage model:

```
unsigned int GetPUFNum(FILE *OUTFILE, volatile unsigned int *CtrlReg, volatile unsigned int *DataRegA,
   volatile unsigned int *DataRegB, volatile unsigned int *DataRegC,
   int param_mask, int do_reset, int PUFNum_cnt, int num_sams,
   int valid_path, int enrollment, int *test_num_ptr,
   int max_public_data_size, char public_data[max_public_data_size], int *public_data_size_ptr)
   {
   unsigned int snapshot_ready, PN_ready, snapshot, PUT_init_val;
   unsigned int test_num, sam_num, ins_point, num_trans, PN_valid;
   unsigned int AvePUFNum, RawPUFNum, PUFNum_range;

   int input_data;

   int num_wait_cycles = 100000;
   int wait_cycles;

   int snapshot_cnter;

// For reporting ’level of effort’ during enrollment
   *test_num_ptr = 0;

// ==========================
// Perform a reset and wait
   if ( do_reset == 1 )
      {
      *CtrlReg = SOFT_RESET_MASK | param_mask;
      *CtrlReg = param_mask;

      wait_cycles = num_wait_cycles;
      while ( wait_cycles > 0 )
```

```
            wait_cycles--;
            }

      // Wait for the PUF engine to be in idle if necessary.
         while ( ((*DataRegA) & ENGINE_IDLE_MASK) == 0 );

      // Write 'start engine' bit in the GPIO control channel and then clear the bit. OR in the skip path bit into param_mask
      // if valid_path is 0 (when we want to skip testing this path).
         if ( valid_path == 0 )
            *CtrlReg = START_ENGINE_MASK | REGEN_SKIP_PATH_MASK | param_mask;
         else
            *CtrlReg = START_ENGINE_MASK | param_mask;
         *CtrlReg = param_mask;

      // Nothing to wait for if we skipped testing this path -- LFSR and insertion point will be updated to next value.
      // NOTE: If we update the LFSR, it may be quite a few cycles before we return to idle in LCTest_Driver.vhd. Wait
      // here until that happens. This is the only one that ever actually spins (waits) and only if the launch rows need
      // to be loaded with new vectors.
         if ( valid_path == 0 )
            {
            while ( ((*DataRegA) & ENGINE_IDLE_MASK) == 0 );
            return 0;
            }
      // =========================
      // =========================
      // Keep starting the engine until you get the requested number of (valid?) PUFNums.
         snapshot_cnter = 0;
         while (1)
            {

      // Get the state of the status signals.
            input_data = *DataRegA;
            snapshot_ready = input_data & (SNAPSHOT_READY_MASK << SNAPSHOT_READY_POS);
            snapshot_ready >>= SNAPSHOT_READY_POS;
            PN_ready = input_data & (PN_READY_MASK << PN_READY_POS);
            PN_ready >>= PN_READY_POS;

      // When DEBUG snapshots is enabled, we stop after every test during the sweep.
            if ( snapshot_ready == 1 )
               {

      // Currently, 17 bits of the relevant part of the delay chain are stored in the DataReg.
               snapshot = input_data & (SNAPSHOT_MASK << SNAPSHOT_POS);
               snapshot >>= SNAPSHOT_POS;

      // I also store the initial value of the path
               PUT_init_val = input_data & (PUT_INIT_VAL_MASK << PUT_INIT_VAL_POS);
               PUT_init_val >>= PUT_INIT_VAL_POS;

      // If the initial value is a '1' then a falling transition is propagating. The '0's on the
      // left side of the snapshot are not printed unless forced.
               if ( snapshot_cnter == 0 )
                  fprintf(OUTFILE, "%05x\t%d\n", snapshot, PUT_init_val);
               else
                  fprintf(OUTFILE, "%05x\n", snapshot);

               *CtrlReg = PROCESSOR_DONE_READING_MASK | param_mask;
               *CtrlReg = param_mask;
               snapshot_cnter++;
               }

      // When DEBUG PN is enabled, we stop after each sweep is complete (success or failure).
            else if ( PN_ready == 1 )
               {

               input_data = *DataRegB;
               test_num = input_data & (TEST_NUM_MASK << TEST_NUM_POS);
               test_num >>= TEST_NUM_POS;

               ins_point = input_data & (INS_POINT_MASK << INS_POINT_POS);
               ins_point >>= INS_POINT_POS;

               num_trans = input_data & (NUM_TRANS_MASK << NUM_TRANS_POS);
               num_trans >>= NUM_TRANS_POS;

               PN_valid = input_data & (PN_VALID_MASK << PN_VALID_POS);
               PN_valid >>= PN_VALID_POS;

               AvePUFNum = input_data & (AVE_PUFNUM_MASK << AVE_PUFNUM_POS);
               AvePUFNum >>= AVE_PUFNUM_POS;

               input_data = *DataRegC;
               sam_num = input_data & (SAM_NUM_MASK << SAM_NUM_POS);
               sam_num >>= SAM_NUM_POS;

               RawPUFNum = input_data & (RAW_PUFNUM_MASK << RAW_PUFNUM_POS);
               RawPUFNum >>= RAW_PUFNUM_POS;

               PUFNum_range = input_data & (PUFNUM_RANGE_MASK << PUFNUM_RANGE_POS);
               PUFNum_range >>= PUFNUM_RANGE_POS;

                fprintf(OUTFILE, "PUFNum count %5d\tTestNum %3d\tInsPt %3d\tSamNum %3d\tNumTrans %d\tValid %d\tAvePUFNum %4d\tRaw-
PUFNum %4d\tPUFNumRange %4d\n",
                  PUFNum_cnt, test_num, ins_point, sam_num, num_trans, PN_valid, AvePUFNum, RawPUFNum, PUFNum_range);

      // Indicate that we are done reading the data
               *CtrlReg = PROCESSOR_DONE_READING_MASK | param_mask;
               *CtrlReg = param_mask;

      // If we've processed the last sample for this PUFNum or PN_valid is false (in which case, we will NOT process
      // any more samples), then increment PUFNum_cnt and restart the engine if we still have PUFNums to produce.
               if ( sam_num == num_sams || PN_valid == 0 )
                  {

      // If enrollment, record in public data the value of the valid bit so we can re-play the right sequence
      // during regeneration. NOTE: We can NOT write the valid bits to a file directly because the bit generation
      // algorithm may have additional criteria and may need to invalidate a PUFNum considered valid in this routine.
                  if ( enrollment == 1 )
                     {
                     if ( *public_data_size_ptr == max_public_data_size )
                        { printf(“ERROR: GetPUFNum(): PUBLIC DATA SIZE EXCEEDED!\n”); exit(1); }
                     public_data[*public_data_size_ptr] = (char)PN_valid;
                     (*public_data_size_ptr)++;
                     }

      // We’re done if all samples are processed and we have a valid PUFNum
                  if ( sam_num == num_sams && PN_valid == 1 )
                     {
                     fprintf(OUTFILE, “Returning VALID!\n”);
                     break;
                     }
                  else
                     {

      // Wait for the PUF engine to return to idle and re-start it to obtain the other samples.
                     while ( ((*DataRegA) & ENGINE_IDLE_MASK) == 0 );
                     *CtrlReg = START_ENGINE_MASK | param_mask;
                     *CtrlReg = param_mask;
                     }
                  }

               snapshot_cnter = 0;
               }
            }

      // For reporting ’level of effort’ during enrollment
         *test_num_ptr = test_num;

         return AvePUFNum;
         }
```

PROJECT: Part 5:

Buy a Sandisk reader and a couple 8 GB disks. This is the easiest way to get a linux OS running on the Zedboard.

Once you complete the build, you disk will have 4 files:

- BOOT.BIN (container for FSBL, bitstream and u-boot)
- zImage (linux kernel)
- ramdisk8M.image.gz (RAM disk file system)
- devicetree.dtb (device tree)

Both Xilinx and Digitent have git repositories for the u-boot and the linux kernel. NOTE: You ONLY need these if you plan to build you own kernel. We can use one of the pre-built kernels for our project (initially) -- see below.

- git clone git://github.com/Xilinx/linux-xlnx.git
- git clone git://github.com/Xilinx/u-boot-xlnx.git
- git clone git://github.com/Xilinx/device-tree.git

- git clone https://github.com/Digilent/u-boot-digilent
- git clone https://github.com/Digilent/linux-digilent.git

ZedBoard tutorial (must read) http://www.zedboard.org/design/1521/11, go to “Zynq Concepts, Tools, and Techniques on ZedBoard” and download ‘ISE 14.5 Version’
You can create your Sandisk from the reference design directory provided in this zip distribution.

Digilent examples and tutorial:
http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,1028&Prod=ZED-BOARD&CFID=2517716&CFTOKEN=84435962
Read through “Getting Started with Embedded Linux - ZedBoard” and “Embedded Linux Hands-on Tutorial”

The Hands-on Tutorial has an example linux device driver which we will get to eventually. Turns out you can use ‘mmap’ as an alternative to a device driver to access the GPIO CtrlRegA and DataRegA registers. This also allows you to use the pre-built kernel images.

Key components of your C program will include:

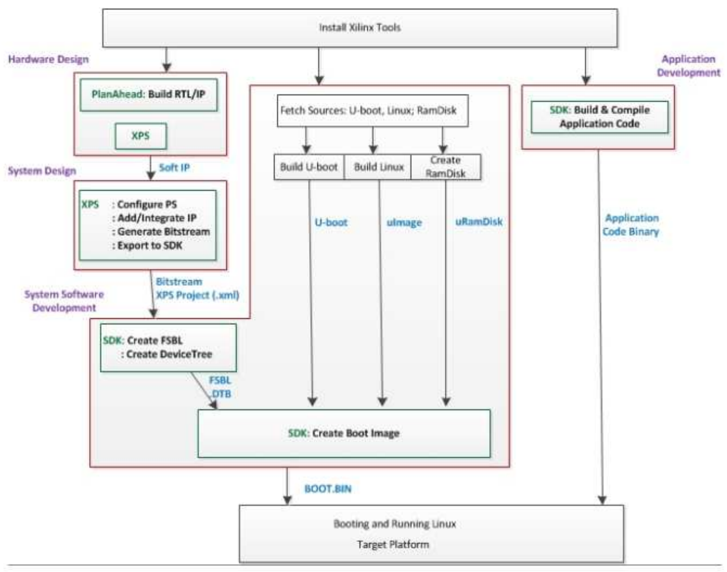
```
// GPIO 0
#define GPIO_0_BASE_ADDR 0x41200000

// GPIO 1
#define GPIO_1_BASE_ADDR 0x41240000

volatile unsigned int *CtrlReg;
volatile unsigned int *DataRegA;
volatile unsigned int *DataRegB;
volatile unsigned int *DataRegC;

int fd = open("/dev/mem", O_RDWR|O_SYNC);

if (fd < 0)
   {
   printf("Error: /dev/mem could NOT be opened!");
   exit (1);
   }

// Add 2 for the DataReg (for an offset of 8 bytes for 32-bit integer variables)
CtrlReg = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, GPIO_0_BASE_ADDR);
DataRegA = CtrlReg + 2;

DataRegB = mmap(0, getpagesize(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, GPIO_1_BASE_ADDR);
DataRegC = DataRegB + 2;
```

Once you have the standalone version running, you can convert to the linux version fairly easily.
Host computer: type ‘ifconfig eth0 192.168.1.11 netmask 255.255.255.0’. Zynq linux OS should be set to 192.168.1.10.

Linux on the Zync (http://www.wiki.xilinx.com/Getting+Started):

The Xilinx Design Flow

The figure shows a high level block diagram of the Xilinx design flow for Zynq AP SoC.


Install Xilinx Tools
Hardware Design
PlanAhead: Build RTL/IP
XPS
Soft IP
System Design
XPS : Configure PS
: Add/Integrate IP
: Generate Bitstream
: Export to SDK
Bitstream
XPS Project (.xml)
System Software Development
SDK: Create FSBL
: Create DeviceTree
FSBL
DTB
Fetch Sources: U-boot, Linux; RamDisk
Build U-boot
Build Linux
Create RamDisk
U-boot
uImage
uRamDisk
SDK: Create Boot Image
BOOT.BIN
Application Development
SDK: Build & Compile Application Code
Application Code Binary
Booting and Running Linux
Target Platform