VarLEON: FPGA Based Processor Variability Emulator for Variation Aware Software

ABHISHEK BHATIA (404-191-904)
Dept. of Electrical Engineering
University of California, Los Angeles, USA
bhatiaabhishek1991@gmail.com
Advisor: Prof. Puneet Gupta

***Abstract*— In the area of design automation, variability emulation is a powerful idea that can help bridge the gap between a conservative design and the opportunistic characteristics in a software. A variability emulator can be used to explicitly control hardware variability to study development of variation-aware software such as in Underdesigned and Opportunistic Computing (UnO) [1]. This project focuses on development of such an emulator comprising of both power and delay variability injection, on a LEON3 microprocessor. The implementation was realized on a Virtex-5 FPGA using Xilinx design tools.**

Keywords*— *Delay Variability, Power Variability, Variable Delay Element, LEON3, FPGA Editor, Kuhn-Munkres Algorithm, GRMON

I. INTRODUCTION

As the manufacturing processes scale by large factors each year, we are now beginning to experience variations of performance, power and reliability across manufactured parts. Apart from manufacturing processes, environment variables such as temperature and humidity may also affect performance. These variations necessitate the need for stringent guardbands and hardware specifications that hardware engineers try to meet. A chip's maximum speed is reported by taking into account the worst case scenarios of manufacturing imperfections, environmental variables and power supply fluctuations. This increases cost and also fails to fully leverage the full potential of the software's inherent flexibility and resilience. Most chips are capable of performing beyond their specifications. To solve such a problem, [1] proposes a novel hardware-software stack interface that adapts to variations in the hardware, thus relaxing the guard-bands in hardware design. The hardware can be under-designed and the software stack along with sensing circuits can sense and adapt to variations. Such a development could benefit from a reconfigurable variability emulator which can be a useful tool to study real world hardware variability scenarios.

This project deals with development of such a variability emulator on a LEON3, a SPARC-V8-based microprocessor implemented on a Xilinx's XUPV5-LX110T Development System which is based around Virtex-5 FPGA. The source code of LEON3 is freely available under the GNU GPL license.

II. DELAY INJECTION METHODOLOGY

The following sections describe the FPGA implementation of the emulator. The power variability was implemented inside the RTL, while the delay variability was introduced later in the FPGA flow.

A. Working Principle of the Variability Emulator

The variability emulator comprises of both power variations as well as delay variations which can be manipulated on-the-fly by writing to a memory-mapped register (0x80000a00) [2] using pointer dereference to the memory address of the register. The power variations are realized using ring oscillators in the LEON3 RTL at strategic locations in the pipeline, specifically for the four major functional groups of ALU, FPU and conditions for branches. Though results for both power and delay variability have been presented in the results, this project focuses on the delay variations which have been realized using tapped buffer chains (which map to LUT delays in FPGA implementation). The basic implementation architecture for the variability emulator is depicted in figure 1.

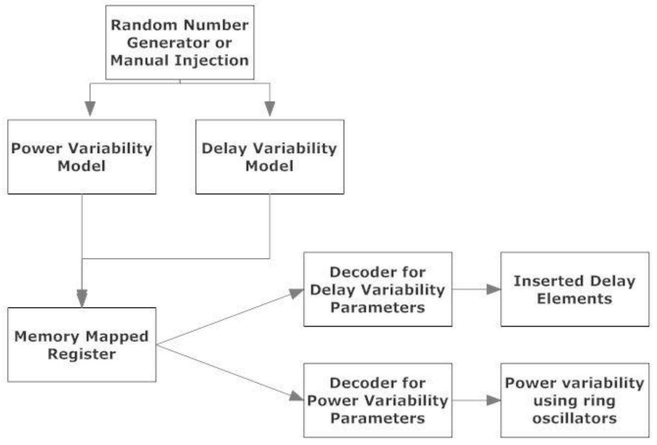

Random Number Generator or Manual Injection
Power Variability Model
Delay Variability Model
Memory Mapped Register
Decoder for Delay Variability Parameters
Inserted Delay Elements
Decoder for Power Variability Parameters
Power variability using ring oscillators

Fig.1 Architecture of the Variability Emulator

The programming configuration for the 32-bit memory-mapped register is depicted in figure 2. The brackets specify which part of the emulator do the bits control.

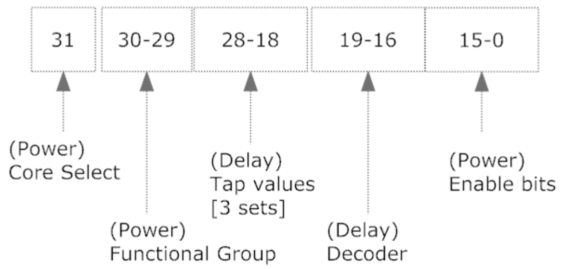


Fig.2 Memory-Mapped Register (0x80000a00)

As shown by the above figure, the delay tap values for the injected variable-delay elements can be configured dynamically from the software itself. Each delay element has 8 taps (3 –bits) and 3 delay elements (9-bits – 28:18) can be configured in one clock cycle synchronous to the clock of the

CPU core. Bits 19:16 are used by the delay decoder to interpret as to which delay elements are to be configured in that particular cycle. The emulator provides for a maximum of 48 delay elements which take 16 cycles (4-bit decoder) for all to be programmed. Refer figure 3 for the architecture for delay variability. The architecture of the decoder has been discussed in Section C. Once inserted on a set of critical paths, each of the delay elements can be configured to have one of the 8 delay values, and once the delay value exceeds the available slack for a given clock period on a particular path, the operation will fail.

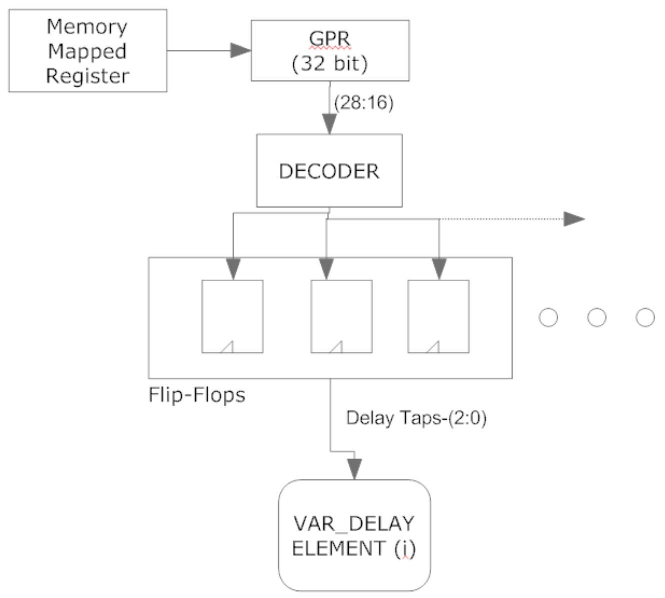


Fig.3 Architecture for Delay Variability

The injected power variations can also be controlled by the same general purpose register according to the following bit assignments:

Bit 31→ Selects the core (CPU 0 or 1) in which the power variations are inserted.
Bit 30:29→ Selects which functional group is associated with the inserted power variations by the following rule.

2'b00: Memory
2'b01: ALU
2'b10: Floating Point Unit
2'b11: Conditions for branches

Bit 15:0→ Selects which set of ring oscillators are enabled. The emulator provides for a set of 16 oscillators hence giving finer control over the variability.

B. The Variable Delay Element Hard Macro

The Variable Delay Element (VAR_ELE) was first designed in RTL as a chain of 8 buffers with taps after each buffer, all of which were multiplexed to one output. Hence the delay value can be configured using 3 bits. It synthesized and implemented using Xilinx ISE tools, and optimized to occupy 2 Slices as shown in figure 4. The design was saved as a hard macro with the bottom slice as the 'Reference Group' implying that the macro would always be placed in this configuration with both the slices and the nets relationally placed. A hard macro is nothing but a special reusable circuit block in which every aspect of the design is preserved from LUT configuration to the physical routing between components. The IOB pads need to be unplaced /deleted and the slice pins connected to those pads need to be declared as external pins. One thing to note is that to save the design as a hard macro, the VCC/power nets have to be removed. But a workaround here is to assign the VDD port as external input pins and connect them to the power sources after instantiating the macro in a design.

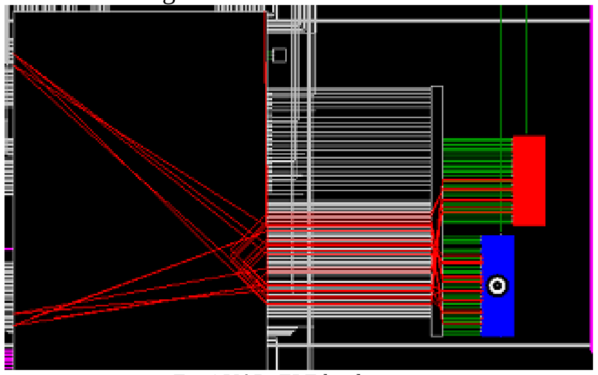


Fig.4 VAR_ELE hard macro

A post-place-and-route simulation was performed to confirm the integrity of the logic (see Appendix A.1 for the waveform). As seen in the simulation, the output signal is progressively delayed from a value of 1.22ns (tap0) to 5.896 ns (tap7) as the tap value ($delay_sel) is incremented.

C. Delay Insertion Flow

A general purpose register is available as a module called GPREG in the leon3 source code itself, which can be instantiated and the parameters specify the address of that register. The Decoder for delay taps was incorporated in the RTL itself inside the top level module of leon3, connected to the General Purpose Register. It can be modeled as a 9bitX16 (18B) set of Flip-Flops which is synchronously written to, by the general purpose register. The output of each such flip-flop was left unconnected and preserved through the synthesis and mapping phase for them to be connected later to the VAR_ELE elements. The Figure 5 depicts the decoder.

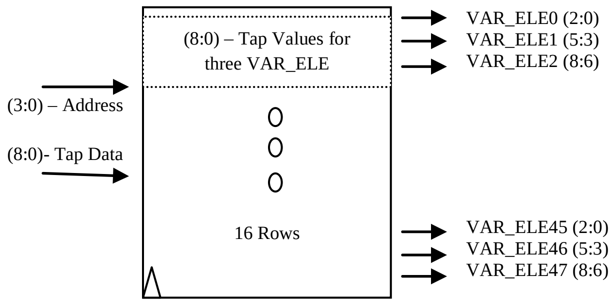


Fig.5 Decoder for Delay Tap Configuration

The first step was to reserve a grid of 13X20 pairs of evenly distributed slice locations before the LEON3 design is mapped. This was achieved using "CONFIG PROHIBIT" constraint in the User Constraint File (.UCF) [4] (Refer Appendix). Pre-reserving slices gives us the flexibility to place the delay elements as near to the target paths as possible thus minimizing the cost (route) of adding extra elements to the design (refer section D). The UCF file was also populated with constraints (NET S) preventing the trimming of the unconnected output ports of the delay decoder. The 'KEEP' attribute was also used in conjecture with the UCF constraints. After the design was placed and routed, timing analysis was done, based on which, critical paths were identified and the elements were inserted in the 'Pre-Route' stage using the FPGA Editor (refer section E). The tap select lines for each of the elements were tied to the open outputs of the above decoder. The resulting design was then routed by the Place-And-Route (PAR) tool. Fig.6 gives an overview of the delay variability insertion flow. Please note that the Macro could only be placed on type 'SLICEM' and hence the right grid needed to be figured out.

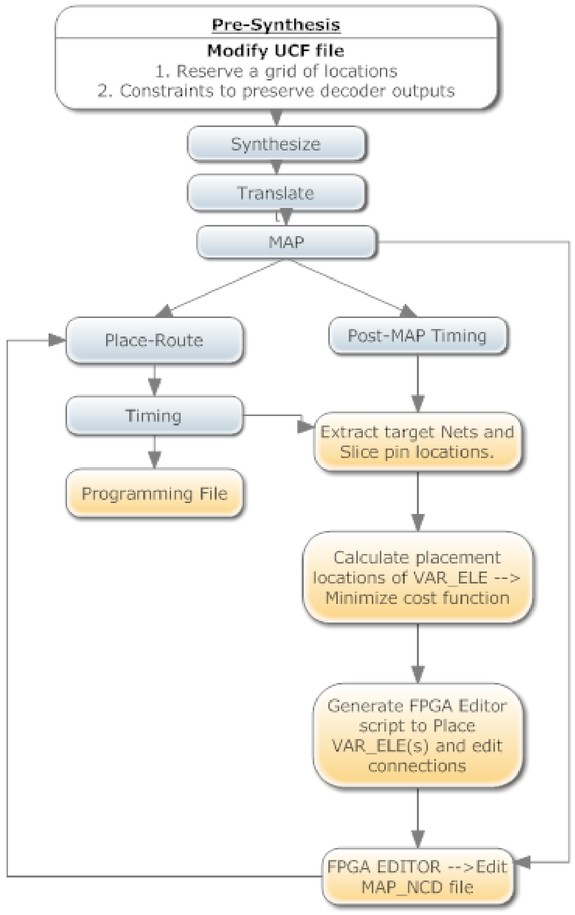

Pre-Synthesis
Modify UCF file
1. Reserve a grid of locations
2. Constraints to preserve decoder outputs
Synthesize
Translate
MAP
Place-Route
Post-MAP Timing
Timing
Extract target Nets and Slice pin locations.
Programming File
Calculate placement locations of VAR_ELE --> Minimize cost function
Generate FPGA Editor script to Place VAR_ELE(s) and edit connections
FPGA EDITOR -->Edit MAP_NCD file

Fig..6 General Insertion Flowchart

D. *Assignment Problem- 'Munkres'*

One of the issues with inserting delay elements on an FPGA board is the fact that the resources available are limited and the elements can only be placed on SLICE locations which were reserved in the UCF file. At the same time, the route overhead of all such insertions should be collectively as low as possible. Consider Fig.7 as a small example of this problem.

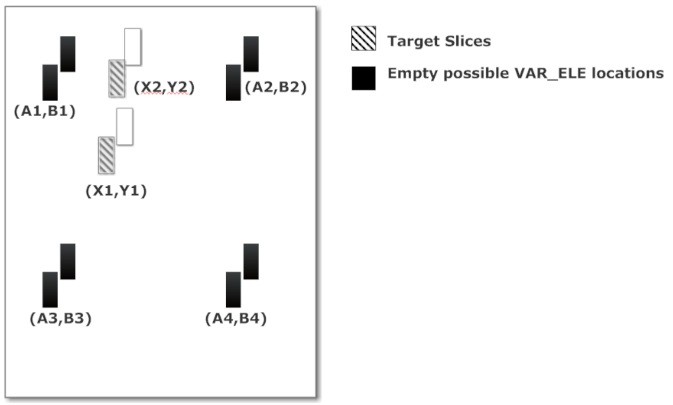


Fig.7 Assignment Problem

Let there be two target slices before which the VAR_ELE(s) need to be inserted. The black locations signify empty reserved locations where they can be possibly inserted. Now, the problem is to find an optimal assignment of two such black locations to the target locations such that the total overhead is as low as possible. In this project, the cost of placing VAR_ELE0 (for X1, Y1) at (Ai,Bi) was C_{1i} = (X1-Ai)+(Y1-Bi). Similarly the cost for placing VAR_ELE1 (for X2, Y2) at a location of (Aj,Bj) was C_{2j} = (X2-Aj)+(Y2-Bj). The problem now reduces to finding *i* and *j* such that the total cost $C_{1i} + C_{2j}$ is minimized. This is a standard assignment problem which can be solved using Kuhn-Munkres algorithm (a.k.a. Hungarian algorithm [5]) in polynomial time [6].

A cost matrix was created using Perl and the algorithm was applied to find the optimal assignment of the locations to the target slices. A Perl package for Munkres Algorithm was used which takes the following matrix as the input and gives a single dimensional array as an output which denotes which column is assigned to which row [7].

	(A1,B1)	(A2,B2)	(A3,B3)	(A4,B4)
(X1,Y1)	C_{11}	C_{12}	C_{13}	C_{14}
(X2,Y2)	C_{21}	C_{22}	C_{23}	C_{24}

E. *FPGA Editor Flow*

In this project, the delay elements were inserted by editing the NCD (Native Circuit Description) file which was generated by the MAP process. A Perl script was developed which apart from solving the assignment problem, also generated a (.scr) script as an input to the FPGA Editor tool [3] which automated the process of insertion of delay elements on the desired paths. The inputs to the Perl script are the net name as well as the SLICE pin location of the target Flip-Flop. The FPGA Editor script first places the VAR_ELE(s) macros on the assigned locations and then connects them across their assigned paths. Fig.8 depicts the flow followed by the FPGA editor script in inserting the delay elements.

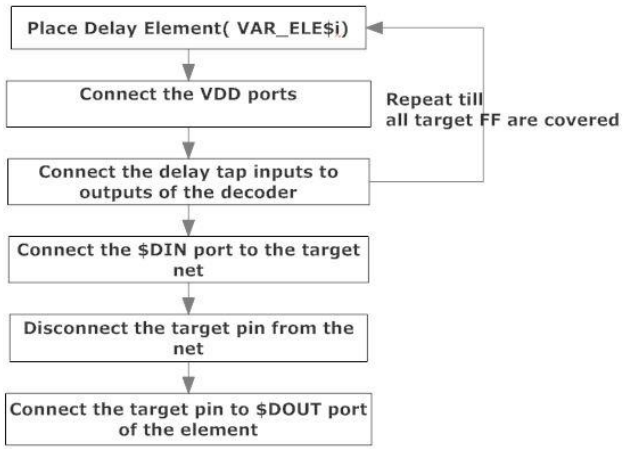

Place Delay Element(VAR_ELE$i)
Connect the VDD ports
Connect the delay tap inputs to outputs of the decoder
Connect the $DIN port to the target net
Disconnect the target pin from the net
Connect the target pin to $DOUT port of the element
Repeat till all target FF are covered

Fig.8 Flow of the FPGA_ED Script

The edited NCD file was then passed onto the PAR tool which takes care of the routing of the design. Insertion of the delay elements needed to be done as late in the flow as possible so that the critical paths are not affected in terms of optimizations, because of the inserted logic. It was decided to perform the insertion just after placement but before the route phase. The PAR tool produces another NCD file which a fully routed version of the MAP_NCD version. There were issues with editing a fully routed design since the PAR tool swaps pins to optimize logic which created errors in inserting the delay elements.

The resultant design was passed onto the bitgen tool which produces the bit programming file using the NCD file and a Physical Constraint File (PCF) as inputs. Xilinx's iMPACT software was used to download to the bit file to the FPGA.

III. TESTING

There was a need to find a concrete way to test the working of the emulator both in terms of delay and power. To the test the functioning of the delay taps, a path in the "Execute" stage of the integer pipeline was chosen due to its critical nature in the architecture. Since in timing slack aspect, it was not the most critical path, the core frequency needed to be increased for it to start failing on changing the taps. The frequency needed to be changed without repeating the whole place and route process so that the layout of the design remains the same for the entire frequency sweep. The power was tested using "Watts Up Pro" power meter attached to the FPGA board and running benchmarks with varying the enabling/disabling the ring oscillators.

The LEON3 design derives its clocks from two PLLs (Phase Locked Loops) which use the on board 200 MHz and 100MHz clock chips integrated on the XUPV5-LX110T. A PLL uses an input clock, an integer multiplier and an integer divider to produce the desired frequencies. The core uses the the PLL with the 100 MHz clock as the input. The default setting is for Multiplier = 8 and Divider = 10, resulting in a core frequency of 80 MHz. To change the frequency after placement and routing, the FPGA Editor was used to modify the fully routed NCD file (refer Fig.9). The parameters of the PLL were modified to generate the desired output frequencies. The design was then bit-streamed and downloaded each time a new frequency was to be tested. The LEON3 that was placed and routed for 80 MHz was found to be able to operate up to 140 MHz (without delay insertion).

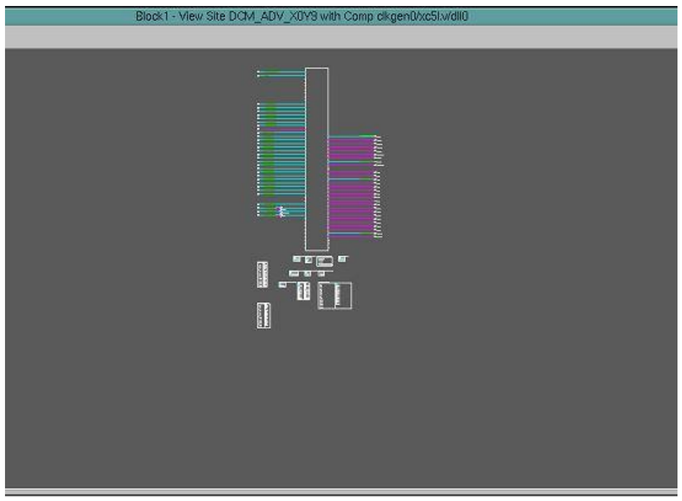
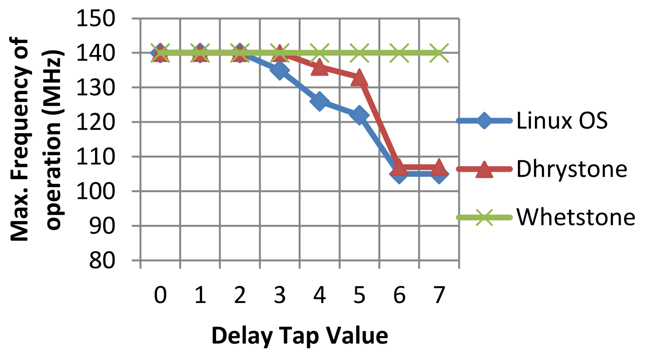

Block1 - View Site DCM_ADV_X0Y9 with Comp clkgen0/xc5l.v/dll0

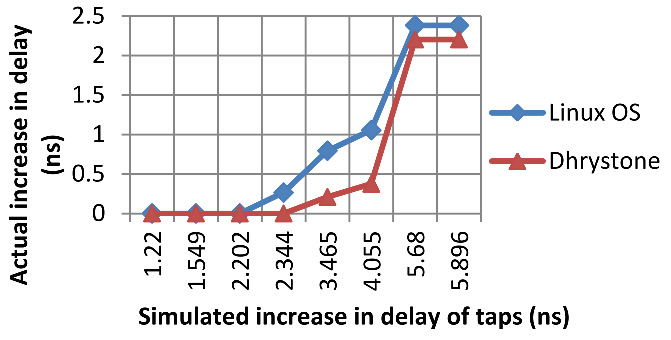
Fig.9 Viewing the PLL in FPGA Editor

IV. EXPERIMENTAL RESULTS

For testing the LEON3 processor as a whole, a popular debug monitor 'GRMON' was used to load and run applications on the target hardware (Virtex-5-FPGA). Benchmarks such as 'Dhrystone', 'Whetstone' and Linux OS were run with different delay tap values ranging from 0-7 for a sweep of frequencies between 80 MHz and 140 Mhz. If the output of the GRMON monitor was as expected, then the benchmarks were deemed as pass. The following graph depicts the max core frequency each benchmark runs for different tap values.



As expected, the programs start failing as the tap value is increased. Whetstone runs on the Floating-Point-Unit module and hence doesn't fail due to the delay element which was introduced in the integer unit. Please note that since the path chosen is not a true critical path, the other paths of the core fail at 140 MHz before the experimental path. Hence the graph terminates at 140 MHz. Another graph (see below) was plotted to see the relationship between the delay of each tap based on timing analysis and the delay actually observed. It is observed that the tap value 6 and 7 have nearly the same delay value since the fanout of for tap 7 is only 1 (the output multiplexer) and that for other stages is 2 (the next stage as well as the output multiplexer). Hence, the last stage does not offer much incremental delay compared to others.



The power variability was tested using a "Watts Up Pro" Power meter connected to the board and running Dhrystone benchmark with varying number of active ring oscillators. The ring oscillators were enabled only if an ALU operation was active. The result for dhyrstone benchmark is showed in the figure below.

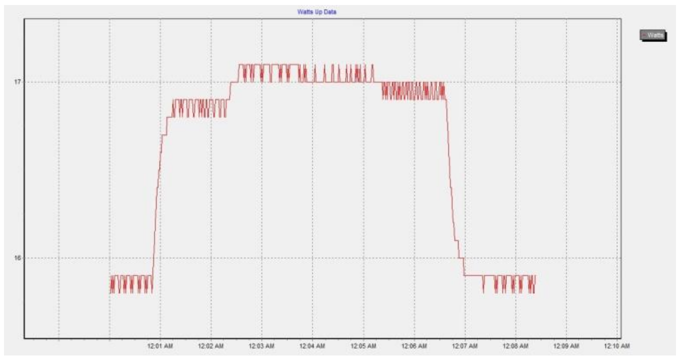


Fig..10 Dhrystone benchmark power variations

V. CONCLUSION

This project focused on conceptual development of a variability emulator for a LEON3 processor on Virtex-5 FPGA platform (ML509). Architecture was first developed for the emulator and then individual components were designed and tested. Various levels of delays can be injected before Flip-Flops and controlled by user registers using memory mapped registers which gives an important ability to reconfigure variability dynamically during execution of standalone programs or applications on LEON3 Linux OS.

This is a first step towards development to a large scale variability emulator and can be improved upon in the future by developing flows to inject delay elements in a fully routed design. Provision can be added for more number of delay elements (>100). Also, it can be an important step in creating variability models based on which software can be developed that dynamically adapts to a modeled hardware and removes the need of an overdesigned hardware [1].

VI. ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Prof. Puneet Gupta for his esteemed guidance and mentorship. I would also like to acknowledge Chengfei Tao for his work and results on the power variability implementation of the emulator.

VII. REFERENCES

[1] P. Gupta *et al.*, "Underdesigned and opportunistic computing in presence of hardware variability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2012, Keynote Paper

[2] *GRLIB IP Core User's Manual Version 1.3.1 - B4135*, Aerolib Gaisler, 2013

[3] *Xilinx FPGA Editor Guide - 2.1i*, Printed in USA

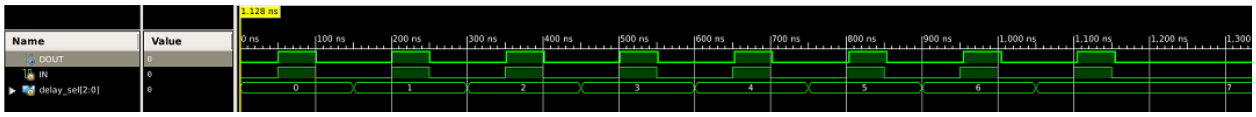
[4] *Xilinx Constraint Guide – 10.1*, Printed in USA

[5] Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, **2**:83–97, 1955. Kuhn's original publication

[6] J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society for Industrial and Applied Mathematics*, **5**(1):32–38, 1957 March.

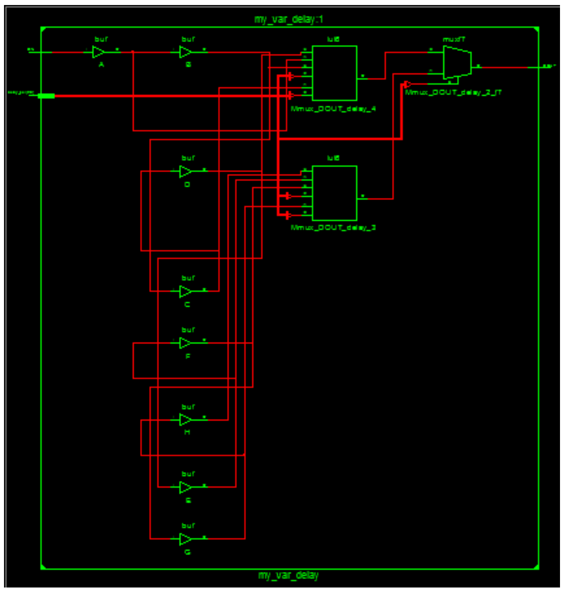
[7] http://search.cpan.org/~anaghakk/Algorithm-Munkres-0.02/lib/Algorithm/Munkres.pm

APPENDIX A



A.1 Post PAR Simulation of the variable delay element (VAR_ELE)

A.2 Schematic of VAR_ELE



A.3 **List of Scripts**

1. ucf_editor_script.pl : Perl Script that modifies the UCF file to include "CONFIG PROHIBIT and "NET S" constraints. It also outputs a text file "LOC_VAR_ELE.txt" that lists all the slice locations that were reserved for the delay element insertion.

2. fpga_editor_script_gen_end.pl : Perl script that takes the target pin locations and target net as inputs. It solves the assignment problem as well as generates a script file named "macro_insertion1.scr" which goes as an input to the FPGA Editor tool.

A4. **Xilinx Commands**

1. /opt/Xilinx/13.2/ISE_DS/ISE/bin/lin/fpga_editor -e leon3mp.ncd → To open the design in the FPGA Editor tool.
2. par -w -intstyle silent -ol high -mt off leon3mp_map.ncd leon3mp.ncd leon3mp.pcf → To place and route the edited mapped design.
3. bitgen -intstyle silent -f leon3mp.ut leon3mp.ncd → To generate bitstream for downloading to FPGA.