Τμήμα Φυσικής
Πανεπιστήμιο Ιωαννίνων
ΠΜΣ στη Φυσική με Ειδικεύσεις
στη Θεωρητική και στην Πειραματική Φυσική

True Random Number Generation and Evaluation Using Silicon Die Process Variations in FPGAs

Ανάπτυξη και αξιολόγηση γεννήτριας πραγματικά τυχαίων αριθμών με FPGA αξιοποιώντας την ανομοιογένεια του υποστρώματος του πυριτίου

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Fotos Theophilos
Φώτος Θεόφιλος

Επίβλεψη: Καθηγητής Μάνθος Νικόλαος
Εργαστήριο Φυσικής Υψηλών Ενεργειών

Ιωάννινα, Οκτώβριος 2022

Abstract

The Purpose of this MSc thesis is the implementation and evaluation of a True Random Number Generator (TRNG) in an FPGA, using Ring Oscillators.

The particular design attempts to leverage the inherent differences found across different regions of silicon on the same chip, so called Process (P) variations, along with global and local Voltage (V) variations and local and global Temperature (T) variations, as sources of randomness. The effect that is observed on the ring oscillators' behavior as a result of those variations is the formation of significant oscillation period variation, also known as Jitter, or Phase Noise.

A simplified design was first implemented and tested to observe the non-deterministic behavior of the Ring Oscillators, and demonstrate their frequency's sensitivity to substrate Process variations, Voltage supply, and Temperature.

Following this, a group of 16 Ring Oscillators were implemented in a Zynq Ultrascale+ MPSoC FPGA evaluation board, to yield a 16-bit true random number ouput. A 16-bit wide output offers certain advantages when used in an evaluation circuit that simplify the testing circuit's implementation. A programmable sampler that is used to undersample (decimate) the Rings' output, was developed, and different sampling rates were tested to observe loss/gain of entropy. Finally, a Statistical Engine was implemented, that calculates the first 10 raw statistical moments and the output stream's bias and captures the distribution of the distance between consecutive instances of '1' in the output bits, which for a truly random sequence should approximate a Binomial distribution.

The TRNG device was evaluated using the synthesized statistical engine, and was first tested in a standalone configuration, without any Post Processing, for different ring lengths and sampling rates. As a result of the theoretical and statistical analysis conducted, medium sized rings were found to be the best candidates for use in a True Random Number Generator, due to them having reduced bias compared to small rings, and yielding greater random number production rates than their largest counterparts.

When a Post Processing stage is attached to the TRNG's output, small rings become the ideal candidates for random number generation, due to elimination of most of the bias combined with increased data rates and reduced silicon real estate utilization.

Η έγκριση της Μεταπτυχιακής Διπλωματικής Εργασίας από το τμήμα Φυσικής του Πανεπιστημίου Ιωαννίνων δεν υποδηλώνει αποδοχή των απόψεων του συγγραφέα (Ν. 5343/4932, άρθρο 202).

Περίληψη

Η Μεταπτυχιακή Διπλωματική Εργασία αυτή αφορά τον σχεδιασμό, υλοποίηση και αξιολόγηση μιας γεννήτριας πραγματικά τυχαίων αριθμών (TRNG, True Random Number Generator, όπως συνήθως αποκαλείται) μέσα σε ένα FPGA. Επιπρόσθετα, περιλαμβάνει την ανάπτυξη ενός συστήματος αξιολόγησης της γεννήτριας ελέγχοντας την απόκλισής της από την ιδανική τυχαιότητα. Στο τέλος, παρουσιάζονται τα αποτελέσματα της ανάλυσης, και εκφράζονται περαιτέρω ιδέες για πιθανή μελλοντική βελτίωση του συστήματος και του χαρακτηρισμού του.

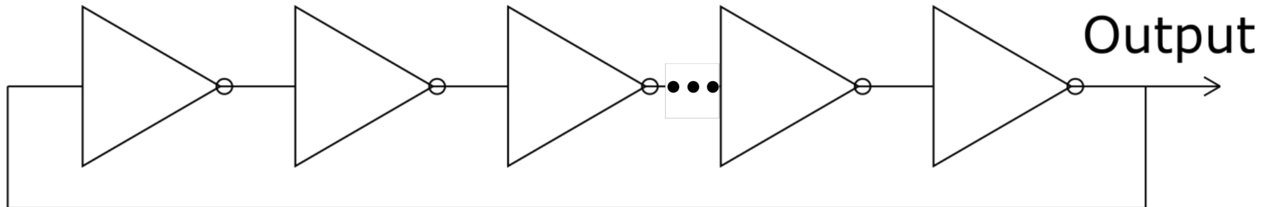
Η παραγωγή τυχαίων αριθμών έχει μεγάλο ενδιαφέρον στα πλαίσια αρκετών καθημερινών διαδικασιών. Κάθε συναλλαγή που συμβαίνει μέσα σε ένα ψηφιακό δίκτυο ή γενικότερα μεσω ψηφιακής διασύνδεσης των συναλλασσομένων, βασίζεται εκτενώς σε διάφορες κρυπτογραφικές υποδομές. Είτε πρόκειται περί μηνύματος SMS, σύνδεσης δύο συσκευών μέσω Bluetooth, σύνδεση στο διαδίκτυο και εξακρίβωση της αυθεντικότητας μιας ιστοσελίδας που προσπελάζεται, είτε για τραπεζική συναλλαγή, η κρυπτογραφία αποτελεί σημαντικό μέρος τους, για λόγους όπως προφύλαξη από κακόβουλη δράση/απάτη. Το κρυπτογραφικό λογισμικό, που λειτουργεί συνεχώς στο παρασκήνιο ώστε να διασφαλίσει την ακεραιότητα των συναλλαγών αυτών, πάντα βασίζεται σε μια πηγή πραγματικών τυχαίων αριθμών, ή αλλιώς μιας πηγής εντροπίας [10].

Μια καλή πηγή εντροπίας πρέπει να παρέχει αρκετούς πραγματικά τυχαίους αριθμούς για τις ανάγκες των εφαρμογών που εξυπηρετεί, ώστε να μην επέλθει εξάντληση της. Επίσης επιθυμητό είναι να είναι σχετικά αναίσθητη σε ένα μεγάλο εύρος περιβαλλοντικών συνθηκών ώστε να μπορεί να χρησιμοποιηθεί σε ποικίλα περιβάλλοντα και εφαρμογές.

Πηγές εντροπίας που βασίζονται σε κλασσικό θερμικό θόρυβο εντός ενός κυκλώματος, επί παραδείγματι, θεωρούνται αρκετά τυχαίες όταν λειτουργούν σωστά, είναι όμως ανάγκη η λεπτή ρύθμιση των παραμέτρων τους και ο συχνός έλεγχος της ορθής τους λειτουργίας, και είναι επιπλέον ευαίσθητες σε περιβαλλοντικές συνθήκες όπως η θερμοκρασία.

Στόχος λοιπόν της εργασίας είναι η δημιουργία ενός συστήματος που παραμένει ευσταθες χωρίς συνεχή έλεγχο, και είναι εύκολα επεκτάσιμο ώστε να παράσχει αυθαίρετα μεγάλους ρυθμούς παραγωγής τυχαίων αριθμών. Ιδανικό στην προκειμένη περίπτωση στοιχείο για την εκπλήρωση αυτού του στόχου φάνηκε ένα κύκλωμα που καλείται δακτυλιοειδής ταλαντωτής [11].

Ο δακτυλιοειδής ταλαντωτής είναι μια συστοιχία περιττού αριθμό λογικών αντιστροφέων (inverters, ήγουν πύλες NOT), συνδεδεμένων σε σειρά και σε τοπολογία βρόχου (με ανάδραση), δηλαδή με την τελευταία έξοδο να επιστρέφει στην πρώτη είσοδο (εικόνα 1).



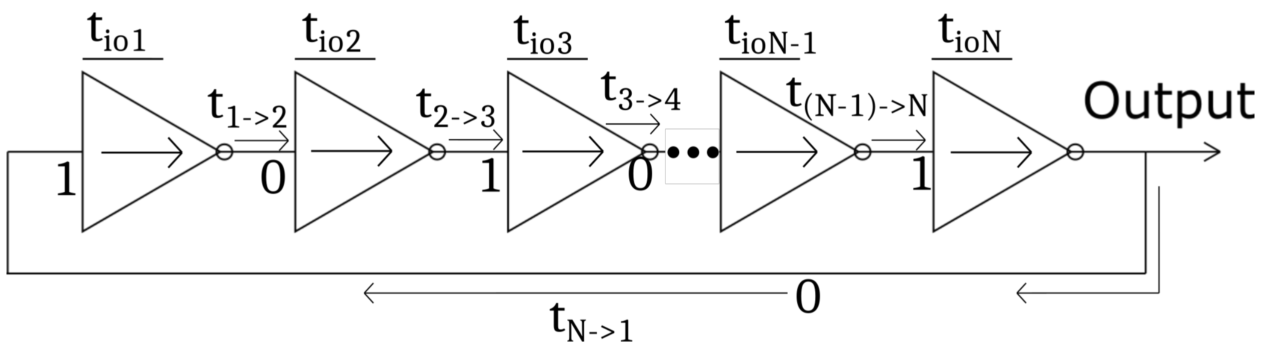
Διάγραμμα 1: Ενας δακτυλιοειδής ταλαντωτής

Το απεικονιζόμενο κύκλωμα είναι λογικά ασταθές: η συνθήκη ταλάντωσης, που είναι ο αριθμός αντιστροφών να είναι περιττός παράγει λογική αστάθεια στην είσοδο του πρώτου αντιστροφέα. Ισχύει

πρώτη είσοδος = τελευταία έξοδος
(λόγω της τοπολογίας βρόχου) αλλά και

τελευταία έξοδος = αντίθετο πρώτης εισόδου
(λόγω του περιττού αριθμού αντιστροφών). Αυτό οδηγεί στη σχέση
πρώτη είσοδος = αντίθετο πρώτης εισόδου.

Όταν η λογική αυτή υλοποιηθεί σε ένα πραγματικό κύκλωμα, υπεισέρχονται και πραγματικές, μη μηδενικές χρονικές καθυστερήσεις ανάμεσα στους αντιστροφείς. Αυτό οδηγεί το κύκλωμα σε κατάσταση ταλάντωσης οπου όταν η είσοδος διατρέξει τον βροχο 2 φορές, η ακολουθία εξόδου επαναλαμβάνεται, δηλαδή δημιουργείται ταλάντωση. Η παραπάνω διατύπωση, τόσο της λογικής αστάθειας όσο και της ταλαντωτικής συμπεριφοράς, απεικονίζεται στο διάγραμμα 2.



Διάγραμμα 2: χρονική ανάλυση του ταλαντωτή και απεικόνιση της λογικής αστάθειας

Στο διάγραμμα αυτό, έχει υποτεθεί ότι ο δακτυλιοειδής ταλαντωτής περιέχει έναν αντιστροφέα με την είσοδο του ίση με την έξοδο (ο πρώτος). λόγω της ανατροφοδότησης της εξόδου πίσω στην πρώτη είσοδο της αλυσίδας. Η έξοδος του κάθε ταλαντωτή είναι το λογικά αντίστροφο της εισόδου του, οπότε η αλυσίδα από μόνη της (χωρίς την ανατροφοδότηση) θα έμενε σε αυτήν την κατάσταση για άπειρο χρόνο. Η λογική αστάθεια προέρχεται απ' την τοπολογία του βρόχου και τον περιττό αριθμό των αντιστροφών: η έξοδος του ταλαντωτή (0), ανατροφοδοτείται στην πρώτη είσοδο της αλυσίδας (1). Αυτό όμως σημαίνει πως πλέον η πρώτη είσοδος είναι 0, η οποία θα παράξει μια έξοδο 1, κ.ο.κ.

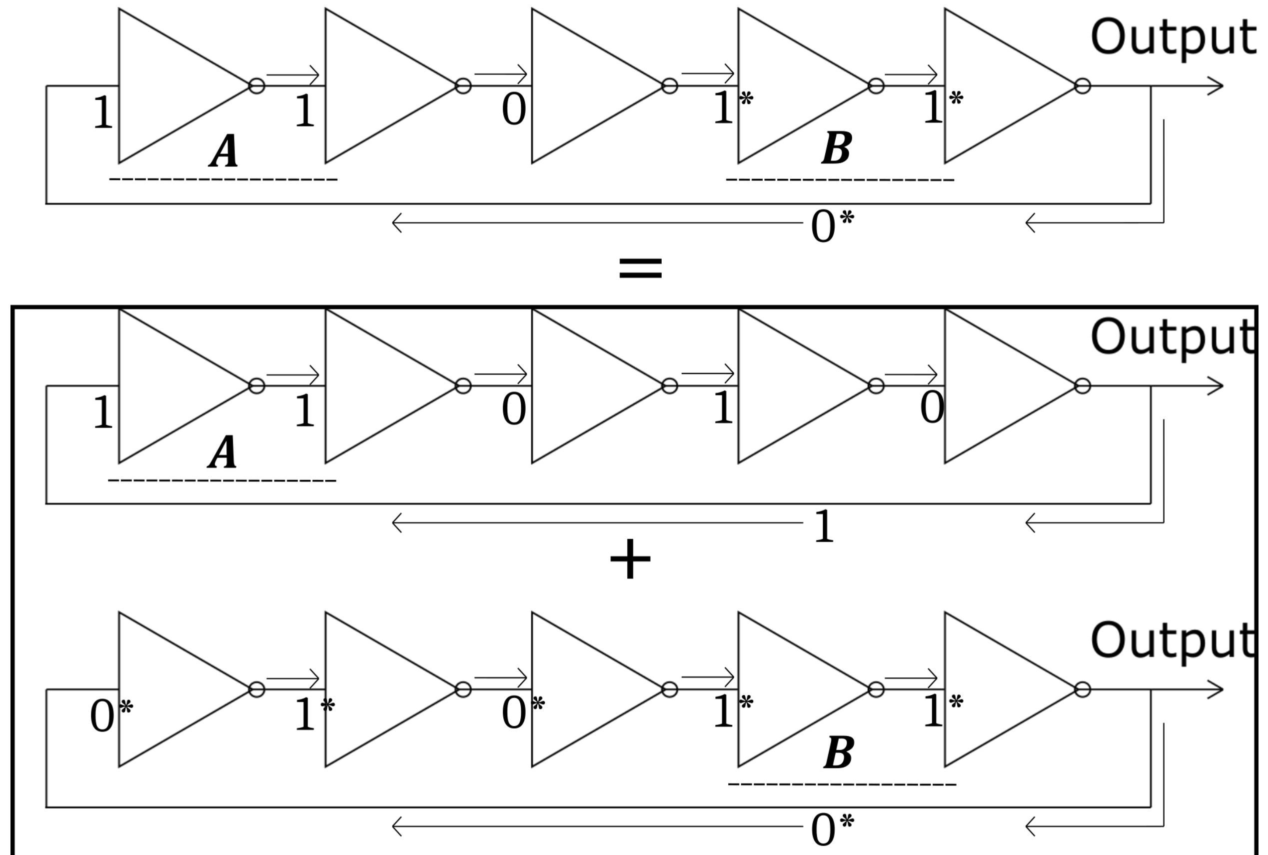
Η γειτνίαση ιδίων τιμών (εκατέρωθεν ενός αντιστροφέα), προκαλεί τη διάδοση μιας διαταραχής μέσα στον ταλαντωτή, καθώς ο αντιστροφέας δρα προς κατάργηση της ισότητας μεταξύ του σήματος εξόδου με το σήμα εισόδου, και επιφέρει την σχέση αντιστροφής αλλάζοντας την εξόδο απ' το ίσον της εισόδου, στο αντίθετο. Αν υποτεθεί η ύπαρξη μιας μόνο NOT εκτός ισορροπίας (είσοδος ίση με έξοδο), πριν τη δράση του αντιστροφέα, η έξοδος αυτή βρίσκεται σε σχέση αντιστροφής με την έξοδο της επόμενης NOT (ισορροπία). Όταν ο αντιστροφέας δράσει, επέρχεται ισότητα μεταξύ εισόδου και εξόδου του επόμενου αντιστροφέα, και ούτω καθ' εξής, ώστε όταν η προηγούμενη NOT ισορροπεί, η επόμενη βγαίνει εκτός ισορροπίας..

Οι χρόνοι t_{io} είναι οι χρόνοι απόκρισης του κάθε αντιστροφέα, οι $t_{a->b}$ είναι οι χρόνοι διάδοσης των σημάτων από τον αντιστροφέα a στον αντιστροφέα b. Ο χρόνος $t_{switch} = t_{io1} + t_{1->2} + t_{io2} + t_{2->3} + ... + t_{ioN} + t_{N->1}$ είναι ο χρόνος εναλλαγής του σήματος της εξόδου (από 0 σε 1 και από 1 σε 0) και $t_{oscillation} = 2t_{switch}$ η περίοδος ταλάντωσης του ταλαντωτή (2 εναλλαγές). Σημειώνεται πως αυτό αποτελεί μια πρώτης τάξης ανάλυση η οποία δεν λαμβάνει υπόψιν ενδεχόμενες αποκλίσεις στους αυτούς χρόνους, που δύνανται να εμφανιστούν από τον έναν χρόνο εναλλαγής στον άλλον.

Στην ως άνω χρονική ανάλυση υπετέθη πως στον ταλαντωτή υπάρχει μόνο ένας αντιστροφέας εκτός ισορροπίας ανα δεδομένη χρονική στιγμή. Όταν όμως το κύκλωμα αυτό έρθει σε λειτουργία χωρίς αρχικοποίηση των σημάτων του δακτυλίου, περιέρχεται σε μια άγνωστη κατάσταση, οπού ο αριθμός των αντιστροφέων εκτός ισορροπίαςείναι άγνωστος.

Εν γένει δεν υπάρχει εγγύηση ύπαρξης μόνο NOT με ίση είσοδο με έξοδο. Μπορεί να υπάρχουν πολλές τέτοιες NOT, οδηγώντας στη δημιουργία μιας πολλαπλής ταλάντωσης. Το διάγραμμα 3 παρουσιάζει ένα παράδειγμα ενός ταλαντωτή που λειτουργεί με μια σύνθετη κατάσταση με 2 ζεύγη ίσων γειτονικών σημάτων, αναγραφόμενα ως A και B. Η πολλαπλή ταλαντώση που εκτελεί ο απεικονιζόμενος δακτύλιος μπορεί να κατανοηθεί ως 2 επί το πλείστον ανεξάρτητες βασικές ταλαντώσεις που διατρέχουν το ίδιο κύκλωμα. Τα αντίστοιχα σήματα των

2 βασικών καταστάσεων αναγράφονται με αστερίσκο (*) για τη Β και χωρίς για την Α. Η απεικόνιση μιας κατάστασης που οδηγεί σε πολλαπλή ταλάντωση φαίνεται στο πάνω μέρος του διαγράμματος 3, και η ανάλυσή του σε βασικές ταλαντώσεις στο κάτω.

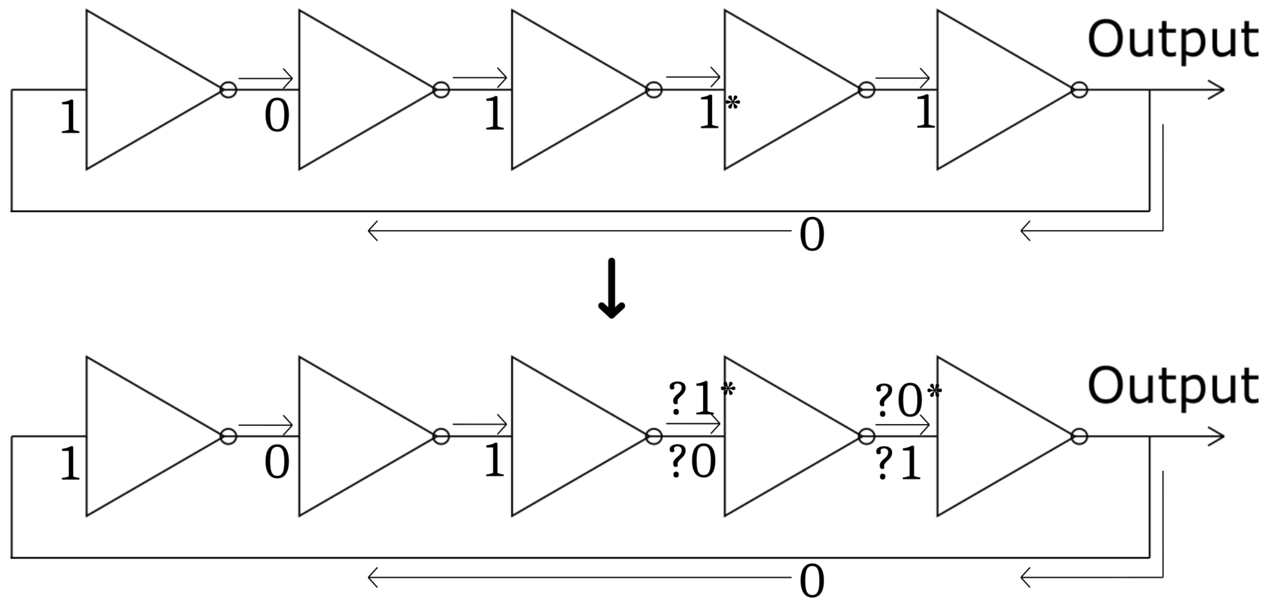


Διάγραμμα 3: ταλαντωτής σε κατάσταση πολλαπλής ταλάντωσης

Η ύπαρξη πολλαπλών ταλαντώσεων είναι ανεπιθύμητη στη συστοιχία των δακτυλίων, τόσο στα πρωταρχικά πειράματα, όσο και στο τελικό σύστημα. Αφενός μεν, παράγουν πολύ περίπλοκες ταλαντώσεις και είναι σχεδόν αδύνατη η καταγραφή βασικών παραμέτρων στους ταλαντωτές, όπως η χαρακτηριστική τους συχνότητα. Αφετέρου δε, οι πολλαπλές ταλαντώσεις μπορεί να εξαλειφθούν σε βάθος χρόνου, διότι η ανεξαρτησία μεταξύ τους δεν ισχύει καθολικά.

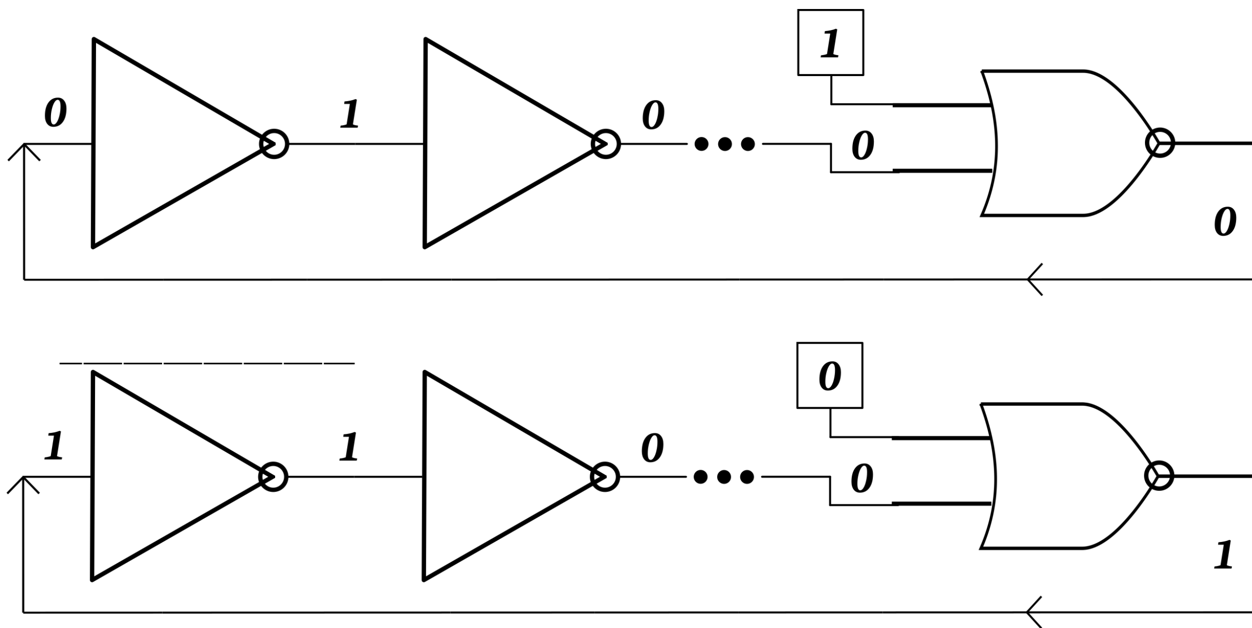
Η εξάλειψη των πολλαπλών ταλαντώσεων οφείλεται σε συνθήκες χρονικού συναγωνισμού επί των σημάτων των ταλαντωτών, ανάμεσα στις διαδιδόμενες διαταραχές. Συνθήκη συναγωνισμού σε ένα σύστημα υπάρχει όταν η κατάσταση τελική ενός συστήματος οφείλεται στην χρονική ακολουθία ή χρονισμό μεταξύ ανεξέλεγκτων γεγονότων [12]. Στην προκειμένη περίπτωση των σύνθετων καταστάσεων, η δράση ανάμεσα στους αντιστροφείς που φέρουν τις διαταραχές

του ταλαντωτή μπορεί να επιφέρει διαφορετικές εξελίξεις στην κατάστασή του, ανάλογα με τη χρονική σειρά δράσης τους (η οποία δεν είναι απολύτως ντετερμινιστικη). Το διάγραμμα 4 δείχνει μια τέτοια συνθήκη συναγωνισμού να απειλεί την ύπαρξη μιας σύνθετης κατάστασης. Το ’1*’ αποτελεί σήμα υπό τον έλεγχο (σήμα εξόδου) του 3ου αντιστροφέα, και ταυτόχρονα σήμα εισόδου του τετάρτου. Οι δύο αντιστροφείς συναγωνίζονται πάνω στη χρονική σειρά προσπέλασης σε αυτό το σήμα, με δυο πιθανές τελικές καταστάσεις να απεικονίζονται στο κάτω μέρος του διαγράμματος (με * και χωρίς).



Διάγραμμα 4: συνθήκη συναγωνισμού σε σύνθετη κατάσταση

Στον αρχικό σχεδιασμό των ταλαντωτών, προστέθηκε μια λειτουργία απενεργοποίησης των ταλαντώσεων, σε πρώτη φάση με στόχο τον ελέγχο της ενεργειακής κατανάλωσης του συστήματος. Αυτό γίνεται αίροντας τη συνθήκη ταλάντωσης στον δακτύλιο, αντικαθιστώντας τον τελευταίο αντιστροφέα στην αλυσίδα με μια NOR αντί για NOT πύλη, την NOR αποκοπής ταλάντωσης. Θέτοντας τη μία της είσοδο σε ’1’. όπως φαίνεται στο άνω μέρος του διαγράμματος 5. Έτσι, παρουσία του σήματος ελέγχου ’1’, διακόπτεται η ανατροφοδότηση της εξόδου προς την είσοδο, και η αλυσίδα των αντιστροφέων εξαναγκάζεται σε ισορροπία (αυστηρώς εναλλασσόμενες είσοδοι-έξοδοι). Με αυτόν τον τρόπο καταστρέφονται και οι καταστάσεις που οδηγούν σε σύνθετη ταλάντωση, οι οποίες εξαφανίζονται όταν οι δακτύλιοι απενεργοποιούνται, αφήνοντας την αλυσίδα των NOT να έρθει σε κατάσταση με αυστηρώς εναλλασσόμενα σήματα μεταξύ των αντιστροφέων. Όταν οι δακτύλιοι επανενεργοποιούνται, με το σήμα ελέγχου να είναι ’0’ (κάτω μέρος του διαγράμματος 5), υπάρχει μόνο ένας αντιστροφέας με ίση είσοδο και έξοδο. Αυτός τονίζεται με τη διακεκκομένη γραμμή στο κάτω μέρος του διαγράμματος 5.



Διάγραμμα 5: λειτουργία ενεργοποίησης/απενεργοποίησης των δακτυλίων με την NOR αποκοπής ταλάντωσης

Οι δακτυλιοειδείς ταλαντωτές επιλέχθηκαν σαν κύριο στοιχείο εντροπίας, διότι η περίοδος ταλάντωσης τους εξαρτάται σε μεγάλο βαθμό από χαώδεις, μη προβλέψιμες παραμέτρους όπως:

- Από το υπόστρωμα στο οποίο έχει υλοποιηθεί ο ταλαντωτής (εξάρτηση από silicon Process - P), δηλαδή από την ταχύτητα των αντιστροφέων και των διασυνδέσεων μεταξύ τους, όπως προκύπτει από τα φυσικά χαρακτηστικά του υποστρώματος του πυριτίου πάνω στο οποίο είναι υλοποιημένοι.
- Μικρομεταβολές στην τοπική τάση τροφοδοσίας των αντιστροφέων ή γενική τάση τροφοδοσίας του ταλαντωτή (εξάρτηση από V - Voltage), που μπορεί να προέρχεται ακόμα και επαγωγικά φαινόμενα από το ίδιο το περιβάλλον του ταλαντωτή, όπως ενδεχομένως από γειτονικούς ταλαντωτές.
- Θερμοκρασιακές μεταβολές (εξάρτηση από T - Temperature), που επηρεάζουν ελαφρώς την ταχύτητα απόκρισης των στοιχείων των ταλαντωτών.

Εν ολίγοις, η περίοδος ταλάντωσης των δακτυλιοειδών ταλαντωτών εμφανίζει τυχαία PVT εξάρτηση, άρα παρέχει πρόσβαση στην εντροπία που μπορούν να προσκομίσουν σέ ένα σύστημα οι βασικές αυτές φυσικές μεταβλητές.

Η υλοποίηση των κυκλωμάτων έγινε στο προγραμματιζόμενο ολοκληρωμένο κύκλωμα ZCU9EG-2FFVB1156E FPGA της Xilinx, προσαρτημένο πάνω στην αναπτυξιακή πλατφόρμα ZCU-102. Ένα FPGA (Field Programmable Gate Ar-

ray), είναι ένα ολοκληρωμένο κύκλωμα που περιέχει ένα πλέγμα στοιχείων γενικής κυκλωματικής λογικής και προγραματιζόμενες διασυνδέσεις ανάμεσά στα στοιχεία αυτά [13, 20]. Μπορεί να αναδιαμορφώσει τις εσωτερικές του διασυνδέσεις με χρήση ενός αρχείου διαμόρφωσης (το καλούμενο bitstream). Δύναται κατ' αυτόν τον τρόπο να υλοποιήσει οποιοδήποτε λογικό κύκλωμα εντός των ορίων των διαθέσιμων ελεύθερων πόρων στο chip. Το αρχείο διαμόρφωσης συνήθως παράγεται μέσω χρήσης μιας γλώσσας περιγραφής υλικού.

Η γλώσσα που χρησιμοποιήθηκε για το σχεδιασμό του συστήματος λέγεται VHDL (Very high speed integrated circuit Hardware Description Language) [14, 15], ερμηνευόμενη με χρήση μιας σουίτας ολοκληρωμένης ανάπτυξης και σχεδιασμου για προϊόντα της Xilinx, το Vivado [16].

Στο FPGA, όλων των ειδών οι λογικές πύλες (όπως η NOT) υλοποιούνται σε ένα βασικό στοιχείο που αποκαλείται LookUp Table (LUT - πίνακας παραμ-πομπής). Αυτό είναι ένα είδος μνήμης που ανασύρει στις εξόδους της τα προϋπολογισμένα αποτελέσματα διαφόρων συνδυασμών των εισόδων του. Με αυτόν τον τρόπο μπορεί να υλοποιήσει πολλές διαφορετικές λογικές πύλες.

Με χρήση αυτών των εργαλείων δημιουργήθηκαν διάφοροι δακτυλιοειδείς ταλαντωτές στο FPGA και έγιναν κάποια πρώιμα πειράματα με σκοπό τη μέτρηση χαρακτηριστικών ποσοτήτων τους και την απόδειξη της εξάρτησής τους από PVT.

Σε πρώτη φάση, απεδείχθη η PVT εξάρτηση χάριν στην εξής σειρά πειραμάτων:

- Η εξάρτηση απ' την τάση απεδείχθη υλοποιώντας ένα δίκτυο θερμαντικών στοιχείων μέσα στο FPGA, το οποίο είναι και βαρύ φορτίο στο τροφοδοτικό της πλακέτας

Πιο συγκεκριμένα, υλοποιήθηκε ένα δίκτυο από συστοιχίες FLIP-FLOP FIFO στο FPGA, καθώς και μια στοιχειώδης διεργασία τροφοδοσίας των FIFO με μια περιοδικά εναλασσόμενη είσοδο (101010...). Η μεταβολή των καταστάσεων των FLIP-FLOP αποτελεί μια ηλεκτρική εναλλαγή που καταναλώνει μεγάλες ποσότητες ενέργειας και επίσης αυξάνει την θερμοκρασία του πυριτίου του FPGA.

Όταν ενεργοποιηθούν τα θερμαντικά στοιχεία βαθμίδα τροφοδοσίας αντιμετωπίζει έναν αυξημένο φόρτο, οπότε εμφανίζεται μια μικρή ελάττωση τάσης στην τροφοδοσία του πυριτίου (από τα ωμικά φαινόμενα που προκαλεί το αυξημένο ρεύμα τροφοδοσίας, για παράδειγμα). Ακριβώς τη στιγμή της ενεργοπ-οίησης των FIFO, πριν δηλαδή αλλάξει η θερμοκρασία στο FPGA, παρατηρήθηκε

πτώση της συχνότητας των ταλαντωτών.

- Η εξάρτηση απ' την θερμοκρασία απεδείχθη παρατηρώντας τη μεταβολή συχνότητας ταλάντωσης των ταλαντωτών κατά τη διάρκεια ενός κύκλου ψύξης του FPGA.

Όταν οι FIFOs δεν λειτουργούν η θερμοκρασία πέφτει. Επίσης περιορίζεται η μεταβολή στην τάση τροφοδοσίας, συνεπώς η μόνη μεταβλητή παράμετρος είναι πτώση της θερμοκρασίας του FPGA. Κατά την πτώση της θερμοκρασίας του FPGA παρατηρήθηκε άνοδος στη συχνότητα ταλαντώσεων.

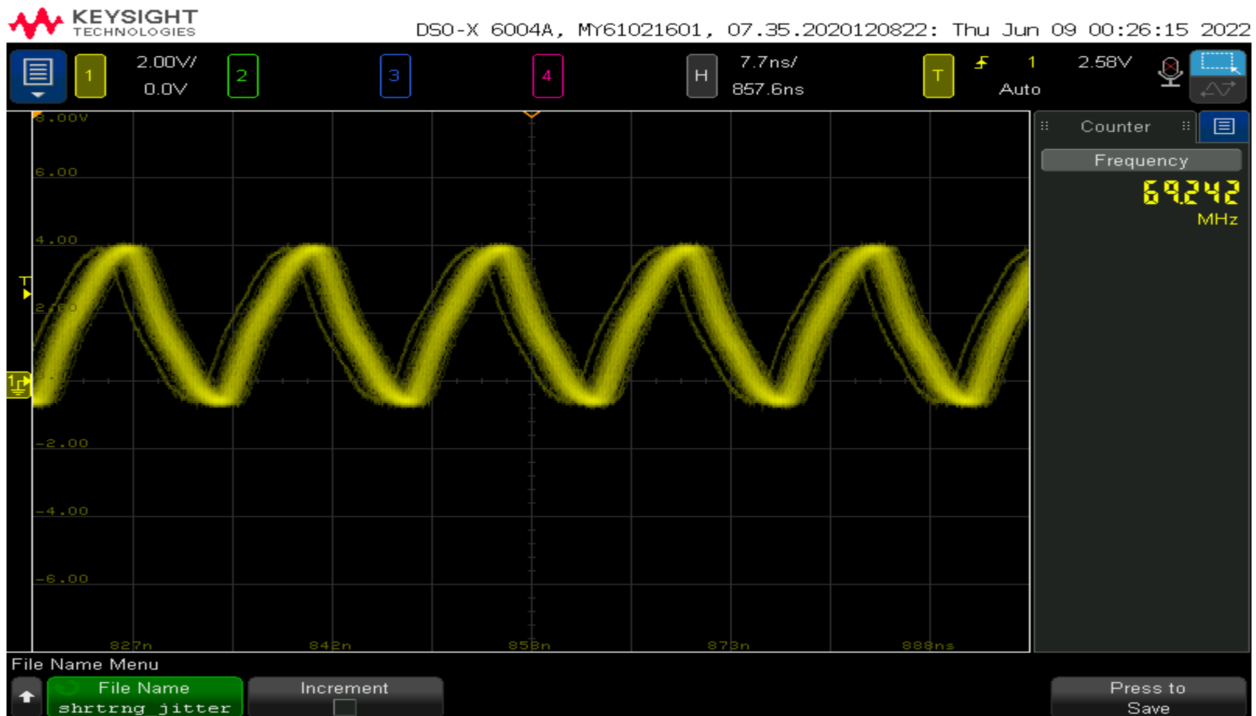
- Τέλος οι δακτύλιοι τοποθετήθηκαν σε διαφορετικές θέσεις στο υπόστρωμα του FPGA και ελέγχθηκε η εξάρτηση της συχνότητάς τους από το υπόστρωμα.

Με την χρήση εντολών σε γλώσσα Tool Command Language (TCL) [17, 18, 19] στο περιβάλλον του Vivado, οι πύλες NOT και οι διασυνδέσεις μεταξύ τους κατασκευάστηκαν με απολύτως συμμετρικό τρόπο. Με την εισαγωγή αυτής της συμμετρίας στην υλοποίηση των δακτυλίων, όλες οι άλλες ποσότητες που επηρεάζουν το χρονισμό τους παραμένουν αμετάβλητες (όπως το μήκος των εκλεγμένων διασυνδέσεων). Η παρατηρούμενη διακύμανση στη χαρακτηριστική συχνότητα μεταξύ δακτυλίων είναι αυτό που δείχνει την εξάρτηση απ' το υπόστρωμα του πυριτίου. Τέλος, ορίστηκε η παράμετρος $<F_{PI}>$ = Αριθμός Αντιστροφέων x Συχνότητα Δακτυλίου. Για το FPGA αυτό και αυτήν την συγκεκριμένη στρατηγική διασύνδεσης μια προσέγγισή της παραμέτρου είναι $<F_{PI}>$ = 2000MHz*#Inverters.

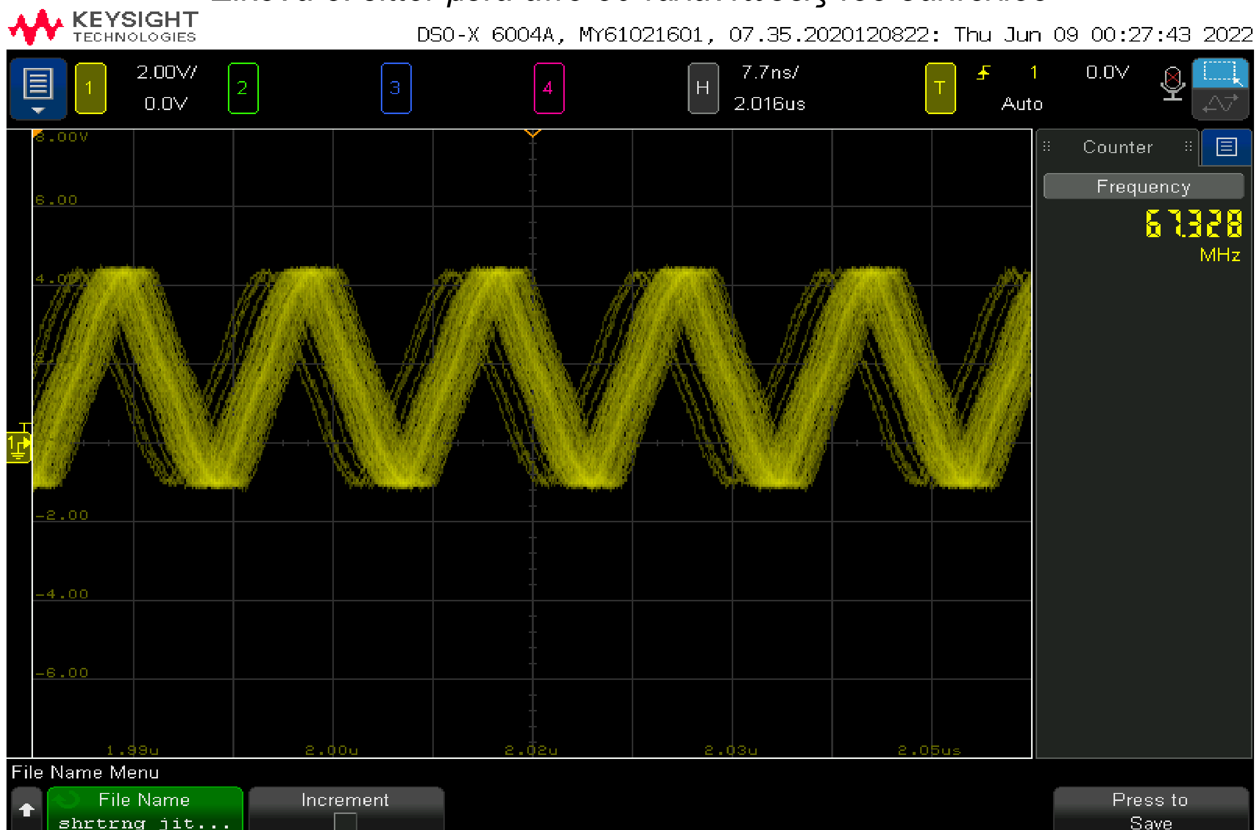
Παρά την ομοιότητα στο σχεδιασμό και υλοποίηση των δακτυλίων και παρά την απόλυτη συμμετρία από δακτύλιο σε δακτύλιο των ηλεκτρικών διασυνδέσεων μεταξύ των αντιστροφέων που τους απαρτίζουν, η απόκλιση από την ταυτόσημη συμπεριφορά των ταλαντωτών είναι γεγονός, και αυτό προκύπτει τόσο από τις διαφορές στην κεντρική συχνότητά ταλάντωσής τους, όσο και από τις μικροδιακυμάνσεις της περιόδου ταλάντωσης τους (Jitter). Στο συγκεκριμένο κύκλωμα το Jitter είναι η πηγή της τυχαιότητας προς αξιοποίηση, καθώς προσδίδει έναν τυχαίο όρο σε κάθε περίοδο ταλάντωσης που είναι παρόμοιος με μια διεργασία Wiener [27], ή πιο συγκεκριμένα έναν γκαουσιανό τυχαίο περίπατο [28], καθώς το μη ντετερμινιστικό jitter ακολουθεί γκαουσιανές κατανομές [29].

Το σύστημα παραγωγής τυχαίων αριθμών λαμβάνει περιοδικά δείγματα με τη χρήση ενός ρολογιού στην αναπτυξιακή κάρτα, με περίοδο μεγαλύτερη της χαρακτηριστικής περιόδου των ταλαντωτών (κύκλωμα υποδειγματοληψίας). Το Jitter σε συνδυασμό με την μεγάλη περίοδο δειγματοληψίας εξασφαλίζουν την τυχαιότητα των δειγμάτων. Στις εικόνες 5 και 6 φαίνεται το η διακύμανση της περιόδου της ταλάντωσης (Jitter) μετά από 50 και 70 περιόδους των δακτυλίων.

Το πάχος του ίχνους των κυματομορφών εκφράζει το μέγεθος του Jitter.



Εικόνα 5: Jitter μετά από 50 ταλαντώσεις του δακτυλίου



Εικόνα 6: Jitter θόρυβος μετά από 70 ταλαντώσεις του δακτυλίου

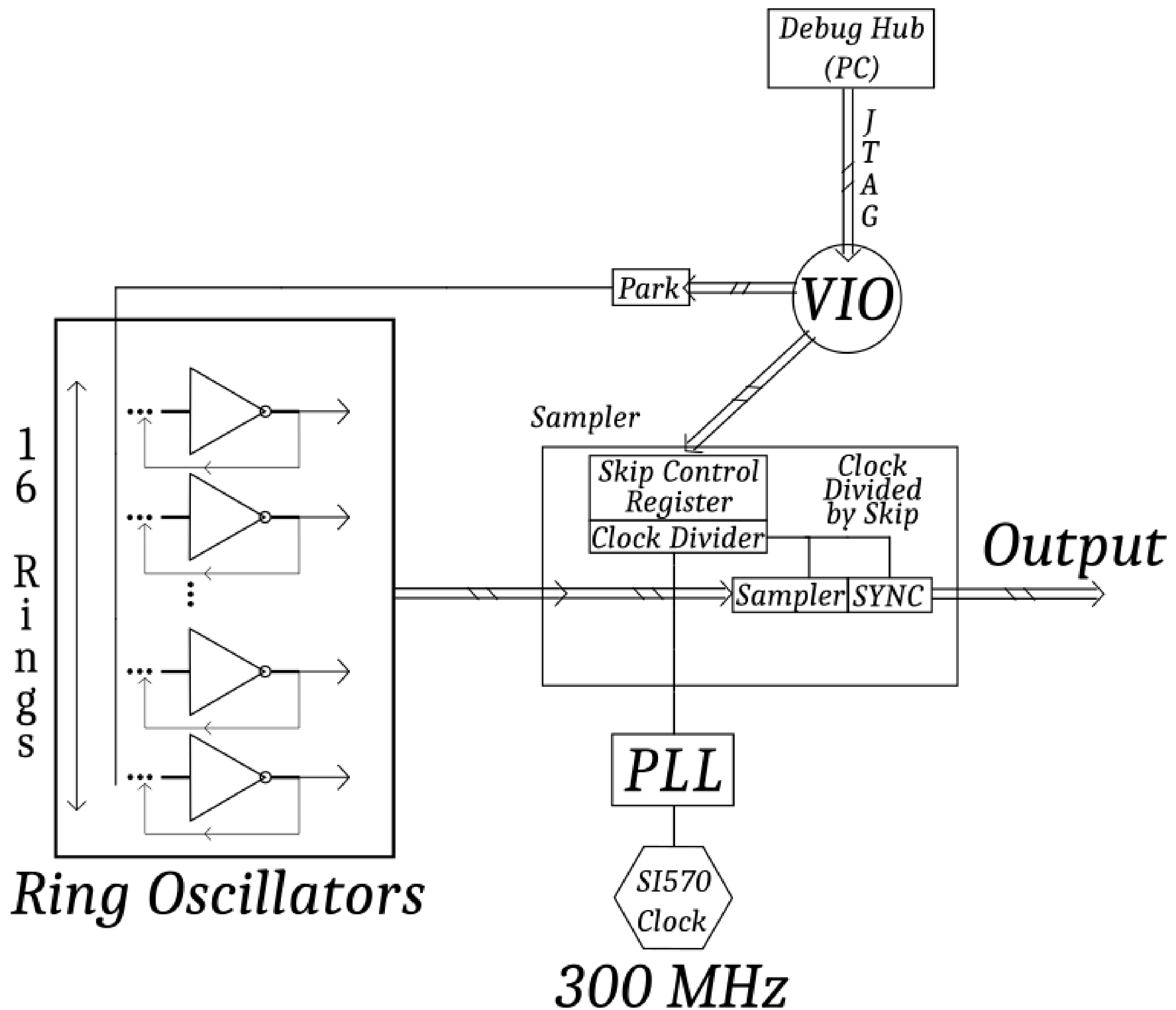
Είναι προφανές απ' τις εικόνες ότι όσο περισσότερες περίοδοι ταλάντωσης των ταλαντωτών παρέλθουν, τόσο πιο πολύ παχαίνει η κυματομοφρή, άρα και η το Jitter. Η απόκλιση δηλαδη απ' την περιοδικότητα αυξάνει συν τω χρόνω.

Η γεννήτρια τυχαίων αριθμών αποτελείται από 16 δακτυλίους, ώστε να παράγει 16-bit (μια λέξη) ανα δειγματοληψία. Ο κάθε δακτύλιος δηλαδή θα παράγει 1 bit ανα δειγματοληψία. Το κύκλωμα δειγματοληψίας σχεδιάστηκε ώστε να έχει ελεγχόμενο διάστημα (μεταβλητός διαιρέτης ρολογιού).

Μετά απ' το κύκλωμα δειγματοληψίας τοποθετήθηκε μια FIFO συγχρονισμού, με σκοπό της αφαίρεσης πιθανής μεταστάθειας των FLIP-FLOP ως αποτέλεσμα της ασυγχρόνης με τους δακτυλίους λειτουργίας του [21, 22, 23, 24]. Αυτό έχει σκοπό να θωρακίσει τα κυκλώματα που θα λάβουν τους τυχαίους μετά την έξοδο του TRNG από μετασταθή συμπεριφορά. Οι παραγόμενοι τυχαίοι αριθμοί περάστηκαν από ένα κύκλωμα αξιολόγησης είτε απευθείας μετά τη δειγματοληψία (ακατέργαστη έξοδος των δακτυλίων), είτε κατηργασμένοι από ένα κύκλωμα μετεπεξεργασίας (Post Processor). Η μετεπεξεργασία αποσκοπεί στην μείωση της στατιστικής προτίμησης υπερ του '1' ή του '0' που ενδεχομένως εμφανίζεται στην έξοδο των δακτυλίων.

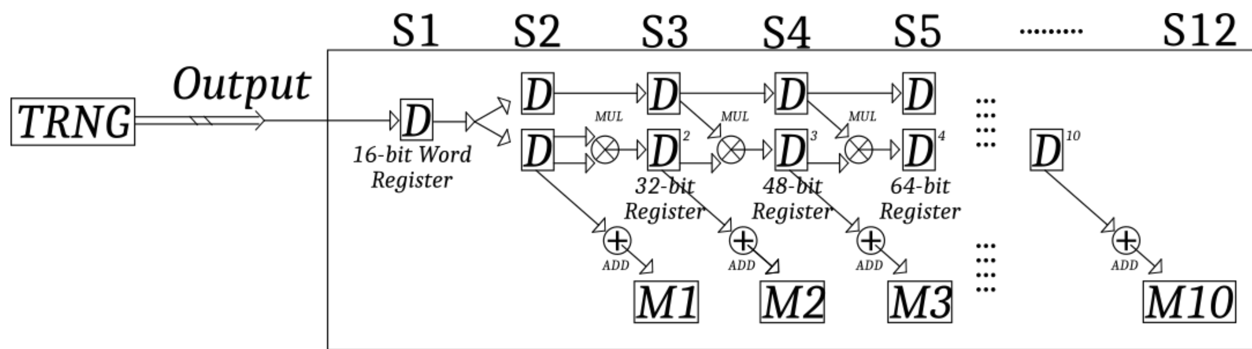
Το κύκλωμα αξιολόγησης υπολογίζει τις 10 πρωτογενείς αλγεβρικές ροπές [3, 25] της κατανομής των αριθμών από τη δειγματοληψία (ροπές γύρω απ' το 0), των οποίων η σύγκριση με τις θεωρητικά αναμενόμενες τιμές μιας ομοιόμορφης κατανομής είναι το πρώτο μέτρο για την ποιότητα των παραγόμενων τυχαίων. Στο ίδιο κύκλωμα προστέθηκε και ένας καταμετρητής του πλήθους των '1', τόσο επί του συνόλου των bit της λέξης, όσο και κατά κάθε ακολουθία bit που παράγεται απο κάθε ξεχωριστό δακτύλιο [1]. Τέλος μετρήθηκε η κατανομή του πλήθους των δειγμάτων που περέρχονται μεταξύ διαδοχικών εμφανίσεων του '1' σε κάθε δείγμα του κάθε δακτυλίου. Μία πραγματικά τυχαία ακολουθία δύο στοιχείων (εν προκειμένω '0' και '1') πρέπει να έχει τη μορφή της διωνυμικής κατανομής [6, 26], και η ως άνω μέτρηση αποτελεί την διωνυμική δοκιμή (binomial test [26]) μέσω της οποίας ελέγχεται η έξοδος της Γεννήτριας Παραγματικά Τυχαίων Αριθμών.

Στα διαγράμματα 7 και 8 φαίνονται συνοπτικά οι αρχιτεκτονικές των κυκλωμάτων της Γεννήτριας Πραγματικά Τυχαίων Αριθμών και αξιολόγησης.



Διάγραμμα 7: αρχιτεκτονική της Γεννήτριας Πραγματικά Τυχαίων Αριθμών

Στο διάγραμμα 7 φαίνεται η ομάδα των ταλαντωτών, που παράγουν τα τυχαία bit, τα οποία περνάνε απ το κύκλωμα δειγματοληψίας (sampler). Ο έλεγχος του λόγου διαίρεσης του πρότυπου ρολογιού SI570 επιλέγεται απ' το PC που επικοινωνεί με την αναπτυξιακή πλατφόρμα μέσω σύνδεσης JTAG. Με την τιμή του καταχωρητή (Skip Control), ελέγχεται ο λόγος διαίρεσης του ρολογιού που χρονίζει τόσο το κύκλωμα της δειγματοληψίας (sampler) όσο και το κύκλωμα συγχρονισμού (SYNC).



Διάγραμμα 8: συστοιχία υπολογισμού πρωτογενών ροπών

Στο διάγραμμα 8 φαίνεται η συστοιχία (Pipeline) που υπολογίζει τις 10 πρωτογενείς στατιστικές ροπές της ακολουθίας των τυχαίων αριθμών. Στην εικόνα το MUL συμβολίζει πολλαπλασιασμό και ADD πρόσθεση.

Η συστοιχία αυτή χρησιμοποιεί τους Ψηφιακούς Επεξεργαστές Σήματος DSP48E2 (Digital Signal Processors, DSPs) που βρίσκονται ενσωματωμένοι μέσα στο FPGA [9]. Οι DSPs χρησιμοποιούνται για να υλοποιήσουν τις προσθέσεις και τους πολλαπλασιασμούς που εμπλέκονται στους υπολογισμούς των ροπών.

Οι πρωτογενείς στατιστικές ροπές υπολογίστηκαν για δακτυλίους μηκών 3, 5, 11, 111, 1111 αντιστροφέων. Επίσης χρησιμοποιήθηκαν διάφοροι ρυθμοί δειγματοληψίας για να προσδιοριστούνε οι συνθήκες ικανοποιητικής τυχαιότητας και μη της εξόδου. Το μέγεθος του δείγματος είναι 2^{20} (1048560) αριθμοί των 16 bit.

Οι ροπές είναι πολύ ευσταθείς αριθμητικά σε όλη την περιοχή τιμών δειγματοληψίας. Εκτός απ’ τους πολύ μεγάλους ρυθμούς δειγματοληψίας σε μεγάλους δακτυλίους, εκεί που η υποδειγματοληψία μετατρέπεται σε υπερδειγματοληψία, οι στατιστικές ροπές παραμένουν σταθερές με μικρές μεταβολές γύρω από μια κεντρική τιμή. Τα αποτελέσματα παρουσιάζονται στον παρακάτω πίνακα, όπου Δακτύλιος_3N, Δακτύλιος_5N... είναι τα αποτελέσματα για δακτυλιοειδείς ταλαντωτές που αποτελούνται από 3, 5... αντιστροφείς. Φαίνεται ότι στην περίπλωση των μικρών δακτυλίων (3, 5) εμφανίζονται μεγάλες αποκλίσεις, ενώ οι μέτριοι προς μεγάλοι (11, 111, 1111 αντιστροφείς) είναι πολύ κοντά στο ιδανικό.

Η ευστάθεια των ροπών από δείγμα σε δείγμα αποδίδεται στο καλό ανακάτεμα των bit λόγω ανεξαρτησίας των δακτυλίων. Η συστηματική απόκλιση των ροπών από δείγμα σε δείγμα από την ιδανική τιμή ερμηνεύεται σε στατιστική προτίμηση υπέρ του ’1’ ή ’0’ στα bit της γεννήτριας τυχαίων (Bias). Ο πίνακας 1

παρουσιάζει τη μέση τιμή των πρωτογενών ροπών που προκύπτουν από δακτυλίους διαφόρων μηκών σε σύγκριση με τις θεωρητικά αναμενόμενες. Στις ποσοστιαίες αποκλίσεις τους, αρνητικό πρόσημο δηλώνει πως η ροπή είναι μικρότερη της θεωρητικής.

	Πρωτογενείς Ροπές (υπό μετασχηματισμό κλίμακας)					
Τάξη Ροπής	Δακτύλιος_3N	Δακτύλιος_5N	Δακτύλιος_11N	Δακτύλιος_111N	Δακτύλιος_1111N	Θεωρητική
1	3261	3343	3414	3436	3438	3436
2	1388	1441	1486	1501	1503	1501
3	6717	7013	7288	7381	7392	7378
4	3485	3648	3813	3870	3877	3868
5	1889	1979	2079	2114	2117	2112
6	1054	1105	1166	1188	1190	1187
7	6015	6303	6675	6815	6825	6805
8	3488	3654	3884	3971	3976	3964
9	2048	2145	2288	2343	2345	2338
10	1216	1272	1362	1396	1398	1393
	Απόκλιση από το Ιδανικό					
Τάξη Ροπής	Δακτύλιος_3N	Δακτύλιος_5N	Δακτύλιος_11N	Δακτύλιος_111N	Δακτύλιος_1111N	
1	-5.09%	-2.71%	-0.64%	0.00%	0.06%	
2	-7.53%	-4.00%	-1.00%	0.00%	0.13%	
3	-8.96%	-4.95%	-1.22%	0.04%	0.19%	
4	-9.90%	-5.69%	-1.42%	0.05%	0.23%	
5	-10.56%	-6.30%	-1.56%	0.09%	0.24%	
6	-11.20%	-6.91%	-1.77%	0.08%	0.25%	
7	-11.61%	-7.38%	-1.91%	0.15%	0.29%	
8	-12.01%	-7.82%	-2.02%	0.18%	0.30%	
9	-12.40%	-8.25%	-2.14%	0.21%	0.30%	
10	-12.71%	-8.69%	-2.23%	0.22%	0.36%	

Πίνακας 1: Πρωτογενείς Ροπές TRNG με διάφορα μήκη δακτυλίων

Τα δεδομένα των πειραμάτων για μέτρηση της προτίμησης υπερ μιας τιμής και απόκτηση των στατιστικών της απόστασης μεταξύ των ’1’, επιβεβαιώνουν αυτό ακριβώς. Οι μικροί δακτύλιοι πάσχουν από μεγάλη προτίμηση ενώ οι μεγαλύτεροι όχι τόσο.

Όπως φαίνεται στον πίνακα 2 η συστηματική προτίμηση υπέρ του ’0’ ή ’1’ παραμένει σταθερό, ανεξαρτήτως από το ρυθμό δειγματοληψίας ενός δακτυλίου με μήκος 5. Ο πίνακας είναι ενδεικτικός και αυτή η διαπίστωση ισχύει για όλα τα μήκη των δακτυλίων. Τιμές μικρότερες του 0 υποδηλώνουν προτίμηση υπέρ του ’0’ και μεγαλύτερες του 0 υποδηλώνουν προτίμηση υπέρ του ’1’.

<table>
<tr><th>Μήκος Δακτυλίου</th><th colspan="3">Ρυθμός Δειγματοληψίας (KHz)</th></tr>
<tr><td rowspan="3">3 Αντιστροφείς</td><td>15</td><td>97.5</td><td>2500</td></tr>
<tr><td colspan="3">Προτίμηση</td></tr>
<tr><td>-6.20%</td><td>-6.26%</td><td>-6.22%</td></tr>
</table>

Πίνακας 2: Προτίμηση για διάφορους ρυθμούς δειγματοληψίας σε δακτύλιο 3 NOT

Η κατανομή των ’1’ όμως είναι εξαιρετικά ευαίσθητη ρυθμό δειγματοληψίας. Ιδιαίτερα στους μεγάλους δακτυλίους, που έχουν μικρότερη χαρακτηριστική συχνότητα, μπορεί να παρατηρηθεί απόκλιση απ’ την διωνυμική κατανομή. Η ιδανική διωνυμική κατανομή έχει ίσες πιθανότητες του ’1’ με το ’0’, κάτι που συνεπάγεται καμία προτίμηση υπέρ μιας τιμής. Σε περίπτωση ύπαρξης προτίμησης, η έξοδος θα πρέπει να ακολουθεί την τροποποιημένη διωνυμική κατανομή, η οποία έχει άνισες πιθανότητες εμφάνισης ’1’ και ’0’.

Κατ’ αρχήν, όσο μεγαλύτερη είναι η προτίμηση υπέρ μίας τιμής, τόσο πιο πολύ η τροποποιημένη διωνυμική κατανομή αποκλίνει από την ιδανική. Όμως και η πειραματικές καμπύλες αποκλίνουν πολύ και απ’ την τροποποιημένη κατανομή όταν οι ρυθμοί δειγματοληψίας γινόνται παραπλήσιοι με τους ρυθμούς των δακτυλίων (συντονισμός).

Συνοπτικά, απόκλιση της τροποποιημένης απ’ την ιδανική διωνυμική καμπύλη σημάινει ύπαρξη προτίμησης υπέρ ’0’ ή ’1’. Απόκλιση της πειραματικής καμπύλης απ’ την τροποποιημένη σημαίνει αποτυχία συσσώρευσης φασικού θορύβου, και ως εκ τούτου περαιτέρω μείωση της τυχαιότητας. Σε δείγμα μεγέθους 2^{20} που εδώ χρησιμοποιήθηκε, απουσία προτίμισης, το ’1' εμφανίζεται $2^{20}/2$ φορές, δηλαδή 524288, το οποίο θεωρείται η ιδανική τιμή.

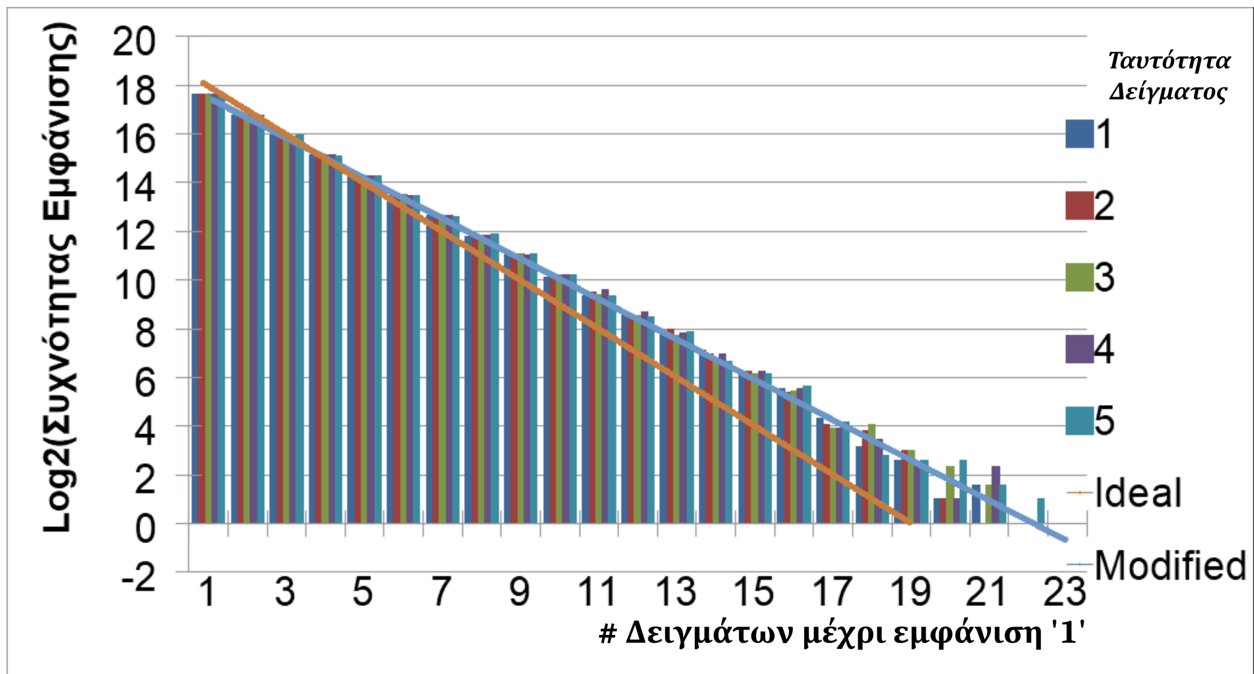
Οι πίνακες 3 ως 17 παρουσιάζουν τα αποτελέσματα της προτίμησης υπέρ του ’1’ για το πρώτο bit (πρωτος δακτύλιος) της κάθε υλοποιημένης ομάδας δακτυλίων, για διάφορους ρυθμούς δειγματοληψίας. Αρνητικές τιμές της προτίμησης δηλώνουν λιγότερα ’1’ από ’0’ (προτίμηση υπέρ του ’0’) και θετικές περιοσσότερα. Συνοδεύονται απ’ τα αντίστοιχα γραφήματα των πειραματικών και θεωρητικών καμπύλων (σε λογαριθμική κλίμακα) και ενδεχόμενα γραφήματα απόκλισης απ’ τις θεωρητικές διωνυμικές καμπύλες. Τα ονόματα των αξόνων παραλείπονται σε όλα τα γραφήματα ιδίου τύπου εκτός του πρώτου, για οικονομ-ία χώρου.

Τα γραφήματα των πειραματικών καμπύλων δείχνουν φαινόμενα αποδοτικής υποδειγματοληψίας ή συντονισμού (μη αποδοτική υποδειγματοληψία). Η παρ-άμετρος «Αριθμός Ταλαντώσεων Δακτυλίων» που αναγράφεται στους πίνακες, εκφράζει το πόσες ταλαντώσεις εκτελεί ο δακτύλιος ανάμεσα σε 2 διαδοχικές δείγματοληψίες. Προκύπτει απ’ το λόγο της συχνότητας δειγματοληψίας προς τη

συχνότητα του ταλαντωτή, με την τελευταία όπως προκύπτει απ' την προσεγγιστική σταθερά <F_{PI}> (δηλαδή Αριθμός Ταλαντώσεων Δακτυλίων) = Συχνότητα Δειγματοληψίας/Εκτιμώμενη Συχνότητα Δακτυλίου).

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
667	3072	6826	97.5	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
458628	524288	-6.26%	43.74%	56.26%

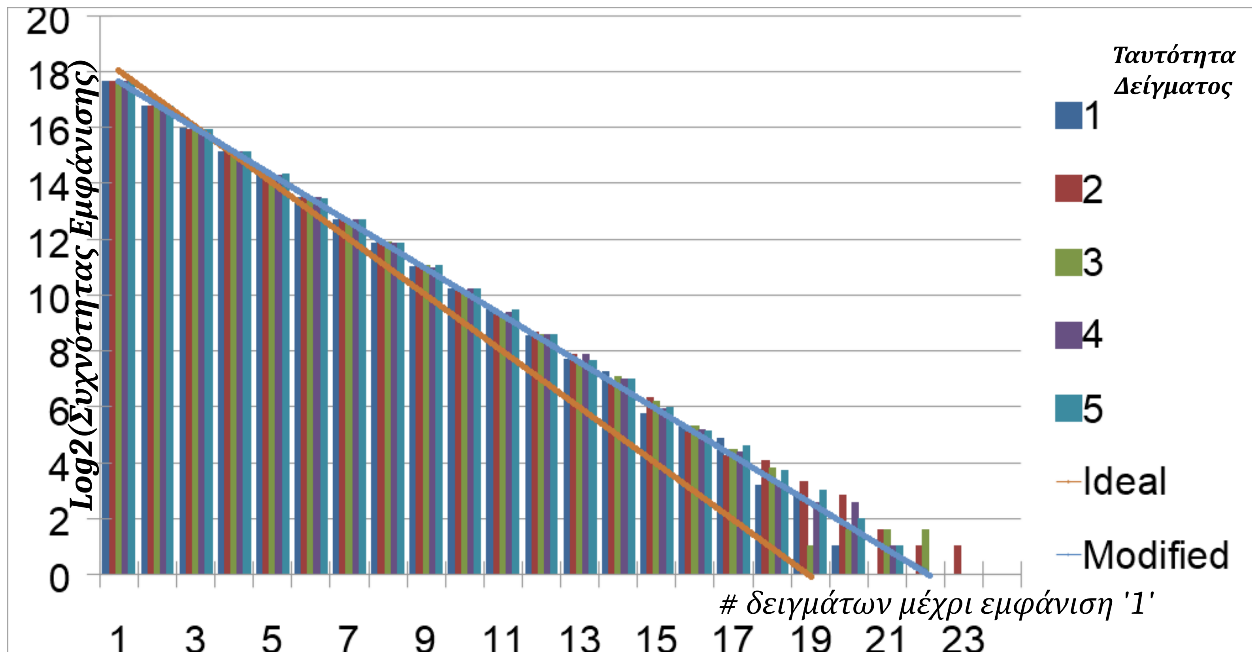
Πίνακας 3: 3 αντιστροφείς, ρυθμός δειγματοληψίας 100KHz



Γράφημα 3: Καμπύλες δακτύλιου 3 αντιστροφέων στα 100KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
667	20000	44444	15	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
459,299	524288	-6.20%	43.80%	56.20%

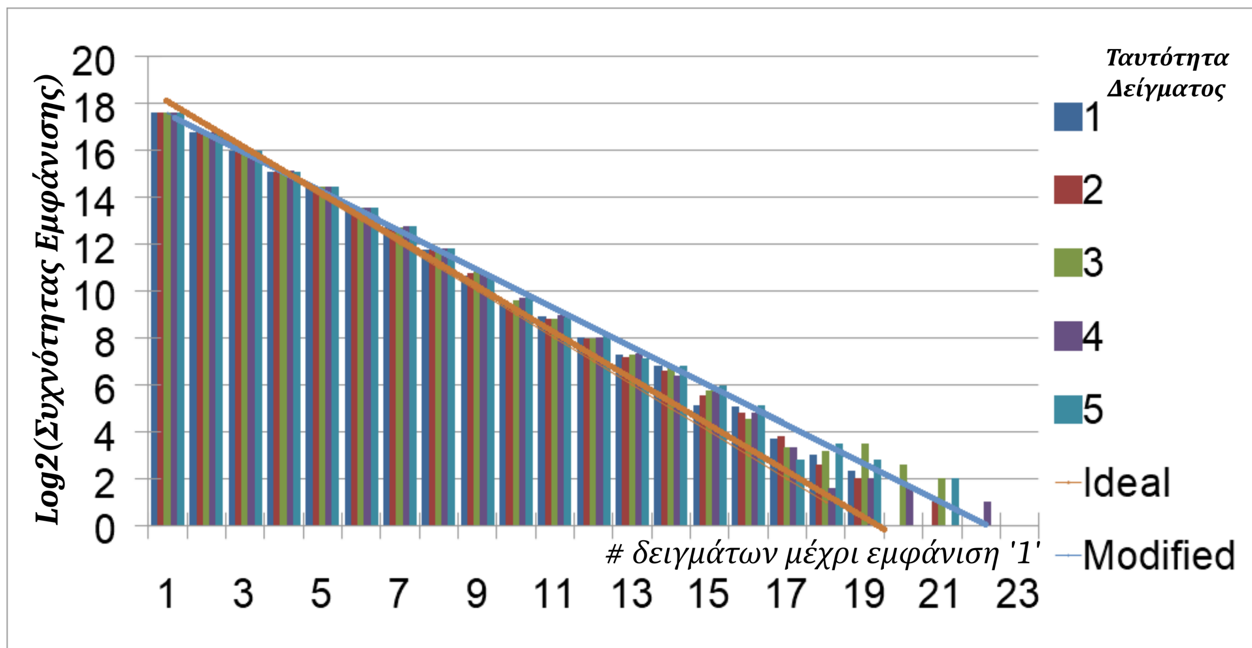
Πίνακας 4: 3 αντιστροφείς, ρυθμός δειγματοληψίας 15KHz



Γράφημα 4: Καμπύλες δακτύλιου 3 αντιστροφέων στα 15KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
667	120	267	2500	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
459,079	524288	-6.22%	43.78%	56.22%

Πίνακας 5: 3 αντιστροφείς, ρυθμός δειγματοληψίας 2.5MHz



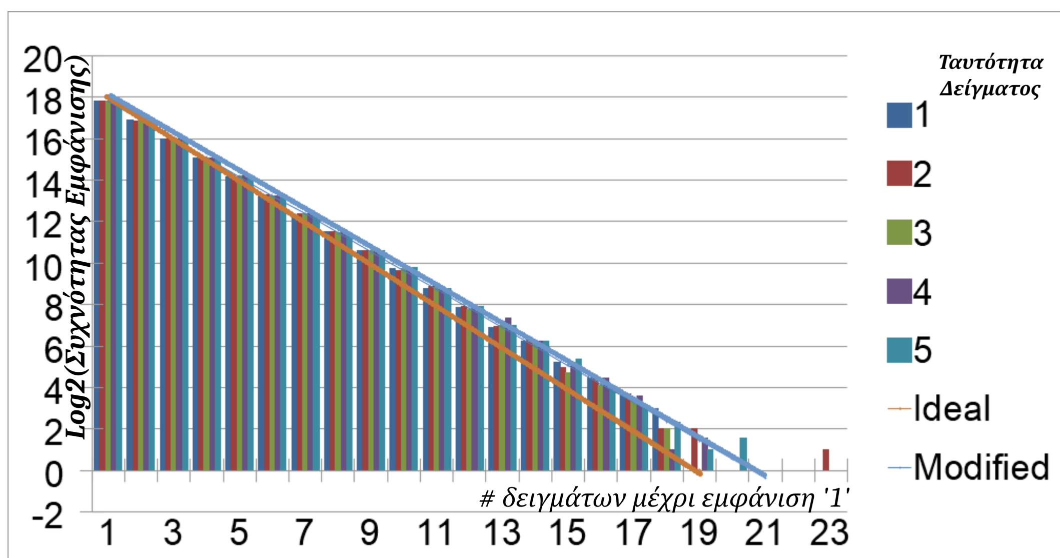
Γράφημα 5: Καμπύλες δακτύλιου 3 αντιστροφέων στα 2.5MHz

Γράφημα 6: Απόκλιση απ' την τροποποιημένη διωνυμική καμπύλη στα 2.5MHz

Όπως είναι εμφανές απ' τα προηγούμενα γραφήματα, οι δακτύλιοι με 3 NOT εμφανίζουν αυξημένη προτίμηση υπέρ του '0', όμως υποδειγματοληψία είναι αποδοτική ακόμα και για τους ταχύτερους ρυθμούς των 2.5 MHz, κάτι το οποίο τονίζεται στο τελευταίο γράφημα, που η απόκλιση απ' την τροποποιημένη διωνυμική καμπύλη είναι μικρή (εντός 1% των θεωρητικών τιμών).

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
400	3072	4096	97.5	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
490011	524288	-3.27%	46.73%	53.27%

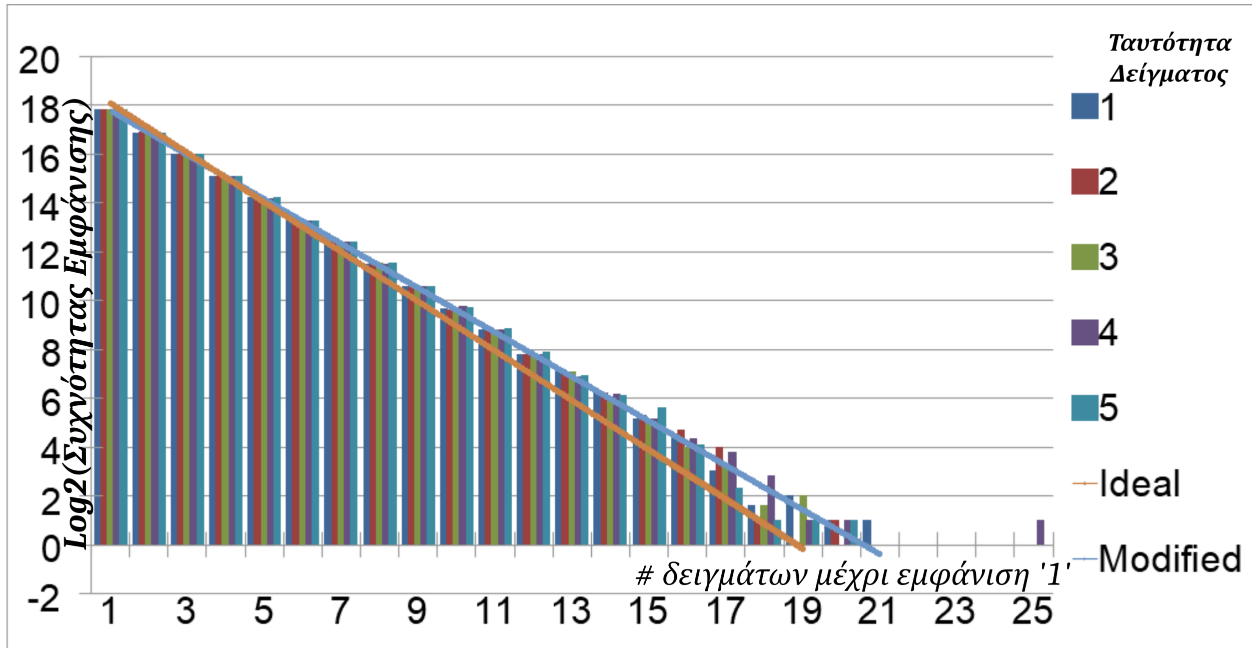
Πίνακας 7: 5 αντιστροφείς, ρυθμός δειγματοληψίας 100KHz



Γράφημα 7: Καμπύλες δακτύλιου 5 αντιστροφών στα 100KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
400	20000	26667	15	
Αριθμός ‘1’	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης‘1’	Πιθανότητα Εμφάνισης ‘0’
489687	524288	-3.30%	46.70%	53.30%

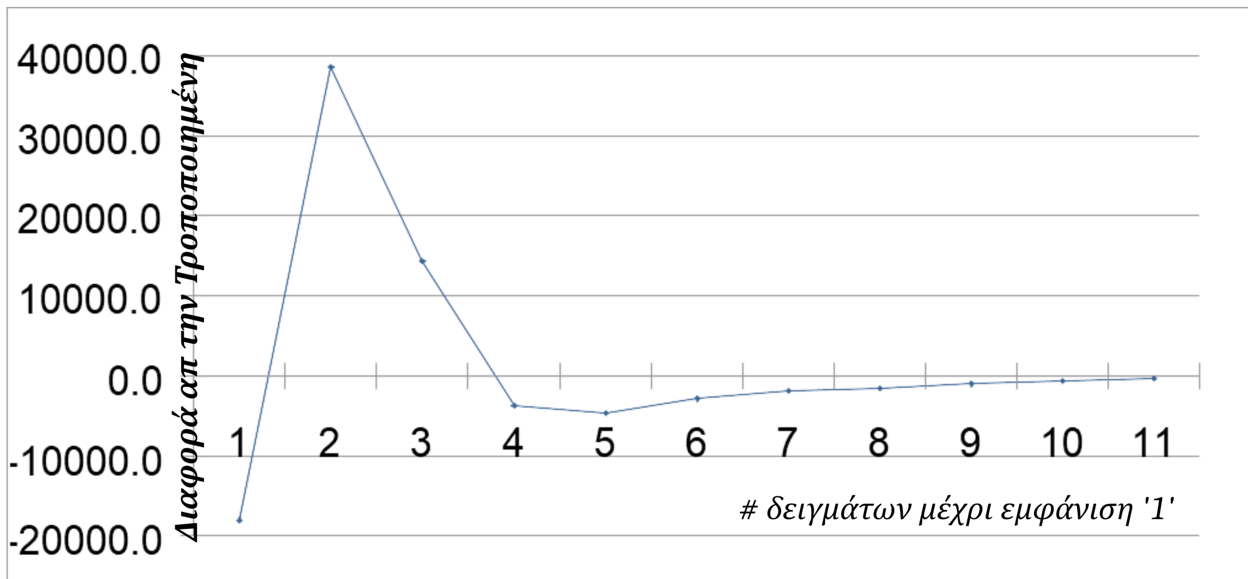
Πίνακας 8: 5 αντιστροφείς, ρυθμός δειγματοληψίας 15KHz



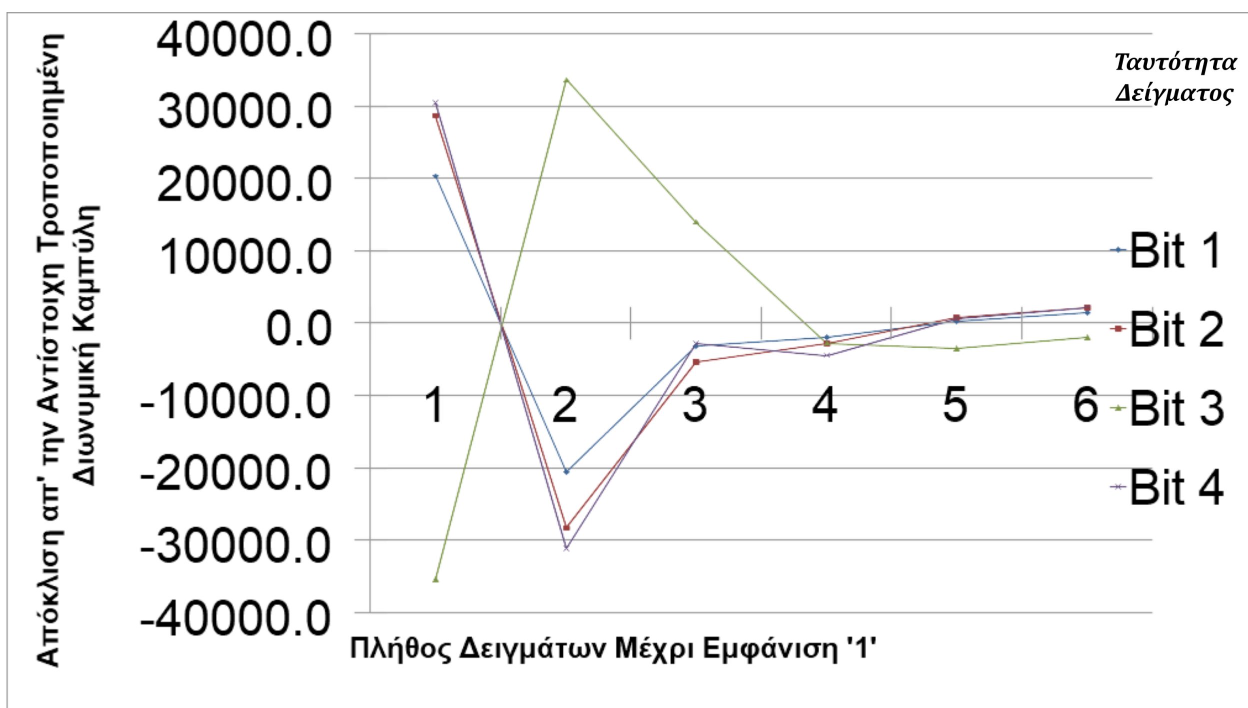
Γράφημα 8: Καμπύλες δακτύλιου 5 αντιστροφών στα 15KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
400	120	160	2500	
Αριθμός ‘1’	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης‘1’	Πιθανότητα Εμφάνισης ‘0’
490022	524288	-3.27%	46.73%	53.27%

Πίνακας 9: 5 αντιστροφείς, ρυθμός δειγματοληψίας 2.5MHz



Γράφημα 9: Απόκλιση απ' την τροποποιημένη διωνυμική καμπύλη στα 2.5MHz

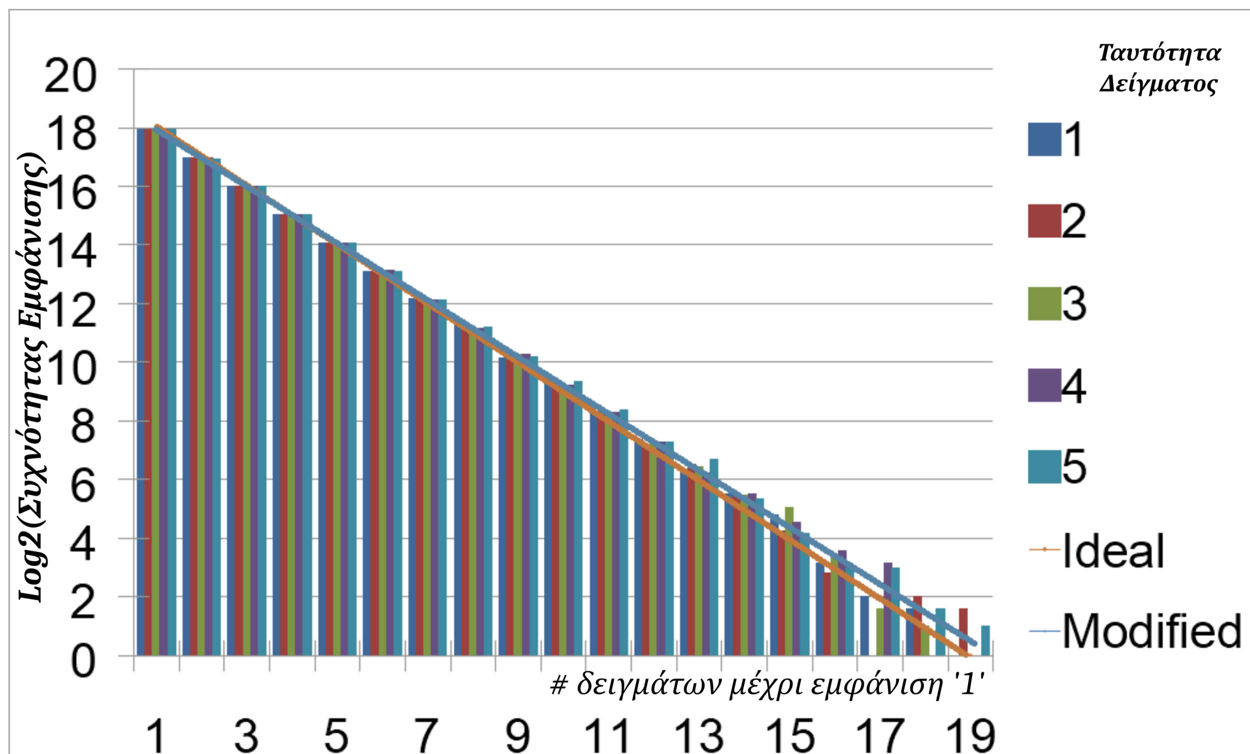


Γράφημα 10: Αποκλίσεις για 4 διαφορετικούς δακτυλίους στα 2.5MHz

Οι δακτύλιοι με 5 NOT εμφανίζουν σημαντική προτίμηση υπέρ του '0', καίτοι μειωμένη σε σχέση με τους αντίστοιχους με 3 αντιστροφείς. Το γράφημα 9 δείχνει την καμπύλη συντονισμού τους σε ρυθμό δειγματοληψίας 2.5 MHz. Εκεί η κατανομή της εξόδου αποτυγχάνει στη διωνυμική δοκιμή, και η έξοδος δεν θεωρείται αρκετά τυχαία. Το γράφημα 10 δείχνει τις καμπύλες συντονισμού στα 2.5MHz για 4 διαφορετικούς δακτυλίους.

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
181.8	3072	1862	97.5	
Αριθμός ‘1’	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης‘1’	Πιθανότητα Εμφάνισης ‘0’
512050	524288	-1.17%	48.83%	51.17%

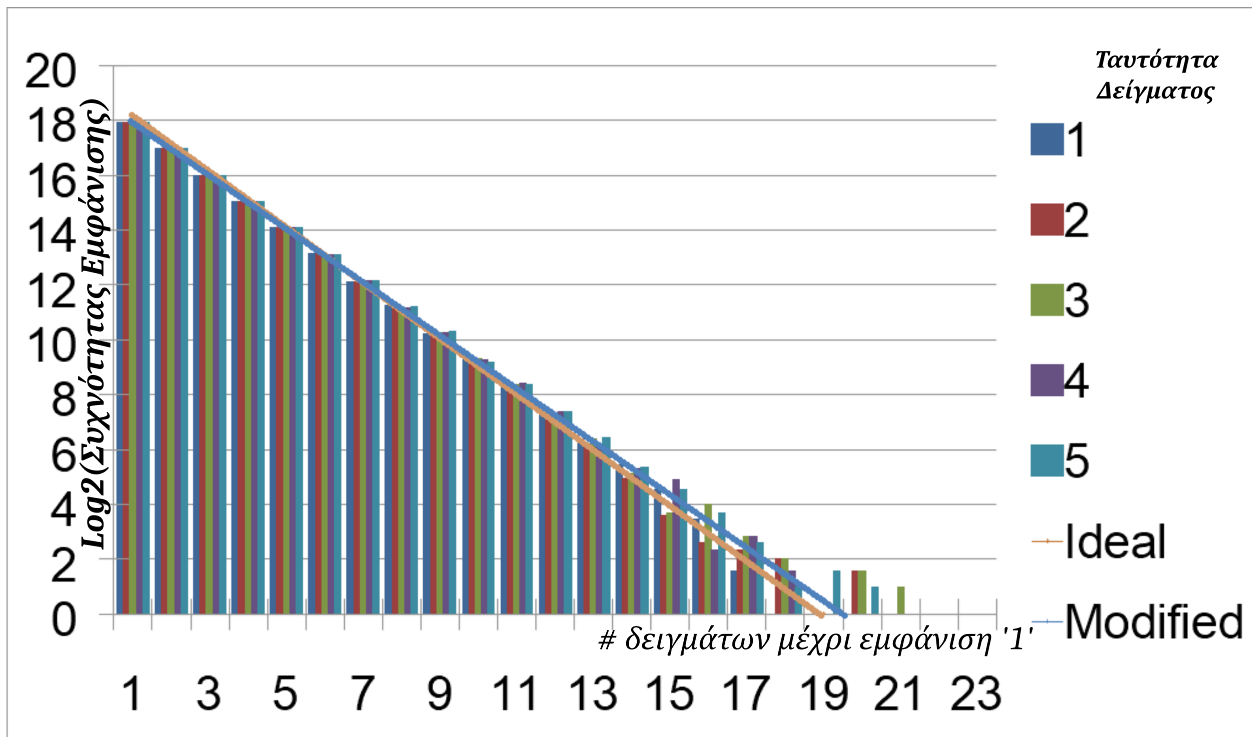
Πίνακας 11: 11 αντιστροφείς, ρυθμός δειγματοληψίας 100KHz



Γράφημα 11: Καμπύλες δακτύλιου 11 αντιστροφών στα 100KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
181.8	20000	12122	15	
Αριθμός ‘1’	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης‘1’	Πιθανότητα Εμφάνισης ‘0’
511729	524288	-1.20%	48.80%	51.20%

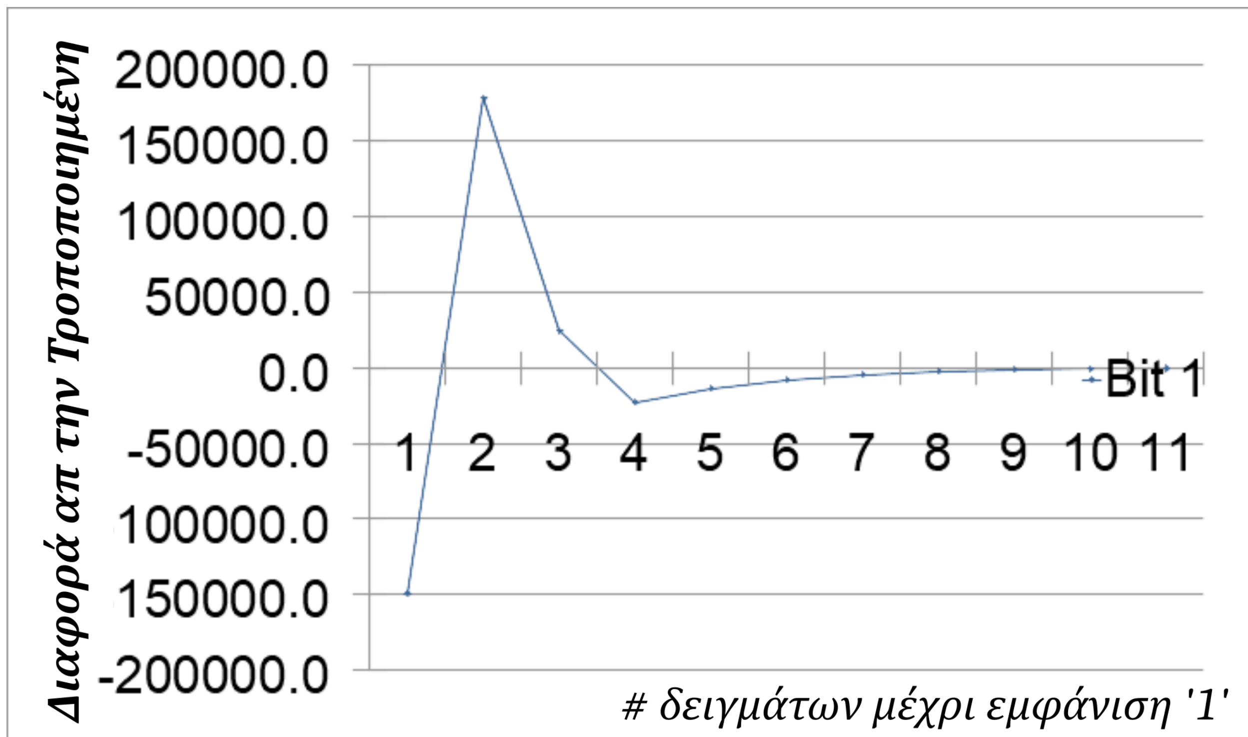
Πίνακας 12: 11 αντιστροφείς, ρυθμός δειγματοληψίας 15KHz



Γράφημα 12: Καμπύλες δακτύλιου 11 αντιστροφέων στα 15KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
181.8	120	73	2500	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
511457	524288	-1.22%	48.78%	51.22%

Πίνακας 13: 11 αντιστροφείς, ρυθμός δειγματοληψίας 2.5MHz

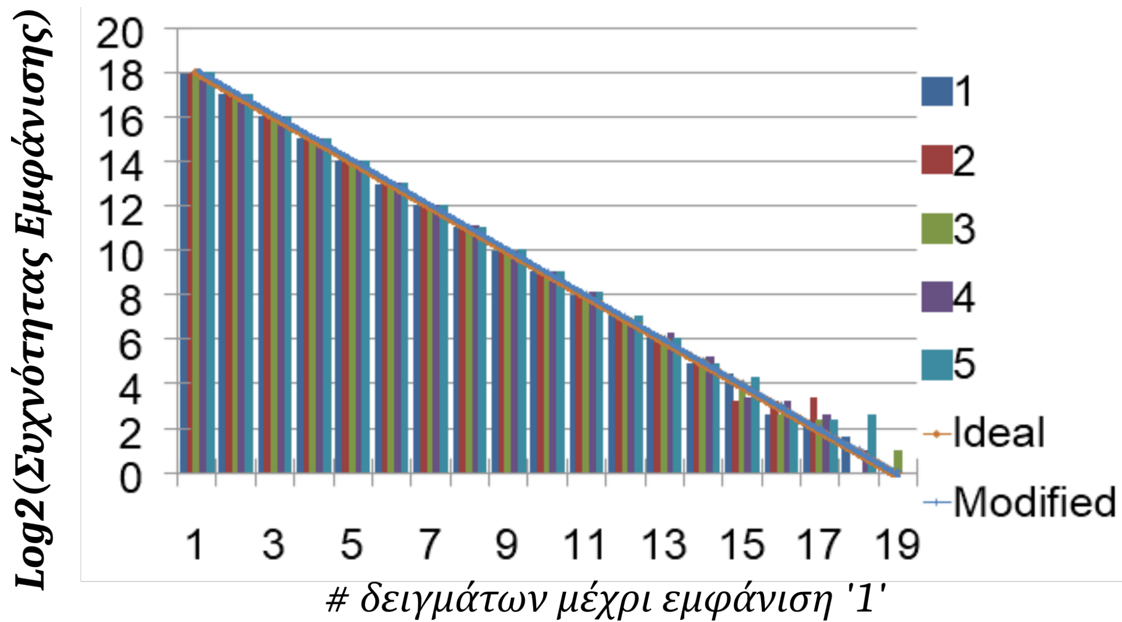


Γράφημα 13: Απόκλιση απ' την τροποποιημένη διωνυμική καμπύλη στα 2.5MHz

Οι δακτύλιοι με μήκος 11 NOT έχουν αρκετά μειωμένη προτίμηση υπέρ του '0' σε σύγκριση με μικρότερους δακτυλίους, όμως εμφανίζουν σημαντικές καμπύλες συντονισμού σε μεγάλους ρυθμούς δειγματοληψίας, όπου και αποτυγχάνουν να ικανοποιήσουν τη διωνυμική δοκιμή.

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
18.02	20,000	1202	15	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
522515	524288	-0.17%	49.83%	50.17%

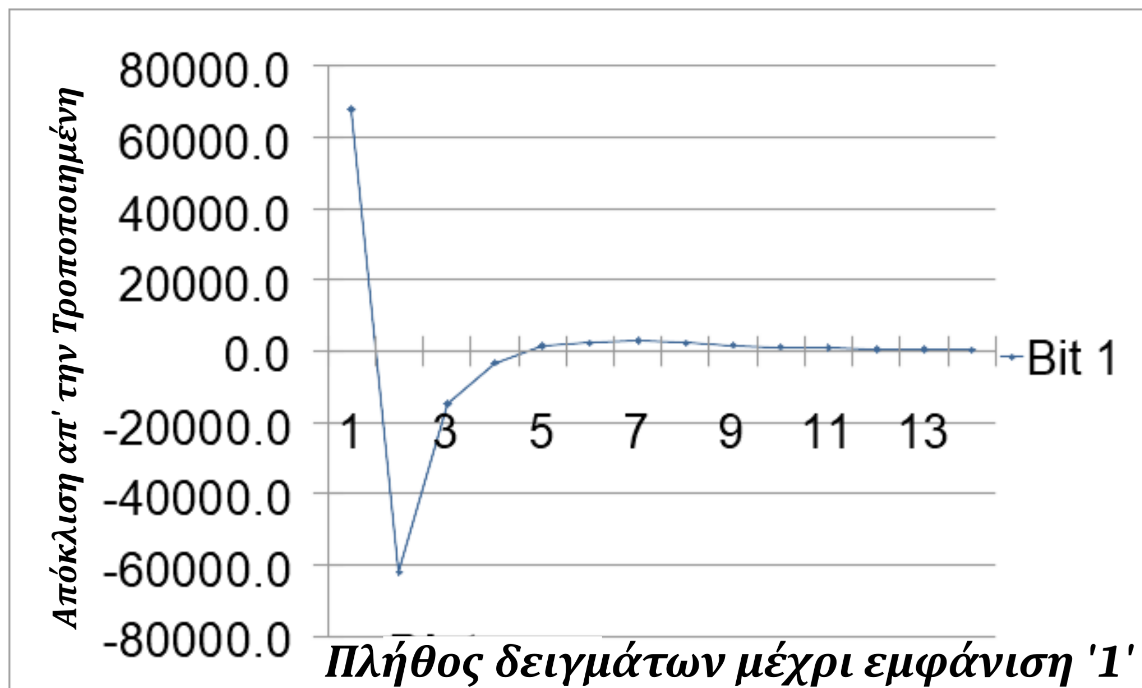
Πίνακας 14: 111 αντιστροφείς, ρυθμός δειγματοληψίας 15KHz



Γράφημα 14: Καμπύλες δακτύλιου 111 αντιστροφών στα 100KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
18.01802	3072	185	97.5	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
523,819	524288	-0.04%	49.96%	50.04%

Πίνακας 15: 111 αντιστροφείς, ρυθμός δειγματοληψίας 100KHz

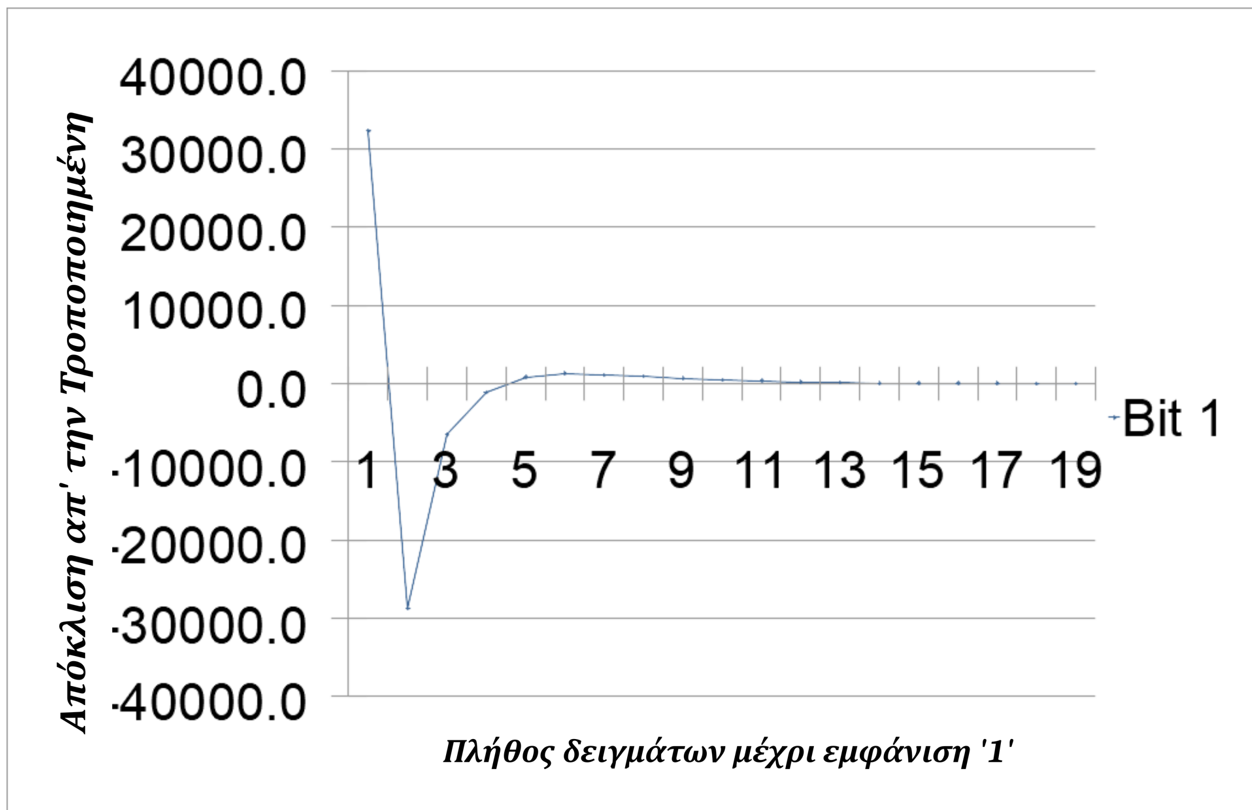


Γράφημα 15: Απόκλιση απ' την τροποποιημένη διωνυμική καμπύλη στα 100KHz

Οι δακτύλιοι μήκους 111 NOT εμφανίζουν πολύ μικρή προτίμηση όμως αποτυγχάνουν να ικανοποιήσουν τη διωνυμική δοκιμή ακόμα και σε ρυθμούς δειγματοληψίας ίσους με 100 KHz.

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	
1.80	131070	786	4.6	
Αριθμός '1'	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης'1'	Πιθανότητα Εμφάνισης '0'
524637	524288	0.03%	50.03%	49.97%

Πίνακας 16: 1111 αντιστροφείς, ρυθμός δειγματοληψίας 5KHz



Γράφημα 16: Καμπύλη συντονισμού δακτύλιου 1111 αντιστροφών στα 5KHz

Οι δακτύλιοι μήκους 1111 NOT έχουν πολύ μικρή προτίμηση όμως αποτυγχάνουν να εμφανίζουν συντονισμό ακόμα και στο κάτω όριο του ρυθμού δειγματοληψίας, που είναι 5KHz.

Στον πίνακα 17 παρατίθενται και στοιχεία μεταβολής της προτίμησης από δακτύλιο σε δακτύλιο ιδίου μήκους. Απ’ αυτά, η προτίμηση φαίνεται να εξαρτάται από τοπικά χαρακτηριστικά του πυριτίου (P) που προκαλούν διαφορές στο χρόνο απόκρισης του κάθε LUT ανάμεσα στην προσπέλαση των δύο πρώτων στοιχείων τους (εδώ το πρώτο ’1’ και ’0’ που υλοποιούν την NOT πύλη) ή/και σε ηλεκτρικά φαινόμενα σχετικά με την τροφοροσία των CLBs/LUTs [30, 31].

	Συχνότητα Εμφάνισης ’1’	Ιδανικό	Bias	Πιθανότητα ’1’	Πιθανότητα ’0’
Bit 1	490022	524288	-3.27%	46.73%	53.27%
Bit 2	508456		-1.51%	48.49%	51.51%
Bit 3	508270		-1.53%	48.47%	51.53%
Bit 4	504859		-1.85%	48.15%	51.85%

Πίνακας 17: Εξάρτηση Bias από P (δακτύλιοι με 5 αντιστροφείς

Τα ως άνω δεδομένα αφορούν τις ακατέργαστες εξόδους των δακτυλίων. Οταν η έξοδος των δακτυλίων υφίσταται μετεπεξεργασία, μέσω του κυκλώματος αφαίρεσης προτίμησης (γραμμικός μετεπεξεργαστής H3 [7, 36]) τότε οι μικρότεροι δακτύλιοι αποδεικνύονται πιο ελκυστικοί για χρήση σε γεννήτρια τυχαίων αριθμών. Στους πίνακες 18 και 19 παρουσιάζονται δεδομένα για τη μετεπεξεργασία των δακτυλίων με μήκος 5 NOT.

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	Ρυθμός Παραγωγής Τυχαίων (KHz)
400	3,072	4096	97.5	48.8
Αριθμός ‘1’	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης‘1’	Πιθανότητα Εμφάνισης ‘0’
525437	524288	0.11%	50.11%	49.89%

Πίνακας 18: 5 αντιστροφείς, ρυθμός δειγματοληψίας 100KHz

Συχνότητα Δακτυλίου (MHz)	Διαιρέτης Ρολογιού	Αριθμός Ταλαντώσεων Δακτυλίων	Ρυθμός Δειγματοληψίας (KHz)	Ρυθμός Παραγωγής Τυχαίων (KHz)
400	120	160	2500	1250
Αριθμός ‘1’	Ιδανική Τιμή	Προτίμηση	Πιθανότητα Εμφάνισης‘1’	Πιθανότητα Εμφάνισης ‘0’
524859	524288	0.05%	50.05%	49.95%

Πίνακας 18: 5 αντιστροφείς, ρυθμός δειγματοληψίας 2.5MHz

Οι πειραματικές καμπύλες των δεδομένων με μετεπεξεργασία είναι πανομοιότυπες τόσο με τις τροποποιημένες θεωρητικές, όσο και με τις ιδανικές.

Η εξαφάνιση της προτίμησης με την αύξηση του μήκους των δακτυλίων αποδίδεται στη στατιστική εξομάλυνση της συμπεριφοράς των LUTs πάνω σε ένα ευρύ πεδίο υποστώματος (αλληλοαναίρεση των μικροαποκλίσεων της συμπεριφοράς LUT).

Οι πιο ελκυστικοί δακτύλιοι για παραγωγή ακατέργαστων τυχαίων είναι αυτό που εκτείνονται σε μήκος ~11 με 111 αντιστροφείς. Έχουν αρκετά μεγάλο μήκος για να μπορούν να μειώσουν την προτίμηση υπέρ του ’1’ μέσω στατιστικής εξομάλυνσης πάνω στο υπόστρωμα. Επίσης μπορούν να συσσωρέυσουν Jitter σε ικανοποιητικά μεγάλους ρυθμούς δειγματοληψίας λόγω του ότι παρέρχονται αρκετές ταλαντώσεις δακτυλίων μεταξύ των δειγμάτων. Ταυτόχρονα είναι και αρκετά μικροί ώστε να χωράνε σαν βοηθητικό κύκλωμα σε ένα μικρό ή με σχετικά περιορισμένο ελέυθερο χώρο FPGA. Σε μεγάλους ρυθμούς δειγματοληψίας (που προσεγγίζουν τα 2.5MHz) η τυχαιότητα της εξόδου μειώνεται, καθώς αυτή αρχίζει να αποκλίνει εμφανώς απ’ τη διωνυμική κατανομή.

Με κόστος μόνο 24 FLIP-FLOP και 1 LUT (5 είσοδοι και μια έξοδος) ανα δακτύλιο, είναι δυνατή η πολύ αποτελεσματική μετεπεξεργασία της εξόδου των μικρών δακτυλίων μέσω του αλγορίθμου H3. Αυτή η λύση παράγει το πιο ικανοποιητικό αποτέλεσμα καθώς οι καμπύλες που αποδίδει είναι σχεδόν πανομοιότυπες με την ιδανική διωνυμική, ακόμα και στους ταχύτερους ρυθμούς δειγματοληψίας των 2.5MHz.

Contents

Abstract	page	I
Περίληψη	page	III
Chapter 1: Introduction	page	1
1.1 Motivation for this Study	page	1
1.2 Entropy Sources and TRNGs	page	2
1.3 FPGAs	page	7
1.4 FPGA components	page	12
1.5 VHDL	page	19
Chapter 2: Design of the TRNG	page	21
2.1 Ring Oscillators as Entropy Sources	page	21
2.2 The TRNG's Features	page	31
2.3 Experimentation	page	35
Chapter 3: Realization of the TRNG System	page	57
3.1 Free Parameters and Expectations	page	57
3.2 Design and Floorplanning	page	60
3.3 Moments of a Uniform Distribution	page	63
3.4 Bias and 1s Frequency	page	65
3.5 Distribution of the Sequence of 1s	page	66
3.6 Design of the Statistical System	page	66
Chapter 4: Results and Interpretation	page	73
Chapter 5: Conclusions	page	89
References	page	91
Appendix A: VHDL and TCL code for the Preliminary Experiments	page	95
Appendix B: VHDL and TCL code of the TRNG System	page	111
Appendix C: VHDL code for the Statistical Engine	page	119
Appendix D: H3 Post Processor Code		149

1. Introduction

This study concerns the process of creating and evaluating of a True Random Number Generator Device. In order to proceed, some preliminary knowledge is needed to establish a frame of reference and get acquainted with the nomenclature. Consequently, the sections that follow shine a light into the motivations behind the study and outline various important concepts that were key in the development and analysis of the device.

1.1 Motivation for this Study

The generaton of totally unpredictable numeric patterns has wide applications in various fields of Science and Engineering. To name but a few applications, Monte Carlo methods require random numbers to solve problems that may be deterministic in nature, various statistical sampling methods involve randomization of sampling, games use random numbers to control the unpredictability of state and outcomes in various degrees (this is especially true in games of chance, like slot machines), and most importantly, randomness lies at the root of every scheme in the field of Cryptography.

Nowadays even the most mundane electronic transaction that takes place over a network requires some form of cryptographic providence. Anything ranging from an SMS message to an E-Mail, a Tweet to a file transfer, unlocking one's car, conducting a Bank Transaction, connecting to Wi-Fi or remotely accessing a printer are processes that involve a substantial amount of cryptographic work to be seen through safely.

Rarely is anything ever transmitted in so-called plaintext form (meaning unencrypted and available for anyone to comprehend), bereft of message authentication codes and cryptographic certificates. This is to shield against malicious attempts that could modify the contents of the transaction, hijack the network or seize it and halt its usability, but also to harden against inadvertent corruption of the contents of the transaction. So the cryptographic infrastructure that powers these transactions guards the integrity and the authenticity of their contents. However every scheme involving a cipher, a cryptographic key exchange or creation or issuance of a certificate starts at a common place: the source of random-

ness.

It is apparent that all of constructs and applications stemming from the aforementioned principles share a common bedrock, and that is access to random numbers. It then becomes a question of how one produces such numbers to service all those potential applications. When such important concepts such as confidentiality, commerce and authenticity rely on random and unpredictable numbers, this provides the necessary motivation to embark on the study of true random number generation.

Those random number streams must satisfy some conditions of unpredictability and some statistical qualities, in degrees that vary depending on how critical the applicaiton they are intended for is. As is explained in the upcoming sections, there are differences between PseudoRandom Number Generation and True Random Number Generation, the latter of which is the focus of this study.

1.2 Entropy Sources and TRNGs

In the field of Computing, Entropy is understood to mean true randomness, unpredictability or, more formally, entropy is "*a measure of the disorder, randomness or variability in a closed system*" [1, 10]. To be useful, Entropy has to be collected through various implementation specific processes, turned into a stream of numbers and subsequently used in cryptographic functions as seeds, nonces and whatnot. Every cryptographic scheme relies on at least one element that is considered to be unpredictable, to operate safely, otherwise it is as good as broken; Supposing, for instance, that a totally self feeding process is used to derive keys, it is possible for a third party to replicate its output and compromise every transaction. An Entropy Source is the provider of such randomness and unpredictability.

TRNG stands for True Random Number Generator. A TRNG is a system that yields a truly random and unpredictable output. Equivalent terms for TRNG include, Non-Deterministic Random Number Generator (NRNG), True Random Bit Generator (TRBG) and Non-Deterministic Random Bit Generator (NRBG) [1].

The terms Entropy Source and TRNG are taken to mean the same thing in the upcoming sections. There could be some distinction made when entropy source refers to the physical source of randomness, whereas TRNG means Entropy Source plus some processing to convert the fundamental randomness to a number stream, but here these terms are used interchangeably.

This is in contrast with a PseudoRandom Number Generator, also called Deterministic Random Number Generators. The latter are constructs that (usually) require an input, and they also posess an internal state. They combine the input and their current state, and usually through a series of rounds they transform it in such a way as to produce a sequence of numbers that are in some sense random. Good PRNGs exhibit the following properties:

- If the internal State (or the initial State) and input is unknown, it is difficult to predict future output values based on the current one
- If input changes slightly, the output varies wildly, in a manner that can generally only be brute-forced to figure out how
- They exhibit a uniform distribution over the range of numbers they produce. For example, a PRNG producing an 8-Bit output should yields each possible 8-Bit with probability 2^{-8}

There are even more properties that are generally desirable in a PRNG, this falls outside the scope of this study. It's output is thought of as "Random" in some sense, however it is 100% dependent on its internal State, so every single PRNG in existence has the following property: If the PRNGs design is known and the State also becomes known, then one can trivially figure out every future output it will produce if it keeps using that state.

To avoid this PRNGs often use reseeding, meaning they get a new input value and combine it with the Initial or Working State to veer off the previous sequence. This Input, or Seed however usually comes from a truly random source. If the Seed is truly random then the PRNGs outputs can be claimed to be exhibiting true randomness, at least the one occuring at the reseeding moments. The more frequent the reseeding is, the more true entropy the PRNG will carry, and subsequently provide the consumer with reliably unpredictable streams.

So, if one desires to construct a sequence of numbers with any amount of real randomness/unpredictability, access to a source of truly random sequences is a precondition. This is the role that a TRNG fulfils. TRNGs provide consumers, such as PRNGs, with a stream of truly random numbers.

Although PRNGs can exist in both in software and hardware forms, TRNGs always tap into the inherent randomness of the physical world. So when talking about a TRNG, there are always components that exist in the physical world, be they a human, a stream of protons from the sun along with a detector, a specialized chip and so forth. So TRNGs always encompass some physical component that provides the entropy inherent in natural processes, monitor this process and convert some signal to bits, and then hand off those bits to a consumer or post-processor to be further processed.

How is entropy provided to various consumers? Operating systems collect entropy for their needs by observing, among other things, mouse movement and keyboard timing from the user. Although humans aren't terribly random, given enough time this strategy can yield some entropic output. What's wrong with this approach? Well, aside from the fact that it takes a long time to yield a sizeable result, humans aren't always available for a device that requires a source of randomness. Embedded devices, like Wireless routers, devices that require entropy during boot or shortly after, or even servers, that under normal operating conditions lack local user input, can't really rely on a human, or they risk entropy depletion. Entropy depletion weakens a system's cryptographic strength. For a server, for example, it could mean creation of weak TLS//SSL keys so that third parties might start guessing parts of their bits, and from there compromise the respective transactions with reduced effort.

What other solutions are there to ensure that cryptographic schemes are backed by the presence of an unpredictable element?

- Hardcoding values

Bad practice in general, as after a while they are used up. Even worse there are techniques to derive them if they support a cryptographic transaction for too long (look at differential cryptanalysis, as an example). They can also be compromized at the stage of hardcoding, through various means, or even after, perhaps by more invasive techniques. This is an indirect leveraging of a natural process, in that the hardcoded value was provided by the output of such a process at some time.

- Observing other device quantities with a degree of non determinism

Like interrupt timings from software or harware, which are non deterministic in complex systems like PCs or servers. Or they could make an array of boot times and get the last bits. Still so many applications are left unserved, and these sources do not yield enough output for more demanding scenarios. As a result, entropy could become depleted, leading to the creation of weak keys etc. This, again, is an indirect leveraging of natural processes, in that the devices used are not dedicated to the creation and monitoring of such processes. Take interrupt timings for instance: while it may be true that they do exhibit some variability, they are not meant to be variable, and they might as well not be at all for the purposes of a PC. That's why this type of random number production is considered indirect.

- Relying on dedicated hardware that leverages a natural process

This solution is the most reliable and performant. Hardware that directly leverages a natural process trumps the other approaches, both because it is applicable in every situation, but also because it scales well and can have a large yield if needed. Such a design is also very safe in the sence that no matter the parameters that the system operates in, the output of the device is guaranteed to remain unpredictable. Operating systems today rely heavily on those types of sources, such as TPM if available and enabled on motherboard ot the RDRAND CPU instruction (also known as Bull Mountain) [2], which acesses a hardware random number generator on the CPU.

Various types of processes can be used to start producing a stream of random numbers. Many designs use thermal noise in electronic circuitry, which is inherently unpredictable, to derive a stream of bits. Such designs are hard to implement correctly because they rely a lot on fine tuning of parameters, and are also susceptible to environmental factors such as temperature, making it difficult to know if they are working correctly. An attacker with physical access to the device could, say, pour liquid nitrogen on it thereby reducing the entropy of its output considerably. This is just an example that leverages a so-called classical noise source. Sources of Classical Noise Include:

- Thermal noise from a resistor, amplified properly to produce a random voltage source
- Avalanche noise from an avalanche diode, or breakdown noise from a reverse-biased Zener Diode.
- Atmospheric noise, picked up by a radio receiver
- Cosmic Radiation

There are other devices, that rely on various microscopic/quantum phenomena, but they usually rely on fine tuning of various parameters and are sensitive to environmental effects. Just to name a few of those quantum processes:

- Shot noise, a quantum mechanical noise source in electronic circuits. A simple example is a lamp shining on a photodiode.
- A nuclear decay radiation source.
- Photons travelling through a semi transparent mirror.
- Amplification of the signal produced on the base of a reverse-biased transistor.

Those designs may be the “Gold Standard” when it comes to randomness, because they tap into the true unpredictability of the quantum realm. But still, collecting the noise from those sources and ensuring their proper behavior poses difficulties. For that reason it is generally desirable, when designing a TRNG, to incorporate techniques that do not rely heavily on finely tuning device parameters and offer stability with regards to environmental conditions (such as temperature, magnetic fields, acceleration or radiation levels). One such technique is the one incorporated into the particular TRNG developed for the purposes of this study: sampling ring oscillator outputs.

1.3 FPGAs

Figure 1.1: An FPGA chip

FPGA stands for Field Programmable Gate Array. An FPGA is semiconductor device that bears a PL (Programmable Logic) Fabric, which in turn is based around elements called Configurable Logic Blocks (CLBs), but also typically contains other types of blocks, such as DSP blocks, IO buffers, RAMs et.c. [13, 20]. CLBs contain basic combinatorial and sequential logic building blocks, that can be used to implement basic logic functions (for example operations such as EX-OR). Individual CLBs are connected to each other via programmable (reconfigurable) interconnects. So, by linking multiple more basic pieces of logic within individual CLBs, arbitrarily complex logic can be implemented (up to the limits of the FPGA, of course).

These integrated circuits are designed so that they are configured post production, and as such they offer a great degree of flexibility, in the sense that, after manufacturing, their behavior is largely specified by a designer through the use of a Hardware Description Language (HDL). Use of a hardware description language introduces a level of abstraction that can be leveraged to create more generic and portable designs, contrary to traditional methods (circuit schematics).

That is to say, a design expressed in an HDL can be transferred to another FPGA family with minimal or no effort in recoding by the designer.

The HDL description of how the integrated circuit should behave is generally interpreted by tools that aid in the automation of the design workflow, such as the stages of synthesis, simulation, fitting floorplanning etc. Should the need arise for a bug-fix or a modification of an implemented design, or even if one wishes to implement a completely novel and unrelated design, the FPGA can be reprogrammed to fit the current requirements. This is in contrast to, say, an Application Specific Integrated Circuit (ASIC), or even a General Purpose Integrated Circuit (like a CPU), realized in immutable silicon dies (usually based on CMOS technology), which are extremely rigid when it comes to modifying their internal behavior. Programming an FPGA in practice means loading a dedicated programmable memory (usually an SRAM, Flash or EEPROM) with a binary file called a bitstream. The bitstream dictates all the connections that will be formed in the routing fabric and all the regions that will be deactivated. As such reprogramming an FPGA is trivial once the new bitstream is available: it is a simple file transfer operation. Contrast this to the inflexibility of ICs such as ASICS that employ immutable routing: Assuming a new design is available, realization means a new order to a Fabrication Plant, a process with incredibly long lead times (the situation was further exacerbated during the COVID-19 pandemic).

As such, FPGAs are ideal for prototyping, where it is all too common for design flaws to arise. It used to be the case that prototyping was the main application of FPGAs, but this has long since changed, as today they are most commonly used as they are, due to the fact that their architecture allows the rapid development of hardware solutions optimized for complex tasks. The fact that FPGAs can perform extremely intensive computational tasks, with a high degree of parallelization, all while being dynamically reconfigurable, opens up an array of viable fields for their application, such as Hardware Acceleration, Computerized Tomography and Radio Frequency Engineering to name a few.

The design process of the FPGAs consists largely of a flow of the general steps of High Level Design (specifications and HDL description), Elaboration (conversion of the HDL into logic primitives), Synthesis (derivation of the netlist), Implementation, and reiteration at any of those stages upon encountering problems. Problems may arise due to various reasons, such as:

- Language rule violations: The source files provided to the compiler do not make sense given the particular set of rules the HDL language is designed to use. This is the simplest case and the compiler will simply notify the designer and halt further action until the source code has been remediated.

- Logic faults: In this scenario the source files are alright with regards to the particular syntax rules of the language, but the step of elaboration and functional simulation yield different results from those expected from the design on the component. This problem is fixed by reevaluating the HDL description of the intended logic.

- Synthesis errors: issues such as invalid netlist derivations (like multi-driven pins) can halt the process of development. The problem is, once again, fixed at the source level.

- Errors during placing and timing analysis: the inferred circuitry does not conform to the timing constraints imposed by the designer and/or by component requirements like fulfilling setup and hold times at various latches, or because of physical constraints, that affect both timing and availability of resources. A component that, for example, requires a certain number of Flip-Flops, cannot be placed in an area on the FPGA die has less.

In the following diagrams, one can see the difference between the traditional development flow, and the more modern one:

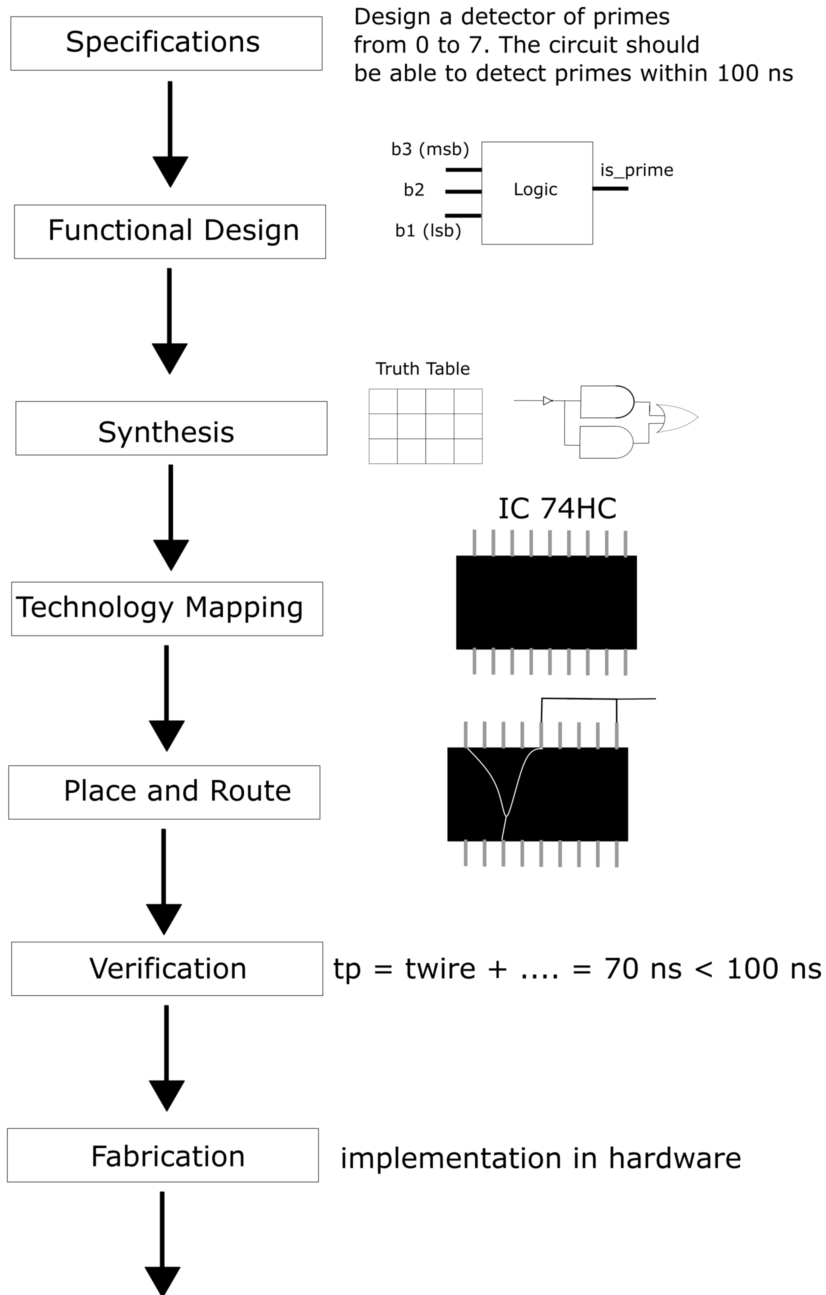


Figure 1.2: The traditional hardware development flow

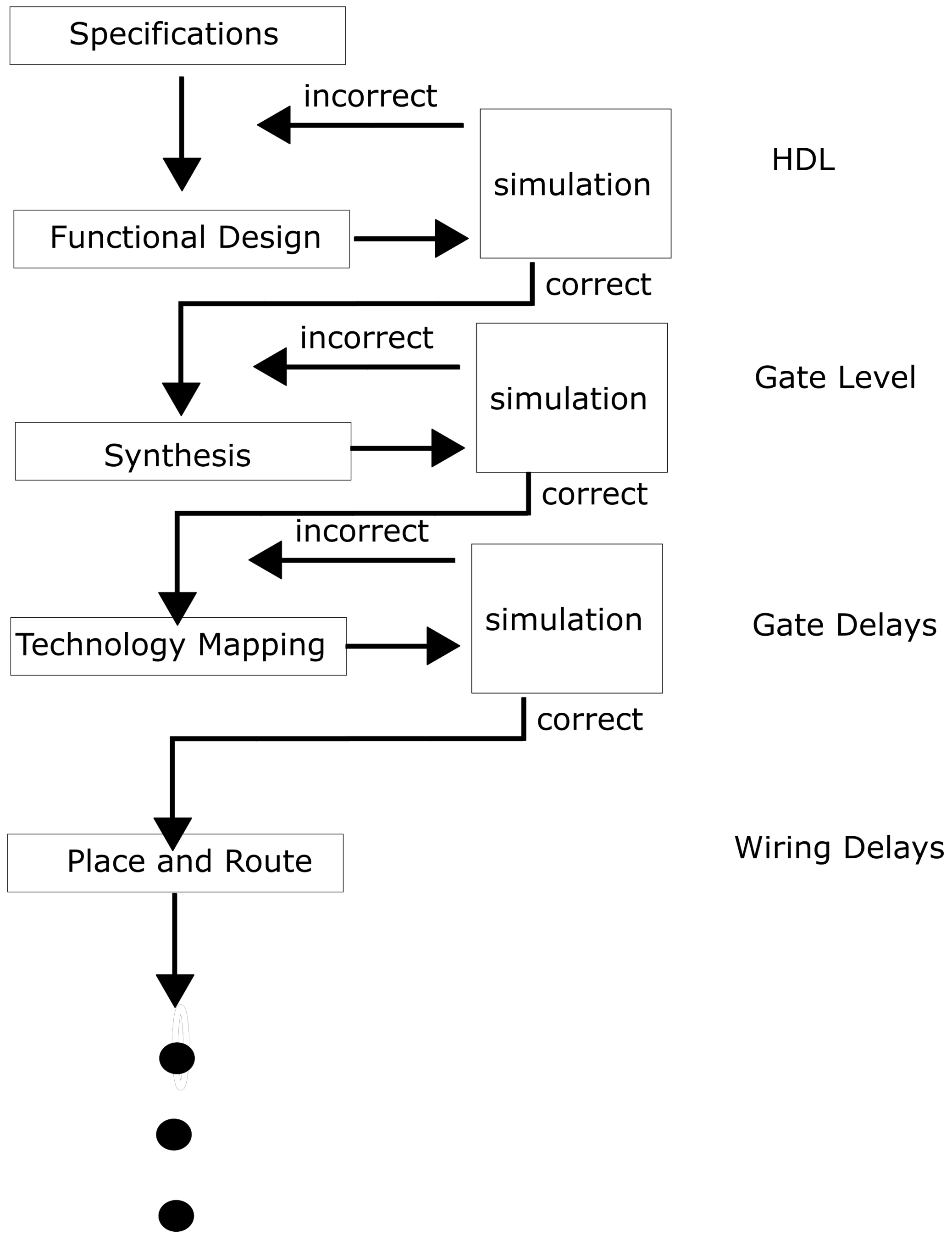


Figure 1.3: The modern hardware development flow

For the implementation of the various systems of interest, the ZCU9EG-2FFVB1156E FPGA from Xilinx was chosen, on board the ZCU102 development board. This FPGA incorporates Xilinx'es Ultrascale+ architecture. Important architectural details will be explained in subsequent chapters. Hardware design typically takes place within an IDE (Integrated Development Environment), which is a software suite that provides an array of tools like HDL compilers, design rule checkers, timing analysers and visual aides in a single package. The IDE, in short, contains all the tools necessary to achieve the modernised version of the workflow, and more. For the purposes of this study, the Vivado® Design Suite [16] from Xilinx Inc. was used.

1.4 FPGA components

When in comes to designing for an FPGA, one must be aware of what some key basic terms are, and also be knowledgable of several concepts and techniques that are relevant. Because the TRNG design makes use of such concepts and practices, there will be a brief analysis here of select terms that will be of use in upcoming sections. Note: It is not the intention of the following section to repeat verbatim what already exists in the Xilinx documentation or to analyse to particular features of every component mentioned ad nauseam, nor is it to serve as a manual on Hardware Design, but merely to shed light on the most important aspects of the mentioned terms to illuminate the reader to adequately comprehend any material that the following chapters will present.

Those FPGA components are

- CLBs
- SwitchBoxes
- Clocks
- PLLs
- DSPs

The CLB is the main resource for implementing general-purpose combinatorial and sequential circuits. Synthesis tools automatically use the highly efficient logic, arithmetic, and memory features of the FPGAs Architecture.These features can be inferred from the code, as well as be directly instantiated for greater control over the implementation. The CLB is made up of the logic elements themselves, which are grouped together in what is called a Slice, along with the interconnect routing resources to connect the logic elements. Utilization of slices, their placement in the device, and routing between them is a job mostly left to the Design Tools.

In order to implement combinatorial logic (e.g. a simple logic gate), CLBs provide a versatile component clale a Lookup Table (LUT), wich is actually an SRAM that gets initialized according to the truth table of the combinatorial logic they implement. The states on its address vector then trigger reading the appropriate output combination from the LUT.

The CLBs found in the Ultrascale+ architecture (the one used in the development) provide advanced, high-performance, low-power programmable logic with: real 6-input look-up table (LUT) capability, dual LUT5 (5-input LUT) option, distributed memory and shift register logic (SRL) ability, dedicated high-speed carry logic for arithmetic functions, wide multiplexers for efficient utilization and dedicated storage elements that can be configured as flip-flops or latches with flexible control signals [4].

Actually, the CLBs in XILINX FPGAs can bear one two types of slices, SLICE_M and SLICE_L. SLICE_M, M for Memory, which is the full slice, can implement memory elements using it's LUTs (distributed 64 bit RAM, as the LUT also has an extra Write Address, Write Enable and Clock signal), and shift registers (delaying LUT output and chaining it), which are part of sequential logic. SLICE_L, L for Logic, is more restricted, and cannot be used for memory (more limited LUT capabilities), and can only implement Combinatorial Logic.

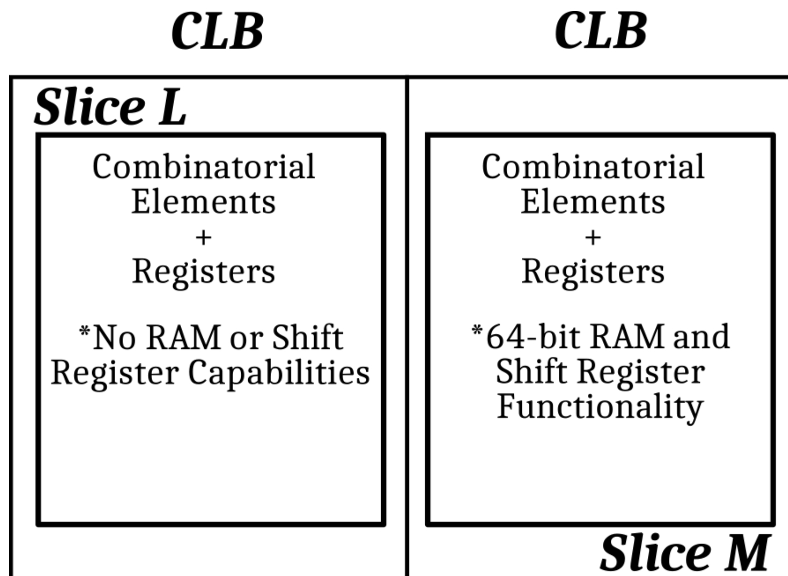


Figure 1.4: CLB Full (M) and Restricted (L) Sices

In the following picture, taken from the Device View that the Vivado IDE offers, one can see a schematic representation of some CLBs and their basic elements. Note that the only difference between the CLBs are some differences in component pins (these are LUTs), which are richer in the case of an M slice as per the reasons previously noted. The internal uninstantiated interconnect network is not drawn.

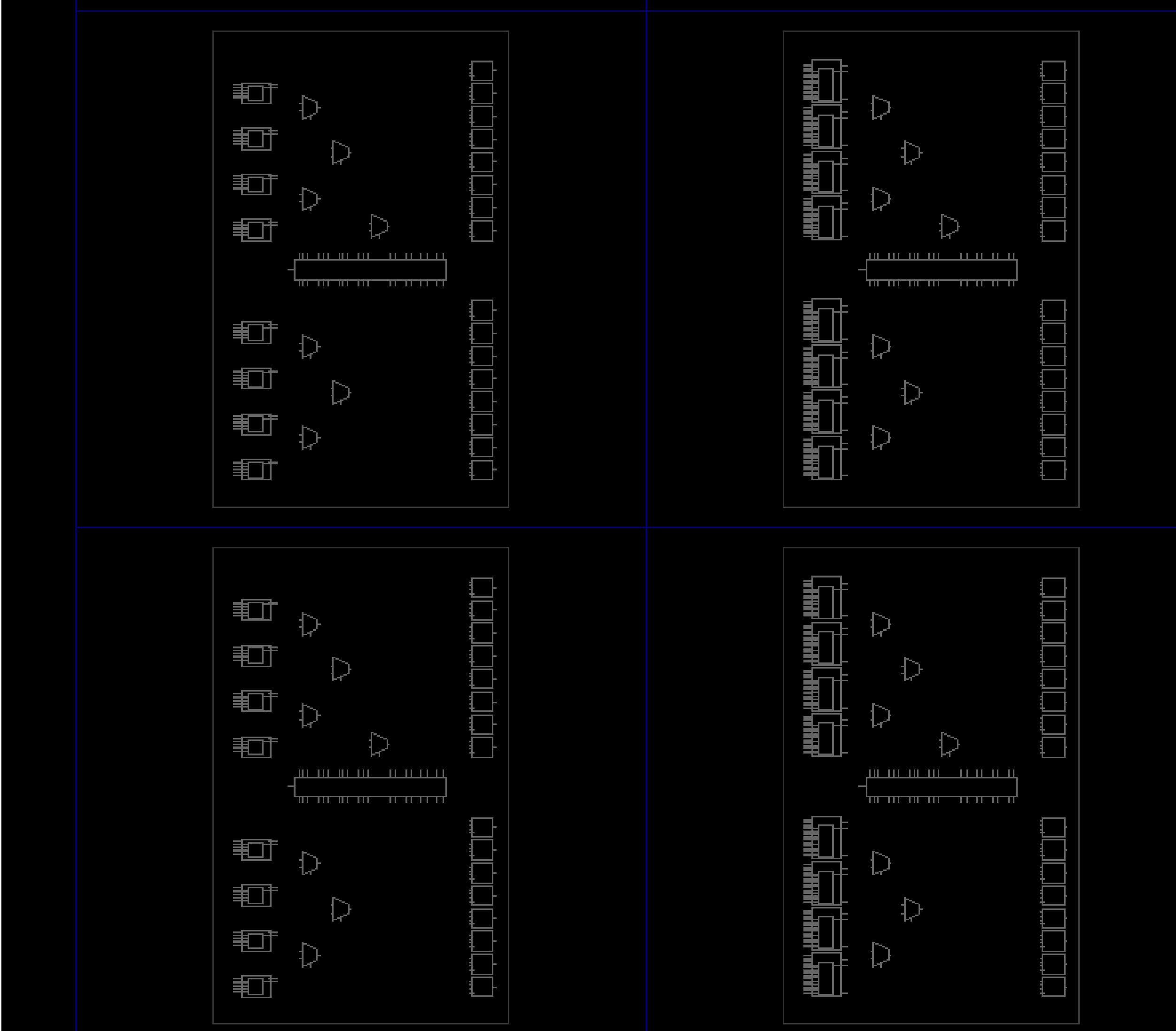


Figure 1.5: CLBs in the FPGA (Vivado Device View), two types of slices visible

The reconfigurability of the interconnects is achieved owing to, in large part, a primary element responsible for forming the necessary connections, called a Switch Matrix, or SwitchBox. An essential part of the PL Fabric (the part of reconfigurable logic of the FPGA), the SwitchBox can map one of its pins to a host of the other pins at it's periphery, thus helping to dynamically control the pathways

that connect the various other FPGA basic elements to each other (although it's capabilities are not unlimited).

So, in short, when implementing some combinatorial or sequential logic, the general purpose programmable logic elements within the CLBs, are most commonly used, and connections between CLBs and other resources commonly utilize the Fabric Routing Resources, with routing lines connecting to the Switch Boxes adjacent to the CLBs. These basic resources include parts such as LUTs (Lookup Tables), Flip-Flops (Registers), MUXes (Multiplexers) and Carry Chains, that implement combinatorial and sequential logic.

Other types of resources that can be used when implementing some form of logic can also include extra-CLB items like DSPs (if present) for arithmetic or RAMs (such as (BRAM and URAM) for memory. Switch Matrices also provide for these other types of access, for instance to create connections between implemented logic and global buffers or clocking pins, DSPs et.c. if so desired. Again, the aforementioned terms will be explained in detail shortly.

The following diagrams illustrate the basic principles discussed so far about how the FPGA actually links different basic logic components together.

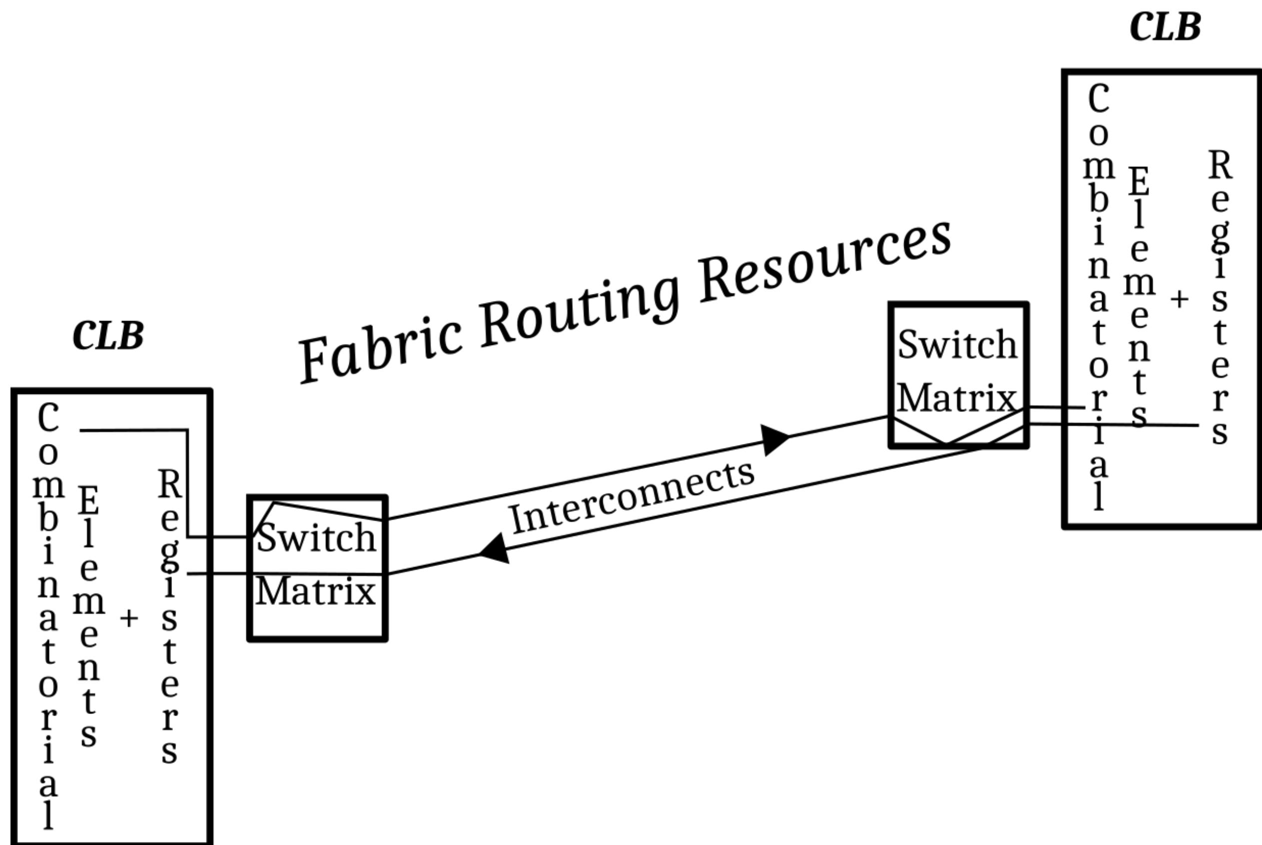


Figure 1.6: Two CLBs linked to one another though Fabric Rounting

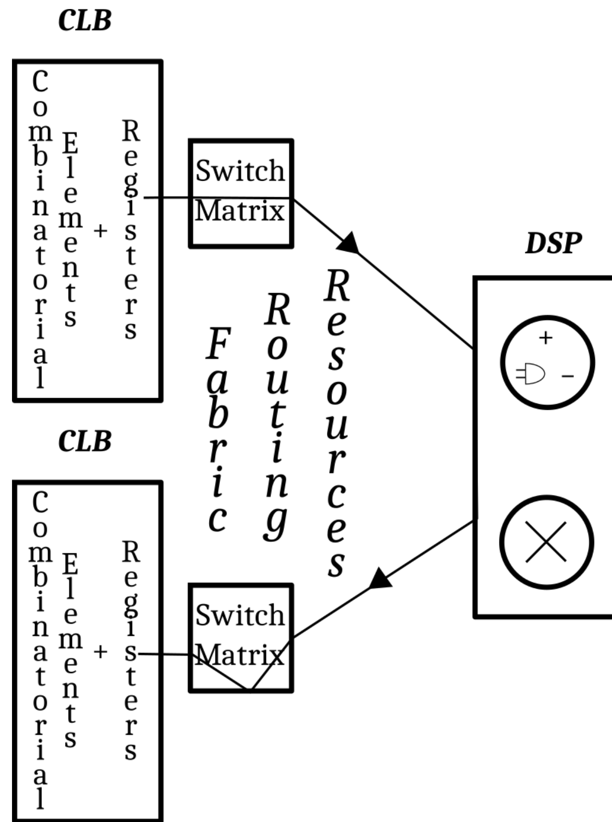


Figure 1.7: CLB registers connect to a DSP

in Figure 1.7 registers from a CLB link to a DSP block and that, in turn, sends its output to some other registers (most likely to implement some sort of arithmetic on the first registers' data and store the result in the latter).

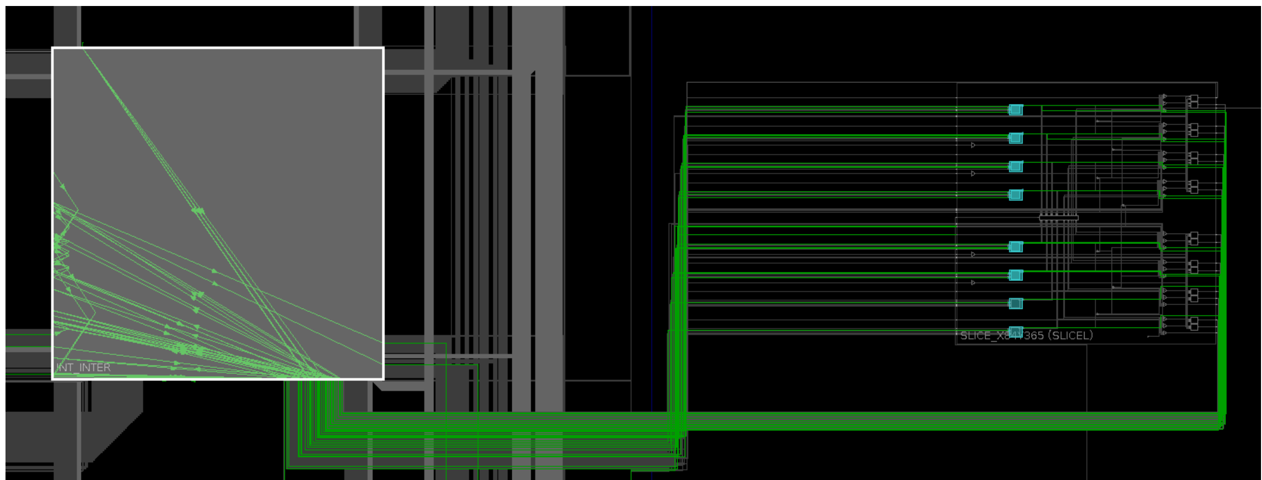


Figure 1.8: picture of a switching matrix as seen from Vivado Device view.

In Figure 1.8 the SwitchBox (left) links some logic elements (in blue) together by matching interconnects (green lines) to each other. This particular arrangement is inferred by the Vivado tools to implement some desctibed logic, and the utilized components are highlighted here.

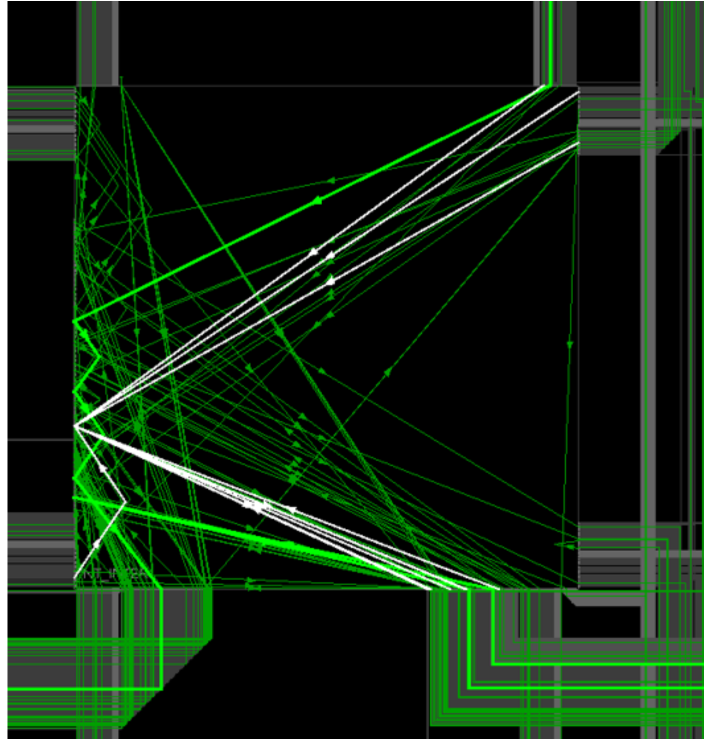


Figure 1.9: connections available to SwitchBox Pins

In Figure 1.9, the particular connections available to a certain pin junction on the SwitchBox are shown in white color (it's routing capabilities are not unlimited).

Clocks are an essential part of sequential logic. They are actually what differentiates sequential logic from combinatorial logic. Clocks are highly periodic (usually) signals, whose edge triggers the transference of data to and from registers. When it comes to synchronizing various processes within the FPGA, a clock signal is needed to establish a common reference.

Processes that generally happen in accordance to the same clock are said to be Synchronous, while others that rely on different clocks or no clock at all are termed Asynchronous. Sequential logic happens in relation to the temporal framework that the clock provides. Combinatorial logic lacks any form of temporal reference and is governed only by it's data inputs. Note however that, that is not to say that combinatorial logic transformations take place instantaneously. It

merely means that it does not happen in accordance to some timing signal. Both combinatorial and sequential logic transformations in hardware require finite time to take place.

In FPGAs, the clock signals are typically routed through the clocking network, which is a dedicated high speed interconnect that spans the whole IC. It is much faster and reliable for this purpose than the Programmaple Logic fabric. One can route clocking signals through the fabric, however this would add a large deal of delay over a large span of PL real estate and one must bear in mind that this might introduce unacceptable timing delays that could introduce timing rule violations. Moreover, when such signals are routed through the fabric, they can pick up a lot of noise and temporal variance (jitter) along the way, wich may not be acceptable depending on the particular design they are used in.

FGPAs offer a set of specialized clocking components (usually crystal oscillators) for general use in synchronous design wherever the designer sees fit. However they usually have a fixed output of frequencies, and if they are programmable, they are so over a limited range. When wider frequency ranges are desired, this is when the various frequency synthesis solutions come into play, most notably the pll.

Pll stands for Phase Locked Loop. It is a control element that generates an output signal with a phase related to the input signal. The simplest pll consists of a Variable Frequency Oscillator, a Phase Detector, and a Feedback Loop. Actually, keeping the phases locked means that the input frequency is the same as the output frequency, but the pll can also synthesize a multiple or even a fraction of an input frequency. This and the property that it removes the noise from an input signal, make it’s use very common as an intermediary stage between the clock and the clocks target. There exist other clock modification options, like MMCMs, clock buffers with dividers and so on, but they will not be of any concern in the context of this study, as they were not uesd.

Dsp stants for Digital Signal Processor. A DSP is a type of processor with a very limited repertoire of instructions (unlike a microprocessor) and a couple of input and output pipeline stages, that is usually used for a very repetitive and computationally intensive task, like the continuous calculation of digital filters. DSPs are very power efficient in comparison to general purpose processors, and FPGAs can have hundreds of them (or more) in their package. Their advantage is that they are designed to be chained together in parallel or in a cascade, so they offer a scalable solution for cases where lots of arithmetic is needed. They usually natively operate on Signed and Unsigned Integer types, not floating point numbers. If one wishes sub radix precision, fixed or floating point behavior can be emulated in a DSP, albeit at the cost of increased clock cycles.

1.5 VHDL

The language used for this project is VHDL [14, 15]. HDL stands for hardware description language, and V stands for VHSIC, which is itself an acronym for Very High Speed Integrated Circuits. Along with Verilog it is one of the two most prevalent languages for hardware description. It is not a programming language, in the sence that most of the statements it uses are meant to infer netlists and circuit schematics. If the statements are too vage, or too complicated, or require a huge amount of resources that the FPGA doesnt contain, implementation will fail.

Unless enveloped by a process block, VHDL code describes concurrent behavior:

```
A <= B;
C(1) <= C(7) NOT A(5);
```

Those two VHDL statements describe some assignments and a logic operation that can be thought of happening in one step, unlike if they were C language statements. As such any parallel behavior can be laid down in any order whatsoever, unless contained in a process block. A process statement in VHDL is used to describe sequential behavior. For example the two statments

```
R <= '0';
R <= '1';
```

will execute sequentially within the process block (but will case an inference of a net with multiple drivers outside the block, which will cause the design to fail).

VHDL has some advantages when in comes to using it as a tool to infer hardware. It enforces strict rules, in particular it is strongly typed, it is non-permissive and as a result and less error-prone than more less rigorous languages. It supports the creation and the use of Packages, that contain pre-built functionality. It allows for the creation of custom types, something very desirable when attempting to extend the languages functionality. It supports enumerated types, which can be used to make a design much more legible.

It is easy to infer basic structures in the FPGA, like RAM, DSP blocks, Latches, Registers, Pipelines, FIFOs and Lookup Tables by not being explicit in the use, say, of an IP component like a DSP48E2 macro (a Xilinx DSP block parametrisation macro), but by structuring the code in such a way as to guide the synthesizer towards a reasonable selection of primitives. For example, a chain of signal assignments inside a process will infer a FIFO. This is important as it can greatly

ease the development process by letting the synthesizer figure out functionality that can be burdensome to code explicitly, and sometimes non portable accross different devices.

In VHDL, the functionality of a component is represented by an ENTITY, which is a formal way of describing Input and Output Ports and Buffers (and perhaps some paramers through the use of Generic declarations), and an architecture that describes the components behavior. A component is written in this way and can later be incorporated in a larger design through instantiation.

A design is usually accompanied by a set of files called Constraint Files. Although not written in VHDL (except for some directives that can be incorporated in VHDL code), they need to be mentioned here, as no real Hardware Design exists separately from constraints.

Constraints inform the design tools various important aspects of the design that may are not apparent just from the HDL code. They serve to inform of various requirements that must be met when implementing a piece of logic, that can be relevant at any point in the design flow. They could be physical constraints, for example where in the IC will the described logic be implemented (could be a general area or a set of very specific elements (like DSPs or CLBs or even particular CLB elements), how much die area the design is alloted, wether pieces of logic can intermingle in the chip, define the pinning and routing of several components explicitly and so on. They could be timing constraints, that serve to inform the design tools of various timing relations (clock periods, generated clocks setup and hold analysis, IO delay, and so on). They can even guide the design tools towards a particular primitive when deciding how to implement a piece of logic (for example to use Distributed RAM or Block RAM when implementing memory).

Two types of constraint files were used in this study: XDC constraints and TCL scpipts [32, 33]. XDC constraints are a combination of industry standard SDC (Synopsis Design Constraints) and Xilinx proprietary physical constraints. They follow the semantics of the Tool Command Language (TCL) programming language [17, 18]. TCL scripts can be used like XDC constraints, but XDC constraints are managed by the design tools, whereas TCL tools are unmanaged (there is no automatic writeback of any additional directives in the script as is often the case in XDC files). Usually they are used to accomplish some tedious tast such as iterating over a list of elements that belong to a component and placing them. this is a job best suited to a general programming language like TCL.

2. Design of the TRNG

This chapter explains the behavior of the basic entity chosed for provision of entropy, the Ring Oscillator. Before going into details to describe the TRNG's form in VHDL, a set of abstract High Level Specifications were laid down to guide the project. Next, some important concepts that can be used to examine random number streams are introduced. Finally, the chapter proceeds into the first set of experiments that serve to defend the selection of the Ring Oscillator as a primary source of entropy, and guide the parameter selection of the subsequent designs.

2.1 Ring Oscillators as Entropy Sources

A ring oscillator is a digital circuit that consists of a loop of an odd number of chained NOT gates (inverters) like the one displayed in the following picture. The requirement that the inverters be odd is the necessary condition for oscillation.

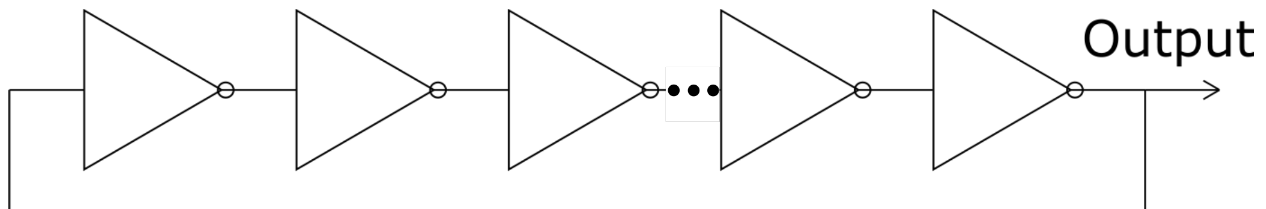


Figure 2.1: a ring oscillator

At any point in the loop the output can be taken by drawing out a wire as shown. This circuit is logically unstable: from the input of the first inverter to the output of the last, a signal undergoes an odd number of inversions. This means that the output of the last inverter is the opposite of the input of the first, but because of the loop topology, it is also the same as the input of the first. Although this is paradoxical when analyzed at an abstract level, wherein no delays between signals are considered, when using a more realistic model, where delays are taken into consideration, it can be understood that the nature of this circuit is to oscillate. As a result, one can observe a stream of HIGHs and LOWs (1s and 0s) at the output. Once the value at the first inverter input terminal runs the length of the Ring Oscillator twice, the output sequence repeats. This is illustrated in the following diagram.

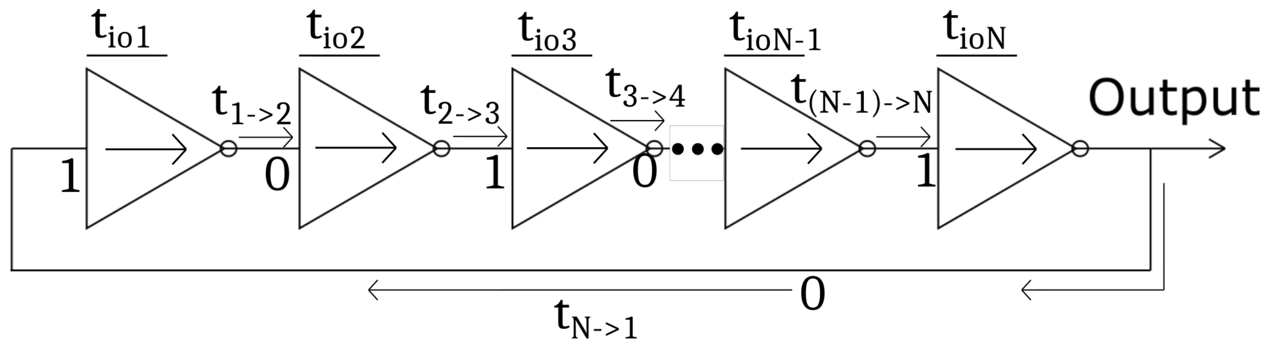


Figure 2.2: the oscillator's timing analysis

The diagram above shows a series of inverters in a ring. If one ignores the ring topology and considers only the inverter chain, this lies at equilibrium, because alternating 1's and 0's between the inverters do not cause any switching to take place over time in the chain. However this doesn't account for the looping wire that ties the final output to the first input. This causes the final 0 to travel back down to the first input, which, after a delay $t_{N->1}$, will reach the first inverters input, and cause a cascade of the inverse values to manifest down the chain of inverters. It should be noted that N is an odd number (condition for oscillation).

The switching time (time the output takes to alternate from a '1' to a '0' or vice versa) is expressed as a function of the delays of the interconnects between inverters ($t_{a->b}$) and the delays of the inversion transformations themselves ($t_{io\#}$) as such: $t_{switch} = t_{io1} + t_{1->2} + t_{io2} + t_{2->3} + ... + t_{ioN} + t_{N->1}$. Then the period of oscillation is $t_{oscillation} = 2t_{switch}$.

After t_{switch} the inverse equilibrium will have settled down the inverter chain, and a travelling 1 will have just reached the first inverter input. After a second switching interval (t_{switch}), the Ring Oscillator will be back at it's former state, hence the expression $t_{oscillation} = 2t_{switch}$ in the diagram above.

An example is shown below with actual numbers. In the following diagram on can se the propagation of a signal through a 3 inverter ring, all the way back to the point of feedback to the first inverters input. The switching time is calculated.

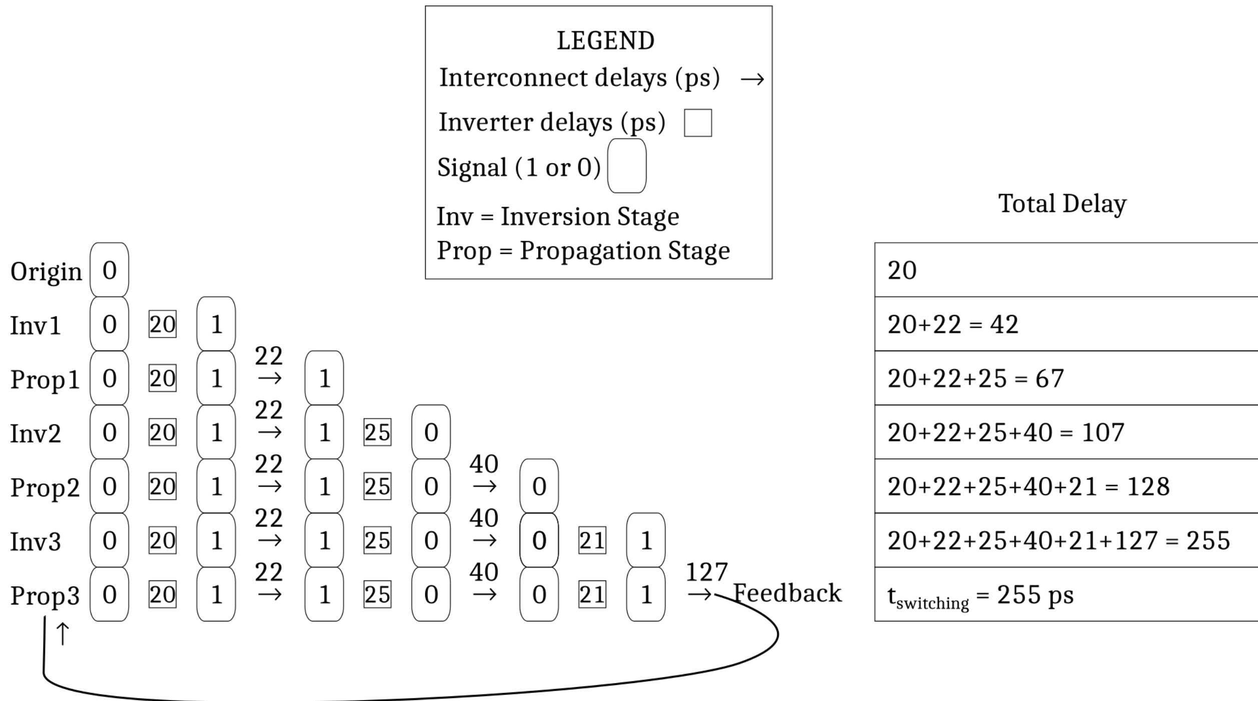
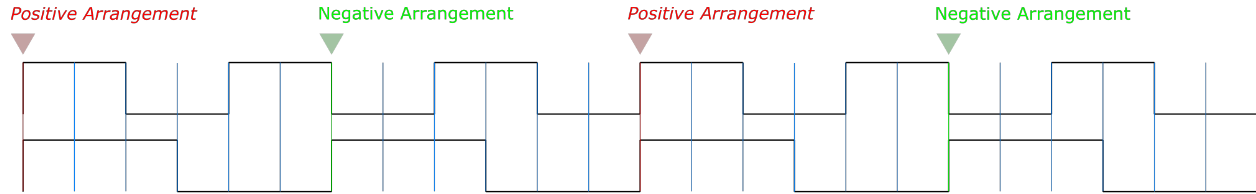


Figure 2.3: Timing of a 3 inverter Ring

It is not immediately obvious where exactly the randomness lies in this contraption: a repeating sequence is the worst kind of “random” that can expected, becoming totally predictable after being observed for some whole periods. The answer is that the true randomness exhibited by this design lies in the sampling of the output, or rather the degree of uncertainty this circuit introduces to whether the output will be gated at a HIGH or a LOW. This is due to, chiefly, three reasons, namely dissonance between sampler and Ring Oscillator output (separate time domains), a degree of non-periodicity inherent to Ring Oscillators, called Jitter in the temporal domain (and is also known as Phase Noise in the frequency domain), and lastly, possible metastabie states occuring during certain sample acquisitions. Moreover, ring oscillator behavior varies over local Process, Voltage and Temperature conditions. The aforementioned terms are explained in more detail in the segment that follows.

In the following picture 2 dissonant signals are presented. One of them could be the output of a ring oscillator. The other would be the clock that latches, or triggers, the sampling element (a Latch or a Flip-Flop). Here it is assumed that sampling is triggered via a rising clock edge.



Ring Oscillator: T = 2 Units
Sampler: T = 3 Units

Figure 2.4: an output and a sampling clock with untuned frequencies

As can be seen, the periods of the two waveforms are not tuned, meaning they do not possess a common integer divisor. More common is the scenario with a preexisting phase differedence as shown in the picture below. The phase difference is evident due to the edge misallignment (indicated by the Δφ bar).

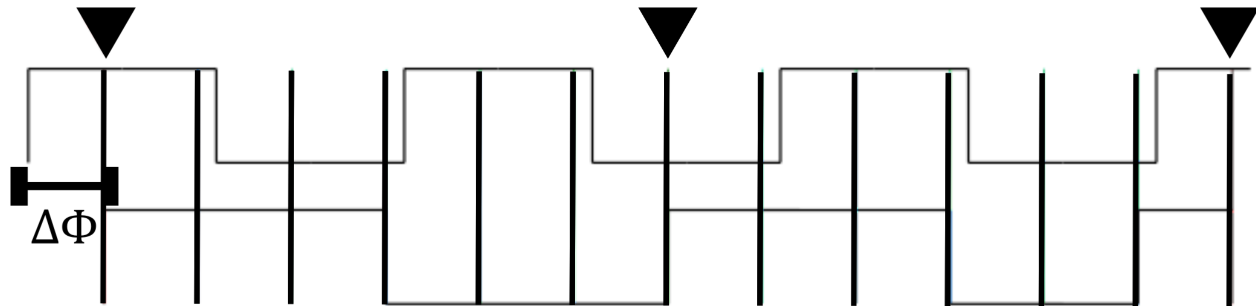


Figure 2.5: output and sampling clock having a phase difference Δφ

In the instance shown here, the periods are coprime, but even more likely in the real world is the case where no common unit of time can be found amongst them, as happens with numbers such as, say, 1 and e, or generally with numbers with an irrational ratio. In the above shown situation, the sampler finds itself once in phase with the output, and once in antiphase (what here is reffered to as positive and negative arrangements, respectively).

Still, the sample sequence acquired in this case is not really different than the previous scenario.However for various relative lengths, as for instance 3 and 7, pattern arise such as

110110110... or *001001001...*

depending on the relative phase between the clock and the oscillator output. In this case, sampling phases between 0 and π yield the first series, while phases between π and 2π yield the second. The following pictures demonstrate this exact case visually. The example assumes perfect ring oscillator periodicity. This also leads to an imbalance to the number of 1's relative to the number of 0's in the output (Bias).

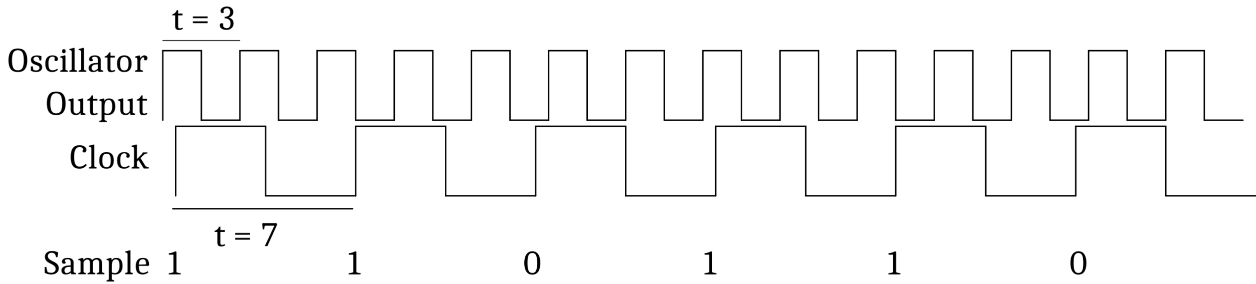


Figure 2.6: sampled sequence for relative clock phase between 0 and π

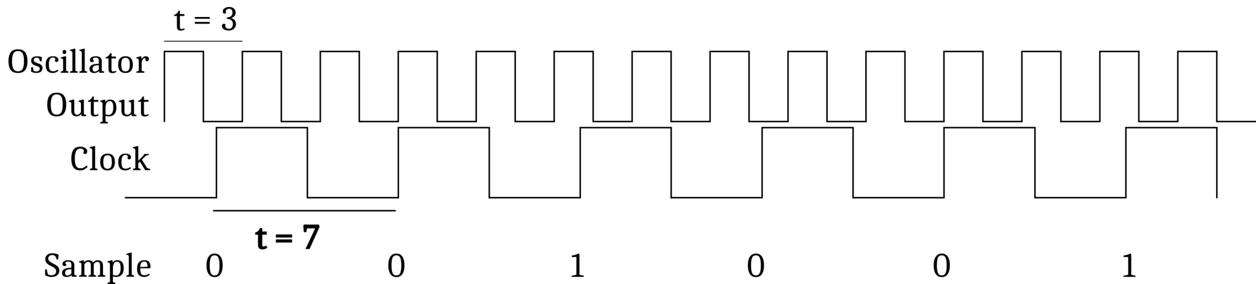


Figure 2.7: sampled sequence for relative clock phase between π and 2π

Concerning the irrational relation case, there is no need for further analysis. In reality it doesn't make a whole lot of difference because the differences in phase after many cycles become so small, it is as if the signals are in phase or antiphase (that is to say, it can be considered to be practically equivalent to the rational case). More importantly, the deviation from perfect periodicity that Ring Oscillators exhibit (see the section on jitter) will quickly corrupt any phase relation that the clock might have had in relation to the oscillator initially, as will it also destroy any practical evidence of irrationality between the two signals.

Dissonance is not a particularly potent source of entropy, not unless it is combined with other phenomena. If not, then the periodic patterns that arise over time prohibitit it's use as an Entropy Source.

The treatment of the Ring Oscillator has thus far assumed strictly alternating adjacent signals in the Ring Oscillator - except for the single travelling inversion, wich implies precisely two adjacent noninverse signals. In actuality, this may not be the case for an implemented circuit. This would depend on inverter input initialization. If there exist more than two adjacent nonalternating signals the Ring Oscillator exhibits an composite state. It can be approximately though of as two independent ripples traveling in the same ring oscillator, and one might assume that this would lead to the output switching to comprise sub-frequencies, or be, in other words, composite, now encompassing superfrequencies. It is true though that there may be some coupling between the switching of the two running ripples. The following diagram visualises a composite state.

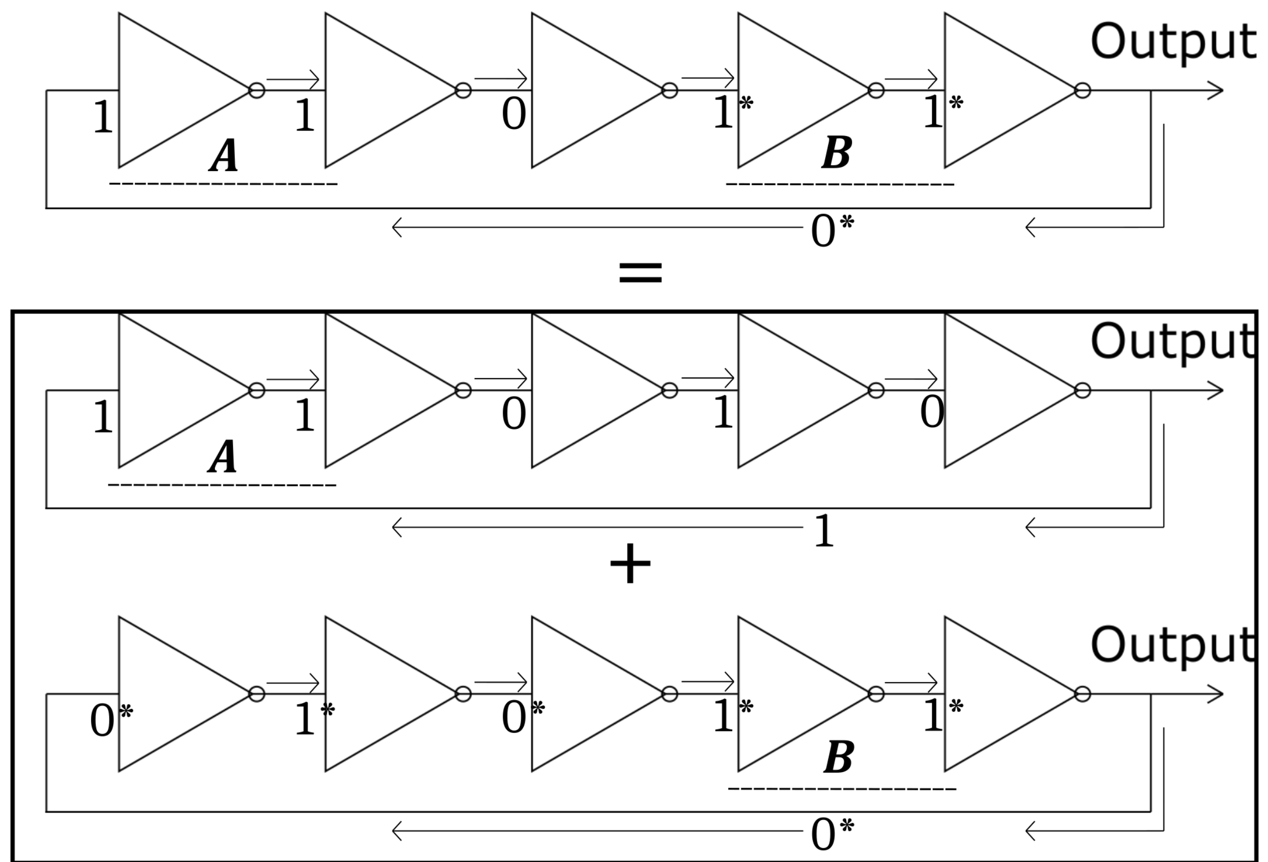


Figure 2.8: A composite state, formed by two pairs of noninverted signals.

Here a composite state is shown that can be though of as 2 largely independent base states (respectively with and without an asterisk) running through the same ring oscillator.

Jitter means deviation from strict periodicity of a presumably periodic signal [29, 34]. It can be categorised as Deterministic and Non-Deterministic. Deterministic Jitter is predictable, and can usually be traced to some periodic relation or some periodic phenomenon interfering with the signal under observation. For example, in a Ring Oscillator inside an FPGA, it could be the periodic switching from a Circuit Board's power supply. Deterministic Jitter is of little interest in creating Truly Random Numbers, due to it's not truly random nature.

Non-Deterministic Jitter, however, is the most potent source of entropy that a Ring Oscillator possesses. In other areas, when it arises, it is generally undesired or even considered pernicious (for example in communication links), nevertheless in this particular case it comprises the most important asset a Ring Oscillator has to yield raw entropy output. Non Deterministic Jitter generally follows a Gaussian Distribution, due to a very large number of underlying contributing factors that form it, as per the central limit theorem [3]. The Jitter expressed here is generally known as Cycle-to-Cycle Jitter, and is the difference between periods of the Ring Oscillator.

The causes for this sort of Jitter are many, and include elctronic noise that causes unpredictable behavior in the components that are used to build the inverters, such as gates for example, or in this case Look-Up Tables, and can cause subtle variations in output resolution times, due to noise induced variability an the gate/lookup table's response time, and variation of the exact time that an input signal's state is considered to belong to one of the two valid logic states, again through noise induced variation in the signal, and variations to the power supply of the inverters, that in turn can cause varying output response times.

In practice, the observed jitter is a combination of the effects of external periodic or predictable phenomena and truly random noise. The former usually yields distinct peaks in the Jitter distribution, while the latter shapes those peaks to the Shape of a Gaussian Distribution [29]. Figure 2.9 demonstrates a hypothetical total jitter curve of a system, that consists of 2 gaussian peaks. The gaussian bells are the jitter's random content. The Peak separation is deterministic.

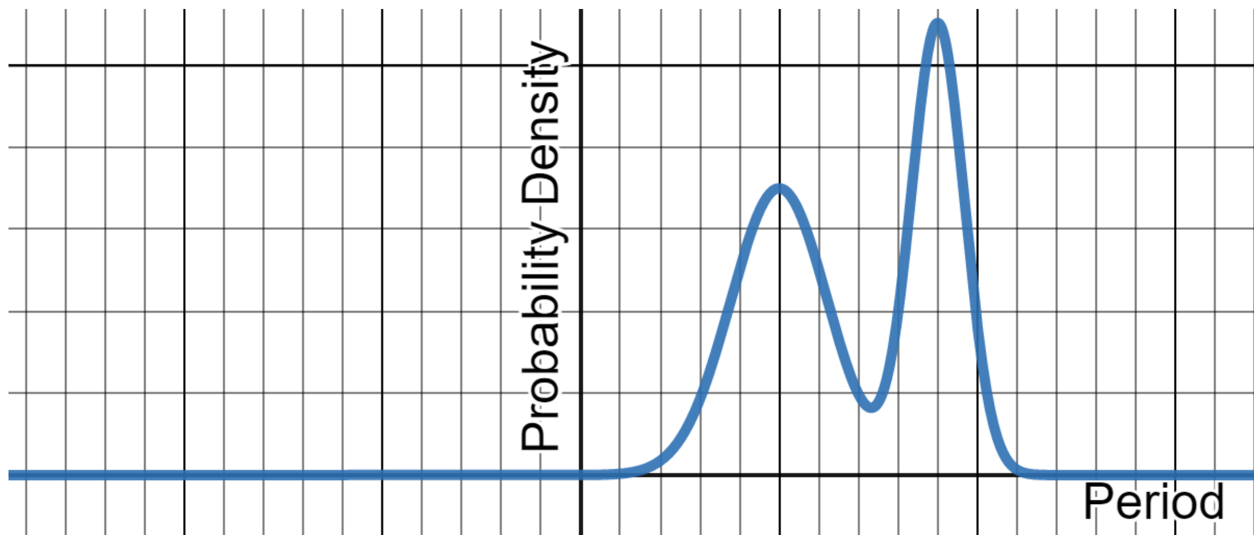


Figure 2.9: Total Jitter as a sum of Random and Deterministic components

Metastability in logic circuits is an unwanted phenomenon that can cause circuit behavior unpredictability and reduced performance [21, 22, 23, 24]. Metastability occurs both in Latches and Flip-Flops when setup and hold time violations occur. Edge triggerd Flip-Flops are relevant here because that's how the sampler was implemented. Setup time is the time needed for data to be stable in a valid level at a data input terminal before the clock trigger in a Flip-Flop. Hold time is the time this data needs to remain stable and valid after the clock trigger.

Flip-Flops have two stable states (they are also called Bi-Stable Multivibrators for that reason). What happens when data at a Flip-Flop's D terminal isn't stable for long enough (or lies in a non-resolvable level) is that the Flip-Flop enters an equally balanced state (an unstable equilibrium) from where it can later resolve in either state. Slight perturbations of the balance of the unstable equilibrium, such as those produced by electronic noise, can cause the output of the Flip-Flop to resolve either way. The issue is that metastability resolution takes longer to resolve than when the Flip-Flop is set up and held properly (normal operation), and sometimes in can even oscillate between the states, as shown in Figure 2.10. As a result, if there is a piece of finely timed logic dependent on the Flip-Flop's output, a metastable situation can cascade into a much larger failure in the form of incorrect operation of whole synthesized components and even more metastability that cascades further down the logic.

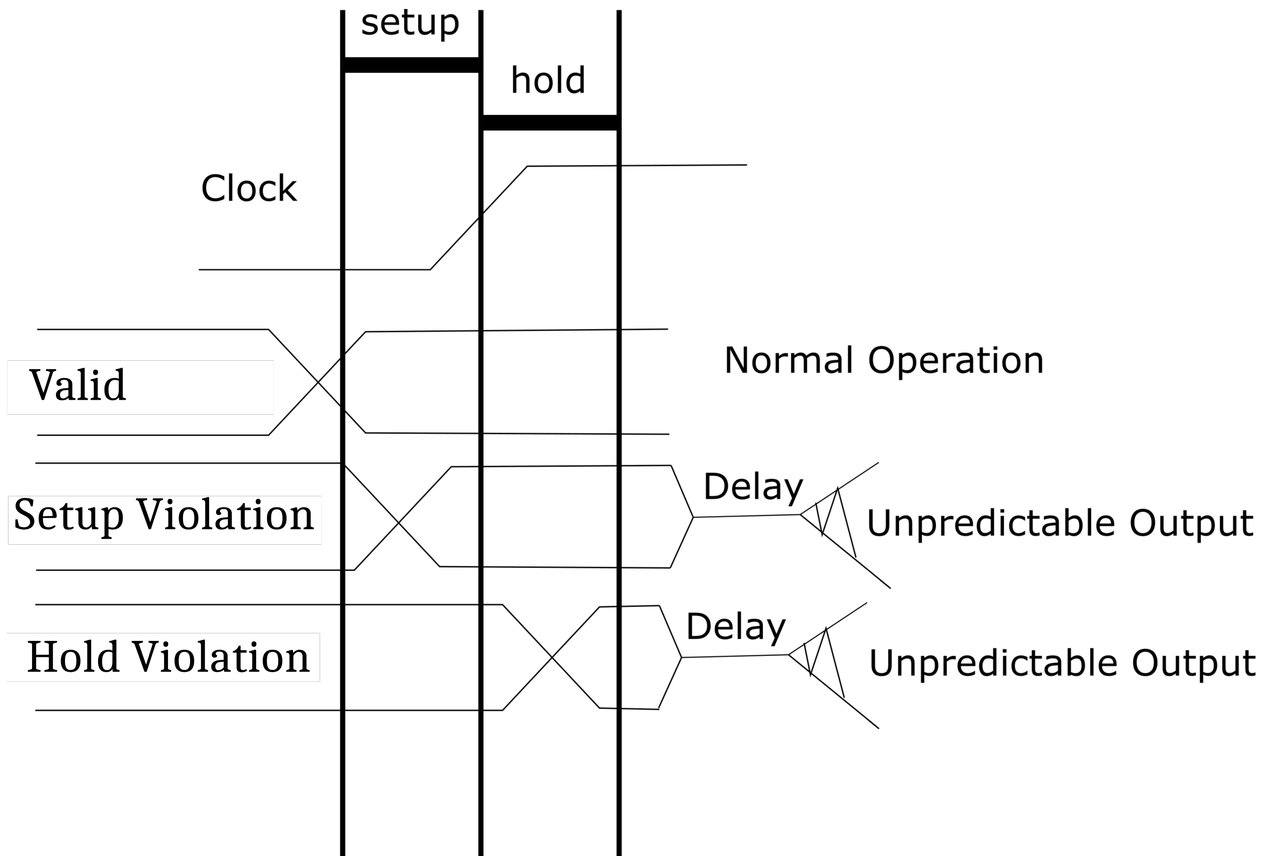


Figure 2.10: Normal vs Metastable Operation of a Flip Flop

Metastability is relevant here in the sense that when the sampler's first Flip-Flop enters a metastable state (by accident due to the inherent asynchrony between the sampler and the TRNG output), this unpredictability of the output signal resolution can be thought of as an extra Source of Entropy. Under normal circumstances, Metastability needs to be eradicated through the use of synchronizers, like register FIFOs, Dual Clock FIFOs and so forth.

To express the degree to witch metastability is involved in this situation, the Mean Time Between Failures (MBTF) is used as a measure. Mean time between failures is dependent upon the jitter of the TRNG output, but also highly dependent of the sampling frequency, with higher sampling frequencies leading to more frequent failures of correct Flip-Flop operation. The equation to calculate MBTF is MBTF = $(f_{input}*f_{clk}*T_{cw})^{-1}$, where f_{input} the input frequency, f_{clk} the clock frequency and T_{cw} the critical time window which is just Tsetup + Thold of the component [21]. When analysing MBTF of two or more components in series, their characteristic average metastability resolution time comes into play.

When the three above basic sources of entropy are combined in a tigltly packed and complex system, with increased surface area for them to act on, what is produced is an evolution in time much like that of a triple pendulum, or

turbulence in a fluid, namely chaotic behavior and quick divergence from projected behavior. This is the true power of the design that was selected for the TRNG. Multiple variables with a significant degree of coupling between them produce a system that cannot be thought of as being described by its parts, but rather the part being derived from the whole. that for example, switching causes power dissipation, this affects the local and global voltage at the inverters's power supply, and inductive phenomena cause this effect to propagate to other inverters or even other ring oscillators, that in turn introduce extra variability in the switching frequency and the chip's voltage and temperature, and so on, and so forth.

Therefore, it might be preferable to have a reasonably tight packing of ring oscillators to create the necessary conditions for the formation of chaotic behavior. This was indeed studied and evidence was gathered that supports this assumption.

It will be shown that ring oscillator behavior, in general, varies over PVT. PVT is shorthand for Process, Voltage and Temperature. Variance over PVT means dependence of Ring parameters (like Ring frequency) on the P, V, and T parameters. Namely, variance over P means differentiation of Ring Oscillator behavior due to innate semiconductor process differences, like the fact that some silicon regions are faster than others on the same die. Variance over V means that if global or local voltage conditions change, so does the Ring Oscillator behavior. Variance over T means Ring Oscillator parameters are dependent on temperature.

2.2 The TRNG's Features

Following are some general specifications that were used to guide the design of the TRNG device.

The TRNG consists of an array of ring oscillators that, when sampled, yields a single raw output bit per Ring Oscillator at a time. This is illustrated in Figure 2.11.

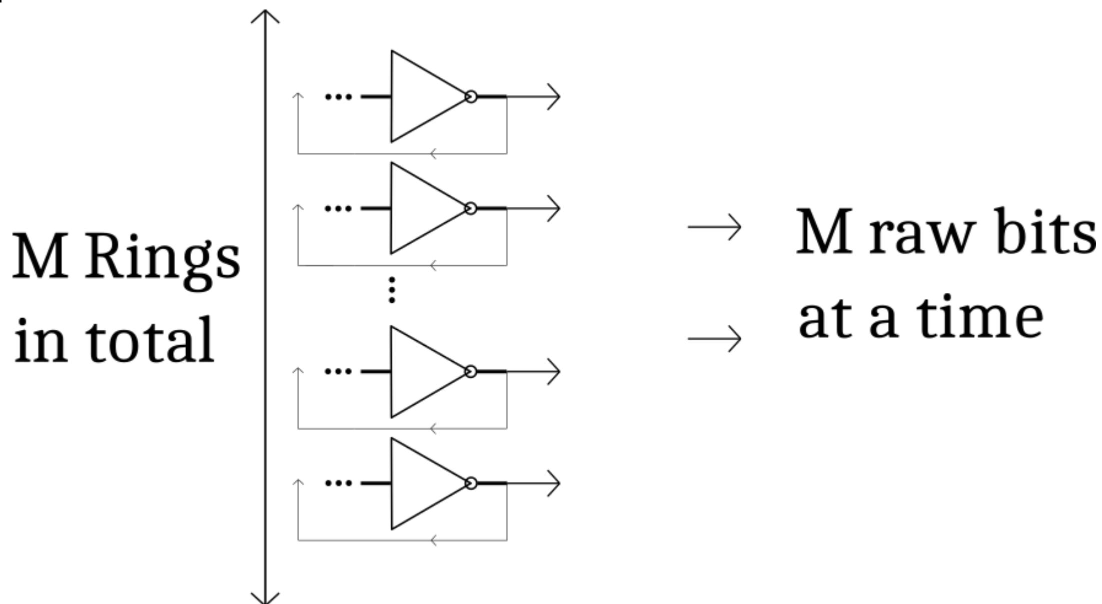


Figure 2.11: general entropy source configuration

The bits of entropy provided by the TRNG are 16 (a word per sample).

They are either raw bits, directly sampled from the ring oscillators' output, or post processed, if a fortification of entropy is deemed desirable. The latter case, implies some sort of reduction. The raw bits can either be reduced laterally (meaning reduction of bitness from more than 16 down to 16) or longitudinally (lentghwise, reduction of data rate) in a post processing stage.

This has implications in the bitness of the raw output stream, meaning that if the lateral solution is employed, possibly 32 raw bits will be necessary (32 Ring Oscillators, for a 2 -> 1 reduction wich is the most common in post processing). If, however, the longitudinal solution is employed, then 16 Ring Oscillators will yield 16 post-processed bits of entropy, albeit at half the rate. Post-Processing is employed due to the fact that raw TRNG output bits may not have adequate enough entropy characteristics. The above are illustrated in the figure 2.12.

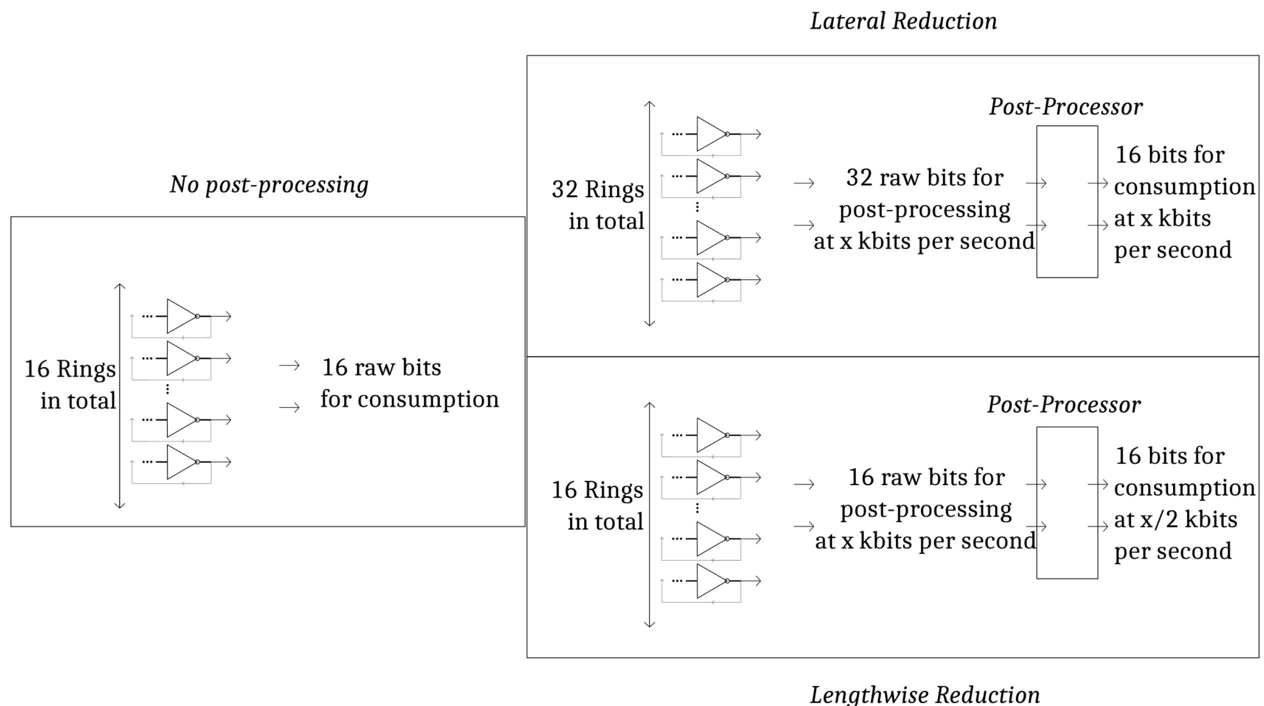


Figure 2.12: the different options regarding the treatment of the output stream (the target is 16 bits at the output stage)

The final output should not be metastable, as any following components that make use of it will potentially be negatively affected. Although metastability might be a part of the output stream's first sampling stage, some means of remediating it must be employed, in order to present stable data to consumers.

The samping rate should be larger than the Ring Oscillator's switching frequency and be reprogrammable. This is to allow for the manifestation of Jitter, that will be useful in building up the entropy at the sampler output. Ideally, some sort of control over the sampling interval is desirable.

The Ring Oscillators should lie at a state without composite states. The reason for this is that, firstly, composite states are not seen as adding substantial entropy to the system. Except for a possible electrical coupling, they run in the same physical oscillator. Secondly, composite states can make preliminary analysis tricky, by creating unclear oscilloscope traces and frequency counts.

More importantly, composite states can become corrupt and vanish over time. So a design cannot be reliably based upon them, barring measures such as forced periodic reinitialization of the rings, composite states are considered volatile. Such measures go against the doctrine of stable TRNG operation accross environmental conditions and without intervention. Composite state corruption is

shown in the following picture.

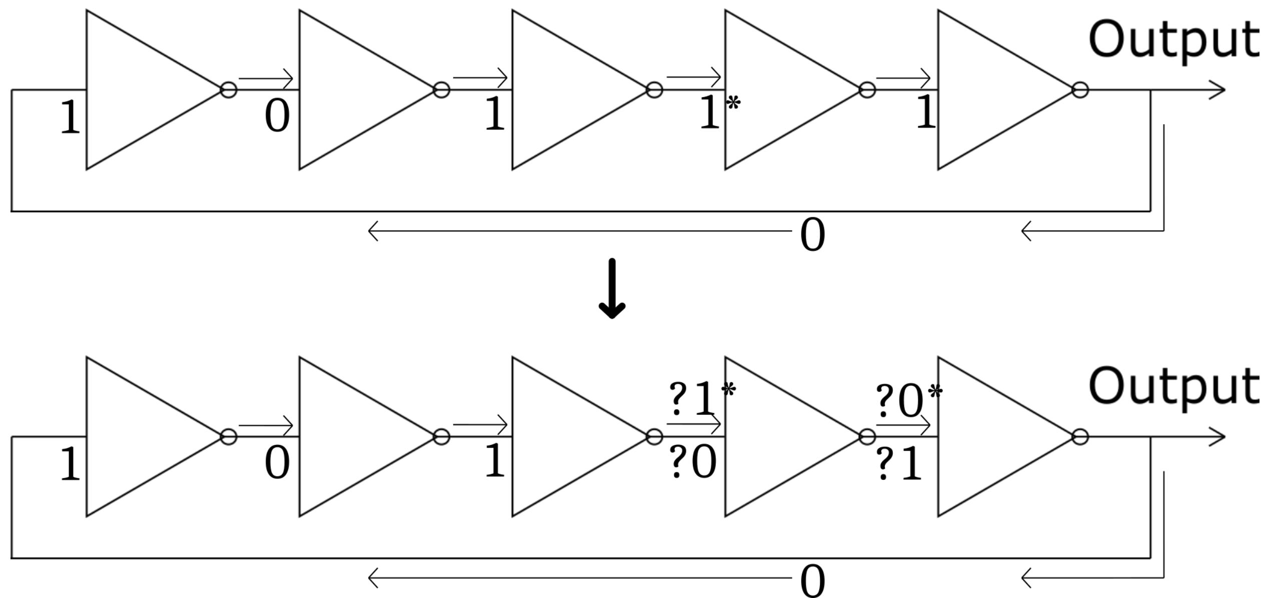


Figure 2.13: Composite State Corruption

Corruption happens because of data races. A data race is a condition where multiple entities access the same piece of data, and at least one of them attempts to modify it. The above picture uses a 5-Inverter ring oscillator as an example to demonstrate this data race. The data in question is the signal between the 3rd and 4th inverters (the ’1’ with the asterisk). The modification is attempted by the 3rd inverter, whereas the reader is the 4th, that uses this signal as an input. It could be that the 3rd inverter flips the data to a ’0’ before th 4th has enough time to react to the ’1’ that was present. Subsequently the next ring signal ‘the rightmost red ’0’) wont change. If it does have enough time to react, then the rightmost sigal will switch to a ’0’. There is no certainty that the middile 1 will survive, and the composite state may vanish.

The TRNG should also have some functionality to allow for halting it and starting it at will. This is to allow controlling power consumption. When the TRNG has not been used in a long time, it may be parked to reduce power consumption in the IC. If at a later stage it’s input is requested, it can be unparked and spun up to provide a stream of entropy.

As for the data rate of the TRNG, this is dynamically adjusted, under the condition that the stream of numbers obtained is adequately random in nature. These range from few KBPS (kilobytes per second) to perhaps multiple MBPS. This is something that can be compensated for by scaling up the design, and varying the sampling rate.

The ring oscillator design was first be tested in a simple manner by drawing some signals to the outside world on the general purpose inputs and outputs of the development board. Then, using an oscilloscope, the intrinsic parameters of the underlying silicon were measured (mostly oscillator frequency as a function of length). Those parameters can be used in a reiterated design to produce the final device. The final device has to be evaluated using statistical testing, and communicate digitally though some sort of link with a PC to report the results of those tests.

2.3 Experimentation

A general purpose FPGA input/output port group is used to measure the Ring's behavior. Fields of interest include, wether any switching is taking place, and if so, what characteristic frequency is to be expected by a Ring of L inverters. L is, as per the condition for oscillation previously mentioned, an odd number. With those questions in mind the first code for the Rings was developed. Following is a snippet of the code used to infer 3 large Rings.

```
entity longrings is
    Port (output : out std_logic_vector(2 downto 0);
          PARK   : in std_logic);
end longrings;

architecture Behavioral of longrings is

   signal ring2001 : std_logic_vector(2000 downto 0);
   signal ring3001 : std_logic_vector(3000 downto 0);
   signal ring4001 : std_logic_vector(4000 downto 0);

   attribute dont_touch : string;
   attribute dont_touch of ring2001 : signal is "TRUE";
   attribute dont_touch of ring3001 : signal is "TRUE";
   attribute dont_touch of ring4001 : signal is "TRUE";

   attribute allow_combinatorial_loops : string;
   attribute allow_combinatorial_loops of ring2001 : signal
   is "TRUE";
   attribute allow_combinatorial_loops of ring3001 : signal
   is "TRUE";
   attribute allow_combinatorial_loops of ring4001 : signal
   is "TRUE";
```

```
begin

      make_ring2001:
      for I in 2000 downto 1 generate
              ring2001(I) <= NOT ring2001(I-1);
              end generate make_ring2001;
              ring2001(0) <= PARK NOR ring2001(2000);

              make_ring3001:
              for I in 3000 downto 1 generate
              ring3001(I) <= NOT ring3001(I-1);
              end generate make_ring3001;
              ring3001(0) <= PARK NOR ring3001(3000);

              make_ring4001:
              for I in 4000 downto 1 generate
              ring4001(I) <= NOT ring4001(I-1);
              end generate make_ring4001;
              ring4001(0) <= PARK NOR ring4001(4000);

      output(0) <= ring2001(2000);
      output(1) <= ring3001(3000);
      output(2) <= ring4001(4000);

      end Behavioral;
```

PARK is used to halt the Ring oscillations. It's purpose is twofold though: it also serves to create a state of strictly alternating 1s and 0s in the loop, effectively initializing it free of composite states. So before taking the measurement, care must be taken to assert Park and initialize the Rings. Secondly, note the use of directives such as `attribute allow_combinatorial_loops of ring2001 : signal is “TRUE”;` If those directives are absent, the tools will complain about the existence of combinatorial timing loops (generally an error). Here, however, it is the purpose of the code to infer a loop, so this directive is necessary, else synthesis will fail. The `attribute dont_touch` serves to prohibit optimizing the consecutive inversions away, since the tools could figure that 2001 NOTs are logically the same as 1 NOT.

For the purposes of measurement, it was decided to use the J3 prototype header on the development board [5]. The design was constrained to include the header pins as it's ouput ports, while the PARK signal was constrained to an external GPIO DIP Switch, SW13.1. The relevant constraints are

```
set_property PACKAGE_PIN G15 [get_ports {output[2]}]
#PROTO 12 4001
set_property PACKAGE_PIN G14 [get_ports {output[1]}]
#PROTO 10 3001
set_property PACKAGE_PIN J14 [get_ports {output[0]}]
#PROTO 8 2001
set_property PACKAGE_PIN AK13 [get_ports {PARK}]
#SW13.1

set_property IOSTANDARD LVCMOS33 [get_ports {output[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {output[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {output[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {PARK}]
```

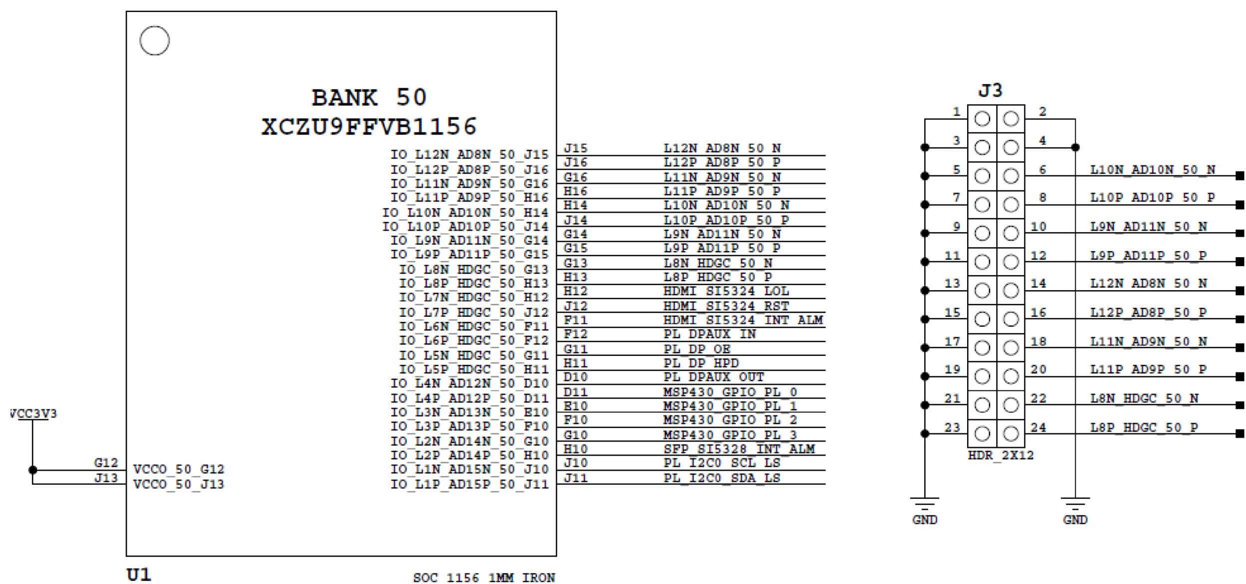
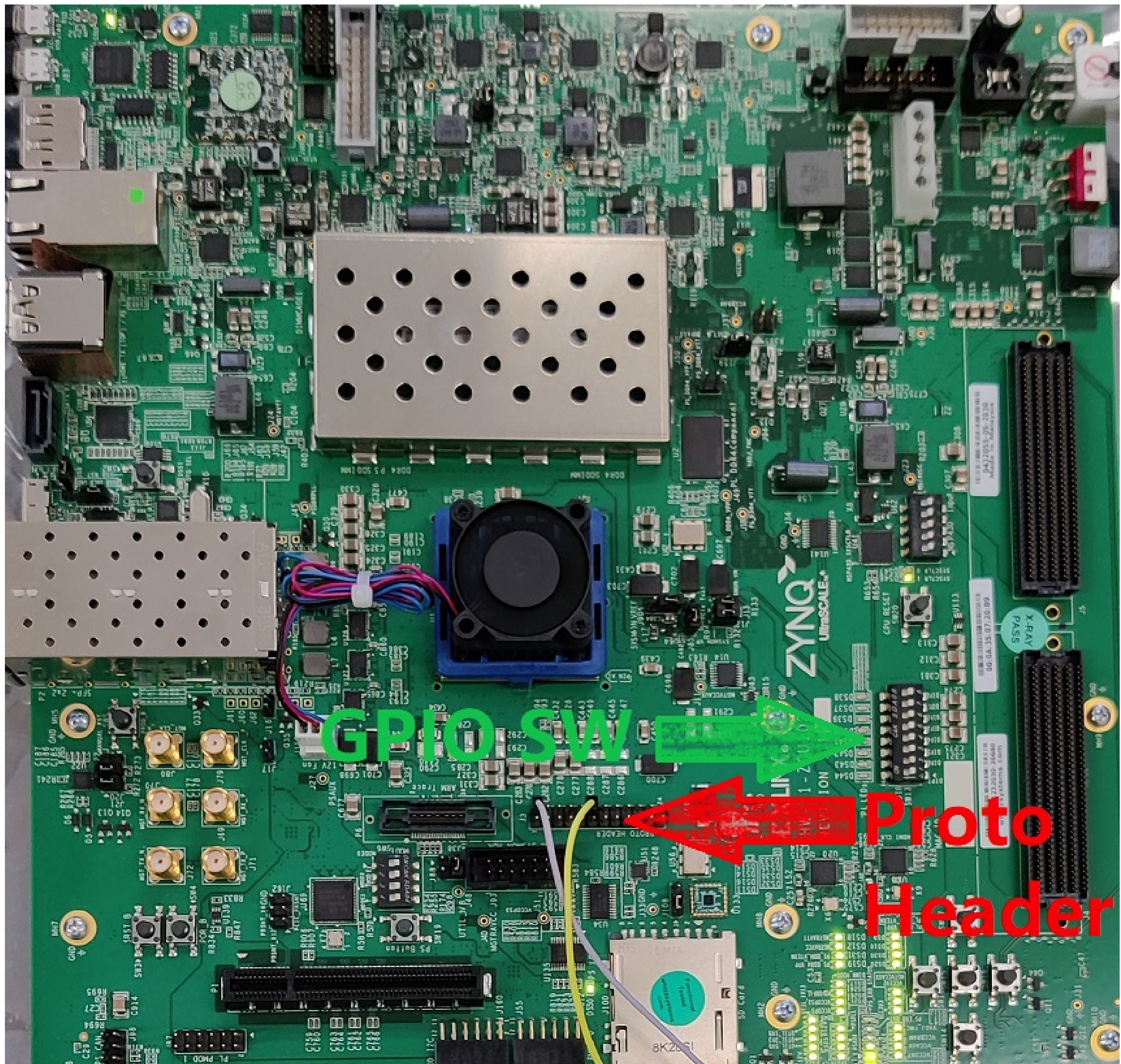
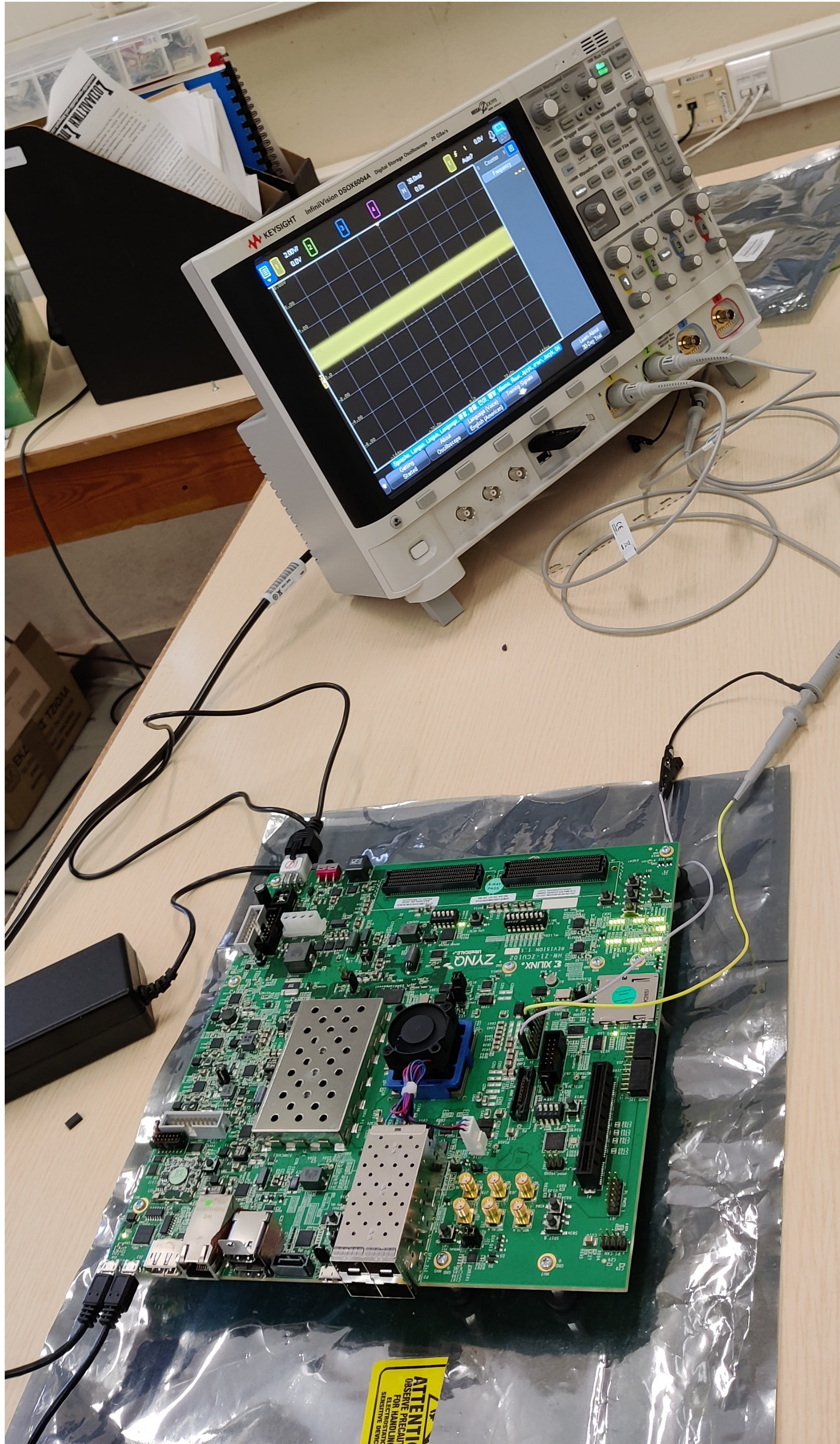


Figure 2.14: J3 PROTO header board pins, image adopted from [5]

Those elements are shown in picture 2.15 of the FPGA board used for this experiment. The oscilloscope's probes can be seen attached to the header pins. Picture 2.16 shows the oscilloscope used to take the measurements.

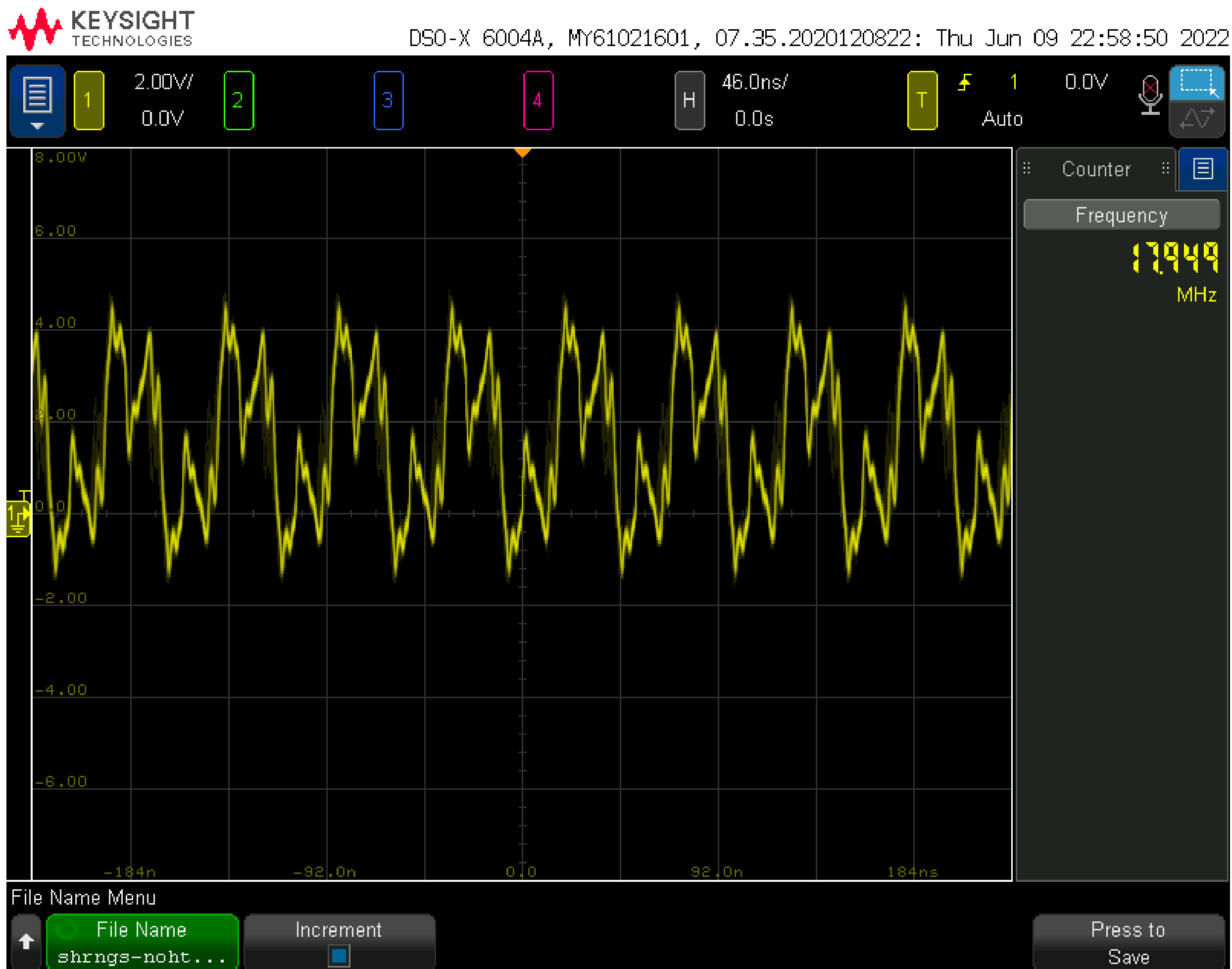


Picture 2.15: The ZCU102 evaluation board used for this study.

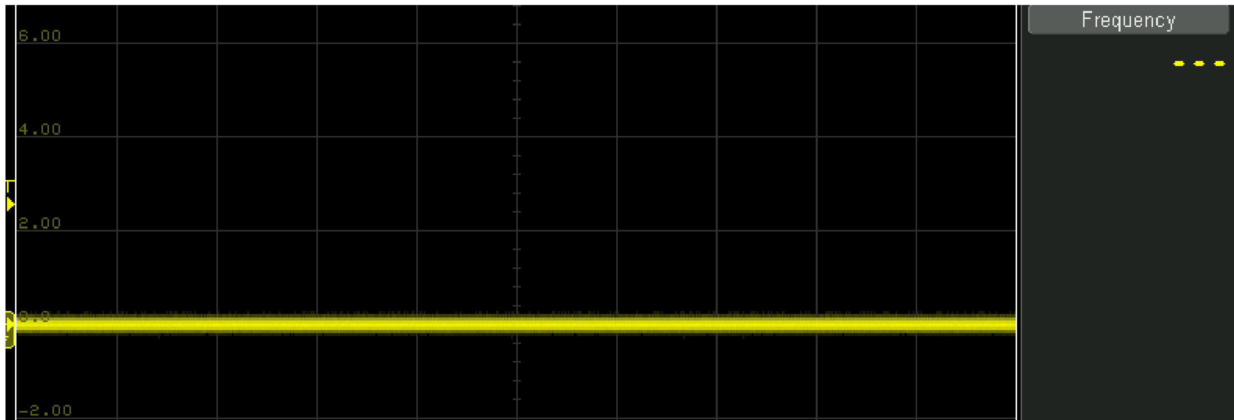


Picture 2.16: The experimental apparatus

The frequency measurements were obtained through the oscilloscope's built-in frequency counter. First a view was taken to validate the suspicion that the rings may start off improperly initialized. As can be seen in Figure 2.18, this is not the waveform expected from an LVDS33 switching (LVDS33 is the IO standard for the particular pin). This waveform implies multiple constituent frequencies, and makes the presence of composite evident. As previously discussed, composite states are undesirable, so their elimination was attempted shortly thereafter via the parking mechanism. Once PARK was asserted, the Ring's behavior was to cease oscillations, as was intended.

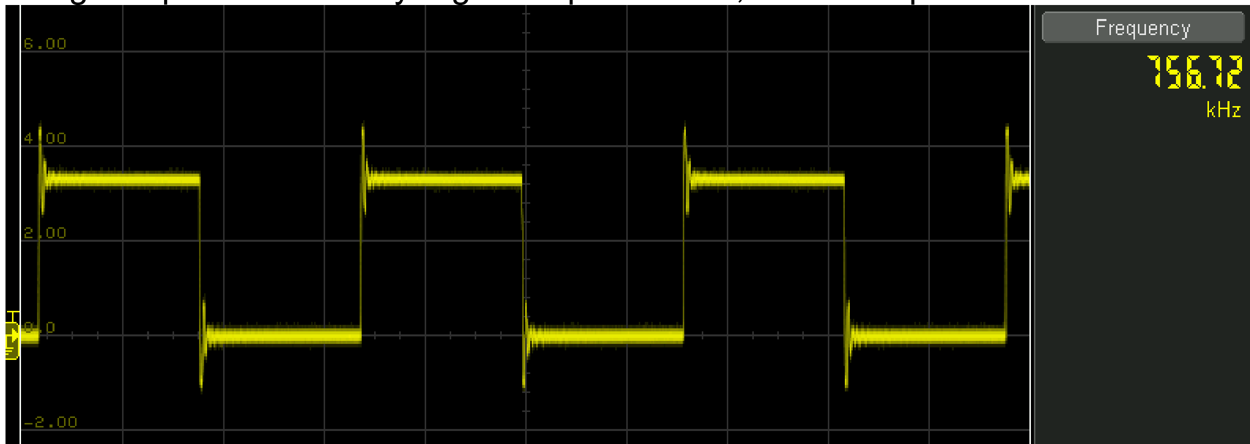


Picture 2.17: Complex oscillations in Ring3001 due to composite states

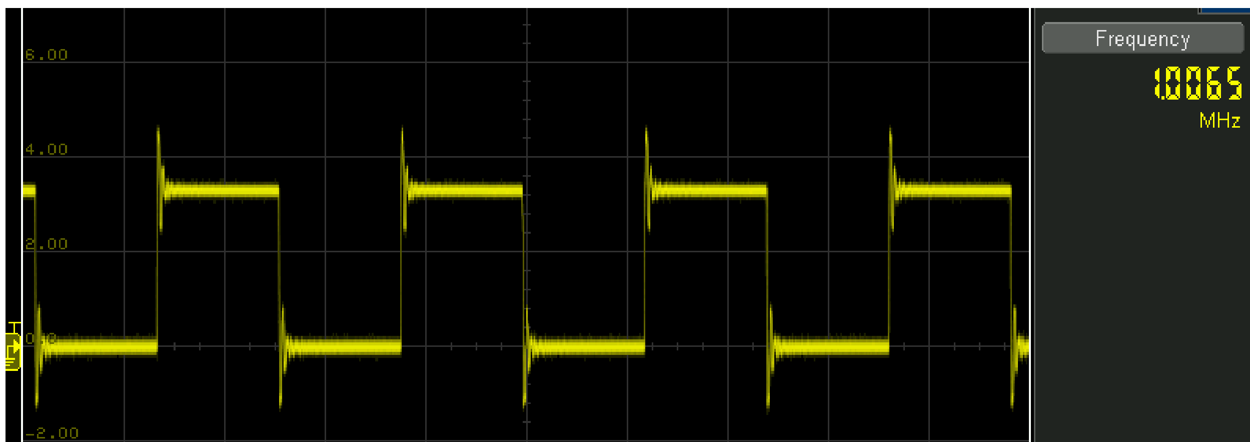


Picture 2.18: a Parked Ring, No Oscillations.

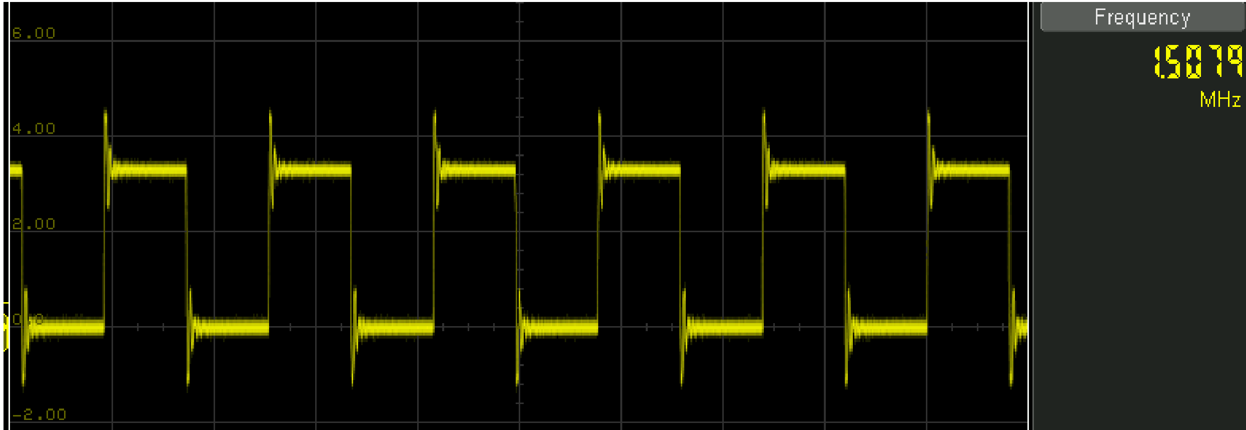
Following that, the Ring was released, and this time a properly initiallized chain of signals produced a very regular square wave, as was expected.



Picture 2.19: Properly Initialized Ring of length 4001



Picture 2.20: Properly Initialized Ring of length 3001



Picture 2.21: Properly Initialized Ring of length 2001

Revisiting Figure 2.2, $t_{oscillation}$ can be taken to be approximately proportional to the number of LUTs: An expression can be obtained that links the ring period to the length of the rings. By summing over all the inverters

$$t_{oscillation} = 2\Sigma(t_{interconnect} + t_{inverter}) = 2L(\langle t_{interconnect}\rangle + \langle t_{inverter}\rangle)$$

By grouping an inverter with it's preceeding interconnect, t_{PI} can be defined as

$t_{PI} = t_{inverter} + t_{interconnect}$,
so it follows that $t_{oscillation} = 2L\langle t_{PI}\rangle$

where L is the number of inverters in a ring (the ring "Length") and $\langle t_{PI}\rangle$ the average delay per inversion. By taking the inverse of this expression, the relation between frequency and ring length is

$F_{oscillation} = 1/(2L\cdot\langle t_{PI}\rangle) \Rightarrow F\cdot L = 1/2\langle t_{PI}\rangle$, and by defining $1/2\langle t_{PI}\rangle$ as $\langle F_{pi}\rangle$
$F\cdot L = \langle F_{pi}\rangle$

The following table shows the calculations for the implemented rings. Note that $\langle F_{pi}\rangle$ is dependent upon the routing scheme employed, as interconnect delays play a significant role in determining the Ring Oscillators' frequency.

Length (Inverters)	Switching Frequency (MHz)	$\langle F_{pi}\rangle$
4001	0.756720	3027.63672
3001	1.006500	3020.5065
2001	1.507900	3017.3079

Table 2.1:$\langle F_{pi}\rangle$ values for the long rings

Using the Vivado's synthesis as show in Figure 2.22, the inverters are implemented as specific LUT instances. A ring oscillator's (labelled ring 2001) NOT gates are synthesized as Lookup Tables.

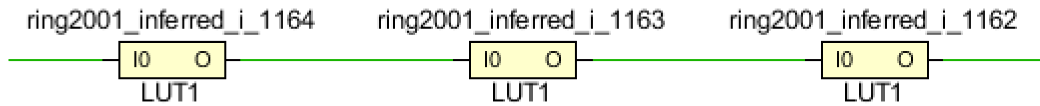


Figure 2.22: part of ring2001's inverter chain, Vivado synthesized schematic

I0	O=!I0
0	1
1	0

Figure 2.23: Truth Table for the LUTs

Following, a piece of code was authored to observe the behavior of relatively short rings. The generic code used allows for selecting the number of rings created easily, without repetition of the same template. The following experiment demonstrates variation of ring parameters over PVT.

Using a piece of selector logic the oscillators that would be multiplexed for output at the PROTO header could be selected. The number of rings and their lengths could now be specified though generic integer values, making the code more general and reusable. The code uses the NMILLS generic parameter which here is set to 16. The outputs are multiplexed in a 16 by 4 multiplexer.

```
entity ShortRings is
       Port (PARK : in std_logic;
             -Ring Selector
             RSelect : in std_logic_vector(1 downto 0);
             --Direct Output to the PROTO Header
             Pinlines : out std_logic_vector(3 downto 0);
end ShortRings;
                              .
                              .
                              .

--generate 4*4-bit mills (effectively a word)
word_mill : generic_mill
generic map (N => NMILLS)
```

```
  port map (out_to_whatever => mills_out, PARK => PARK);
  --select any mill at your discretion to send to the PROTO
header
  with RSelect select
  Pinlines <= mills_out(3 downto 0)   when "00",
                   mills_out(7 downto 4)   when "01",
                   mills_out(11 downto 8)  when "10",
                   mills_out(15 downto 12) when "11";

  (This is just a part of the code - The code for instances
  referenced in this chapter can be found in Appendix A)
```

The experiment demonstrates variance over Process and Voltage using a heater netowrk, that serves both as a heavy load to affect the boards power supply voltage, and as a means to change the temperature. A register FIFO with an alternating input was used as a heating element. A network of 15 long Register FIFOs was inferred in the FPGA. The following diagram presents a network of 4 heaters (instead of 16, for simplicity's sake).

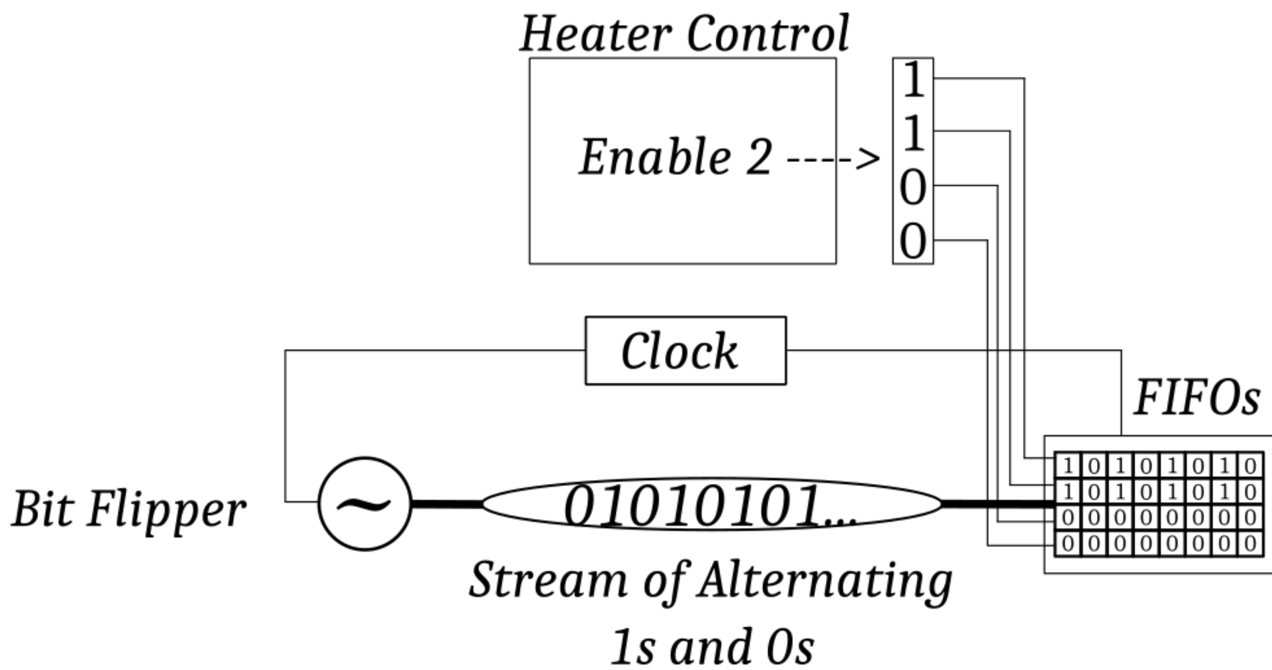


Figure 2.24: Heating Component Block Diagram

The Heater element was developed with an enable signal, and a selector for 16 different sequential heater activations (all heaters off, 1 on, 2 on, up to 15 on), in order to not only examine the effects of both temperature and voltage supply on ring frequency. The heater control block accepts an integer up to the total number of FIFOs and translates it to a vector with the same number of bits set to HIGH. These act as enable signals for the respective FIFOs, while the rest of them are disabled. As a result, there is some control over the power drawn. The FIFO's alternating input has the purpose of creating a series of interchanging 1s and 0s from the FIFO's head all the way down the FIFO's output. Once the FIFO is full, every register flips it's state to the opposite one at the clock edge and this switching generates heat. It also consumes a significant amount of power. This makes it a heavy load, which can cause small fluctuations to the silicon die's power supply input, by increasing the current.

Thus, It is expected to get a variation of the oscillation frequency at the moment the heaters are turned on, and before a temperature increase take splace.

Heaters where turned on either sequentially, or all at once. The bigger the number of heating elements that were activated, the more the frequency of oscillations dropped Even before a temperature rise. Figure 2.26 shows one such heating and cooling cycle for the FPGA.

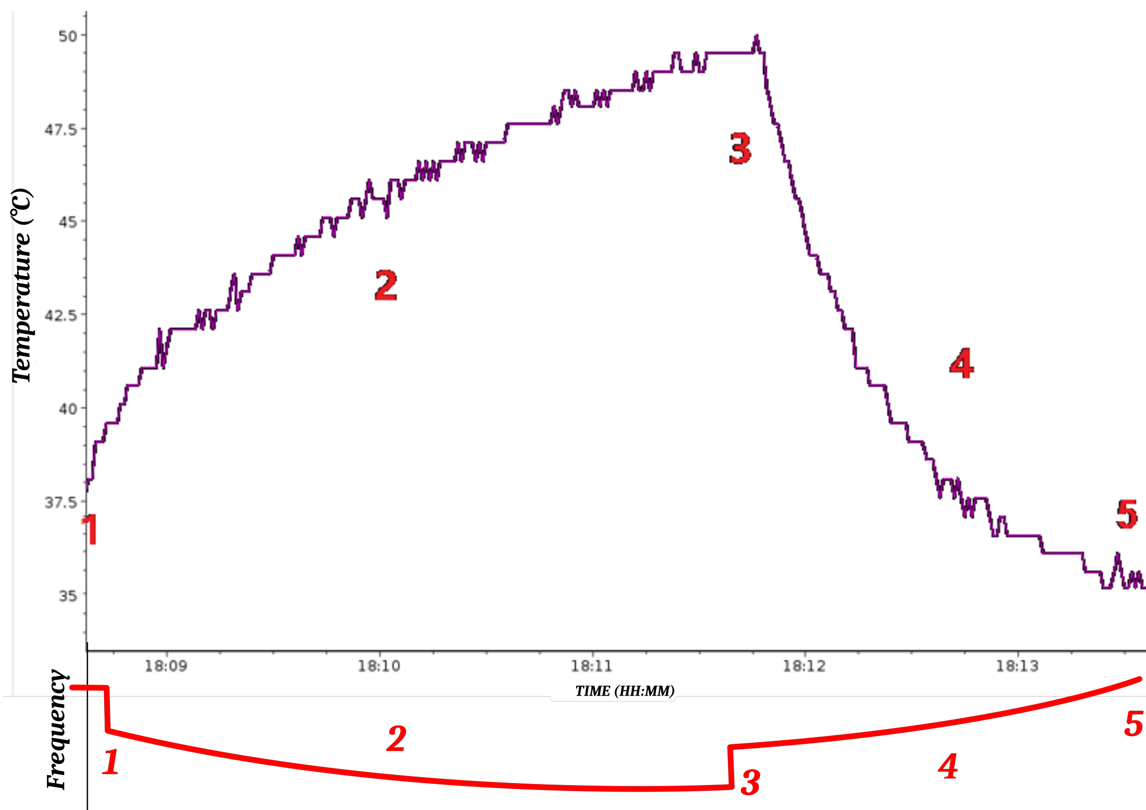


Figure 2.25: Heating and Cooling Cycle

- Before allowing the temperature to fluctuate, on chip temperature (measured by an embedded temperature sensing element) was stable. Before each activation of the heaters, the die was allowed to reach an equilibrium at 35 degrees celcius. The same composite states effects were observed in regard to initialization of the Rings, so before any of the frequency measurements were taken, a cycle of parking and unprking the rings was carried out.

- Then the heaters would be turned on. The more heaters turned on the more the oscillation frequency dropped. Even though the die's temperature at the moment of activaton did not change, the Ring Oscillator frequency dropped instantaneously, as soon as the heaters were turned on. It also dropped more intensely when turning on more heaters, thus proving that the Rings' frequency is dependent upon voltage supply.

- Having observed variation over Voltage, the FPGA cooling fan was disabled, and the heaters were turned on. Die temperature started rising (2), reaching around 50 degrees celcius. During this time, the frequency kept dropping.

- At point 3 the heaters were truned off and the frequency rose abruptly, but not to previous levels. The difference between the frequency at point 2 right before turning the heaters off and right after are attributed to power supply variations.

- Then, the frequency rose gradually as the die cooled (4). This demonstrates variance over Temperature.

- At point 5 the frequency had been restored to its original value. The total change of frequency was of the magnitute of up to 200 Khz due to any of the previous reasons on the Ring of length 161. This is a 1% variation from the central frequency of 18 MHz.

- Using the multiplexer inferred, all the Rings' outputs were brought to the surface, 4 at a time and the results were similar in every case.

This set of tests prooves the variance of ring oscillator frequency over Process and Voltage variations. Following is the high level schematic of the circuits implemented for the purposes of proving PVT ring oscillator dependence, acompanied by the view of the Ring Oscillators in the FPGA (from Vivado's Device View). The ring oscillators can be seen to utilize the LUTs in the SLICEs.

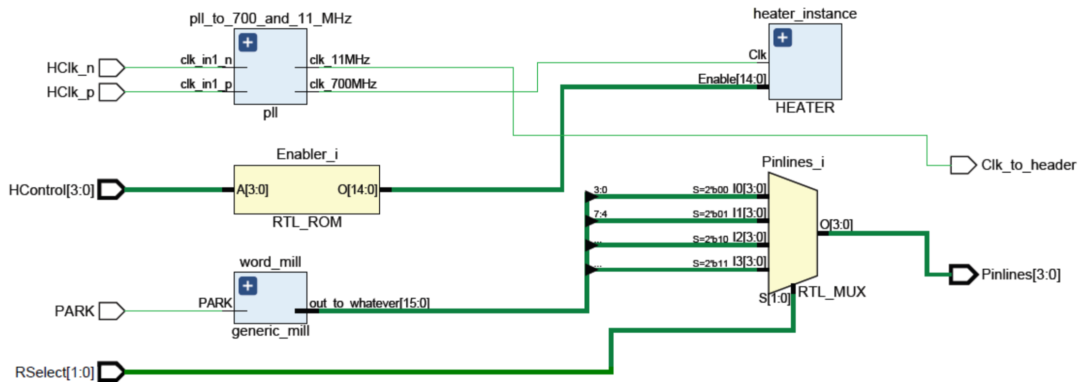


Figure 2.26: The elaborated schematic of the described experimental device

Figure 2.27: Part of the Ring Oscillators, seen to utilize lookup tables

In order to demonstrate variance over Process, absolute symmetry between the rings is necessary, to eliminate all other possible variables from affecting the ring's frequency like differences in routing, certain optimizations that may take place differently accross rings (combining 2 LUTs into 1) and different pin selection (meaning selection of specefic LUT Input and Output pins) and placement the automated tools choose for the rings.

From the implemented design that Vivado composed, the various interconnect delays of the placed design can be examined. The tools never chose absurdly long routes to route the LUTs. However the tools do not choose a symmetrical implementation of the rings as show in the vivado device window in Figure 2.28. Going back to the Long Rings design, Figure 2.28 shows the three implemented

rings, highlighted in different colors, and a zoomed in view on ring4001. As can be seen, letting the placement to the CAD tools produces a highly irregular pattern for the LUTs. Also, the LUTs belonging to different rings can intermingle: it can be seen how the cells that belong to ring4001 (yellow) are interspersed with the other two rings. In the zoomed in view two used slices are noted with a text overlay. Two unused slices can also be noted.

Figure 2.29 makes the routing and pin selections visible too. The pin selection implemented automatically is asymmetrical (other LUTs use the A3 pin as input, others use A4 et.c). What can also be seen is that, the tools combined two independent inversions in one LUT: the signals Q3778 = NOT Q3777, and Q3774 = NOT Q3773. This is called LUT combining. As every LUT6 has more than 2 inputs and 2 outputs, the compilation tools figured each of them can be used to implement two independent NOTs. The NOTs are grouped together for combining at the tools' discretion, and the grouping can't be predicted. The design also exerts no control over where the Ring Oscillators would end up in the silicon die. Lastly, Figure 2.30 shows the asymmetry of SwitchBox connections.

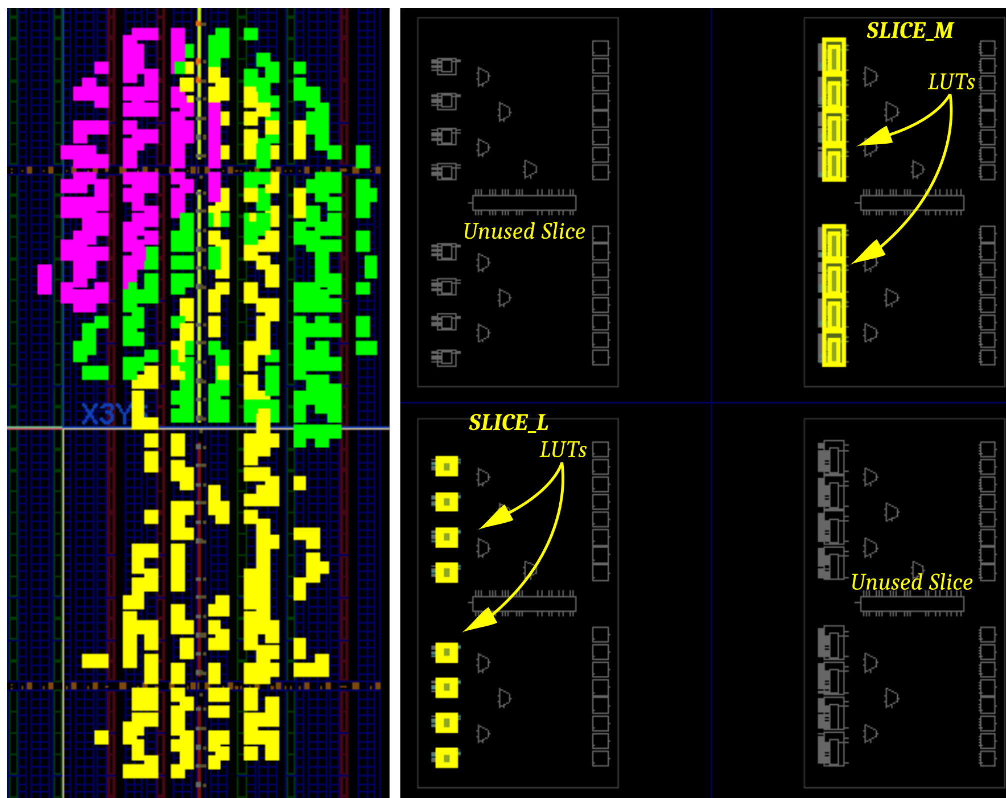


Figure 2.28: Left: ring2001 (purple), ring3001 (green), ring4001 (yellow)
Right: Slice utilization

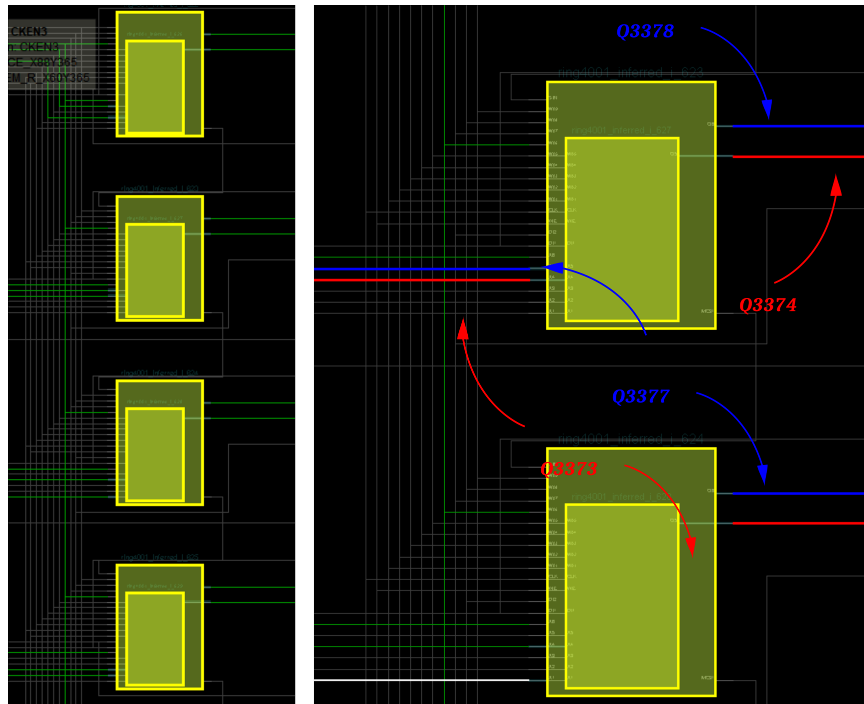


Figure 2.29: Left: Irregular LUT pinning
Right: LUT combining

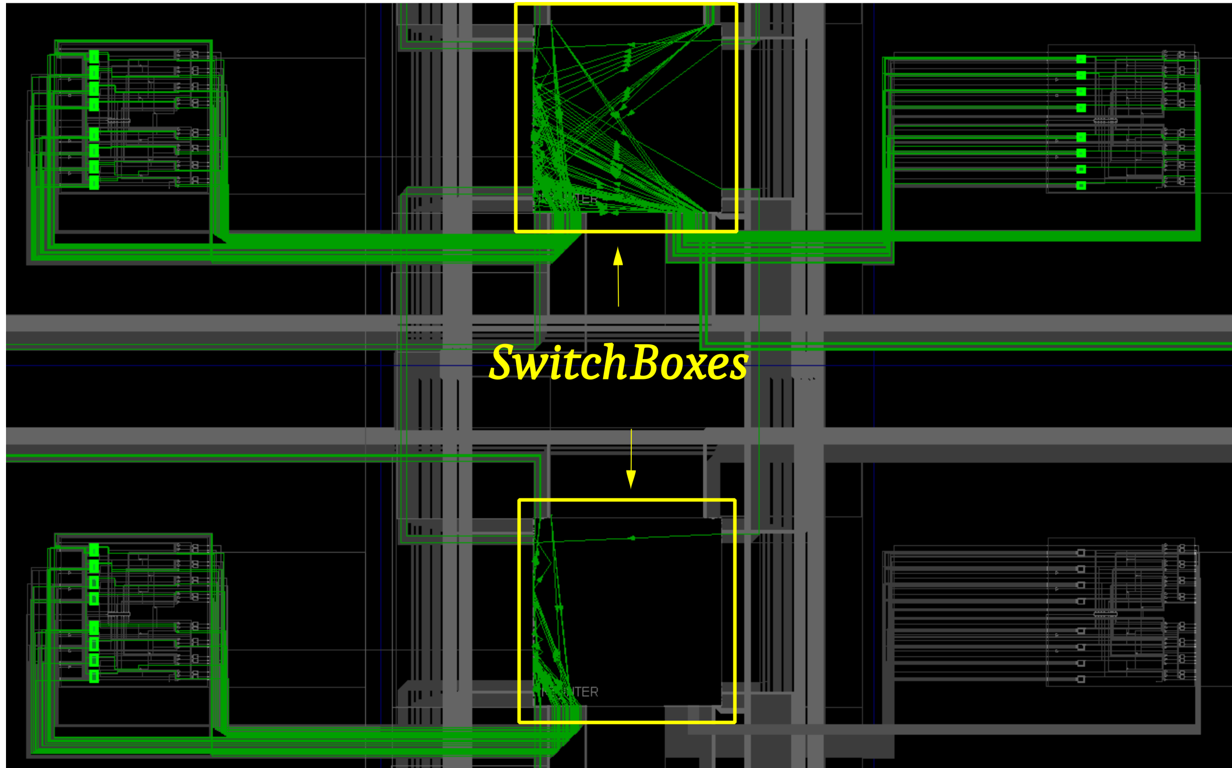


Figure 2.30: SwitchBox asymmetrical configuration

The non-periodic routing of the rings can also be established by dumping the net delays of the ring signals. Using the following TCL script, the list “delays” is created and set to be empty (step 1). Then the signals of the ring (ring[0], ring[1] et.c.) are selected in increasing order and their delays are appended to the list (step 2). Then the list is shown as an output to the console (step 3).

```
1: set delays {}

2: for {set i 0} {$i <= 2000} {incr i} {lappend delays\
   [get_property SLOW_MAX      [get_net_delays -of_objects\
   [get_nets "ring2001["$i"]"]]]}

3: puts $delays
```

The output is expressed in picoseconds and looks like:

84 160 272 199 206 57 40 43 85 160 249 209 202 41 88 104 95 158 160 194 199 206 57 40 43 56 263 209 202 41 88 104 95 209 200 41 88 104 180 227 262 154 206 57 40 44 102.....

which is non-periodic.

Lastly, LUT combining between different components is also observed. Figure 2.31 shows the combining of an inverter belonging to ring4001 with an inverter of ring 3001 in the same LUT.

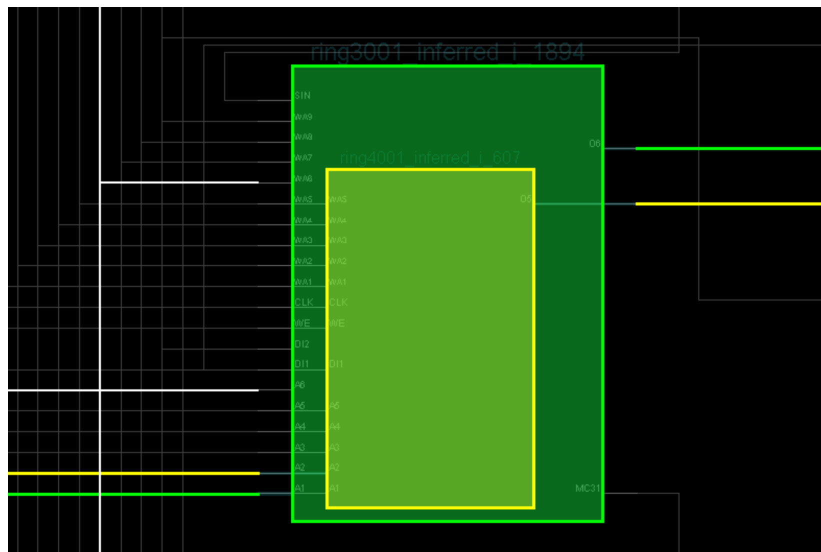


Figure 2.31: LUT combining between rings, same color coding as before

In order to remediate the above situation, some kind of floorplanning solution needed to be developed. At first, a simple approach of using PBlocks was attempted, shown in Figure 2.32. PBlocks are regions within the FPGA, where cells can be added. The placer is, in that way, instructed to place the cells within the PBlock region. However this did not work, as the PBlock boundaries are soft by design.

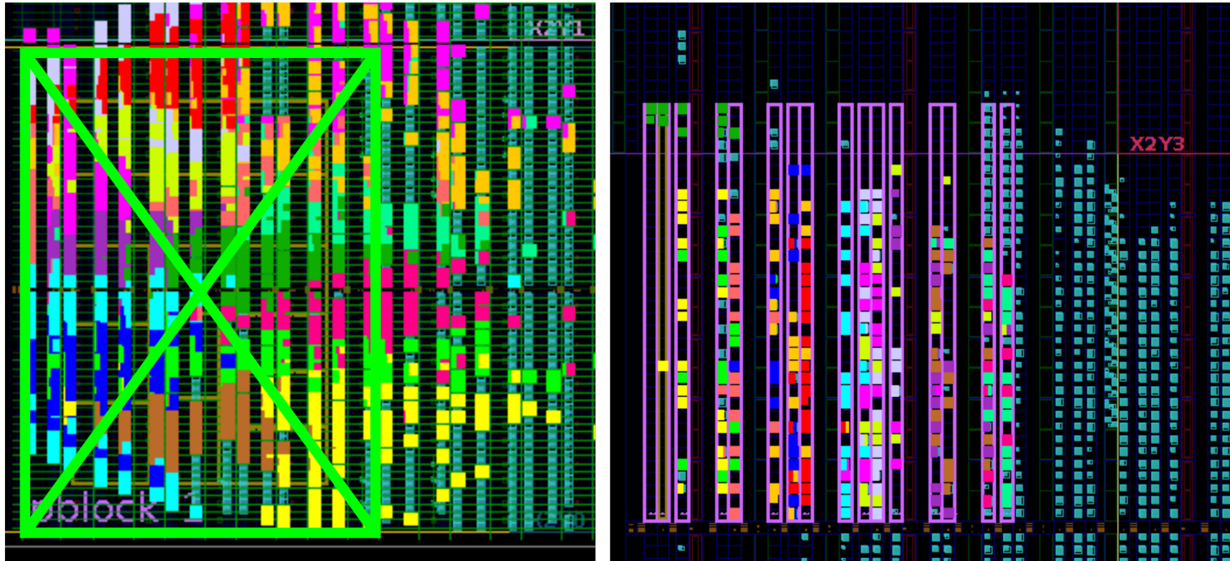


Figure 2.32: Left: Single PBlock (green)
Right: A PBlock for each Ring Oscillator (purple)

The next step attempted was to change the Ring code to explicitly instantiate LUT1 instances, using the intrinsic templates that come with the FPGA's support package. the NOR would be implemented as a LUT2 as follows.

```
constant Len : Integer := 161;
signal ring : std_logic_vector(Len -1 downto 0);

--Attributes needed so the design doesn't simplify logic and so that combinatorial and timing loops are permitted
attribute dont_touch : string;
attribute dont_touch of ring : signal is "TRUE";
attribute ALLOW_COMBINATORIAL_LOOPS : string;
attribute ALLOW_COMBINATORIAL_LOOPS of ring : signal is "TRUE";
```

```
begin

    gen_r : for I in 0 to Len -2 generate
    LUT1_inst_r : LUT1
    generic map (
        INIT => "01")
    port map (
        O => ring(i + 1),   -- LUT general output
        I0 => ring(i)  -- LUT input
    );
    end generate gen_r;

    LUT2_inst_r : LUT2
    generic map (
        INIT => "0001"  -- Logic function
    )
    port map (
        O => ring(0),   -- 1-bit output: LUT
        I0 => ring(Len -1), -- 1-bit input: LUT
        I1 => PARK  -- 1-bit input: LUT
    );

    out_to_sampler <= ring(Len -1);
```

Still, this did not have any effect on the LUT combining technique the tools use to make more efficient use of FPGA real estate. There is also the problem with the routing signal routing and LUT pinning. Subsequently, a solution was this time attempted by using global synthesis and implementation directives.

- BLOCK_SYNTH.MAX_LUT_INPUT useless: minimum is 4, we need 1 (2 for NOR)
- BLOCK_SYNTH.LUT_COMBINING useless: setting to 0 does not stop opt_design from combining two LUTs when called with -remap (which is always)
- Neither is LUT_REMAP of any use, as setting it to False does not yield anything

The solution was to explicitly iterate over the cells of interest using a TCL macro constraint. As was previously explained, a TCL script constraint is not managed by the tools, so the tools do not modify it in any way, or decide where it is to be used. It is used to separate normal XDC constraints from TCL programs meant to carry out tedious tasks best suited to a general purpose programming language.

Here the TCL script was used to explicitly place and pin the LUT cells. The script is long and complex, and will not be presented here. It is however shown in entirety in Appendix B. Suffice it to say, it relies on the tags LUT1_inst_r and LUT2_inst_r to collect the proper LUT cells and applies 3 important constraints.

```
set_property LUTNM DISABLED [get_cells -hier *LUT1_inst*]

set_property LOCK_PINS {I0:A6} [get_cells -hier *LUT1_inst*]

set_property LOCK_PINS {I0:A6 I1:A5} [get_cells -hier *LUT2_inst*]
```

LUTNM DISABLED is the actual directive to disable LUT combining. LOCK_PINS explicitly maps the inputs of LUT1s to the A6 pin, and the inputs of LUT2s to A6 and A5.

The script then separates the LUTs according to the Ring Oscillator they belong to and arranges them in a vertical collumn, iterating over the SLICE LUT basic elements (BELs) from bottom to top. The constraint set_property BEL X6LUT $cell was useful for this purpose, where X is a letter from A to H, and $cell is the particular LUT cell being mapped to the specific BEL. The script defines a function that cycles over the BELs from A to H every 8 LUTs encountered.

After this is accomplished, the cell is placed into a particular SLICE, using a LOC constraint as such: set_property LOC SLICE_X${thelocx}Y${thelocy} [get_cells $r]. The variable thelocy is incremented evey 8 LUTs (one SLICE has 8 LUTs), also every time a ring is fully iterated over, but reset every second ring iteration. thelocx is incremented every 4 rings. The expression SLICE_X${thelocx}Y${thelocy} gets evaluated to something like SLICE_X36Y1, which are slice coordinates. This flow produces a 4 by 4 Ring Oscillator configuration. The only case that the script can fail is when the Rings are large enough to produce invalid slice coordinates. This simply means that the placement falls outside the FPGA boundaries. A last note on the LUTNM directive: even though the placement loop described above does not allow for LUT combining to take place between the rings, ring LUTs can still be combined. This is because logic belonging to some other component can still be automatically placed in the free part of a ring LUT.

The effects of the above can be seen in the Figure 2.33. The figure shows the spatial configuraiton of rings R1 through R16, and also shows how the cells are cycled through the BELs and SLICEs

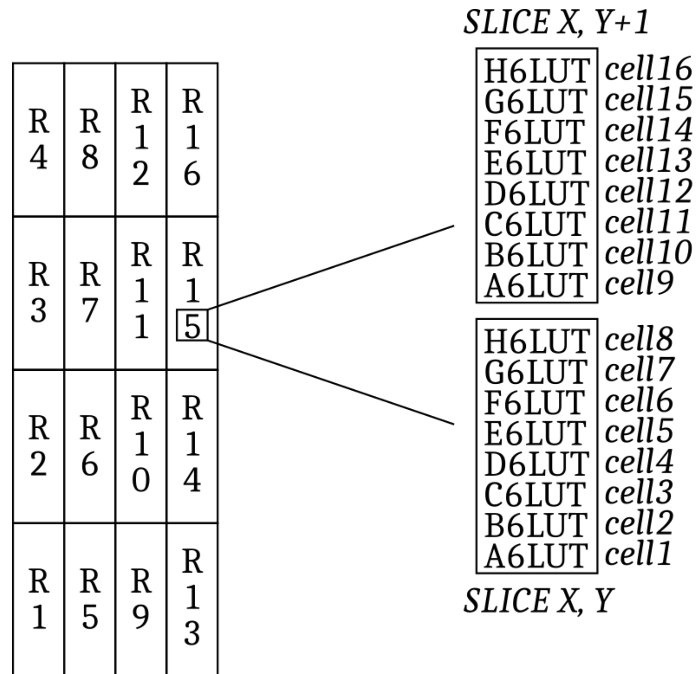


Figure 2.33: Floorplanning Strategy

When creating such a cell grid, a Relative Placement Macro (RPM) can alternatively be used, however in this case no advantages over using a LOC constraint were identified. Figure 2.34 displays the actual 4 by 4 configuration as seen by the Device view.

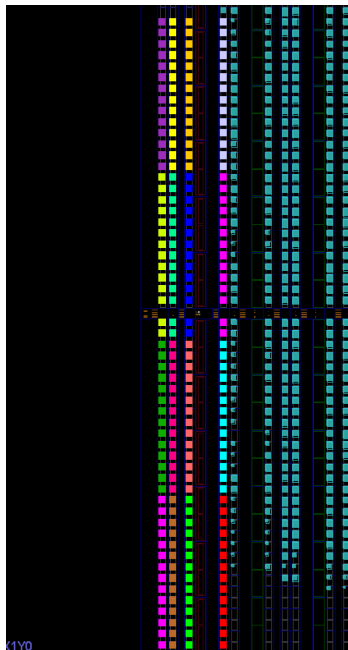


Figure 2.34: Implemented Rings in a 4 by 4 configuration

The TCL constraint file relies on the preexeistence of cells, so it must be run during implementation. This is accomplished by setting the USED_IN attribute of the file if Vivado. If this attribute is not set properly, the script is run during synthesis as well, and there is a warning from the compiler.

The new design also solves the routing symmetry problems. Althouth the nets are not explicitly placed, the arrangement of the ring cells in collumns guides the tools to chose a symmetrical routing solution. This can be verified by running the same TCL commands as before:

```
1:  set delays {}
2:  for {set i 0} {$i <= 160} {incr i} {lappend delays
    [get_property SLOW_MAX  [get_net_delays -of_objects
    [get_nets word_mill/generate_N_sin
    gle_bit_mills[0].mill_instance/ring4[$i]]]]}
3:  puts $delays
```

The output in this case is

40 86 39 87 144 176 139 41
40 86 39 87 144 176 139 41

which is periodic with a period of 8. That is because the routing strategy is the same at every CLB. This can also be visually confirmed by taking a look at the device view. This is shown in Figure 2.35.

Figure 2.35: Symmetrical routing and pin selections

The slight variation of the ring oscillation frequency due to underlying Silicon Process differences was observed on these totally symmetrical rings. The oscillation frequency for a ring of length 161, for example would vary at the hundreds of KHz when implemented at different parts of the die. Indicatively, the scopes taken for four adjacent rings are presented in Figure 2.36. In the upper right corner of every scope, the frequency is shown. Similar results were obtained for the other rings too. This experiment demonstrates variance of Ring parameters over Silicon Process (P).

Figure 2.36 : Frequency from Rings of Length 161

In coclusion, the previous tests manage to demonstrate Ring sensitivity over Process Voltage and Temperature. The new <F_{PI}> value for the symmetrical floorplan and routing, without LUT combining is <F_{PI}> = 2000MHz*#Inverters. This can be used to estimate the characteristic frequency of ring oscillators of various lengths, when implemented with the particular TCL macro mentioned previously.

3. Realization of the TRNG System

Based on the results the experiments described in chapter 2 provide, the TRNG's design was refined and a statistical system for the evaluation of the TRNG's performance was developed.

3.1 Free Parameters and Expectations

The previous set of tests brought forth a very important quality of the Ring Oscillators: Jitter. Ring Oscillators exhibit phase instability from oscillation to oscillation. In the time domain, it means uncertainty of when exactly the transition from HIGH to LOW or vice versa will happen after some oscillator cycles in the future. The more oscillator cycles that are allowed to pass, the more this uncertainty grows.

In the oscilloscope, this can be seen by triggering at a certain edge, and looking at the trace with respect to that edge some time forward after the trigger. The waveform is seen oscillating horizontally, as a result of Jitter. As a matter of fact this phase noise (Jitter) is quite considerable for relatively long rings. Two such traces were taken for demonstration. They are from a shorter ring of length 51.

The traces where taken 700ns and 1000ns post trigger, for a ring of frequency around 67MHz. This translates to ~50 and ~70 ring oscillator periods respectivlely, after the oscilloscope was triggered by a rising edge. The traces are shown in Figure 3.1.

Figure 3.1: The waveform after 50 (top) and 70 (bottom) oscillations

In Figure 3.1 waveforms with the jitter are shown. After just tens of cycles, the variation from true periodic behavior increases to considerable levels. The more time that is allowed to pass, the more unpredictable the switching point is. After just 70 ring periods, the horizontal waveform swing, which is a measure of the phase deviation from perfect periodic behavior, approaches almost 40% of a full period. If more time is allowed to pass between sample acquisitions, all traces of the previous periodicity are destroyed.

This jitter is a cumulative result of all the previous mechanisms that affect a ring oscillator's parameters, as described in Chapter 2.

As explained, the more time that passes, the more the effects of Jitter build up. These are the addition of an unpredictable phase deviation from cycle to cycle. To allow for the buildup of this phase noise, the ring oscillators output has to be sampled in relatively long intervals. If the intervals are long enough, all periodicity will be overshadowed by phase noise, making all previous periodic behavior irrelevant in predicting when exactly the next switching will take place. The device that samples at a much lower rate than the toggling rate of the Ring Oscillator, is called the Undersampler. Therefore, one free parameter is the Sampling Interval, which means how many cycles are allowed to pass between two samplings.

Another free parameter is the ring length. Ring length could prove to be of consequence, as a ring's stages should contribute to the total jitter by means of a gaussian random walk [28, 35]. If a ring with M stages is considered, then for a mean random (gaussian) jitter with variance $\langle\sigma_s^2\rangle = \sigma^2$ per inverter stage, it follows that the total variance per ring cycle will be $M \cdot \sigma^2$, so the random jitter piles up following a square root law: $\sigma_{per\ cycle} = \sigma \cdot M^{1/2}$.

However, larger rings also exhibit reduced characteristic frequency, thus more time is required for realising the same number of elapsed ring cycles between the samples, which also contributes to the total jitter as follows: For D ring cycles, $\sigma^2_{tot} = D \cdot \sigma^2_{per\ cycle} = D \cdot M \cdot \sigma^2$, so $\sigma_{tot} = \sigma \cdot (D \cdot M)^{1/2}$. Both parameters have to be explored to some degree over a range of values. Note that σ is the total jitter introduced per gate through both global and local non deterministic sources. It could very well be dependent upon the switching frequency of the rings.

The sampling frequency range was set to be variable from 2.5MHz down to 4.6 KHz. How those numbers translate to ring oscillator periods is dependent on the ring's length, upon which the ring's frequency is dependent. The rings used ranged from extremely long rings of length 1111, medium sized rings of length 111 and 11, and relatively short rings that do not span more than one CLB, with lengths 7, 5, 3 and even 1.

A third free parameter is the ring locations. This is chosen as part of the floorplanning strategy. The requirements set for proceeding to a systematic study were to choose a constant starting point for the N by M ring oscillator configuration, and disabling LUT combining, so as to not only spread over more silicon, but to also disallow any third component from using available ring LUT pins.

3.2 Design and Floorplanning

The origin selected for the rings was the bottom left corner of the X1Y0 region as shown in the Figure 3.2. This place offers the maximum freedom for both vertical ring placement, upwards. This becomes important when implementing large vertical rings in the FPGA.

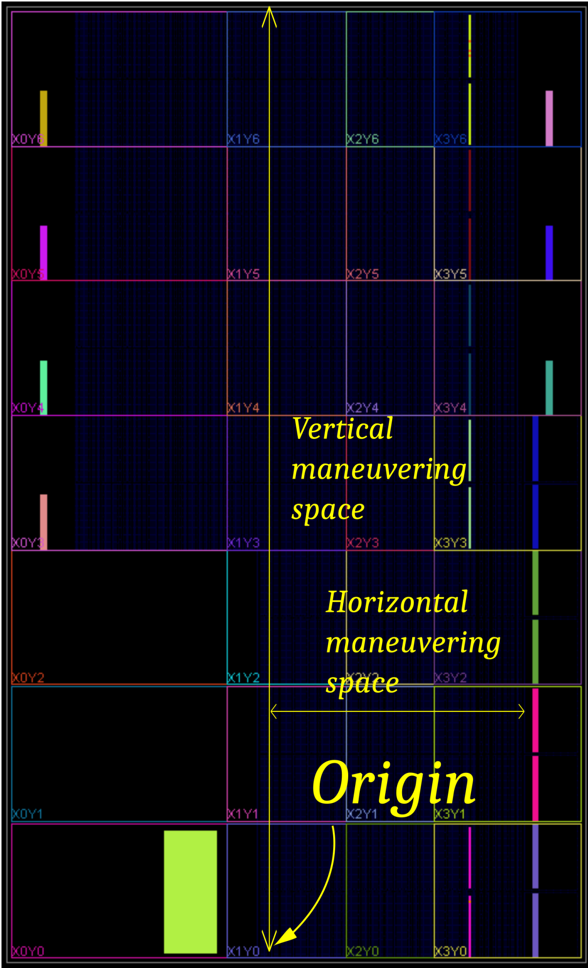


Figure 3.2: The origin of the rings

The floorplanning script was slightly modified from that described in chapter 2. Due to the vertical Inverter placement of each ring, a 4 by 4 configuration can't be accommodated for rings of length ~1000 Inverters (they exceed the FPGA's boundaries), so a 2 by 8 configuration was deemed preferable.

Figure 3.3 shows the placement of the rings according to this new configuration and an implemented instance of the described rings.

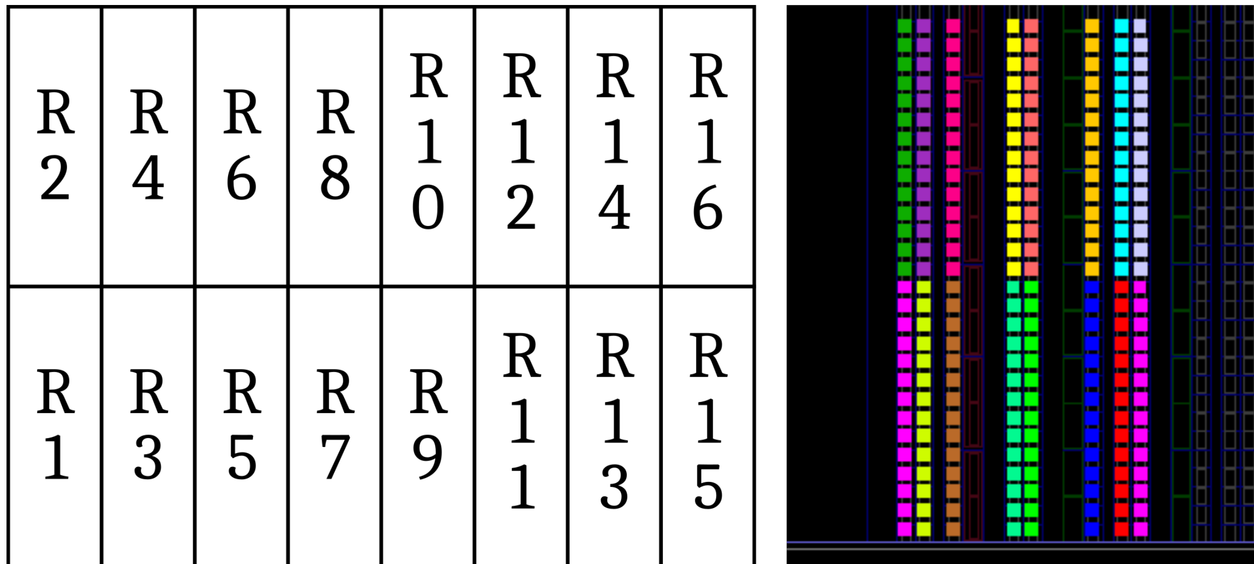


Figure 3.3: Left: ring placement in the 2 by 8 configuration
Right: implemented rings in 2 by 8 configration

In regard to the TRNG design, it consists of those 16 rings placed in the aforementioned way, plus a programmable sampler. The sampler contains a user programmable register. This register provides the Skip Interval, which is the interval of clock ticks between two successive samples (not the Sampling Interval). The Clock signal originates from external programmable clock on the board, SI570, and is passed through a PLL.

The sampler uses the divided clock to capture samples from the Rings' outputs. This happens through a sampling register. In order for the TRNG output to be stable (as opposed to metastable), a SYNC (synchronizer) register is piped to the sampling registers output [21, 22, 23], clocked on the common divided clock. The output of the synchronizer is considered the TRNGs random number stream.

The SI570 clock has a default frequency of 300MHz, and the PLL is used to clean the clock signal from all the noise and jitter that may have formed during it's flight from the clock chip to the FPGA clocking network.

Input from the user is provided through the use of a VIO probe. This input includes the Skip Interval and the PARK signal, as far as the TRNG is concerned. The VIO probe is an IP core from Xilinx[8] that can monitor and drive signals in the FPGA in real time. It interfaces to the hub on a PC through the JTAG cable used to linnk to the FPGA. It offers the capability to create input and output probes, which are used to read from and write to the FPGA respectively. The width of those buses is also customizable [8]. The VIO probe was configured through the IP integrator wizard [35] that is part of the Vivado suite.

The following diagram shows the high level architectural structure of the TRNG system, which comprises the Ring Oscillators, the tunable sampler (with the SYNC register), and the Control Infrastracture.

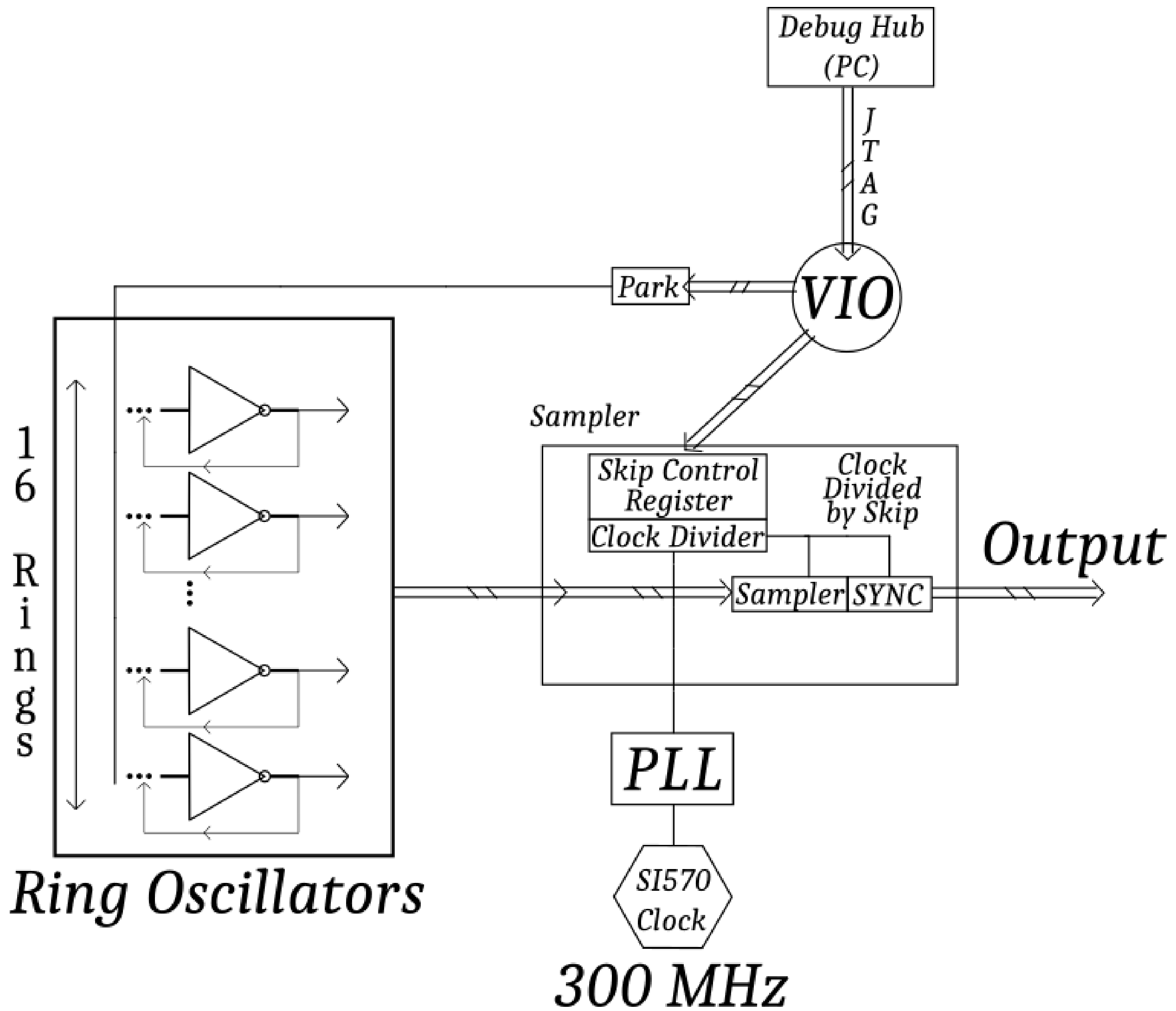


Figure 3.4: The TRNG architecture

In the case of a Post Processed design, the post processor is tied to the synchronizer's output, and provides the TRNG's post processed output.

The VHDL code for the TRNG is listed in Appendix B. The code for the post processor is listed in Appendix D.

3.3 Moments of a Uniform Distribution

The algebraic moment μ_p of a distribution of numbers $\{x_n\}$ around a value c is defined as the expectation value $\mu_p = \Sigma(x_n - c)^p f(x_n)$ [6]. Here f(x) is the probability function of receiving a patricular number x out of the distribution.

The Raw or Crude moments μ'_p are taken around 0, i.e. $c = 0$, so $\mu'_p = \Sigma(x_n)^p f(x_n)$. Empirically, for a sample of N numbers, if $s(x)$ is the frequency of x within the sample, it is $f(x) = s(x)/N$, where N is the sample size. Therefore

$\mu'_p = \Sigma_x(x)^p \cdot s(x)/N$, or more simply

$\mu'_p = \Sigma_n(x_n)^p /N$, for n = 1 to N.

Although central moments are most commonly used to describe a distribution, wherein $c = \langle x \rangle = \Sigma_n(x_n) /N$ (the average x), there is a particular algorithmic advantage of considering raw moments instead: They can be computed without implementing extra memory. Straightforward central moment calculation requires storing the values in some sort of memory, calculating the mean term, and then using the mean term while reiterating over them to extract the central moments themselves. By constrast, the unnormalized raw moment of order p is but a sum of the pth power of the numbers {x} of the sample. A simple division by N at the end of the summation gives the proper, normalized raw moment. No need to implement memory to store all the sampled numbers, just a register to hold the power value, and another wide enough to repeatedly sum it into. Raw Moments can be expressed in terms of central Moments by means of an Inverse binomial transform,

$$\mu'_n = \sum_{k=0}^{n} \binom{n}{k} \mu_k \, {\mu'_1}^{n-k},$$

with $\mu_0 = 1$, $\mu_1 = 0$ [25].

In case the central moments are required, after the calculation of the raw moments, it is possible to express them in terms of the raw moments, by means of an forward binomial transform (inverse of inverse) [25]:

$$\mu_n = \sum_{k=0}^{n} \binom{n}{k} (-1)^{n-k} \mu'_k \, {\mu'_1}^{n-k},$$

where μ_k is th kth central moment.

A Uniform Continuous Distribution of numbers from a to b is a distribution that has a constant probability density for x within the interval [a,b], and 0 outside. For the Uniform Discrete Distribution, like the one this TRNG aspires to at least approximate, it simply means ***f(x) = 1/W***, where ***W*** stands for the number of individual values possible within the [a, b] interval (the cardinality of the set that the numbers take values from).

The theoretical raw moment of order p of a Discrete Uniform Distribution for a number from a to b are just the sum of x^p over (b-a+1). The numbers that the TRNG produces are 16-bits wide and will be treated as unsigned integers. Therefore the range of numbers is for 0 to 2^16-1 = 65535, and the theoretical 10 first raw moments are as follows:

$$\sum_{i=0}^{65\,535} i^k \times \frac{1}{65\,536}$$, where k = {1,2...10}

k=1 65535/2
k=2 2863245995/2
k=4 7378416157370421521/2
k=5 201478413347682581299200
k=6 22635408953845511612687175485/2
k=7 648997493940803049423952086220800
k=8 75613111568987562610479093479680053521/2
k=9 2229904382563824487976712787769489579458560
k=10 265705270245739278002454396465343726009161978725/2

where k = order of the moment

The division by 65536 can be avoided, and the sum kept as is (change of scale), and as a result, the theoretical values of the scale-transformed raw moments, or rather, the raw sums of samples from a 16-bit uniform distribution, over the 2^{16} possible values, are simply

$\sum_{i=0}^{65535} i^k$, where k = {1,2,...10}

k=1 2147450880
k=2 93822844764160
k=3 4611545282012774400
k=4 241775940644713972400128
k=5 13204089297153725648024371200
k=6 741717080599609724524533366292480
k=7 42532699762904468647048123922566348800
k=8 2477690439892584451620178935142155993776128
k=9 146139013615702801644041849259261269079396188160
k=10 8706630295412384661584425663376383213868219718860800

where k = order of the moment

3.4 Bias and 1’s frequency

In a binary number stream sequence (of 0s and 1s), the bias ***e*** is defined as [1]

$$e = \frac{1}{2}(P(x_i = 1) - P(x_i = 0))$$

where P(x=1) is the probability of getting a ’1’ and P(x=0) the probability of getting a ’0’ in the sequence. Empirically, those are the frequency of ones and the frequency of zeroes, respectively. If the bias is known then one can obtain the probabilities of getting a 1 or a 0 from the sequence as

$$P(x_i = 1) = \frac{1}{2} + e \text{ and } P(x_i = 0) = \frac{1}{2} - e$$

It is desirable that the bias of the output of the TRNG be very small, ideally zero. A small bias indicates that the TRNG output is as random as possible. Assuming a large positive bias means one could reasonably predict more bits in the output sequence would be 1 rather than 0. This in turn makes the output stream more vulnerable to brute force attacks.

For this study’s purposes, bias is the relative deviation from the ideal relative frequency of ones, 0.5. Given a sample of length W, the ideal ones’ frequency would be W/2, whereas any difference S is the expansive expression for bias (the non normalized image of bias), meaning F(1) = W/2 + S, F(0) = W/2 - S.

3.5 Distribution of the sequence of 1s

When considering where the true nature of randomness lies in a stream of bits, one important characteristic is that any bit be independent of any other bit that came before it. Ideally, in a zero-bias case, when sampling the TRNG output, the first bit has 1/2 chance to turn out ‘1’, the next again 1/2, and so on. Same goes for the value 0.

In other words, when given 1 as a starting value, the probability that a new 1 is received right after that is 1/2. The probability that it is received two samples after (meaning the probability of getting a 101 sequence) is then $P(0) \cdot P(1) = 1/4$. In general the probability that one gets ’1’ N positions next to the first one, and N-1 zeros in between as a consequence, is

$$P(\text{distance} = N) = P(0)^{N-2} \cdot P(1)^2 = (1/2)^{N-2} \cdot (1/2)^2 = 2^{-N}.$$

Note that thus far only a bias of zero was considered. If, however, the stream exhibits a nonzero bias ***e***, then one can obtain the modified probability

$$P(\text{distance} = N) = P(0)^{N-2} \cdot P(1)^2 = (0.5 - e)^{N-2} \cdot (0.5 + e)^2 = B \cdot 2^{-N},$$

with the modification factor $B(e) = (1+2e)^2 \cdot (1-2e)^{N-2}$ [6, 26].

This is a very potent test to evaluate the TRNG output. It a form of the binomial test [26] on the outcome of getting a sequence of N-1 0s and a ’1’ in the end.

3.6 Design of the Statistical System

The TRNG’s output is piped into a system that calculates it’s various statistical quantities. Those include the Raw Moments of the random numbers’ distribution, the count of 1s and 0s (as a measure of bias) and the number of 0s between consecutive appearances of 1. For the purposes of evaluation a sample size of 2^{20} = 1,048,560 samples was chosen. The values of the Raw Moments expected from this sample are

$$\mu_n'' = \sum_{i=0}^{65\,535} 16\, i^k$$, where k={1,2...10}

k=1 34359214080
k=2 1501165516226560
k=3 73784724512204390400
k=4 3868415050315423558402048
k=5 211265428754459610368389939200
k=6 11867473289593755592392533860679680
k=7 680523196206471498352769982761061580800
k=8 39643047038281351225922862962274495900418048
k=9 2338224217851244826304669588148180305270339010560
k=10 139306084726598154585350810614022131421891515501772800

The Moment Calculator, is a Pipeline that leverages DSPs to calculate all powers of the TRNG output words up to the tenth. The calculated individual power is then summed into a register that holds the Moment sum. The calculation is lossless, meaning that there is no arithmetic truncation or loss of accuracy during the Moment calculation.

The diagram in Figure 3.5 simplifies the explanation of the pipeline. MUL stants for Multiplication and ADD for addition.

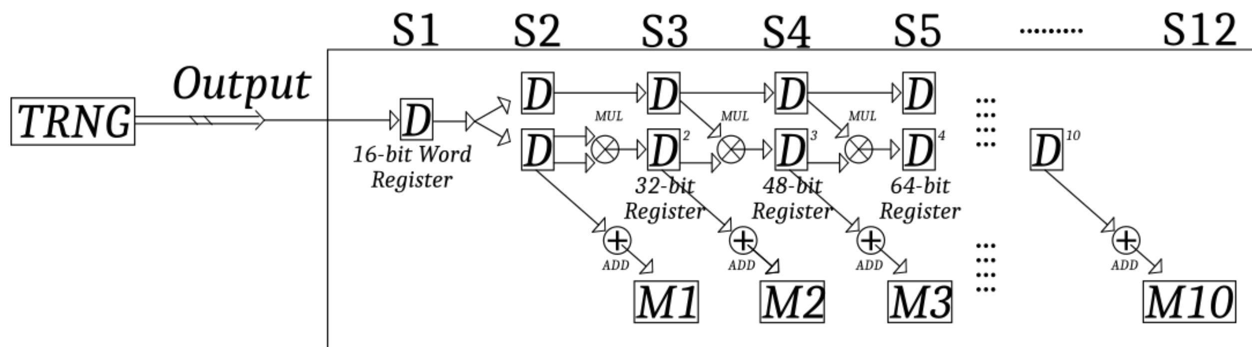


Figure 3.5: The raw moments pipeline, S# is the stage and M# the #th moment

As can be seen in the Figure 3.5, the pipeline consists of 12 stages. Every stage is executed at a clock's rising edge, and after an overhead of 12 clock cycles, all the stages of the pipeline are active. Beyond that point, the pipeline continuously calculates powers of the input data and sums them to the moment register at every clock cycle (iteration interval of 1).

The multiplication of two numbers N and M bits wide without overflow produces an output that is M+N bits wide [37]. Similarily, S repetitive summations of N-bit numbers, one on top of the other, can be thought of as a product $S*(2^N -1)$ in the worst case scenario, 2^N being the largest N bits wide integer. The width of S is $Ceil(Log_2(S))$, Ceil being ceiling function, which rounds to the closest greated integer. Hence the repetitive summation requires $N + Ceil(Log_2(S))$ bits to avoid an overflow. In this particular case, Nth powers of 16 bit numbers produce Nx16-bits wide products (seen in figure 3.5 as 16-, 32-, 48-, 64-bit register and so on). There are 2^{20} consecutive summations of those particular numbers to yield the raw moments, therefore the required register bitness for the Nth moment registers is $Nx16 + Ceil(Log_2(2^{20})) = Nx16 + 20$ bits.

The Moments' calculator is a relatively generic code that does not explicitly target a particular component in the FPGA. It does however guide the tools to the inference of DSP blocks for the purposes of calculating the pertinent sums and products. A part of the logic that yields the fifth moment and carries out the calculation of the sixth power (to be used in the followng pipeline stage) is presented below.

```
--MOMENT5
regMOMENT5 <= regMOMENT5 + P1;
D7 <= D6;
HX1 <= P2*D6;--MAIN
HX2 <= P2*D6;--CARRY
```

Here regMOMENT5 is the register that holds the partial sum of the 5th moment, P1 and P2 are registers holding fifth power of the 16-bit word, D6, and HX1 and HX2 are the registers that hold the calculation of the sixth power of the 16 bit word. Registers with suffix 1 are the main registers, to be used for the partial sums, while registers with suffix 2 are to be used for the calculation of powers. Lastly the word D6 is propagated for later use into register D7.

The arithmetic expression that uses * as the multiplication operator requires using the IEEE.NUMERIC_STD.ALL package (declared in the source header declarations). Due to the way the code is written, with pure 2-term products and simple additions, and the relatively large bitness of the code (16 and over), the tools are guided to using DSP blocks to carry out the intended tasks.

The DSP blocks in this FPGA contain the DSP48E2 core from XILINX, found in Ultrascale and Ultrascale+ FPGA families [9]. Follwing is the DSP48E2 basic functionality diagram.

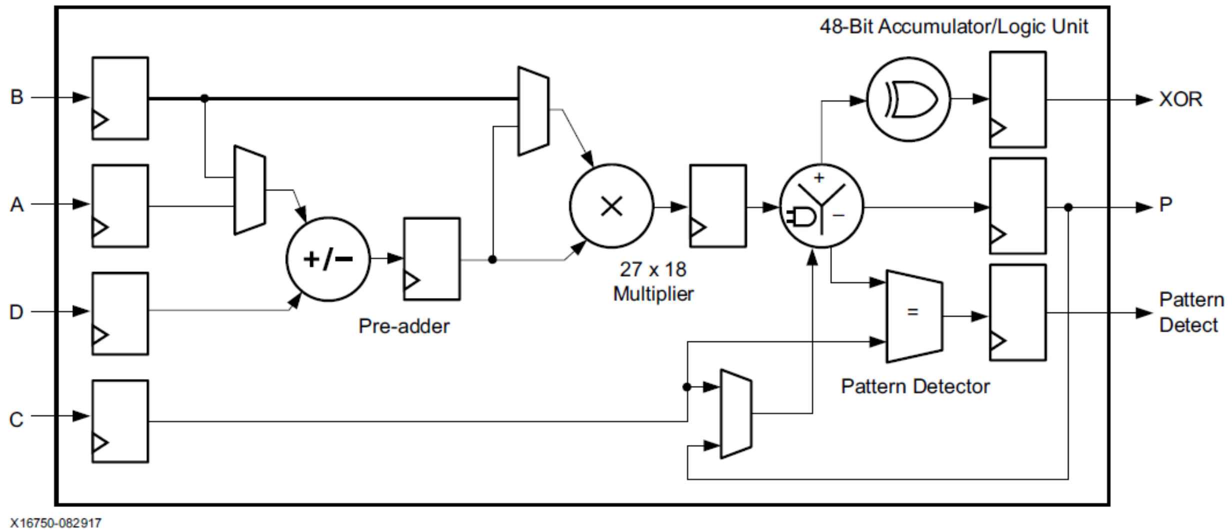


Figure 3.6: DSP48E2 basic functionality (adopted from [9])

The A and B input ports that can be seen lead to the A and B embedded registers, 18 and 27 bits wide respectively, that are used in the DSP to hold summation or multiplication variables. This particular DSP's arithmetic and logic unit (ALU) operates, on two's complement numbers, wherein the first bit is the sign. This is very common for DSPs. However, two's complement numbers are signed integers.

To emulate unsigned number behavior, the true input must be set to include an extra bit along with the true input data. This bit has to be set to 0 always, to restrict the DSP's behavior to only the unsigned subset of the two's complement range. For example, the 16 bit input 1111 1011 1100 0111, unsigned, has to take the form 0 1111 1011 1100 0111 in the DSP's input register.

The pipeline uses a synchronous (to the pipeline clock) signal (labeled Set in the code) that flushes all internal buffers, resets the sample counter and thus prepares the pipeline for a new run. Failure to flush buffers will result in incorrect results. A Start signal (also synchronous) initiates the pipeline, as long as the Set signal is not asserted.

The statistical pipeline's behavior was simulated for fidelity, and also tested on a trial vector. The trial vector was the sequence of {0,1,...65535} repeating 16 times. This also yields the expectation value for the unnormalized raw moments of a 16 bit uniform distribution, over a sample of size 2^{20}. The test code is in Appendix C. This test yielded the numbers expected, for clock frequencies up to 100 MHz.

The final code was constrained for skip intervals down to 60, that generate a 2.5 MHz divided clock. It is therefore guaranteed to work correctly at this skip interval and greater.

The first numbers that this system returned were close to the theoretical raw moments within a 1% margin. However, from run to run, they tended to oscillate around a central value, and almost systematically lie on one side the theoretical value. This side, meaning wether the values were less or more, was dependent upon the particular ring oscillator circuit inferred (various ring lengths were tested). This was interpreted as a potential indicator that ring oscillators exhibit some sort of bias.

Following that, an other piece of VHDL code that counts the total number of ones accross the whole word was developed. This circuit measures the total number of ones over the 16-bits of every produced word, and sums it over the sample size. Ideally, the bias would be 0, so half of the $16x2^{20}$ bits produced would be one, so the results were compared to the expectation value $16x2^{20}/2$ = 8,388,608. The following diagram shows it's operation.

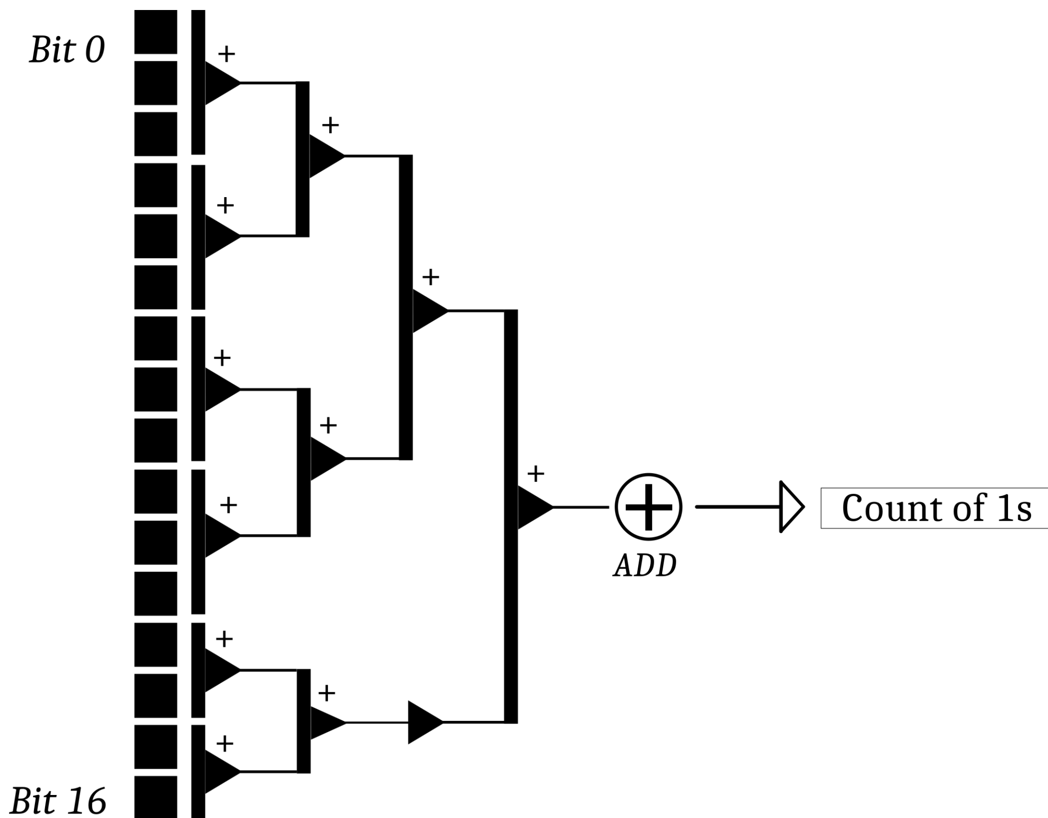


Figure 3.6: the pipeline for counting the total number of ones, showing the consecutive additions of bits

This circuit confirmed the existence of bias in the ring oscillators. As was mentioned in the preliminary chapters, when sampling perfectly periodic waveforms, some bias can occur because of the relation of the sampling frequency to the oscillator frequency. Due to the fact that jitter is known to be present in ring oscillators, this does not hold true for large undersampling ratios. Even in relatively poorly undersampled outputs, the jitters effects would introduce variable phase relations in the two waveforms, and that would even out any bias that would otherwise be generated this way.

A next step was to study wether the bias was more prevalent in a certain subset of the 16 bit word. In order to study wether the bias was unbalanced accross the 16-bit word, another circuit similar to the above was used, that just sums accross every 4 bits of the word instead of 16. It showed measurable differences accross the biases of the 4-bit parts. This hinted to the existence of a per ring oscillator characteristic bias.

Next, the per bit bias counter was developed which provided the best insight into the Rings' behavior. It turns out that the rings do exhibit some bias, sometimes significant. It is unclear wether this bias is a manifestation of a substrate variation, electrical imbalances that affect LUT output rise and fall times, or both. The per bit expectation count is now $2^{20}/2$ = 524,288.

Finally, the most important test was developed, that captures the distribution of the distance between consecutive 1s. This test focuses on each bit separately. Denoting the distance between consecutive ones at a ring's output with d, let P(d) be the probability that two consecutive ones occur with a distance d (d greater than or equal to 1). If enough jitter has indeed been accumulated through the sampling process, this distrubution must be a binomial distribution.

Ideally, a bias of zero would yield

$\mathbf{P(d) = P_1*(P_0)^{d-1}, 0.5*(0.5)^{d-1}=0.5^d=2^{-d}.}$

If however a bias e is observed, one can still compare the fidelity of the curve to $\mathbf{P(d) = (0.5 +e)(0.5 -e)^{d-1}}$, as mentioned in chapter 2.

This circuit uses a distrubuted RAM to store the number of samples passed (distance) from the appearence of a '1' in the output stream to the next appearence. It uses a maxumim of 32 registers, each for a distance d=1,2,...32. This should be more than enough, as it is unlikely to enounter a one in a 2^{-32} event in 2^{20} samples. Once the distance is measured, the appropriate register is incremented and the distance counter is reset. This is considered to be the most potent test for the TRNG, as it can reveal both information about the bias, but also uncover true randomness, or lack thereof, in the per bit output stream.

The Binomial Curve measurement circuit's high-level mode of operation is presented in Figure 3.7 in a graphical way.

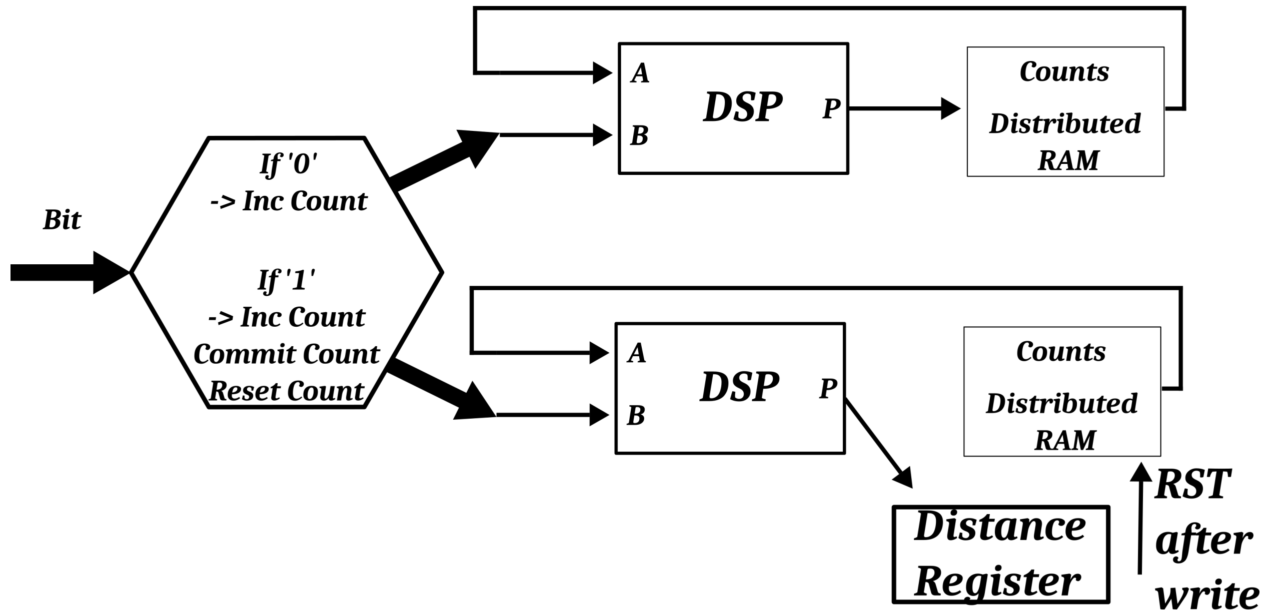


Figure 3.7: The circuit responsible for collecting the '1's distribution

The binomial test is used here as a P-Test, with a margin of 1%.

4. Results and Interpretation

The following results are from the binomial distribution tests. The results are shown here for various ring sizes and various clock skip intervals. For convenience the clock skip intervals are translated to the approximate Number of Elapsed Ring Periods and Sampling Frequency. Alongside the measured curves, the ideal theoretical curve is presented (with zero bias) and the modified binomial curve for the measured bias of the output. When examining the spectrum of the sample count between consecutive ones (referenced as # Samples Until Resampling a '1' in the graphs), different conclusions can be drawn regarding the effectiveness of the sampling, the bias of the ring oscillators, the quality of the switching and the overall randomness of the TRNG's output.

Fidelity to the ideal curve is the desired outcome that speaks for the quality of the TRNG output stream. However, fidelity to the modified curve can still offer interesting information related to the total jitter manifested within the system. The closer the measured curves are to the modified binomial curve, the more phase noise was accumulated through the sampling process. Fidelity to the modified binomial curve means the result is as random as it can be, and the only deviation from randomness is bias. The bias is measured using the sum of all the distance registers, as effectively they increment every time a '1' is encountered. There is no need to include the results of the special bias measuring circuit, as they are identical. In the following tables, it is the average sum over the samples taken.

Parts of the spectrum that lie far over the 20th position normally shouldn't exist (or be very rare). If they do exist, it could be an indicator of problematic switching. If an excess of such instances is found then this is considered a very bad sign about the ring oscillator's suitability for use in a TRNG. It means that it outputs frequent series of zeroes, and that makes it predictable. When looking at the low end of the sampling interval, some tuning effects are to be expected. Because the Jitter's effects don't have enough time to accumulate in such cases, the spectrum is expected to lie more heavily in the leftmost side of the chart (small distances between ones).

Following are the results for non post processed ring outputs with various numbers of inverters and sampling rates. The linear axis view is not so useful as the rightmost terms collected become invisible, so the logarithmic view is used in almost all cases shown. In every case, 5 samples were collected, labelled 1 through 5 in the graphs.

In Chart 4.1, the results for Ring with 3 Inverters are presented in a graphic fashion. The linear view is shown only in the first case. It can be seen that it is hard to discern the relationship of the collected patterns with the theoretical curves. The logarithmic representation gives a clearer view of the curves' relation, so this is adopted for almost all the graphs.

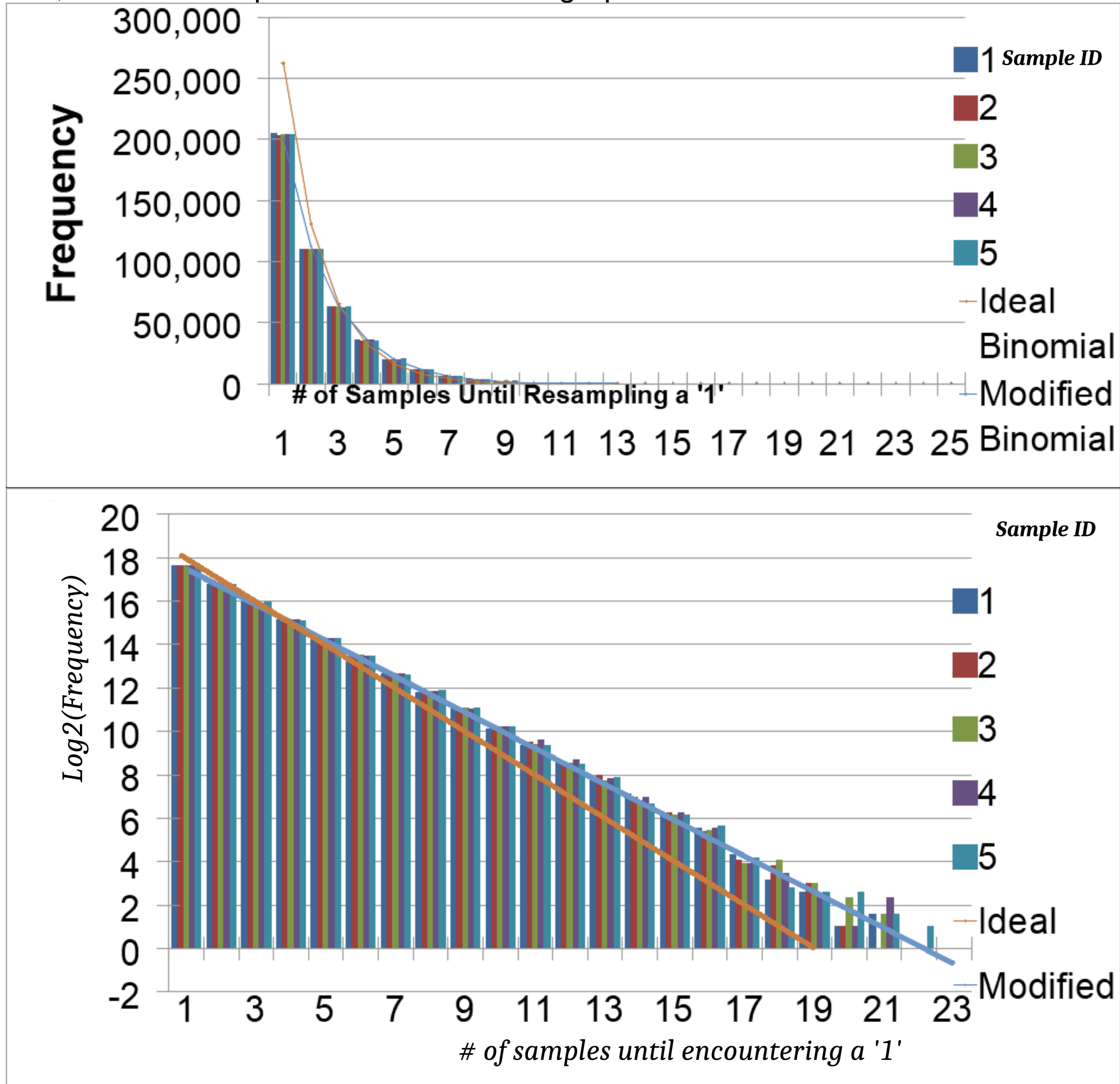


Chart 4.1: 3 Inverters, 100 KHz linear (top) and logarithmic view (bottom), 1st ring

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
667	3072	6826	97.5	
1s Frequency	Ideal	Bias	P(1)	P(0)
458,628	524288	-6.26%	43.74%	56.26%

Table 4.1: Parameters

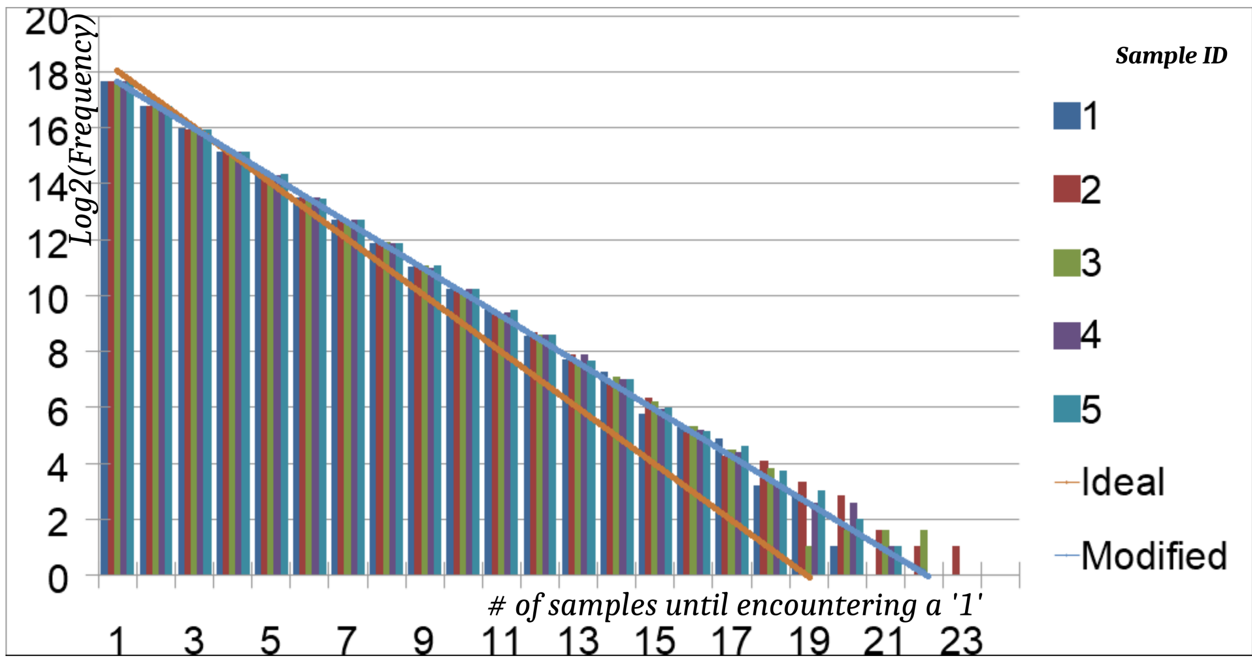


Chart 4.2: 3 Inverters at 15 KHz, logarithmic view

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
667	20000	44444	15	
1s Frequency	Ideal	Bias	P(1)	P(0)
459,299	524288	-6.20%	43.80%	56.20%

Table 4.2: 3 Inverters at 15 KHz, logarithmic view

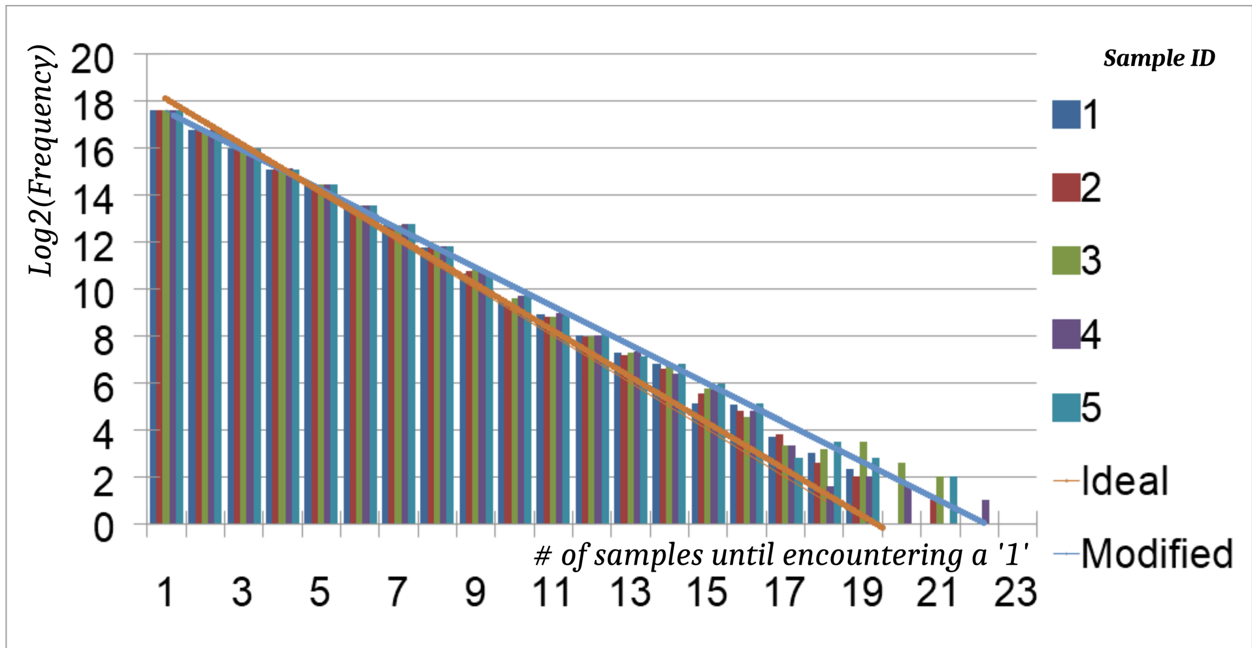


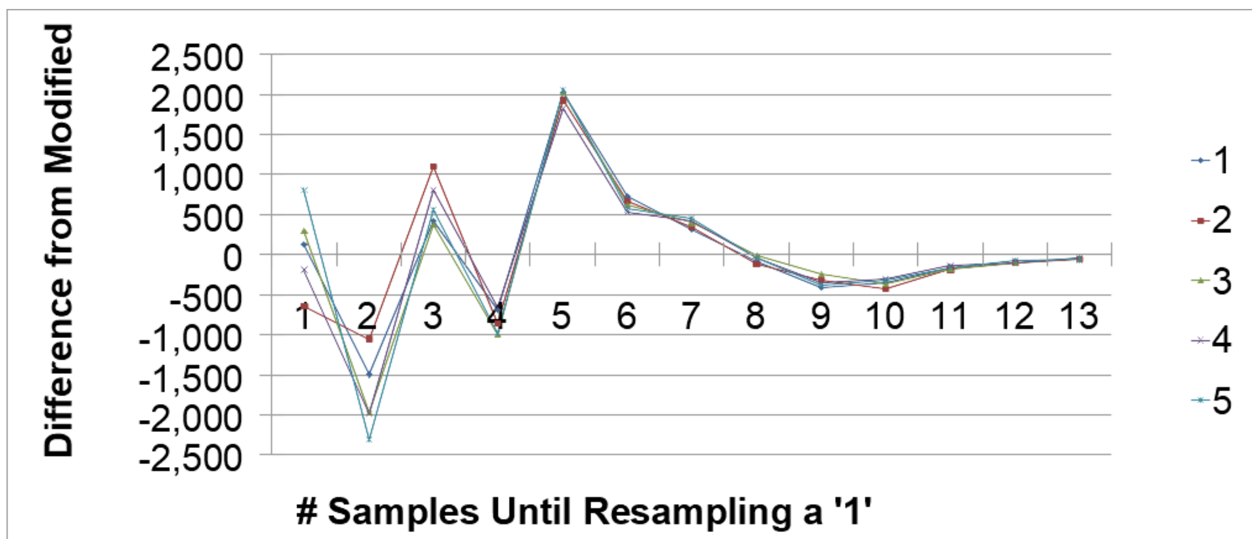
Chart 4.3: 3 Inverters at 2.5MHz logarithmic view, 1st ring

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
667	120	267	2500	
1s Frequency	Ideal	Bias	P(1)	P(0)
459,079	524288	-6.22%	43.78%	56.22%

Table 4.3: Parameters

Up until this point, even for the 2.5MHz sampling rate case, the Elapsed Ring Osicllations parameter is still considerably large. This is a result of the elevated frequency of the ring with size 3, which is shown in the tables.

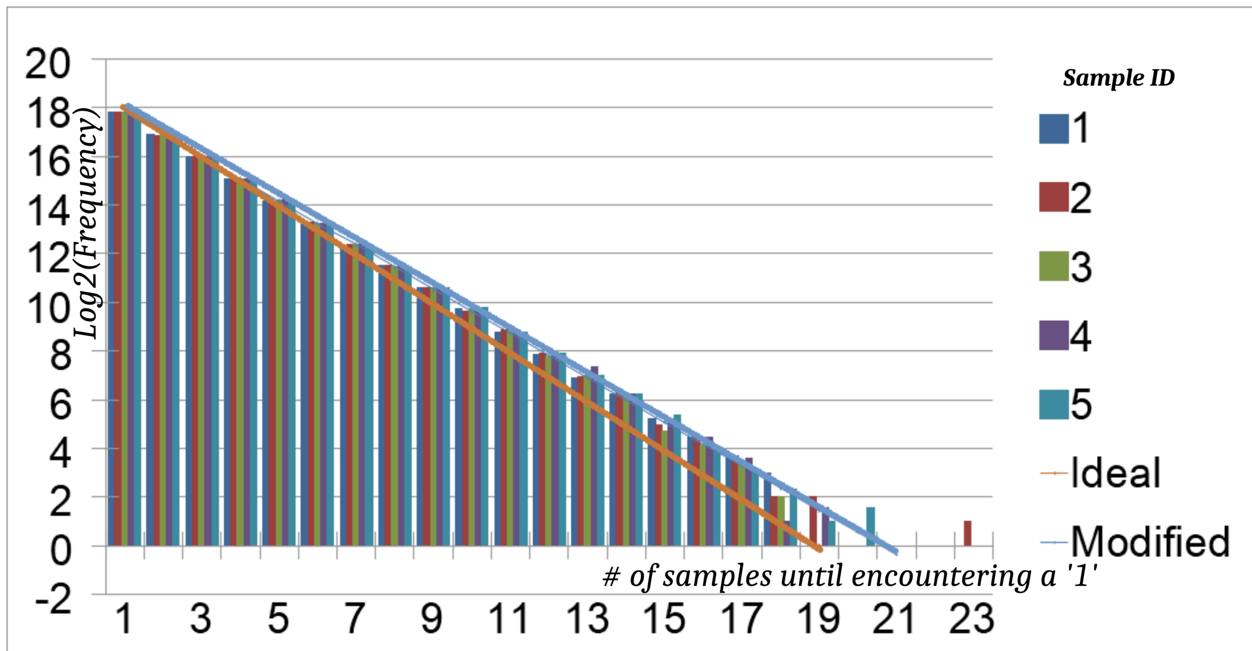
The curves' fidelity to the modified is, as mentioned, taken to be an indicator of the effectiveness of the undersampling process to allow for jitter manifestation. Even in the 2.5MHz sampling case, the curves do not deviate visibly from the modified binomial curve. This is clearly visible in chart 4.4, where the deviations from the binomial test are taken (they are less than 1%). In short. 3-Inverter rings do not tune, even for the highest sampling rates of 2.5 MHz.



Graph 4.4: 3 Inverters tuning phenomena at 2.5MHz, 1st ring

Note: the axis names in almost all the graphs are removed to give space to the actual graph, for purposes of curve legibility. They are the same as the ones presented, namely: Distance-Frequency for primary linear curves, Distance-Log_2(Frequency) for the logarithmic curves and Distance-Difference from Modified Curve for tuning patterns.

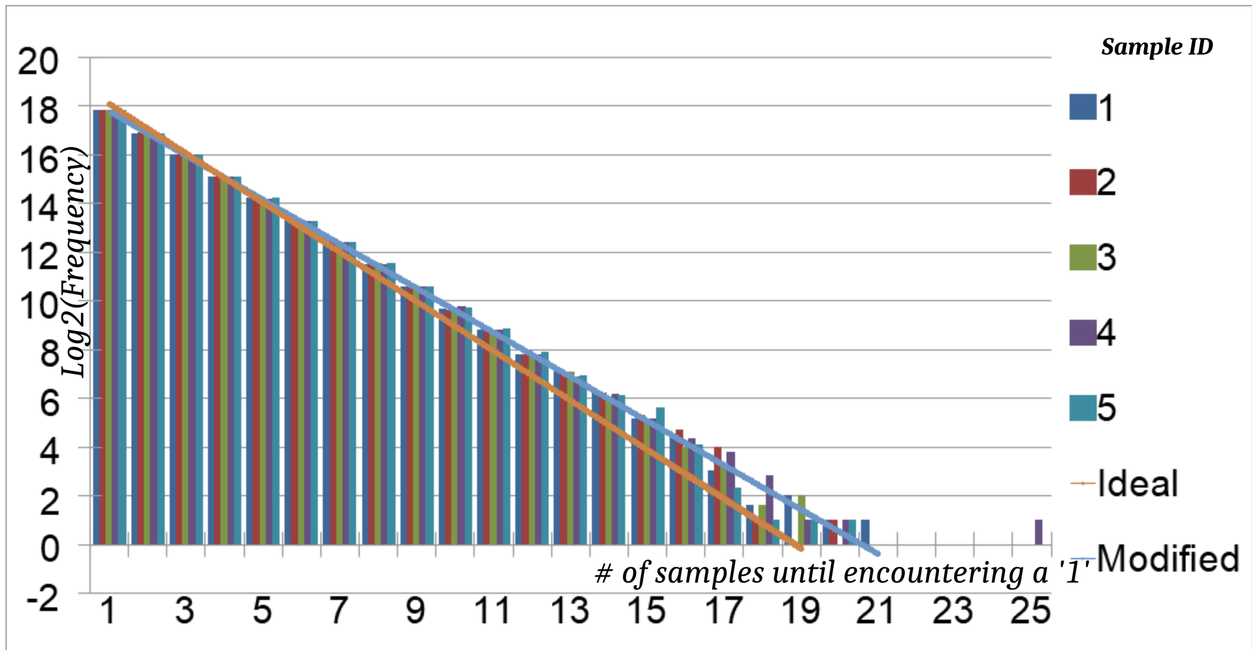
Following, the results ROs of length 5 are presented, in the same format.



Graph 4.5: 5-Inverter Rings at 100KHz

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
400	3072	4096	97.5	
1s Frequency	Ideal	Bias	P(1)	P(0)
490011	524288	-3.27%	46.73%	53.27%

Table 4.5: Parameters

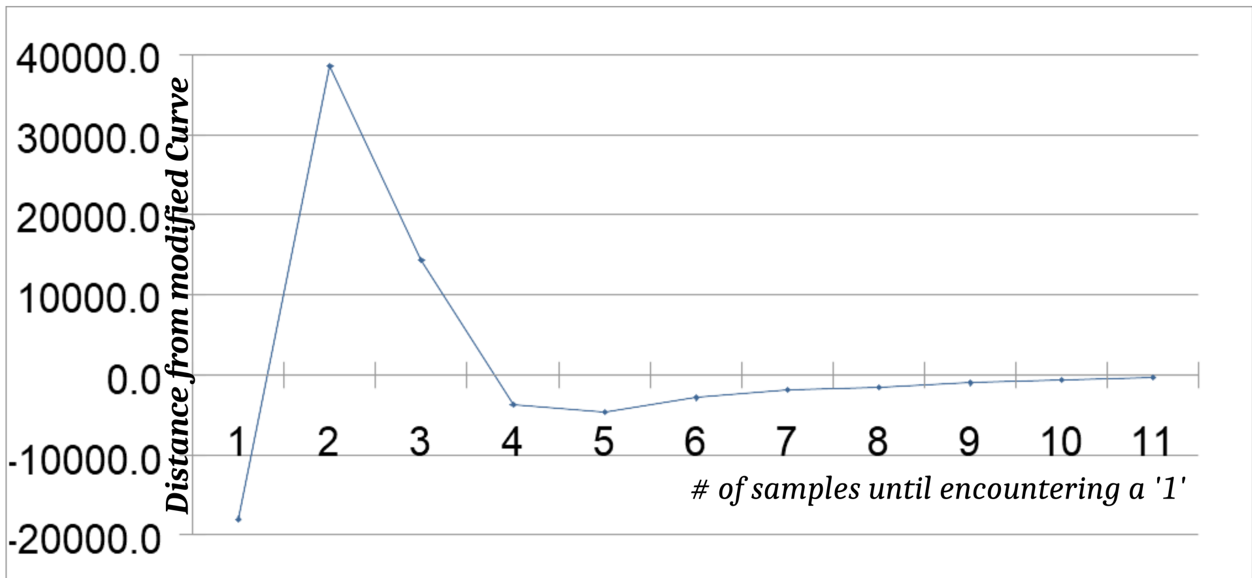


Graph 4.6: 5-Inverter Rings at 15KHz

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
400	20000	26667	15	
1s Frequency	Ideal	Bias	P(1)	P(0)
489687	524288	-3.30%	46.70%	53.30%

Table 4.6: Parameters

At a sampling rate of 2.5MHz, a tuning pattern emerges, as shown in the graph 4.7.



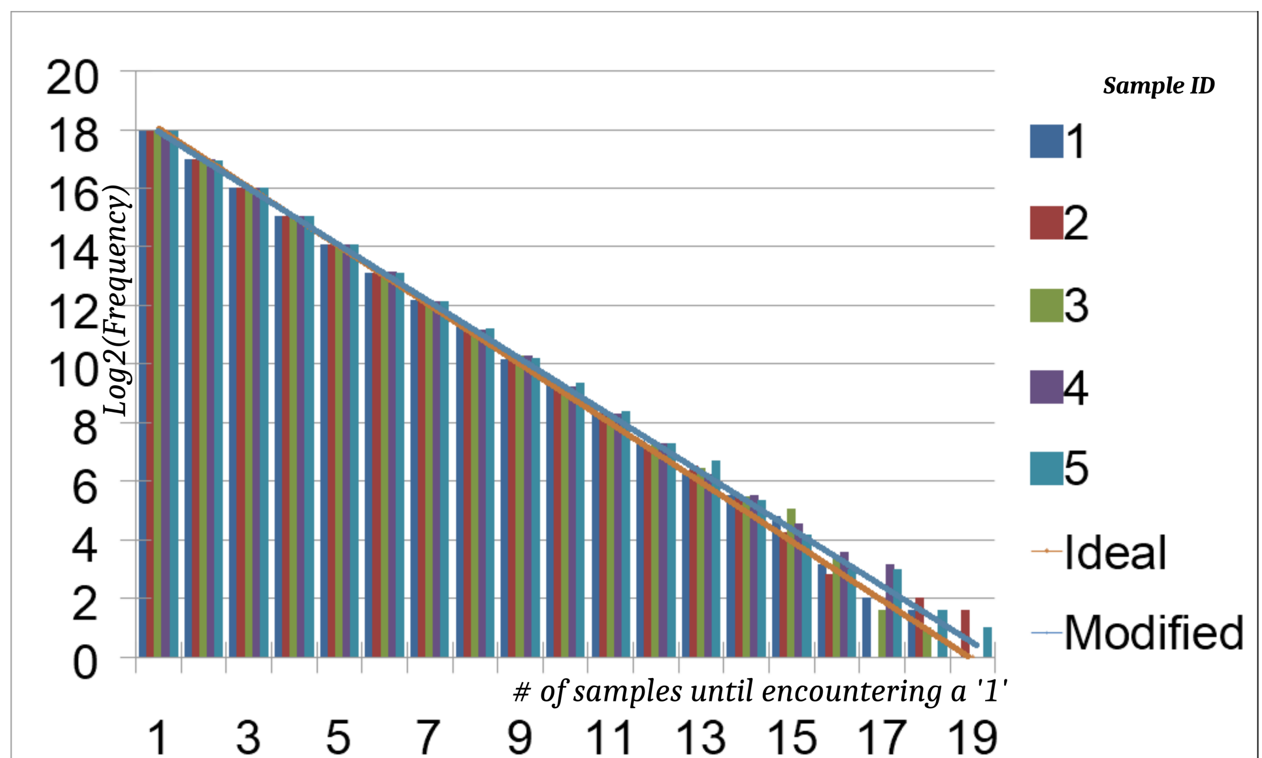
Graph 4.7: Tuning pattern for a 5-Inverter ring oscillator at 2.5 MHz

Ring Frequency (MHz)	Skip	Elapsed Ring Oscillations	Sampling Frequency (KHz)	
400	120	160	2500	
1s Frequency	Ideal	Bias	P(1)	P(0)
490022	524288	-3.27%	46.73%	53.27%

Graph 4.7 shows tuning phenomena between the 5-Inverter ring and the sampler, meaning the elapsed ring oscillations aren't enough to facilitate large jitter buildup. Consequently, this causes some preference for smaller terms (left side of the graph), as an expression of some periodic behavior still present in the sampler's time frame. This, however, is not easily visible in the logarithmic graph.

Here, in particular, there is an overaccumulation over the 2nd and 3rd terms. Beyond that there is a deficit (compared to the modified curve), as is to be expected. Percentage relations are not particularly useful in this case, as there is a considerable drop of magnitude from left to right of the modified curve. This means that small statistical deviations (order of magnitude 1) can yield large percentage differences in the rightmost terms.

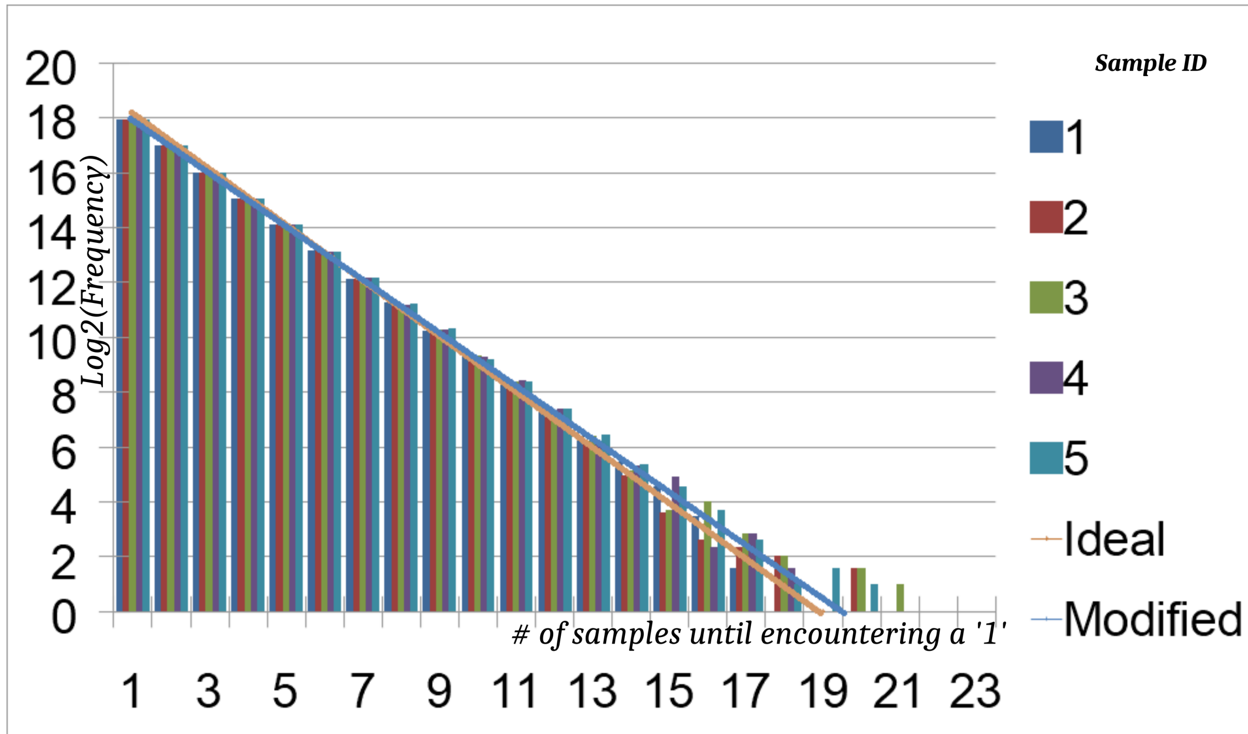
Following are the results from rings 11 Inverters long.



Graph 4.8: 11-Inverter Rings at 100KHz

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
181.8	3072	1862	97.5	
1s Frequency	Ideal	Bias	P(1)	P(0)
512,050	524288	-1.17%	48.83%	51.17%

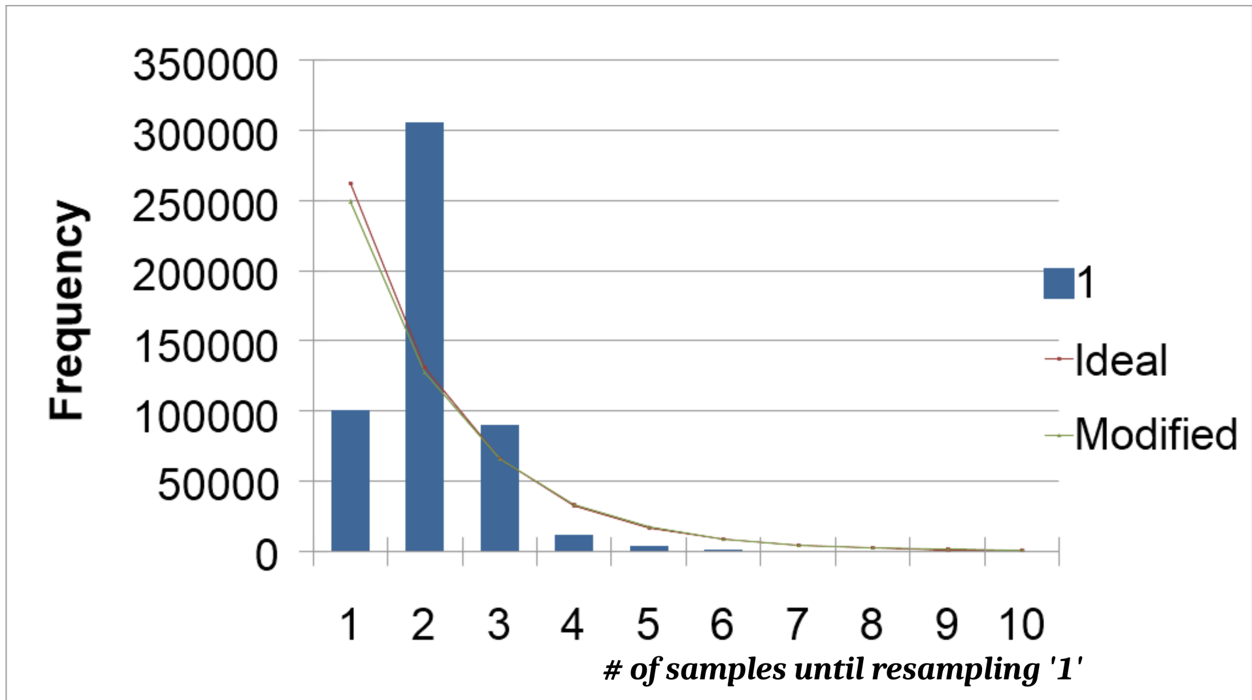
Table 4.8: Parameters



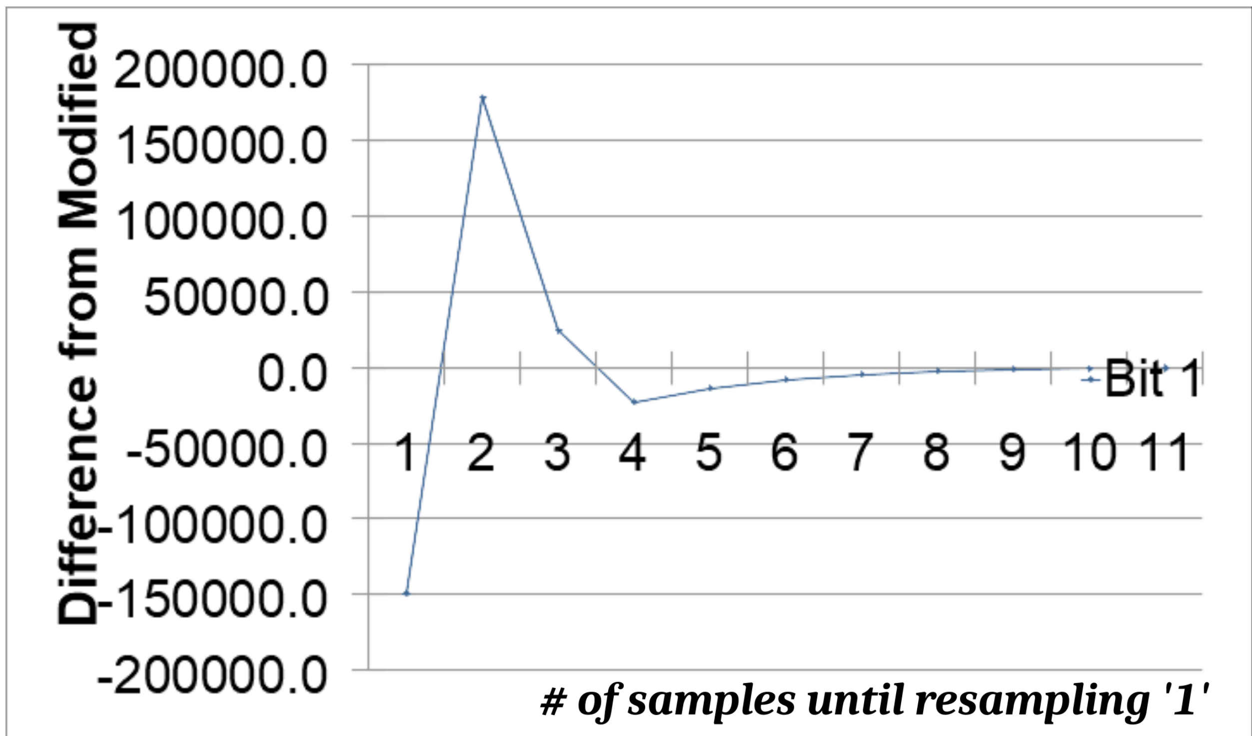
Graph 4.9: 11-Inverter Rings at 15 KHz

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
181.8	20000	12122	15	
1s Frequency	Ideal	Bias	P(1)	P(0)
511729	524288	-1.20%	48.80%	51.20%

Table 4.9: Parameters



Graph 4.10: 11-Inverter Rings at 2.5MHz tuning pattern, primary curve, 1st Ring



Graph 4.11: 11-Inverter Ring at 2.5MHz, tuning tattern, difference from the Modified Binomial curve

Ring Frequency (MHz)	Skip	Elapsed Ring Oscillations	Sampling Frequency (KHz)	
181.8	120	73	2500	
1s Frequency	Ideal	Bias	P(1)	P(0)
511457	524288	-1.22%	48.78%	51.22%

Table 4.11: Parameters

Graph 4.11 shows tuning phenomena similar to the ones encountered in th 5-Inverter Rings.

Next, the results from the 111 inverters version are shown.

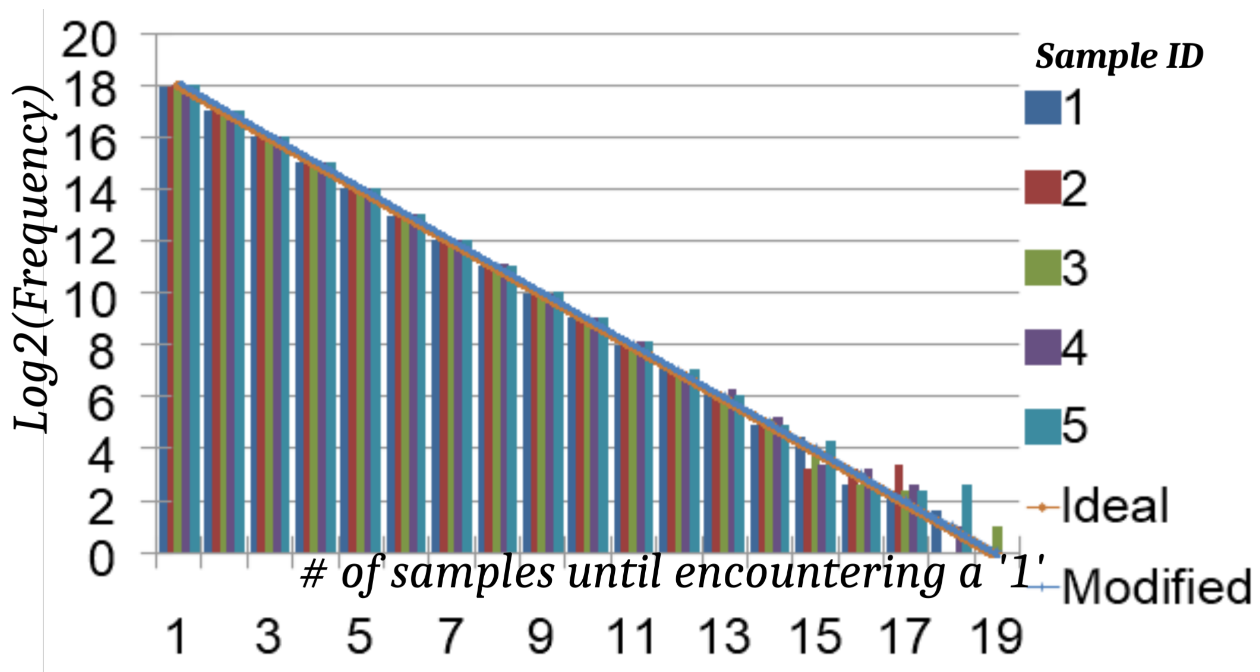


Chart 4.12: 111 Inverters 30KHz, logarithmic view

Ring Frequency (MHz)	Skip	Elapsed Ring Oscillations	Sampling Frequency (KHz)	
18.0	20000	1202	15	
1s Frequency	Ideal	Bias	P(1)	P(0)
522,515	524288	-0.17%	49.83%	50.17%

Table 4.12: Parameters

Tuning starts to happen even at the 200KHz sampling rate, so just one result is included (first ring tuning pattern at 200 KHz). This is the result of a reduced D parameter.

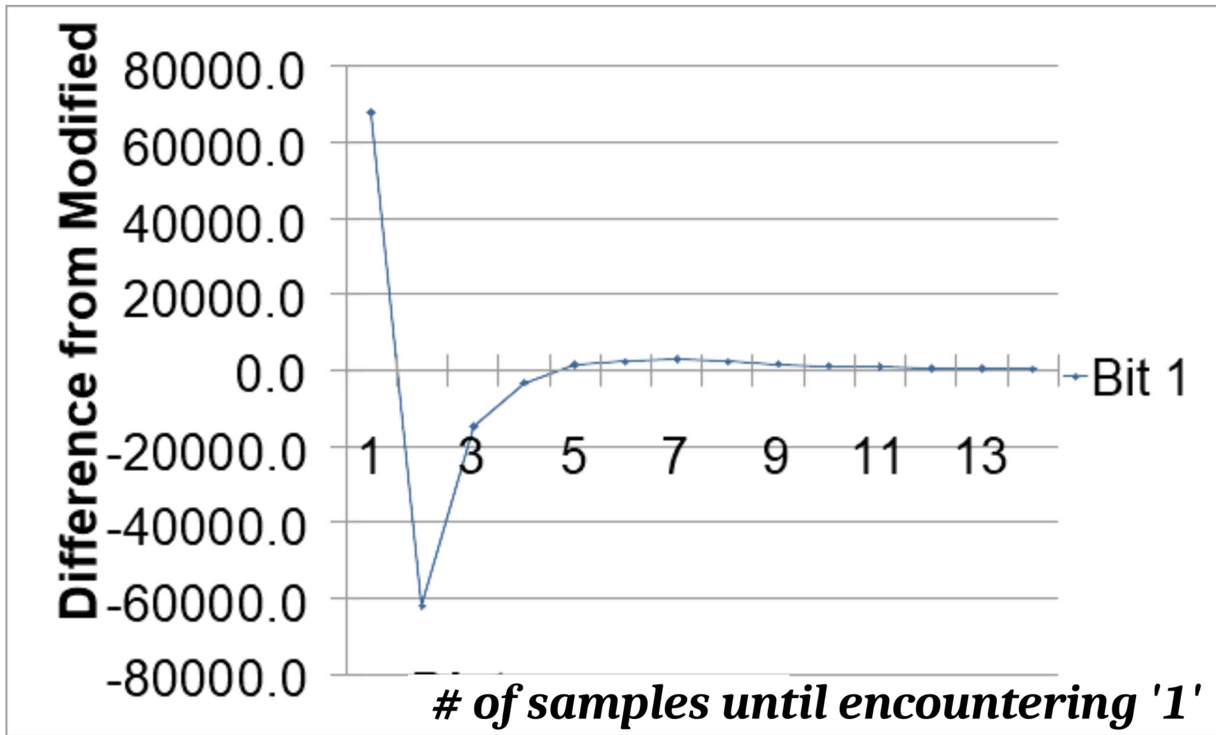


Chart 4.13: 111 Inverters, tuning for the first Ring at 100 KHz

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
18.01802	3072	185	97.5	
1s Frequency	Ideal	Bias	P(1)	P(0)
523,819	524288	-0.04%	49.96%	50.04%

Table 4.13: Parameters

Lastly, the results from the ring with 1111 inverters are shown.

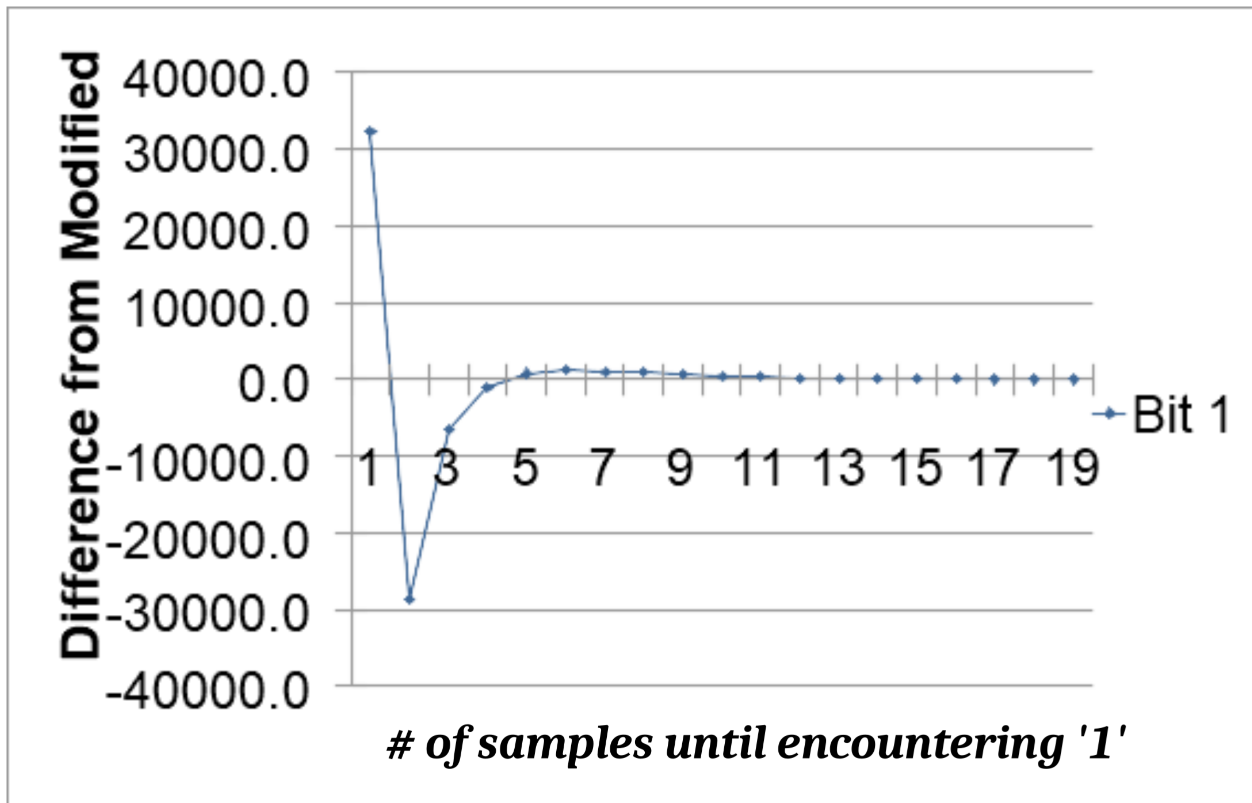


Chart 4.14: 1111 Inverters tuning phenomena, 4.6 KHz, 1st Bit

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	
1.80018	131070	786	4.6	
1s Frequency	Ideal	Bias	P(1)	P(0)
524,637	524288	0.03%	50.03%	49.97%

Table 4.14: Parameters.

This one exhibits significant tuning even for the maximum Skip count the system can be configured with (2^{16} = 65535), although the Elapsed Ring Oscillations parameter is still reduced.

The following graphs demonstrate the differences in bias for four different bits of the random number stream, of them belonging to a specific ring oscillator. In one graph the differences both in bias and in tuning patterns are visible accross the 4 ring oscillators. The following tables also summarize the results of the bias calculations per Ring Oscillator.

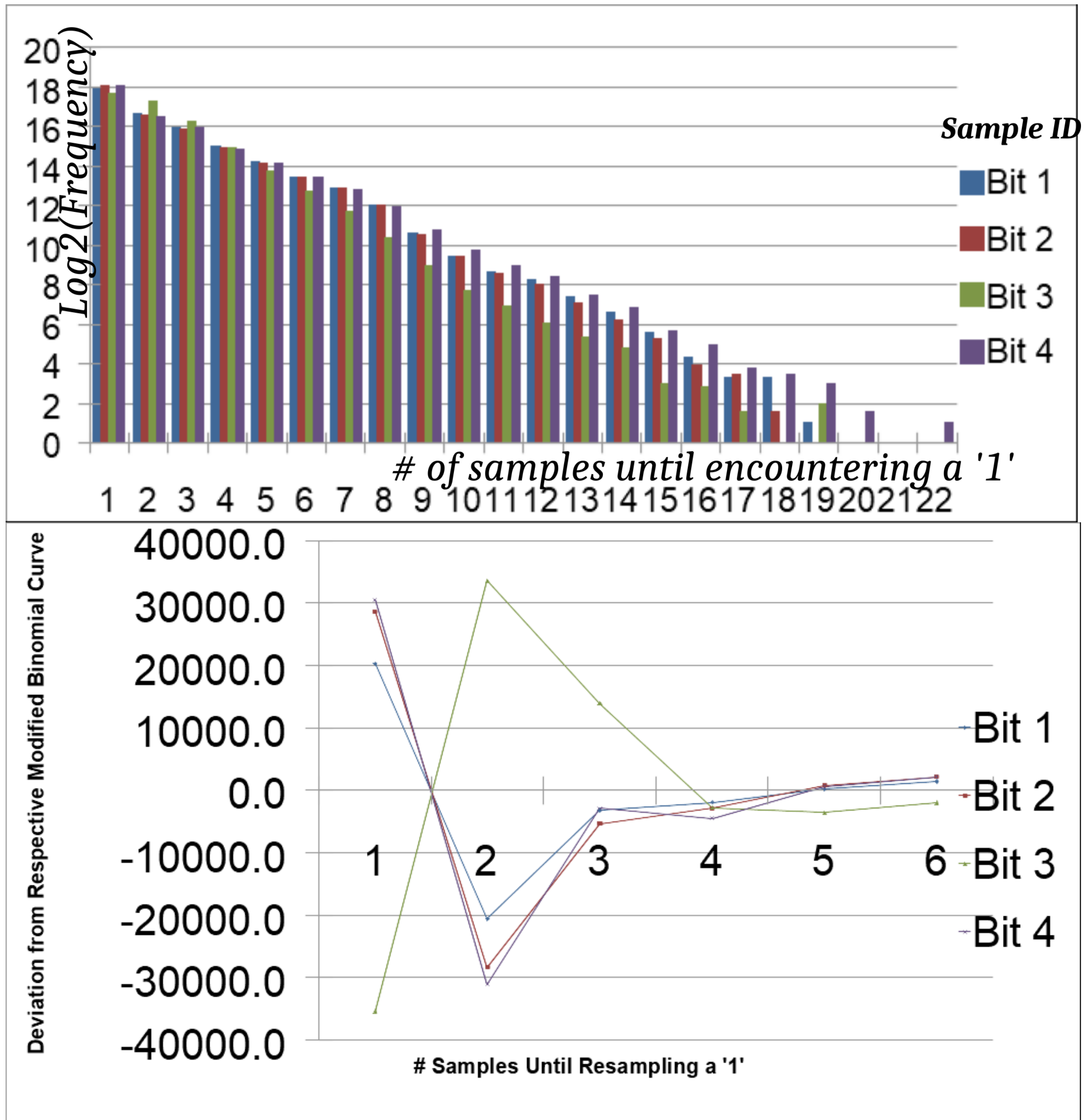


Chart 4.15: 5-Inverter per-bit distributions (top) and 5MHz Tuning Patterns (bottom)

	Ones Frequency	Ideal	Bias	P(1)	P(0)
Bit 1	490,022	524288	-3.27%	46.73%	53.27%
Bit 2	508456	524288	-1.51%	48.49%	51.51%
Bit 3	508270	524288	-1.53%	48.47%	51.53%
Bit 4	504859	524288	-1.85%	48.15%	51.85%

Table 4.16: 5-Inverter Ring Bias-Process Variations

This Process-sensitive attribute of the bias is corroborated by similar curves obtained by all the rings.

Finally, the moments are accumulated in the following table (in scale transformed units), and the relative deviations are shown.

	Moments (scale-transformed)					
Moment Order	3-Inv Ring	5-Inv Ring	11-Inv Ring	111-Inv Ring	1111-Inv Ring	Theoretical
1	3261	3343	3414	3436	3438	3436
2	1388	1441	1486	1501	1503	1501
3	6717	7013	7288	7381	7392	7378
4	3485	3648	3813	3870	3877	3868
5	1889	1979	2079	2114	2117	2112
6	1054	1105	1166	1188	1190	1187
7	6015	6303	6675	6815	6825	6805
8	3488	3654	3884	3971	3976	3964
9	2048	2145	2288	2343	2345	2338
10	1216	1272	1362	1396	1398	1393
	Deviation from Ideal					
Moment Order	3-Inv Ring	5-Inv Ring	11-Inv Ring	111-Inv Ring	1111-Inv Ring	
1	-5.09%	-2.71%	-0.64%	0.00%	0.06%	
2	-7.53%	-4.00%	-1.00%	0.00%	0.13%	
3	-8.96%	-4.95%	-1.22%	0.04%	0.19%	
4	-9.90%	-5.69%	-1.42%	0.05%	0.23%	
5	-10.56%	-6.30%	-1.56%	0.09%	0.24%	
6	-11.20%	-6.91%	-1.77%	0.08%	0.25%	
7	-11.61%	-7.38%	-1.91%	0.15%	0.29%	
8	-12.01%	-7.82%	-2.02%	0.18%	0.30%	
9	-12.40%	-8.25%	-2.14%	0.21%	0.30%	
10	-12.71%	-8.69%	-2.23%	0.22%	0.36%	

Table 4.17: 10 Raw Moments

Finally, the remediation of the small rings deficiencies with a Post Processor was attempted. The H3 Linear Post Processor described in Appendix D was used, that Halves the data rate by halving the output of each ring oscillator [7, 36], using the following process: It First separates the 16-bit input into an upper half H and a lower half L, and then proceeds to reduce them according to:
H EXOR L EXOR ROTL(L,1) EXOR ROTL(L,2) EXOR ROTL(L,4), as illustrated in figure 4.16. Here ROTL(L, N) means left rotation of the L vector by N positions. The resulting output bits' bias is a polynomial of the original bias. This particular Post-Processor is able to eradicate all powers of the bias expansion in the output bits up to the third [7]. This means the output bias' most significant term of the original is e^4 (since bias is a number less than 1). The H3 post processor was chosen instead of a Von Neumann extractor, due to the fact that it offers a greater and constant output rate, whereas the Von Neumann extractor has a smaller, and non constant output rate.

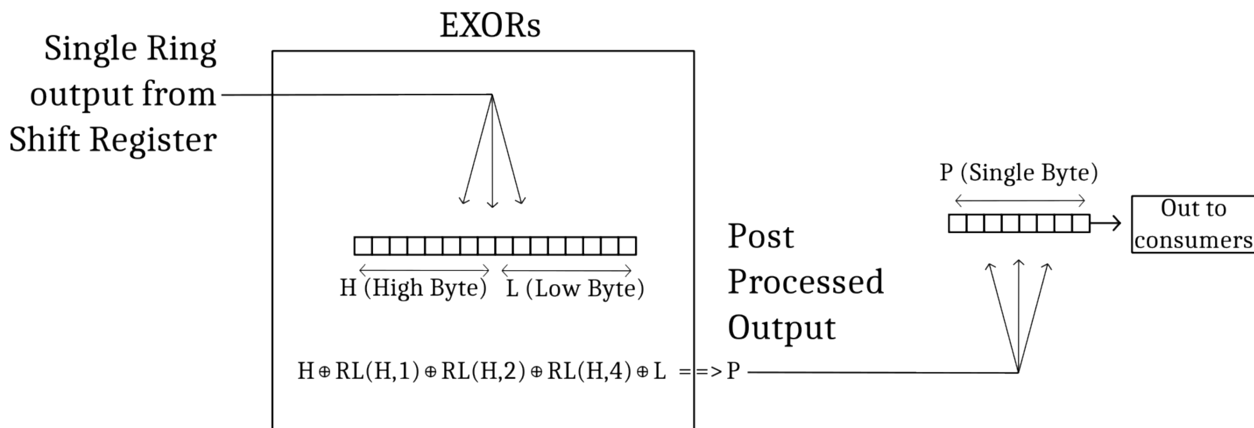


Figure 4.16: The H3 post processor's mode of operation

In tables 4.18 and 4.19, the correction of the small rings' bias with the use of the H3 Post Processor is shown.

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	Data Output Rate (KHz)
400	3,072	4096	97.5	48.8
1s Frequency	Ideal	Bias	P(1)	P(0)
525437	524288	0.11%	50.11%	49.89%

Table 4.18: 5-Inverter Ring Post Processed Output Parameters

Ring Frequency (MHz)	Skip	Elapsed Ring Ocillations	Sampling Frequency (KHz)	Data Output Rate (KHz)
400	120	160	2500	1250
1s Frequency	Ideal	Bias	P(1)	P(0)
524859	524288	0.05%	50.05%	49.95%

Table 4.19: 5-Inverter Ring Post Processed Output Parameters

The bias vanishes almost completely and the experimental curves are indistinguishable from the ideal. An indicative curve is presented in figure 4.17, from a 5-inverter ring at the maximum sampling rate of 2.5 MHz. This translates to an output of 1.25 MHz x 16-bits = 2.5 MByte/s.

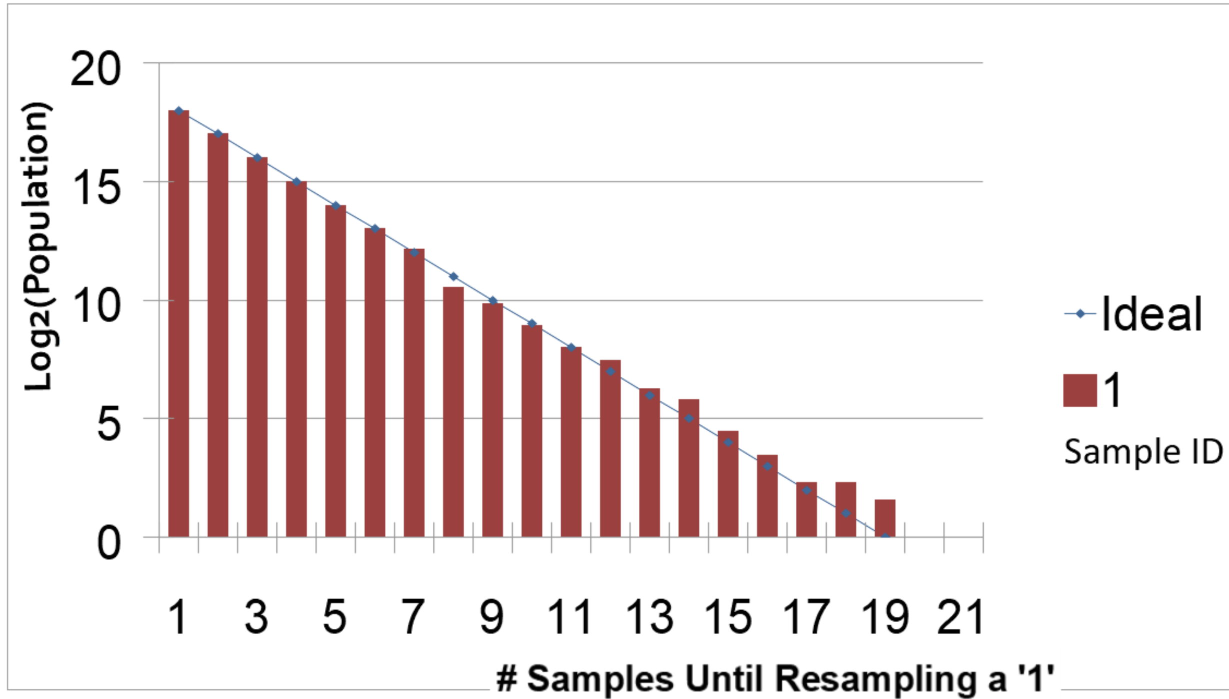


Figure 4.17: 5-Inverter rings sampled at 2.5 MHz, output rate of 1.25 MHz

5. Conclusions

Based on the results presented in chapter 4, the final conclusions are drawn regarding the various Rings' suitability for use in TRNGs.

The moments remained stable accross almost all sampling rates. The only case of instability was with the 1111 Inverter long ring, in a 5MHz sampling test (oversampling, not undersampling anymore). This caused a ~2% sway in moment magnitude accross runs (because of sequences samples without switching/jitter at the start and the end of the run, that yield large statistical imbalances. All the other cases, regardless of Binomial Distribution status or tuning phenomena, were very stable in their magnitude. This, again, is attributed to the inherent difference between the oscillators, wich causes adequate mixing accross the bits of the TRNG output. That is to say the uniformity (but not necessarily the unpredictability) of the distribution is easily achievable through the use of ring oscillators.

The moments deviation from the ideal seems to be essentially tied to the bias of the rings. The distributions that best resemble a uniform (based on the raw moments) are the ones from the largest rings. The smallest rings suffer large deviations, and that is attributed to their large bias.

Medium sized rings seem to offer enough substrate and electrical variability to even out duty cycle anomalies, while at the same time offering reasonable output rates for high quality random numbers. Bias, therefore, seems to even out towards zero when spreading accross multiple CLBs.

The bias is attributed to the duty cycle of the Ring Oscillators, which seems to deviate from the 50% value that would be expected. Part of the reason for the duty cycle deviation from 50% could be intrinsic Process properties, such as different access times between LUT memory elements. It could also be electrical [30], such as assymetry between CMOS rise and fall times caused by local voltage supply imbalances. Hence, when the individual rings span a large silicon die area, which also encompasses a large set of these particular parametric imbalances, these types of variabilities even out toword zero.

Summing those results up, for non post processed outputs, the most attractive solutions seem to be Medium Sized Rings (11 to 111 inverters long, they are more versatile in terms of space, they have excellent bias and moment performance. They can either be used in a small auxillary module that streams ramdom 16-bit numbers, or fashioned into a scaled up version with hundreds of rings for massive random number generation.

Larger rings demonstrate an inability to accumulate jitter, even for relatively large numbers of elapser ring oscillations. This indicates a relation between the total jitter variance σ^2 and the characteristic oscillator frequency, like $\sigma^{2\,=}\,\sigma^2(f)$.

Finally, when the H3 Linear Corrector was used at post processing, at the cost of 24 FLIP-FLOPs and 1 LUT (mapping 5 inputs to 1 output) per ring, and a data rate half of the sampling rate, the bias was almost completely corrected to 0, even for the highest sampling rates of 2.5MHz, and the experimental curves of the small rings becom indistinguishable from the ideal binomial curves for 0 bias.

This makes the post processed design with small rings the most attractive option of all for the purposes of creating a TRNG.

As part of possible future work, it is proposed that the silicon process' contribution to the ring's bias, if any, is reduced even more, by use of the following technique: replacement of select pairs of NOT gates by pairs of buffers in the ring, so as to invert their timing contribution toward a 50% duty cycle. This is shown in figure 5.1.

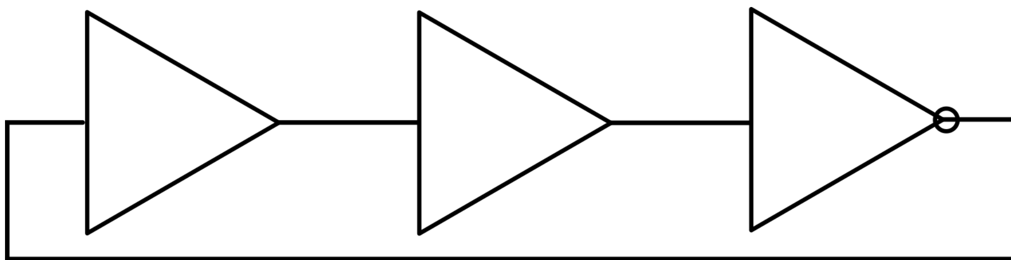


Figure 5.1: Replacing 2 Inverters with 2 buffers in a Ring oscillator, to invert their contribution toward a Duty Cycle

This process is adaptive, meaning that multiple inverter pairs, and numbers of inverter pairs have to be tested to find the best possible solution, that yields the lowest bias.

The TRNG can also be equipped with a SHA mixer [1], as is customary in TRNGs that are meant for use in cryptography.

References

1: NIST Special Publication 800-90C “Recommendation for Random Bit Generator (RBG) Constructions”, second Draft, 2016

2: Hofemeier, Gael (2011-06-22). "Find out about Intel's new RDRAND Instruction". Intel Developer Zone Blogs. Retrieved 30 December 2013

3: Montgomery, Douglas C.; Runger, George C. *Applied Statistics and Probability for Engineers* (6th ed.). Wiley 2014

4: Xilinx Inc., Ultrascale+ CLB user guide UG574, 2017

5: Xilinx Inc., ZCU102 Evaluation Board User Guide UG1182, 2019

6: Christian Walck: Hand-book on Statistical Distributions for experimentalists, 1996, last revision 2007

7: Dichtl, Markus, Bad and Good Ways of Post-processing Biased Physical Random Numbers. Lecture Notes in Computer Science, 2007

8: Xilinx Inc., Virtual Input Output V3.0 Product Guide PG159, 2018

9: Xilinx Inc., UltraScale Architecture DSP Slice User Guide UG579, 2021

10: Wikipedia, Entropy Source https://en.wikipedia.org/wiki/Entropy_(computing), 2022

11: Wikipedia, Ring Oscillator https://en.wikipedia.org/wiki/Ring_oscillator, 2022

12: D. A. Huffman, The Synthesis of Sequential Switching Circuits, 1954

13: Xilinx Inc. FPGAs & 3D ICs https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html, 2022

14: IEEE VHDL analysis and stantardization group https://open-

source.ieee.org/vasg

15: Volnei A. Pedroni, Circuit Design with VHDL, 2020

16: Xilinx Inc., Vivado Design Suite User Guide: Synthesis UG901, 2021

17: Official TCL webgage https://www.tcl.tk, 2022

18: Xilinx Inc., Vivado Design Suite TCL Command Reference Guiide UG835, 2022

19: Xilinx Inc., Vivado Design Suite Using TCL Scripting UG894, 2022

20: Xilinx Inc, The Programmable Logic Data Book 2000, 2000.

21: Xilinx Inc., XAPP094 Version 2.1 Metastable Recovery Application Note, November 1997

22: Xilinx Inc., XCell Issue 22, 3rd Quarter 1996, Article “Metastability Recovery in Xilinx FPGAs” Page 30

23: Xilinx Inc., XAPP077 Version 1.0 Metastability Considerations Application Note, 1997

24: Altera White Paper: Understanding Metastability in FPGAs version 1.2, 2009

25: Papoulis, A.. Probability, Random Variables, and Stochastic Processes, 2nd ed., McGraw Hill 1984

26: J. Christopher Westland, Audit Analytics, 2020

27: Wiener Process, N.Wiener Collected Works vol.1, 1976

28: Wikipedia Gaussian Random Walk, https://en.wikipedia.org/wiki/Random_walk, 2022

29: Ransom Stephens Jitter 360, The meaning of Total Jitter, 2005

30: An In-Depth Analysis of Ring Oscillators: Exploiting their configurable Duty Cycle, Javier Agustin, Marisa Lopez-Vallejo, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, VOL. 62, NO. 10, 2015

31: Boyan Valtchanov, Alain Aubert, Florent Bernard, Viktor Fischer Modeling

and observing the Jitter in Ring Oscillators implemented in FPGAs, 2008

32: Xilinx Inc., Vivado Design Suite User Guide, Using Constraints, UG903, 2022

33: Xilinx Inc.,Constraints Guide UG625, 2013

34: Wikipedia, Jitter https://en.wikipedia.org/wiki/Jitter, 2022

35: Xilinx Inc., Vivado Design Suite User Guide, Designing With IP, UG896, 2022

36: Patrick Lacharme, Post Processing Functions for a Biased Physical Random Number Generator, 2008

37: Parhami, Behrooz, Computer Arithmetic: Algorithms and Hardware Designs, Oxford University Press, 2000

Appendix A. VHDL code for the preliminary experiments

A.1 LongRings

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity longrings is
   Port (output : out std_logic_vector(3 downto 1);
        PARK   : in std_logic);
end longrings;

architecture Behavioral of longrings is

signal ring2001 : std_logic_vector(2000 downto 0);
signal ring3001 : std_logic_vector(3000 downto 0);
signal ring4001 : std_logic_vector(4000 downto 0);
attribute dont_touch : string;
attribute dont_touch of ring2001 : signal is "TRUE";
attribute dont_touch of ring3001 : signal is "TRUE";
attribute dont_touch of ring4001 : signal is "TRUE";
attribute allow_combinatorial_loops : string;
attribute allow_combinatorial_loops of ring2001 : signal is "TRUE";
attribute allow_combinatorial_loops of ring3001 : signal is "TRUE";
attribute allow_combinatorial_loops of ring4001 : signal is "TRUE";

begin

   make_ring2001:
   for I in 2000 downto 1 generate
   ring2001(I) <= NOT ring2001(I-1);
   end generate make_ring2001;
   ring2001(0) <= PARK NOR ring2001(2000);

   make_ring3001:
   for I in 3000 downto 1 generate
   ring3001(I) <= NOT ring3001(I-1);
   end generate make_ring3001;
   ring3001(0) <= PARK NOR ring3001(3000);
```

```
    make_ring4001:
    for I in 4000 downto 1 generate
    ring4001(I) <= NOT ring4001(I-1);
    end generate make_ring4001;
    ring4001(0) <= PARK NOR ring4001(4000);

    output(1) <= ring2001(2000);
    output(2) <= ring3001(3000);
    output(3) <= ring4001(4000);

end Behavioral;
```

A.2 Ring Oscillator Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VComponents.all;


entity mill is
  Port ( out_to_sampler : out std_logic;
        PARK : in std_logic);
end mill;


architecture Behavioral of mill is

constant Len : Integer := 5;
signal ring4 : std_logic_vector(Len4 -1 downto 0);

--Attributes needed so the design doesn't simplify logic and so that combinatorial
and timing loops are permitted
attribute dont_touch : string;
attribute dont_touch of ring4 : signal is "TRUE";

attribute ALLOW_COMBINATORIAL_LOOPS : string;
attribute ALLOW_COMBINATORIAL_LOOPS of ring4 : signal is "TRUE";

begin
```

```
   gen_r4 : for I in 0 to Len4 -2 generate
   LUT1_inst_r4 : LUT1
   generic map (
      INIT => "01")
   port map (
      O => ring4(i + 1),   -- LUT general output
      I0 => ring4(i)  -- LUT input
   );
   end generate gen_r4;

   LUT2_inst_r4 : LUT2
   generic map (
      INIT => "0001"  -- Logic function
   )
   port map (
      O => ring4(0),   -- 1-bit output: LUT
      I0 => ring4(Len4 -1), -- 1-bit input: LUT
      I1 => PARK  -- 1-bit input: LUT
   );


   out_to_sampler <= ring4(Len4 -1);


end Behavioral;
```

A.3 Generic Ring Oscillator Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity generic_mill is
   generic (N : integer);
   Port (output_bus : out std_logic_vector(N - 1 downto 0);
        PARK : in std_logic);
end generic_mill;

architecture Behavioral of generic_mill is

component mill
   port(out_to_sampler : out std_logic;
```

```
        PARK : in std_logic);
end component;

begin

generate_N_single_bit_mills:
for I in N-1 downto 0 generate
   mill_instance : mill port map (out_to_sampler => output_bus(I), PARK =>
PARK);
end generate generate_N_single_bit_mills;

end Behavioral;
```

A.4 Short Rings

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ShortRings is
  Port (PARK : in std_logic;


       --no sampling implemented as of yet
       --SClk : in std_logic;

       --Heating Element Clock and activation control
       HClk_p : in std_logic;
       HClk_n : in std_logic;
       HControl : in std_logic_vector(3 downto 0);

       --Ring Selector
       RSelect : in std_logic_vector(1 downto 0);

       --Direct Output to the PROTO Header
       Pinlines : out std_logic_vector(3 downto 0);
       Clk_to_header : out std_logic);
end ShortRings;
```

```
architecture Behavioral of ShortRings is

    -----------------------------------------------------
    -----------------------------------------------------
    --------Related to the Ring Oscillators

    constant NMILLS : Integer := 4;
    signal mills_out : std_logic_vector(4*NMILLS -1 downto 0);

    component generic_mill
       generic (N : integer);
       Port (out_to_whatever : out std_logic_vector(4*N - 1 downto 0);
             PARK : in std_logic);
     end component;

    --warning! very important: synthesizer will trim all mills_out signals if this is not used
     --seems that the dont touch attribute in the mill library isn't enough because mills_out is defined here
    attribute dont_touch : string;
    attribute dont_touch of mills_out : signal is "TRUE";

    -----------------------------------------------------
    -----------------------------------------------------
    --------Related to the Heating Elements

    --single ended clock for heater switching
    signal HClk : std_logic;
    --attribute dont_touch of HClk : signal is "true";
    constant HEATERLENGTH : Integer := 2000;
    constant HEATERSNUM : Integer := 15;
    signal Enabler : std_logic_vector(HEATERSNUM -1 downto 0);

    component pll
    Port
       (-- Clock in ports
       -- Clock out ports
       clk_700MHz          : out    std_logic;
       clk_11MHz          : out    std_logic;
       clk_in1_p         : in     std_logic;
       clk_in1_n         : in     std_logic
       );
    end component;
```

```
    component HEATER is
    Generic (Length : Integer;
           NHeaters : Integer);
    Port (Clk : in std_logic;
         Enable : in std_logic_vector(NHeaters -1 downto 0)
         );
    end component;


begin

   pll_to_700_and_11_MHz : pll
   port map (
  -- Clock out ports
   clk_700MHz => HClk,
   clk_11MHz => Clk_to_header,
   -- Clock in ports
   clk_in1_p => HClk_p,
   clk_in1_n => HClk_n
 );



    --generate 4*4-bit mills (effectively a word)
    word_mill : generic_mill
    generic map (N => NMILLS)
    port map (out_to_whatever => mills_out, PARK => PARK);

    --select any mill at your discretion to send to the PROTO header
    with RSelect select
    Pinlines <= mills_out(3 downto 0)   when "00",
                mills_out(7 downto 4)   when "01",
                mills_out(11 downto 8)  when "10",
                mills_out(15 downto 12) when "11";



    --This select statement is used for translating HControl
    with HControl select
    Enabler <= (others => '0')   when "0000",
```

```
            (0 => '1', others => '0') when "0001",
            (0 to 1 => '1', others => '0') when "0010",
            (0 to 2 => '1', others => '0') when "0011",
            (0 to 3 => '1', others => '0') when "0100",
            (0 to 4 => '1', others => '0') when "0101",
            (0 to 5 => '1', others => '0') when "0110",
            (0 to 6 => '1', others => '0') when "0111",
            (0 to 7 => '1', others => '0') when "1000",
            (0 to 8 => '1', others => '0') when "1001",
            (0 to 9 => '1', others => '0') when "1010",
            (0 to 10 => '1', others => '0') when "1011",
            (0 to 11 => '1', others => '0') when "1100",
            (0 to 12 => '1', others => '0')  when "1101",
            (0 to 13 => '1', others => '0') when "1110",
            (0 to 14 => '1', others => '0') when "1111";


        heater_instance : HEATER
        generic map (Length => HEATERLENGTH, NHeaters => HEATERSNUM)
        port map (Clk => HClk, Enable => Enabler);

end Behavioral;
entity ShortRings is
  Port (PARK : in std_logic;

        --no sampling implemented as of yet
        --SClk : in std_logic;

        --Heating Element Clock and activation control
        HClk_p : in std_logic;
        HClk_n : in std_logic;
        HControl : in std_logic_vector(3 downto 0);

        --Ring Selector
        RSelect : in std_logic_vector(1 downto 0);

        --Direct Output to the PROTO Header
        Pinlines : out std_logic_vector(3 downto 0);
        Clk_to_header : out std_logic);
end ShortRings;

architecture Behavioral of ShortRings is
```

```
    -----------------------------------------------------
    -----------------------------------------------------
    --------Related to the Ring Oscillators

    constant NMILLS : Integer := 4;
    signal mills_out : std_logic_vector(4*NMILLS -1 downto 0);

    component generic_mill
       generic (N : integer);
       Port (out_to_whatever : out std_logic_vector(4*N - 1 downto 0);
            PARK : in std_logic);
     end component;

    --warning! very important: synthesizer will trim all mills_out signals if this is not
used
     --seems that the dont touch attribute in the mill library isn't enough because
mills_out is defined here
    attribute dont_touch : string;
    attribute dont_touch of mills_out : signal is "TRUE";

    -----------------------------------------------------
    -----------------------------------------------------
    --------Related to the Heating Elements

    --single ended clock for heater switching
    signal HClk : std_logic;
    --attribute dont_touch of HClk : signal is "true";
    constant HEATERLENGTH : Integer := 2000;
    constant HEATERSNUM : Integer := 15;
    signal Enabler : std_logic_vector(HEATERSNUM -1 downto 0);

    component pll
    Port
       (-- Clock in ports
       -- Clock out ports
       clk_700MHz          : out    std_logic;
       clk_11MHz          : out    std_logic;
       clk_in1_p         : in     std_logic;
       clk_in1_n         : in     std_logic
       );
    end component;
```

```
   component HEATER is
   Generic (Length : Integer;
          NHeaters : Integer);
   Port (Clk : in std_logic;
        Enable : in std_logic_vector(NHeaters -1 downto 0)
        );
   end component;

begin

  pll_to_700_and_11_MHz : pll
  port map (
 -- Clock out ports
  clk_700MHz => HClk,
  clk_11MHz => Clk_to_header,
  -- Clock in ports
  clk_in1_p => HClk_p,
  clk_in1_n => HClk_n
 );

   --generate 4*4-bit mills (effectively a word)
   word_mill : generic_mill
   generic map (N => NMILLS)
   port map (out_to_whatever => mills_out, PARK => PARK);

   --select any mill at your discretion to send to the PROTO header
   with RSelect select
   Pinlines <= mills_out(3 downto 0)   when "00",
               mills_out(7 downto 4)   when "01",
               mills_out(11 downto 8)  when "10",
               mills_out(15 downto 12) when "11";

   --This select statement is used for translating HControl
   with HControl select
   Enabler <= (others => '0')   when "0000",
           (0 => '1', others => '0') when "0001",
           (0 to 1 => '1', others => '0') when "0010",
           (0 to 2 => '1', others => '0') when "0011",
           (0 to 3 => '1', others => '0') when "0100",
           (0 to 4 => '1', others => '0') when "0101",
           (0 to 5 => '1', others => '0') when "0110",
           (0 to 6 => '1', others => '0') when "0111",
```

```
            (0 to 7 => '1', others => '0') when "1000",
            (0 to 8 => '1', others => '0') when "1001",
            (0 to 9 => '1', others => '0') when "1010",
            (0 to 10 => '1', others => '0') when "1011",
            (0 to 11 => '1', others => '0') when "1100",
            (0 to 12 => '1', others => '0')  when "1101",
            (0 to 13 => '1', others => '0') when "1110",
            (0 to 14 => '1', others => '0') when "1111";


    heater_instance : HEATER
    generic map (Length => HEATERLENGTH, NHeaters => HEATERSNUM)
    port map (Clk => HClk, Enable => Enabler);

end Behavioral;
```

A.5 Heater

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity HEATER is
  Generic (Length : Integer;
         NHeaters : Integer);
  Port (Clk : in std_logic;
       Enable : in std_logic_vector(NHeaters -1 downto 0)
       );
end HEATER;

architecture Behavioral of HEATER is

  type sig2d is array(0 to NHeaters-1) of std_logic_vector(Length -1 downto 0);
  signal array2d : sig2d;
  signal flip : std_logic := '0';

  --important so the synthesizer won't optimize the heater chain on basis of logic
  attribute dont_touch : string;
  attribute dont_touch of array2d : signal is "true";


begin
```

```
   flipping_data:
   process(Clk)
   begin
      if (rising_edge(Clk)) then
         flip <= NOT flip;
      end if;
   end process;

   generate_heaters:
   for N in 0 to NHeaters -1 generate
      process(Clk)
      begin
         if (rising_edge(Clk) AND Enable(N) = '1') then
         array2d(N)(0) <= flip;
           for I in 1 to Length -1 loop
               array2d(N)(I) <= array2d(N)(I-1);
            end loop;
         end if;
      end process;
   end generate generate_heaters;

end Behavioral;
```

A.6 Latch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Latch is
   Generic (Length : Integer);
   Port (Clk : in std_logic;
        Enable : in std_logic;
        led : out std_logic);
end Latch;

architecture Behavioral of Latch is

   signal data : std_logic_vector(Length -1 downto 0);
   signal flip : std_logic := '0';

   attribute dont_touch : string;
```

```
    attribute dont_touch of data : signal is "true";

begin

    process(Clk)
    begin
    if(rising_edge(Clk)) then
        flip <= NOT flip;
        data(0) <= flip;
            for I in 0 to Length -2 loop
                data(I+1) <= data(I);
            end loop;
        end if;
    end process;

    led <= data(Length -1);

end Behavioral;
```

A.6 XDC Constraints for LongRings

```
set_property PACKAGE_PIN G15 [get_ports {output[3]}]
#PROTO 12 4001
set_property PACKAGE_PIN G14 [get_ports {output[2]}]
#PROTO 10 3001
set_property PACKAGE_PIN J14 [get_ports {output[1]}]
#PROTO 8 2001
set_property PACKAGE_PIN AK13 [get_ports {PARK}]
set_property IOSTANDARD LVCMOS33 [get_ports {output[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {output[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {output[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {PARK}]
```

A.7 XDC Constraints for ShortRings

```
set_property PACKAGE_PIN G15 [get_ports {Pinlines[3]}]
#PROTO 12
set_property PACKAGE_PIN G14 [get_ports {Pinlines[2]}]
```

```
#PROTO 10
set_property PACKAGE_PIN J14 [get_ports {Pinlines[1]}]
#PROTO 8
set_property PACKAGE_PIN H14 [get_ports {Pinlines[0]}]
#PROTO 6
set_property PACKAGE_PIN AK13 [get_ports PARK]
#SW 13.1
set_property PACKAGE_PIN AL13 [get_ports {RSelect[0]}]
#SW 13.2
set_property PACKAGE_PIN AP12 [get_ports {RSelect[1]}]
#SW 13.3
set_property PACKAGE_PIN AN13 [get_ports {HControl[0]}]
#SW 13.5
set_property PACKAGE_PIN AM14 [get_ports {HControl[1]}]
#SW 13.6
set_property PACKAGE_PIN AP14 [get_ports {HControl[2]}]
#SW 13.7
set_property PACKAGE_PIN AN14 [get_ports {HControl[3]}]
#SW 13.8
#set_property PACKAGE_PIN AE13 [get_ports {led}]
#led 2
set_property PACKAGE_PIN AL8 [get_ports HClk_p]
#SI570 clock
set_property PACKAGE_PIN H13 [get_ports Clk_to_header]
#10.937...MHz pll output
#PROTO 24

set_property IOSTANDARD LVCMOS33 [get_ports {Pinlines[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Pinlines[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Pinlines[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Pinlines[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {RSelect[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {RSelect[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {HControl[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {HControl[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {HControl[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {HControl[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports PARK]
set_property IOSTANDARD DIFF_SSTL12 [get_ports HClk_p]
set_property IOSTANDARD LVCMOS33 [get_ports Clk_to_header]
```

A.8 TCL Script Constraints

```
set_property LUTNM DISABLED [get_cells -hier *LUT1_inst*]

set_property LOCK_PINS {I0:A6} [get_cells -hier *LUT1_inst*]

set_property LOCK_PINS {I0:A6 I1:A5} [get_cells -hier *LUT2_inst*]

##get mill identifiers
set millindex [get_cells -hier *generate_N_single_bit_mills*]
##set millindex [lsort -dictionary $millindex]
set xorigin 36
set yorigin 0

##we can access the cellgroups as such
##get_cells *[lindex $millindex $i]/*
##the / is deeded to access the cell directory under the master cell
##hierarchy generate_N_single_bit_mills[i].mill_instance
##Even better: separate LUT1 and LUT2 as such
##get_cells *[lindex $millindex $i]/*.LUT1_inst_r4*
##get_cells *[lindex $millindex $i]/*.LUT2_inst_r4*
##and so on, or even more better
## loop in r4 with a $com variable
##the plan is to make rectangles

##define a mapping function
proc standardmapping {i cell} {
switch [expr $i % 8] {
0 {
set_property BEL A6LUT $cell }
1 {
set_property BEL B6LUT $cell }
2 {
set_property BEL C6LUT $cell }
3 {
set_property BEL D6LUT $cell }
4 {
set_property BEL E6LUT $cell }
5 {
set_property BEL F6LUT $cell }
6 {
set_property BEL G6LUT $cell }
```

```
7 {
set_property BEL H6LUT $cell }
}
}

##get all cells
set r4cells [get_cells -hier *LUT1_inst_r4]
##sort them in a dictionary fashion
set r4cells [lsort -dictionary $r4cells]
##to select origin by hand for RLOC_ORIGIN highlight a slice and type
##join  "X[get_property  RPM_X  [get_selected_objects]]Y[get_property  RPM_Y
[get_selected_objects]]"
##set_property RLOC_ORIGIN X2400Y18 [lindex $cells 0]
##however it is supported only in HDL attribute form
##cannot be set by XDC or TCL for that matter

##the millindex integer iterator
set index 0
##will be reset in a case statement
set i1 0
set i2 0
set i3 0
set i4 0

set pack1 $millindex
##get all cells
set r4cells1 [get_cells *[lindex $pack1 0]/*LUT1_inst_r4]
##sort them in a dictionary fashion
set r4cells1 [lsort -dictionary $r4cells1]
foreach mill_i $millindex {
set r4list [get_cells *${mill_i}/*.LUT1_inst_r4]
set r4list [lsort -dictionary $r4list]
##watch out, LUT2 does not have a . because it
##does not come from gen_rn directives
set r4nor [get_cells *${mill_i}/*LUT2_inst_r4]
lappend r4list $r4nor
set H4 [expr [llength $r4list]/8 + 1]
##offset +4 every 4 mills
set offset [expr 1 * [expr $index/2]]
##reset collumn iterators
if {{$index % 4} == 0} {
set i4 0
set i3 0
```

```
set i2 0
set i1 0}

foreach r4 $r4list {
switch [expr $index % 4] {
0 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
##switch for the bels
standardmapping $i4 $r4
}
1 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8 + $H4]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
standardmapping $i4 $r4
}
2 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8 2*$H4]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
standardmapping $i4 $r4
}
3 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8 + 3*$H4]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
standardmapping $i4 $r4
}
}
incr i4
}
incr index
set i4 0
}
```

Appendix B. VHDL and TCL code for the TRNG System

B.1 Generic Ring Oscillator Module
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```
entity generic_mill is
   generic (N : integer);
   Port (output_bus : out std_logic_vector(N - 1 downto 0);
         PARK : in std_logic);
end generic_mill;

architecture Behavioral of generic_mill is

component mill
   port(out_to_sampler : out std_logic;
        PARK : in std_logic);
end component;

begin

generate_N_single_bit_mills:
for I in N-1 downto 0 generate
      mill_instance : mill port map (out_to_sampler => output_bus(I), PARK =>
PARK);
end generate generate_N_single_bit_mills;

end Behavioral;
```

B.2 Ring Oscillator Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VComponents.all;
```

```
entity mill is
  Port ( out_to_sampler : out std_logic;
         PARK : in std_logic);
end mill;


architecture Behavioral of mill is

constant Len4 : Integer := 111;
signal ring4 : std_logic_vector(Len4 -1 downto 0);

--Attributes needed so the design doesn't simplify logic and so that combinatorial
and timing loops are permitted
attribute dont_touch : string;
attribute dont_touch of ring4 : signal is "TRUE";
attribute ALLOW_COMBINATORIAL_LOOPS : string;
attribute ALLOW_COMBINATORIAL_LOOPS of ring4 : signal is "TRUE";

begin

   gen_r4 : for I in 0 to Len4 -2 generate
   LUT1_inst_r4 : LUT1
   generic map (
      INIT => "01")
   port map (
      O => ring4(i + 1),   -- LUT general output
      I0 => ring4(i)  -- LUT input
   );
   end generate gen_r4;

   LUT2_inst_r4 : LUT2
   generic map (
      INIT => "0001"  -- Logic function
   )
   port map (
      O => ring4(0),   -- 1-bit output: LUT
      I0 => ring4(Len4 -1), -- 1-bit input: LUT
      I1 => PARK  -- 1-bit input: LUT
   );

   out_to_sampler <= ring4(Len4 -1);

end Behavioral;
```

B.3 Synchronizer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Synchronizer is
   Generic ( QWidth : Integer);
   Port ( clk : in std_logic;
         data_in : in std_logic_vector(QWidth -1 downto 0);
         data_out : out std_logic_vector(QWidth -1 downto 0));
end Synchronizer;

architecture Behavioral of Synchronizer is

--note: data metastable could be implemented in a generic vector fashion
--so as to have a selectable synchronization length
signal data_metastable : std_logic_vector(QWidth -1 downto 0) := (others => '0');

begin

   SCFIFO : process(clk)
   begin
      if (rising_edge(clk)) then
         --TICK 1
         data_metastable <= data_in;

         --TICK 2
         data_out <= data_metastable;
      end if;
   end process SCFIFO;

end Behavioral;
```

B.3 XDC Constraints

```
set_property PACKAGE_PIN AL8 [get_ports si570_clk_p]
set_property IOSTANDARD DIFF_SSTL12 [get_ports si570_clk_p]

set_clock_groups -asynchronous -group {clk_300_clk_wiz_0} -group
{clk_120_clk_wiz_0}

create_generated_clock -name deciclock -source [get_ports si570_clk_p] -di-
vide_by 60 [get_pins Stats_Instance/sclk_reg/Q]

set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
connect_debug_port dbg_hub/clk [get_nets clk_120]
```

B.4 TCL Constrains File

```
set_property LUTNM DISABLED [get_cells -hier *LUT1_inst*]

set_property LOCK_PINS {I0:A6} [get_cells -hier *LUT1_inst*]

set_property LOCK_PINS {I0:A6 I1:A5} [get_cells -hier *LUT2_inst*]

##get mill identifiers
set millindex [get_cells -hier *generate_N_single_bit_mills*]
##set millindex [lsort -dictionary $millindex]
set xorigin 36
set yorigin 0

##we can access the cellgroups as such
##get_cells *[lindex $millindex $i]/*
##the / is deeded to access the cell directory under the master cell
##hierarchy generate_N_single_bit_mills[i].mill_instance
##Even better: separate LUT1 and LUT2 as such
##get_cells *[lindex $millindex $i]/*.LUT1_inst_r3*
##get_cells *[lindex $millindex $i]/*.LUT2_inst_r3*
##and so on, or even more better
## loop in r1 r2 r3 r4 with a $com variable

##the plan is to make rectangles
```

```
##define a mapping function
proc standardmapping {i cell} {
switch [expr $i % 8] {
0 {
set_property BEL A6LUT $cell }
1 {
set_property BEL B6LUT $cell }
2 {
set_property BEL C6LUT $cell }
3 {
set_property BEL D6LUT $cell }
4 {
set_property BEL E6LUT $cell }
5 {
set_property BEL F6LUT $cell }
6 {
set_property BEL G6LUT $cell }
7 {
set_property BEL H6LUT $cell }
}
}

##get all cells
set r4cells [get_cells -hier *LUT1_inst_r4]
##sort them in a dictionary fashion
set r4cells [lsort -dictionary $r4cells]
##to select origin by hand for RLOC_ORIGIN highlight a slice and type
##join "X[get_property RPM_X [get_selected_objects]]Y[get_property RPM_Y
[get_selected_objects]]"
##set_property RLOC_ORIGIN X2400Y18 [lindex $cells 0]
##however it is supported only in HDL attribute form
##cannot be set by XDC or TCL for that matter

##the millindex integer iterator
set index 0
##will be reset in a case statement
set i1 0
set i2 0
set i3 0
set i4 0
```

```
set pack1 $millindex
##get all cells
set r4cells1 [get_cells *[lindex $pack1 0]/*LUT1_inst_r4]
##sort them in a dictionary fashion
set r4cells1 [lsort -dictionary $r4cells1]


foreach mill_i $millindex {
set r4list [get_cells *${mill_i}/*.LUT1_inst_r4]
set r4list [lsort -dictionary $r4list]
##watch out, LUT2 does not have a . because it
##does not come from gen_rn directives
set r4nor [get_cells *${mill_i}/*LUT2_inst_r4]
lappend r4list $r4nor
set H4 [expr [llength $r4list]/8 + 1]


##offset +2 every 4 mills
set offset [expr 1 * [expr $index/2]]
##reset collumn iterators
if {{$index % 4} == 0} {
set i4 0
set i3 0
set i2 0
set i1 0}

foreach r4 $r4list {
switch [expr $index % 4] {
0 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
##switch for the bels
standardmapping $i4 $r4
}
1 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8 + $H4]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
standardmapping $i4 $r4
}
2 {
set theloc4x [expr $xorigin + $offset]
```

```
set theloc4y [expr $yorigin + $i4/8]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
standardmapping $i4 $r4
}
3 {
set theloc4x [expr $xorigin + $offset]
set theloc4y [expr $yorigin + $i4/8 + $H4]
set_property LOC SLICE_X${theloc4x}Y${theloc4y} [get_cells $r4]
standardmapping $i4 $r4
}
}

incr i4
}
incr index


set i4 0
}
```

Appendix C. VHDL code for the Statistical Engine

C.1 Top Level Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
Library UNISIM;
use UNISIM.vcomponents.all;

entity WordRng is
   Port ( --PARK : in std_logic;
          si570_clk_p : in STD_LOGIC;
          si570_clk_n : in STD_LOGIC);
end WordRng;

architecture Behavioral of WordRng is
--number of Ring Oscillator Modules, and also bitness control of the design
constant NMills : Integer := 16;
--carrier signal of the 300MHz Clock from the pll output
signal si570_clk : std_logic;
--decimator clock
signal sampling_clk : std_logic := '0';
--carrier signal for clocking IO probe
signal clk_IO : std_logic := '0';
--output of Ring Oscillator Modules
signal mills_out : std_logic_vector(NMills -1 downto 0);
--park signal controllable from IO
signal PARK : std_logic;

-----------------accumulation register parametrisation------------------
constant QWidth : Integer := NMills;
constant Acc : Integer := 20;
constant SampleSize : Integer := 2**Acc;
--adding a QWidth number 2^20 = 1024*1024 times needs QWidth + 20 bits
--adding the square of a QWidth number 2^20 = 1024*1024 times needs 2*QWidth + 20 bits
--and so on
constant MILLFREQ : Integer := 50;
```

```
--MHz rough base frequency of observed Ring Oscillations
constant SI570FREQ : Integer := 300;
--MHz frequency of incoming clock
constant SKIP : Integer := (2**8) * (SI570FREQ/MILLFREQ);
signal variable_skip : std_logic_vector(15 downto 0);
--this is the downsampling parameter used to allow expression of jitter in our TRNG
--it accounts for the division of the SI570 clock too

----------------constant containing the register widths-----------------
type IntArray is array(Integer range<>) of Integer;
constant WMoment : IntArray(1 to 10) := (
QWidth+Acc, 2*QWidth+Acc, 3*QWidth+Acc, 4*QWidth+Acc, 5*QWidth+Acc,
6*QWidth+Acc, 7*QWidth+Acc, 8*QWidth+Acc, 9*QWidth+Acc,
10*QWidth+Acc);

signal decimated_stream : std_logic_vector(QWidth -1 downto 0);

--warning! very important: synthesizer will trim all mills_out signals if this is not used
--seems that the don't touch attribute in the mill library isn't enough because mills_out is defined here
attribute dont_touch : string;
attribute dont_touch of mills_out : signal is "true";
--attribute dont_touch of MASKIP: in std_logic_vector(15 downto 0);CC : signal is "true";

component generic_mill
   Generic (N : Integer);
   Port (output_bus : out std_logic_vector(N - 1 downto 0);
        PARK : in std_logic);
end component;


component clk_wiz_0
   Port(-- Clock in ports
       -- Clock out ports
       clk_300         : out    std_logic;
       clk_120         : out    std_logic;
       clk_in1_p        : in     std_logic;
       clk_in1_n        : in     std_logic
      );
end component;
```

```
COMPONENT vio_1
  PORT (
    clk : IN STD_LOGIC;
    probe_out0 : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END COMPONENT;

component Decimator is
    Generic (QWidth : Integer);
    Port ( clk : in std_logic;
           SKIP: in std_logic_vector(15 downto 0);
           feed : in std_logic_vector(QWidth -1 downto 0);
           decimated_stream : out std_logic_vector(QWidth -1 downto 0));
end component;

component Stats is
    Generic(QWidth, Acc, SampleSize, WMoment1, WMoment2,
            WMoment3, WMoment4, WMoment5,
            WMoment6, WMoment7, WMoment8,
            WMoment9, WMoment10 :Integer);
    Port(clk : in std_logic;
         clk_IO : in std_logic;
         SKIP: in std_logic_vector(15 downto 0);
         feed : in std_logic_vector(QWidth -1 downto 0);
         PARK : out std_logic
        );
end component;

begin

    --generate a word rng
    word_mill : generic_mill
    Generic map (N => NMills)
    Port map (output_bus => mills_out, PARK => PARK);

    PLL_300_120 : clk_wiz_0
    Port map (
    -- Clock out ports
    clk_300 => si570_clk,
    clk_120 => clk_IO,
    -- Clock in ports
    clk_in1_p => si570_clk_p,
    clk_in1_n => si570_clk_n
```

```
    );

    vio_skipcontrol : vio_1
    PORT MAP (
    clk => CLK_IO,
    probe_out0 => variable_skip
    );

   --decimate the Ring Oscillator Module output so as to allow for the manifesta-
tion of jitter
    DCM : Decimator
    Generic map(QWidth => QWidth)
     Port map(clk => si570_clk, SKIP => variable_skip, feed => mills_out, decimat-
ed_stream => decimated_stream);

    Stats_Instance: Stats
      generic map (QWidth => QWidth, Acc => Acc, SampleSize => SampleSize,
WMoment1 => WMoment(1),
                  WMoment2 => Wmoment(2), WMoment3 => Wmoment(3), WMo-
ment4 => Wmoment(4), WMoment5 => Wmoment(5),
                  WMoment6 => Wmoment(6), WMoment7 => Wmoment(7), WMo-
ment8 => Wmoment(8), WMoment9 => Wmoment(9),
               WMoment10 => Wmoment(10))
     port map (clk_IO => clk_IO, clk => si570_clk, SKIP => variable_skip, feed =>
decimated_stream,  PARK => PARK);

end Behavioral;
```

C.2 Decimator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity Decimator is
   Generic (QWidth, SKIP : Integer);
   Port ( clk : in std_logic;
        feed : in std_logic_vector(QWidth -1 downto 0);
        decimated_stream : out std_logic_vector(QWidth -1 downto 0));
end Decimator;

architecture Behavioral of Decimator is

--generated divided clock
--purpose is to allow the manifestation of jitter
signal dec_clk : std_logic := '0';

component Synchronizer is
   Generic ( QWidth : Integer);
   Port ( clk : in std_logic;
        data_in : in std_logic_vector(QWidth -1 downto 0);
        data_out : out std_logic_vector(QWidth -1 downto 0));
end component;

begin

   --It will be passed through the normal switching fabric
   --meaning high latency and skew, however here the use is strictly local,
   --so it's fine here
   --maybe the use of a BUFR (regional buffer) would be simpler
   CLOCK_DIVIDER : process(clk)
   variable i : Integer range 0 to SKIP-1 := 0;
   begin
      if (rising_edge(clk)) then
         i := i + 1;
         if (i = SKIP -1) then
            i := 0;
            dec_clk <= NOT dec_clk;
         end if;
      end if;
   end process;

   SYNCH : Synchronizer
   Generic map(QWidth => QWidth)
   Port map(clk => dec_clk, data_in => feed, data_out => decimated_stream);

end Behavioral;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Decimator is
   Generic (QWidth, SKIP : Integer);
   Port ( clk : in std_logic;
          feed : in std_logic_vector(QWidth -1 downto 0);
          decimated_stream : out std_logic_vector(QWidth -1 downto 0));
end Decimator;

architecture Behavioral of Decimator is

--generated divided clock
--purpose is to allow the manifestation of jitter
signal dec_clk : std_logic := '0';

component Synchronizer is
   Generic ( QWidth : Integer);
   Port ( clk : in std_logic;
          data_in : in std_logic_vector(QWidth -1 downto 0);
          data_out : out std_logic_vector(QWidth -1 downto 0));
end component;

begin

   --It will be passed through the normal switching fabric
   --meaning high latency and skew, however here the use is strictly local,
   --so it's fine here
   --maybe the use of a BUFR (regional buffer) would be simpler
   CLOCK_DIVIDER : process(clk)
   variable i : Integer range 0 to SKIP-1 := 0;
   begin
      if (rising_edge(clk)) then
         i := i + 1;
```

```
        if (i = SKIP -1) then
           i := 0;
           dec_clk <= NOT dec_clk;
        end if;
     end if;
  end process;

  SYNCH : Synchronizer
  Generic map(QWidth => QWidth)
  Port map(clk => dec_clk, data_in => feed, data_out => decimated_stream);


end Behavioral;
```

C.3 Statistical module for 10 Raw Moments and 4-bit Lateral 1s sums

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Stats is
  Generic(QWidth, Acc, SampleSize, SKIP, WMoment1,
        WMoment2, WMoment3, WMoment4, WMoment5,
        WMoment6, WMoment7, WMoment8,
        WMoment9, WMoment10 : Integer);
  Port (clk : in std_logic;
      clk_IO : in std_logic;
      PARK : out std_logic;
      feed : in std_logic_vector(QWidth -1 downto 0)
     );
end Stats;

architecture Behavioral of Stats is

--regional divided clock
--we do this instead of passing a global divided clock through the switching fabric
--as it would add tremendous skew and latency to be used as a common clock
between
--the statistical engine and the synchronizer
```

```
signal sclk : std_logic := '0';

signal bStart : std_logic := '0';
signal bSet : std_logic := '0';
signal reserved : std_logic := '0';

signal bStage1Ready : std_logic := '0';
signal bStage2Ready : std_logic := '0';
signal bStage3Ready : std_logic := '0';
signal bStage4Ready : std_logic := '0';
signal bStage5Ready : std_logic := '0';
signal bStage6Ready : std_logic := '0';
signal bStage7Ready : std_logic := '0';
signal bStage8Ready : std_logic := '0';
signal bStage9Ready : std_logic := '0';
signal bStage10Ready : std_logic := '0';
signal bStage11Ready : std_logic := '0';
signal bStage12Ready : std_logic := '0';
signal bStage13Ready : std_logic := '0';
signal bMomentsReady : std_logic := '0';

signal onescount1 : Unsigned(1 downto 0) := (others => '0');
signal onescount2 : Unsigned(1 downto 0) := (others => '0');
signal onescount3 : Unsigned(1 downto 0) := (others => '0');
signal onescount4 : Unsigned(1 downto 0) := (others => '0');
signal onescount5 : Unsigned(1 downto 0) := (others => '0');
signal onescount6 : Unsigned(1 downto 0) := (others => '0');
signal onescount7 : Unsigned(1 downto 0) := (others => '0');
signal onescount8 : Unsigned(1 downto 0) := (others => '0');
signal onesPartsum1, onesPartsum2, onesPartsum3, onesPartsum4 : Un-
signed(4 downto 0) := (others => '0');
signal onesBsum1,onesBsum2 : Unsigned(4 downto 0) := (others => '0');
signal bit1, bit2, bit3, bit4, bit5, bit6, bit7, bit8, bit9, bit10, bit11, bit12, bit13, bit14,
bit15, bit16 : Unsigned(0 downto 0);
signal CountOfOnes : Unsigned (4 downto 0) := (others => '0');
signal TotalCountOfOnes : Unsigned (Acc + 4 downto 0) := (others => '0');


COMPONENT vio_0
  PORT (
    clk : IN STD_LOGIC;
    probe_in0 : IN STD_LOGIC_VECTOR(35 DOWNTO 0);
    probe_in1 : IN STD_LOGIC_VECTOR(51 DOWNTO 0);
```

```
    probe_in2 : IN STD_LOGIC_VECTOR(67 DOWNTO 0);
    probe_in3 : IN STD_LOGIC_VECTOR(83 DOWNTO 0);
    probe_in4 : IN STD_LOGIC_VECTOR(99 DOWNTO 0);
    probe_in5 : IN STD_LOGIC_VECTOR(115 DOWNTO 0);
    probe_in6 : IN STD_LOGIC_VECTOR(131 DOWNTO 0);
    probe_in7 : IN STD_LOGIC_VECTOR(147 DOWNTO 0);
    probe_in8 : IN STD_LOGIC_VECTOR(163 DOWNTO 0);
    probe_in9 : IN STD_LOGIC_VECTOR(179 DOWNTO 0);
    probe_in10 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in11 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in12 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in13 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in14 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in15 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in16 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in17 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in18 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in19 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in20 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in21 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in22 : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
    probe_in23 : IN STD_LOGIC_VECTOR(24 DOWNTO 0);
    probe_out0 : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_out1 : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_out2 : OUT STD_LOGIC_VECTOR(0 DOWNTO 0)
  );
END COMPONENT;

    --registers
    signal D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11: UNSIGNED(QWidth -1
downto 0) := (others => '0');
    signal SQ1, SQ2 : UNSIGNED(2*QWidth -1 downto 0) := (others => '0');
    signal C1, C2 : UNSIGNED(3*QWidth -1 downto 0) := (others => '0');
    signal Q1, Q2 : UNSIGNED(4*QWidth -1 downto 0) := (others => '0');
    signal P1, P2 : UNSIGNED(5*QWidth -1 downto 0) := (others => '0');
    signal HX1, HX2 : UNSIGNED(6*QWidth -1 downto 0) := (others => '0');
    signal HP1, HP2 : UNSIGNED(7*QWidth -1 downto 0) := (others => '0');
    signal O1, O2 : UNSIGNED(8*QWidth -1 downto 0) := (others => '0');
    signal E1, E2 : UNSIGNED(9*QWidth -1 downto 0) := (others => '0');
    signal DEC1, MREGDEC1 : UNSIGNED(10*QWidth -1 downto 0) := (others
=> '0');
    signal regMOMENT1, oregMOMENT1 : UNSIGNED(WMoment1 -1 downto 0)
:= (others => '0');
```

```
    signal regMOMENT2, oregMOMENT2 : UNSIGNED(WMoment2 -1 downto 0)
:= (others => '0');
    signal regMOMENT3, oregMOMENT3 : UNSIGNED(WMoment3 -1 downto 0)
:= (others => '0');
    signal regMOMENT4, oregMOMENT4 : UNSIGNED(WMoment4 -1 downto 0)
:= (others => '0');
    signal regMOMENT5, oregMOMENT5 : UNSIGNED(WMoment5 -1 downto 0)
:= (others => '0');
    signal regMOMENT6, oregMOMENT6 : UNSIGNED(WMoment6 -1 downto 0)
:= (others => '0');
    signal regMOMENT7, oregMOMENT7 : UNSIGNED(WMoment7 -1 downto 0)
:= (others => '0');
    signal regMOMENT8, oregMOMENT8 : UNSIGNED(WMoment8 -1 downto 0)
:= (others => '0');
    signal regMOMENT9, oregMOMENT9 : UNSIGNED(WMoment9 -1 downto 0)
:= (others => '0');
   signal regMOMENT10, oregMOMENT10 : UNSIGNED(WMoment10 -1 downto
0) := (others => '0');

   begin


   CLOCK_DIVIDER : process(clk)
   variable i : Integer range 0 to SKIP-1 := 0;
   begin
      if (rising_edge(clk)) then
         i := i + 1;
         if (i = SKIP -1) then
            i := 0;
            sclk <= NOT sclk;
         end if;
      end if;
   end process;


   Moments_Processing : process(sclk)

   constant NStages : Integer := 12;
   constant MaxSampleLabel : Integer := SampleSize -1;
   constant FinalStageLabel : Integer := NStages -1;
   constant MaxCount : Integer := MaxSampleLabel + NStages;
   variable count : Integer range 0 to MaxCount := 0;
```

```
    begin
    --allow for proper state of Ring Oscillators to emerge by waiting for an unpark
signal
     --however now we can have a signal race between the PARK signal and the
actual deactivation of the Ring Oscillator
     --care should be taken so that bSample is set to True only on a properly un-
parked Ring and set to false NOT AFTER
    --the Rings are Parked


        --synchronous reset so that we can merge all the regiesters in the dsp block
        if (rising_edge(sclk)) then
            if(bset = '1') then

                --set all the flags and signals to the initial value
                bStage1Ready <= '0';
                bStage2Ready <= '0';
                bStage3Ready <= '0';
                bStage4Ready <= '0';
                bStage5Ready <= '0';
                bStage6Ready <= '0';
                bStage7Ready <= '0';
                bStage8Ready <= '0';
                bStage9Ready <= '0';
                bStage10Ready <= '0';
                bStage11Ready <= '0';
                bMomentsReady <= '0';
                count := 0;

                --flush registers
                   --warning: failure to flush a stage register will result in erroneous re-
sults
                --in reruns of the statistical engine
                D1 <= (others => '0');
                D2 <= (others => '0');
                D3 <= (others => '0');
                D4 <= (others => '0');
                D5 <= (others => '0');
                D6 <= (others => '0');
                D7 <= (others => '0');
                SQ1 <= (others => '0');
                SQ2 <= (others => '0');
                C1 <= (others => '0');
```

```
    C2 <= (others => '0');
    Q1 <= (others => '0');
    Q2 <= (others => '0');
    P1 <= (others => '0');
    P2 <= (others => '0');
    HX1 <= (others => '0');
    HX2 <= (others => '0');
    HP1 <= (others => '0');
    HP2 <= (others => '0');
    O1 <= (others => '0');
    O2 <= (others => '0');
    E1 <= (others => '0');
    E2 <= (others => '0');
    DEC1 <= (others => '0');
    regMOMENT1 <= (others => '0');
    regMOMENT2 <= (others => '0');
    regMOMENT3 <= (others => '0');
    regMOMENT4 <= (others => '0');
    regMOMENT5 <= (others => '0');
    regMOMENT6 <= (others => '0');
    regMOMENT7 <= (others => '0');
    regMOMENT8 <= (others => '0');
    regMOMENT9 <= (others => '0');
    regMOMENT10 <= (others => '0');

    onescount1 <= (others => '0');
    onescount2 <= (others => '0');
    onescount3 <= (others => '0');
    onescount4 <= (others => '0');
    onescount5 <= (others => '0');
    onescount6 <= (others => '0');
    onescount7 <= (others => '0');
    onescount8 <= (others => '0');
    onesPartsum1 <= (others => '0');
    onesPartsum2 <= (others => '0');
    onesPartsum3 <= (others => '0');
    onesPartsum4 <= (others => '0');
    onesBsum1 <= (others => '0');
    onesBsum2 <= (others => '0');
    CountOfOnes <= (others => '0');
    TotalCountOfOnes <= (others => '0');

end if;
```

```
if(bStart = '1' AND bSet = '0') then
--the ones count

    --stage 1
    if(count <= MaxSampleLabel) then
        bit1 <= Unsigned(feed(0 downto 0));
        bit2 <= Unsigned(feed(1 downto 1));
        bit3 <= Unsigned(feed(2 downto 2));
        bit4 <= Unsigned(feed(3 downto 3));
        bit5 <= Unsigned(feed(4 downto 4));
        bit6 <= Unsigned(feed(5 downto 5));
        bit7 <= Unsigned(feed(6 downto 6));
        bit8 <= Unsigned(feed(7 downto 7));
        bit9 <= Unsigned(feed(8 downto 8));
        bit10 <= Unsigned(feed(9 downto 9));
        bit11 <= Unsigned(feed(10 downto 10));
        bit12 <= Unsigned(feed(11 downto 11));
        bit13 <= Unsigned(feed(12 downto 12));
        bit14 <= Unsigned(feed(13 downto 13));
        bit15 <= Unsigned(feed(14 downto 14));
        bit16 <= Unsigned(feed(15 downto 15));
    end if;
    --stage 2
    if(count <= MaxSampleLabel + 1) then
        onescount1 <= resize(bit1,2) + resize(bit2,2);
        onescount2 <= resize(bit3,2) + resize(bit4,2);
        onescount3 <= resize(bit5,2) + resize(bit6,2);
        onescount4 <= resize(bit7,2) + resize(bit8,2);
        onescount5 <= resize(bit9,2) + resize(bit10,2);
        onescount6 <= resize(bit11,2) + resize(bit12,2);
        onescount7 <= resize(bit13,2) + resize(bit14,2);
        onescount8 <= resize(bit15,2) + resize(bit16,2);
    end if;
    --stage 3
    if(count <= MaxSampleLabel + 2) then
        onesPartsum1 <= resize(onescount1,5) + resize(onescount2,5);
        onesPartsum2 <= resize(onescount3,5) + resize(onescount4,5);
        onesPartsum3 <= resize(onescount5,5) + resize(onescount6,5);
        onesPartsum4 <= resize(onescount7,5) + resize(onescount8,5);
    end if;
    --stage 4
```

```
 if(count <= MaxSampleLabel + 3) then
   onesBsum1 <= resize(onesPartsum1,5) + resize(onesPartsum2,5);
   onesBsum2 <= resize(onesPartsum3,5) + resize(onesPartsum4,5);
   --stage 5
 end if;
 if(count <= MaxSampleLabel + 4) then
   CountOfOnes <= onesBsum1 + onesBsum2;
 end if;
 if(count <= MaxSampleLabel + 5) then
   TotalCountOfOnes <= TotalCountOfOnes + CountOfOnes;
 end if;

--TICK STAGE 1 (SAMPLE INPUT)
if(count <= MaxSampleLabel) then
D1 <= UNSIGNED(feed);
end if;
if (count = MaxSampleLabel) then
bStage1Ready <= '1';
end if;

--TICK STAGE 2
if(count <= MaxSampleLabel + 1) then
--MOMENT1
regMOMENT1 <= regMOMENT1 + D1;

D2 <= D1;
end if;
if (count = MaxSampleLabel + 1) then
bStage2Ready <= '1';
end if;

--TICK STAGE 3
if(count <= MaxSampleLabel + 2) then
SQ1 <= D2*D2;--MAIN
SQ2 <= D2*D2;--CARRY
D3 <= D2;
end if;
if (count = MaxSampleLabel + 2) then
bStage3Ready <= '1';
end if;

--TICK STAGE 4
if(count <= MaxSampleLabel + 3) then
```

```
--MOMENT2
regMOMENT2 <= regMOMENT2 + SQ1;

D4 <= D3;
C1 <= SQ2*D3;--MAIN
C2 <= SQ2*D3;--CARRY
end if;
if (count = MaxSampleLabel + 3) then
bStage4Ready <= '1';
end if;

--TICK STAGE 5
if(count <= MaxSampleLabel + 4) then

--MOMENT3
regMOMENT3 <= regMOMENT3 + C1;

D5 <= D4;
Q1 <= C2*D4;--MAIN
Q2 <= C2*D4;--CARRY
end if;
if (count = MaxSampleLabel + 4) then
bStage5Ready <= '1';
end if;

--TICK STAGE 6
if(count <= MaxSampleLabel + 5) then

--MOMENT4
regMOMENT4 <= regMOMENT4 + Q1;

D6 <= D5;
P1 <= Q2*D5;
P2 <= Q2*D5;
end if;
if (count = MaxSampleLabel + 5) then
bStage6Ready <= '1';
end if;

--TICK STAGE 7
if(count <= MaxSampleLabel + 6) then

--MOMENT5
```

```
regMOMENT5 <= regMOMENT5 + P1;

D7 <= D6;
HX1 <= P2*D6;--MAIN
HX2 <= P2*D6;--CARRY
end if;
if (count = MaxSampleLabel + 6) then
bStage7Ready <= '1';
end if;

--TICK STAGE 8
if(count <= MaxSampleLabel + 7) then

--MOMENT6
regMOMENT6 <= regMOMENT6 + HX1;

D8 <= D7;
HP1 <= HX2*D7;--MAIN
HP2 <= HX2*D7;--CARRY
end if;
if (count = MaxSampleLabel + 7) then
bStage8Ready <= '1';
end if;

--TICK STAGE 9
if(count <= MaxSampleLabel + 8) then

--MOMENT7
regMOMENT7 <= regMOMENT7 + HP1;

D9 <= D8;
O1 <= HP2*D8;--MAIN
O2 <= HP2*D8;--CARRY
end if;
if (count = MaxSampleLabel + 8) then
bStage9Ready <= '1';
end if;

--TICK STAGE 10
if(count <= MaxSampleLabel + 9) then

--MOMENT8
regMOMENT8 <= regMOMENT8 + O1;
```

```
            D10 <= D9;
            E1 <= O2*D9;--MAIN
            E2 <= O2*D9;--CARRY
            end if;
            if (count = MaxSampleLabel + 9) then
            bStage10Ready <= '1';
            end if;

            --TICK STAGE 11
            if (count <= MaxSampleLabel + 10) then

            --MOMENT9
            regMOMENT9 <= regMOMENT9 + E1;

            DEC1 <= E2*D10;--MAIN (NO CARRY NEEDED)
            end if;
            if (count = MaxSampleLabel + 10) then
            bStage11Ready <= '1';
            end if;

            --TICK STAGE 12
            if (count <= MaxSampleLabel + 11) then

            --PREG DEC10
            regMOMENT10 <= regMOMENT10 + DEC1;
            end if;
            if (count = MaxSampleLabel + 11) then
            bMomentsReady <= '1';
             --Implement some logic to take the upper QWidth*MomentNumber bits
as the Integer part and
              --the lower Acc bits as the fractional part, or simply divide the number
after VIO by 2^Acc
            end if;

            if (count < MaxCount) then
            count := count + 1;
            end if;

         end if;

      end if;
```

```
    end process Moments_Processing;

    --todo: configure VIO probes to get only th upper WMoment - Acc bits to facili-
tate division by 2^Acc
    VIO : vio_0
    PORT MAP (
    clk => clk_IO,
    probe_in0 => std_logic_vector(regMOMENT1),
    probe_in1 => std_logic_vector(regMOMENT2),
    probe_in2 => std_logic_vector(regMOMENT3),
    probe_in3 => std_logic_vector(regMOMENT4),
    probe_in4 => std_logic_vector(regMOMENT5),
    probe_in5 => std_logic_vector(regMOMENT6),
    probe_in6 => std_logic_vector(regMOMENT7),
    probe_in7 => std_logic_vector(regMOMENT8),
    probe_in8 => std_logic_vector(regMOMENT9),
    probe_in9 => std_logic_vector(regMOMENT10),
    probe_in10(0) => bStage1Ready,
    probe_in11(0) => bStage2Ready,
    probe_in12(0) => bStage3Ready,
    probe_in13(0) => bStage4Ready,
    probe_in14(0) => bStage5Ready,
    probe_in15(0) => bStage6Ready,
    probe_in16(0) => bStage7Ready,
    probe_in17(0) => bStage8Ready,
    probe_in18(0) => bStage9Ready,
    probe_in19(0) => bStage10Ready,
    probe_in20(0) => bStage11Ready,
    probe_in21(0) => bMomentsReady,
    probe_in22 => std_logic_vector(CountOfOnes),
    probe_in23 => std_logic_vector(TotalCountOfOnes),
    probe_out0(0) => bSet,
    probe_out1(0) => bStart,
    probe_out2(0) => PARK
    );

end Behavioral;
```

C.4 Statistical module for Ones Distance Distribution and Bias

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Stats is
    Generic(QWidth, Acc, SampleSize, WMoment1,
            WMoment2, WMoment3, WMoment4, WMoment5,
            WMoment6, WMoment7, WMoment8,
            WMoment9, WMoment10 : Integer);
    Port (clk : in std_logic;
          clk_IO : in std_logic;
          SKIP : in std_logic_vector(15 downto 0);
          PARK : out std_logic;
          feed : in std_logic_vector(QWidth -1 downto 0)
         );
end Stats;

architecture Behavioral of Stats is

--regional divided clock
--we do this instead of passing a global divided clock through the switching fabric
--as it would add tremendous skew and latency to be used as a common clock
between
--the statistical engine and the synchronizer
signal sclk : std_logic := '0';

signal bStart : std_logic := '0';
signal bSet : std_logic := '0';
signal reserved : std_logic := '0';
```

```
signal bReady : std_logic := '0';

signal onescount1 : Unsigned(1 downto 0) := (others => '0');
signal onescount2 : Unsigned(1 downto 0) := (others => '0');
signal onescount3 : Unsigned(1 downto 0) := (others => '0');
signal onescount4 : Unsigned(1 downto 0) := (others => '0');
signal onescount5 : Unsigned(1 downto 0) := (others => '0');
signal onescount6 : Unsigned(1 downto 0) := (others => '0');
signal onescount7 : Unsigned(1 downto 0) := (others => '0');
signal onescount8 : Unsigned(1 downto 0) := (others => '0');
signal onesPartsum1H, onesPartsum1MH, onesPartsum1ML, onesPartsum1L :
Unsigned(4 downto 0) := (others => '0');
signal onesBsum, onesCarrysum : Unsigned(4 downto 0) := (others => '0');
signal bit1, bit2, bit3, bit4, bit5, bit6, bit7, bit8, bit9, bit10, bit11, bit12, bit13, bit14,
bit15, bit16 : Unsigned(0 downto 0);
signal CountOfOnesH, CountOfOnesMH, CountOfOnesML, CountOfOnesL : Un-
signed (4 downto 0) := (others => '0');
signal TotalCountOfOnesH, TotalCountOfOnesMH, TotalCountOfOnesML, Total-
CountOfOnesL : Unsigned (Acc + 4 downto 0) := (others => '0');

signal OnesDistance : Unsigned(15 downto 0) := (others => '0');

type ram1024 is array(0 to 31) of Unsigned(Acc -1 downto 0);
type bitstoregroup is array(0 to 15) of Unsigned(2**Acc -1 downto 0);

signal bitstore, pstore : Unsigned(15 downto 0) := (others => '0');
signal distance0, distance1, distance2, distance3, distance4, distance5, dis-
tance6, distance7,
        distance8, distance9, distance10, distance11, distance12, distance13, dis-
tance14, distance15 : ram1024 := (others => (others => '0'));
signal RamElements : ram1024 := (others => (others => '0'));
signal SelectBit : std_logic_vector(3 downto 0) := (others => '0');
signal trackbit : Unsigned(0 downto 0);




COMPONENT vio_0
  PORT (
    clk : IN STD_LOGIC;
    probe_in0 : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_in1 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in2 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
```

```
    probe_in3 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in4 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in5 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in6 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in7 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in8 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in9 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in10 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in11 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in12 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in13 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in14 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in15 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in16 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in17 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in18 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in19 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in20 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in21 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in22 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in23 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in24 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in25 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in26 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in27 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in28 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in29 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in30 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in31 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_in32 : IN STD_LOGIC_VECTOR(19 DOWNTO 0);
    probe_out0 : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_out1 : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_out2 : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
    probe_out3 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
  );
END COMPONENT;

    begin


    CLOCK_DIVIDER : process(clk)
    variable i : Integer range 0 to 2**16 -1 := 0;
    begin
```

```
    if (rising_edge(clk)) then
      i := i + 1;
      if (i = to_integer(UNSIGNED(SKIP)) -1) then
        i := 0;
        sclk <= NOT sclk;
      end if;
    end if;
  end process;

  Moments_Processing : process(sclk)

  constant NStages : Integer := 12;
  constant MaxSampleLabel : Integer := SampleSize -1;
  constant FinalStageLabel : Integer := NStages -1;
  constant MaxCount : Integer := MaxSampleLabel + NStages;
  variable count : Integer range 0 to MaxCount := 0;
   variable ind0, ind1, ind2, ind3, ind4, ind5, ind6, ind7, ind8, ind9, ind10, ind11,
ind12, ind13, ind14, ind15 : Integer range 0 to 31 := 0;

  begin
   --allow for proper state of Ring Oscillators to emerge by waiting for an unpark
signal
    --however now we can have a signal race between the PARK signal and the
actual deactivation of the Ring Oscillator
    --care should be taken so that bSample is set to True only on a properly un-
parked Ring and set to false NOT AFTER
  --the Rings are Parked

    --synchronous reset so that we can merge all the regiesters in the dsp block
    if (rising_edge(sclk)) then
      if(bset = '1') then

        count := 0;

        --flush registers
          --warning: failure to flush a stage register will result in erroneous re-
sults
        --in reruns of the statistical engine

        bReady <= '0';
        distance0 <= (others =>(others => '0'));
        distance1 <= (others =>(others => '0'));
        distance2 <= (others =>(others => '0'));
```

```
			distance3 <= (others =>(others => '0'));
			distance4 <= (others =>(others => '0'));
			distance5 <= (others =>(others => '0'));
			distance6 <= (others =>(others => '0'));
			distance7 <= (others =>(others => '0'));
			distance8 <= (others =>(others => '0'));
			distance9 <= (others =>(others => '0'));
			distance10 <= (others =>(others => '0'));
			distance11 <= (others =>(others => '0'));
			distance12 <= (others =>(others => '0'));
			distance13 <= (others =>(others => '0'));
			distance14 <= (others =>(others => '0'));
			distance15 <= (others =>(others => '0'));
			OnesDistance <= (others => '0');
			bitstore <= (others => '0');
			pstore <= (others => '0');
			RamElements <= (others => (others => '0'));

		end if;


		if(bStart = '1' AND bSet = '0') then
		--the ones count

			--stage 1
			if(count <= MaxSampleLabel) then
				bitstore <= Unsigned(feed);
			end if;
			 --stage 2
			if((count <= MaxSampleLabel + 1) AND (count > 0)) then
				pstore <= bitstore;
			end if;

			 --stage 3
			if((count <= MaxSampleLabel + 2) AND (count > 1)) then
					distance0(ind0) <= distance0(ind0) + resize(Unsigned(pstore(0 to
0)),20);
					distance1(ind1) <= distance1(ind1) + resize(Unsigned(pstore(1 to
1)),20);
					distance2(ind2) <= distance2(ind2) + resize(Unsigned(pstore(2 to
2)),20);
					distance3(ind3) <= distance3(ind3) + resize(Unsigned(pstore(3 to
3)),20);
```

```
                distance4(ind4) <= distance4(ind4) + resize(Unsigned(pstore(4 to
4)),20);
                distance5(ind5) <= distance5(ind5) + resize(Unsigned(pstore(5 to
5)),20);
                distance6(ind6) <= distance6(ind6) + resize(Unsigned(pstore(6 to
6)),20);
                distance7(ind7) <= distance7(ind7) + resize(Unsigned(pstore(7 to
7)),20);
                distance8(ind8) <= distance8(ind8) + resize(Unsigned(pstore(8 to
8)),20);
                distance9(ind9) <= distance9(ind9) + resize(Unsigned(pstore(9 to
9)),20);
                distance10(ind10) <= distance10(ind10) + resize(Un-
signed(pstore(10 to 10)),20);
                distance11(ind11) <= distance11(ind11) + resize(Unsigned(pstore(11
to 11)),20);
                distance12(ind12) <= distance12(ind12) + resize(Un-
signed(pstore(12 to 12)),20);
                distance13(ind13) <= distance13(ind13) + resize(Un-
signed(pstore(13 to 13)),20);
                distance14(ind14) <= distance14(ind14) + resize(Un-
signed(pstore(14 to 14)),20);
                distance15(ind15) <= distance15(ind15) + resize(Un-
signed(pstore(15 to 15)),20);
        if((ind0 < 31) AND pstore(0) = '0') then
           ind0 := ind0 + 1;
        end if;
        if(pstore(0) = '1') then
           ind0 := 0;
        end if;
        if((ind1 < 31) AND pstore(1) = '0') then
           ind1 := ind1 + 1;
        end if;
        if(pstore(1) = '1') then
           ind1 := 0;
        end if;
        if((ind2 < 31) AND pstore(2) = '0') then
           ind2 := ind2 + 1;
        end if;
        if(pstore(2) = '1') then
           ind2 := 0;
        end if;
        if((ind3 < 31) AND pstore(3) = '0') then
```

```
   ind3 := ind3 + 1;
end if;
if(pstore(3) = '1') then
   ind3 := 0;
end if;
if((ind4 < 31) AND pstore(4) = '0') then
   ind4 := ind4 + 1;
end if;
if(pstore(4) = '1') then
   ind4 := 0;
end if;
if((ind5 < 31) AND pstore(5) = '0') then
   ind5 := ind5 + 1;
end if;
if(pstore(5) = '1') then
   ind5 := 0;
end if;
if((ind6 < 31) AND pstore(6) = '0') then
   ind6 := ind6 + 1;
end if;
if(pstore(6) = '1') then
   ind6 := 0;
end if;
if((ind7 < 31) AND pstore(7) = '0') then
   ind7 := ind7 + 1;
end if;
if(pstore(7) = '1') then
   ind7 := 0;
end if;
if((ind8 < 31) AND pstore(8) = '0') then
   ind8 := ind8 + 1;
end if;
if(pstore(8) = '1') then
   ind8 := 0;
end if;
if((ind9 < 31) AND pstore(9) = '0') then
   ind9 := ind9 + 1;
end if;
if(pstore(9) = '1') then
   ind9 := 0;
end if;
if((ind10 < 31) AND pstore(10) = '0') then
   ind10 := ind10 + 1;
```

```
          end if;
          if(pstore(10) = '1') then
             ind10 := 0;
          end if;
          if((ind11 < 31) AND pstore(11) = '0') then
             ind11 := ind11 + 1;
          end if;
          if(pstore(11) = '1') then
             ind11 := 0;
          end if;
          if((ind12 < 31) AND pstore(12) = '0') then
             ind12 := ind12 + 1;
          end if;
          if(pstore(12) = '1') then
             ind12 := 0;
          end if;
          if((ind13 < 31) AND pstore(13) = '0') then
             ind13 := ind13 + 1;
          end if;
          if(pstore(13) = '1') then
             ind13 := 0;
          end if;
          if((ind14 < 31) AND pstore(14) = '0') then
             ind14 := ind14 + 1;
          end if;
          if(pstore(14) = '1') then
             ind14 := 0;
          end if;
          if((ind15 < 31) AND pstore(15) = '0') then
             ind15 := ind15 + 1;
          end if;
          if(pstore(15) = '1') then
             ind15 := 0;
          end if;

       end if;

       if (count = MaxSampleLabel + 11) then
          bReady <= '1';
--           --Implement some logic to take the upper QWidth*MomentNumber bits as the Integer part and
--         --the lower Acc bits as the fractional part, or simply divide the number after VIO by 2^Acc
```

```
--              --also commit the first 32 elements of the distribution
--              --a bit late but that's not problematic

--                  RamElements(j) <= mega(j);

            end if;

            if (count < MaxCount) then
            count := count + 1;
            end if;

            if(bReady = '1') then
               case SelectBit is
                  when "0000" => RamElements <= distance0;
                  when "0001" => RamElements <= distance1;
                  when "0010" => RamElements <= distance2;
                  when "0011" => RamElements <= distance3;
                  when "0100" => RamElements <= distance4;
                  when "0101" => RamElements <= distance5;
                  when "0110" => RamElements <= distance6;
                  when "0111" => RamElements <= distance7;
                  when "1000" => RamElements <= distance8;
                  when "1001" => RamElements <= distance9;
                  when "1010" => RamElements <= distance10;
                  when "1011" => RamElements <= distance11;
                  when "1100" => RamElements <= distance12;
                  when "1101" => RamElements <= distance13;
                  when "1110" => RamElements <= distance14;
                  when "1111" => RamElements <= distance15;
               end case;
            end if;


         end if;

      end if;


   end process Moments_Processing;

    --todo: configure VIO probes to get only th upper WMoment - Acc bits to facili-
tate division by 2^Acc
```

```
    VIO : vio_0
    PORT MAP (
    clk => clk_IO,
    probe_in0(0) => bReady,
    probe_in1 => std_logic_vector(RamElements(0)),
    probe_in2 => std_logic_vector(RamElements(1)),
    probe_in3 => std_logic_vector(RamElements(2)),
    probe_in4 => std_logic_vector(RamElements(3)),
    probe_in5 => std_logic_vector(RamElements(4)),
    probe_in6 => std_logic_vector(RamElements(5)),
    probe_in7 => std_logic_vector(RamElements(6)),
    probe_in8 => std_logic_vector(RamElements(7)),
    probe_in9 => std_logic_vector(RamElements(8)),
    probe_in10 => std_logic_vector(RamElements(9)),
    probe_in11 => std_logic_vector(RamElements(10)),
    probe_in12 => std_logic_vector(RamElements(11)),
    probe_in13 => std_logic_vector(RamElements(12)),
    probe_in14 => std_logic_vector(RamElements(13)),
    probe_in15 => std_logic_vector(RamElements(14)),
    probe_in16 => std_logic_vector(RamElements(15)),
    probe_in17 => std_logic_vector(RamElements(16)),
    probe_in18 => std_logic_vector(RamElements(17)),
    probe_in19 => std_logic_vector(RamElements(18)),
    probe_in20 => std_logic_vector(RamElements(19)),
    probe_in21 => std_logic_vector(RamElements(20)),
    probe_in22 => std_logic_vector(RamElements(21)),
    probe_in23 => std_logic_vector(RamElements(22)),
    probe_in24 => std_logic_vector(RamElements(23)),
    probe_in25 => std_logic_vector(RamElements(24)),
    probe_in26 => std_logic_vector(RamElements(25)),
    probe_in27 => std_logic_vector(RamElements(26)),
    probe_in28 => std_logic_vector(RamElements(27)),
    probe_in29 => std_logic_vector(RamElements(28)),
    probe_in30 => std_logic_vector(RamElements(29)),
    probe_in31 => std_logic_vector(RamElements(30)),
    probe_in32 => std_logic_vector(RamElements(31)),
    probe_out0(0) => bSet,
    probe_out1(0) => bStart,
    probe_out2(0) => PARK,
    probe_out3 => SelectBit
    );

end Behavioral;
```

C.5 XDC Constraints File

```
set_property PACKAGE_PIN AL8 [get_ports si570_clk_p]
set_property IOSTANDARD DIFF_SSTL12 [get_ports si570_clk_p]

set_clock_groups -asynchronous -group clk_300_clk_wiz_0 -group clk_120_clk_wiz_0
create_generated_clock -name deciclock -source [get_ports si570_clk_p] -divide_by 60 [get_pins Stats_Instance/sclk_reg/Q]

set_false_path -from [get_clocks -of_objects [get_pins PLL_300_120/inst/plle4_adv_inst/CLKOUT1]] -to [get_clocks deciclock]
set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
connect_debug_port dbg_hub/clk [get_nets clk_120]
```

Appendix D. H3 Post Processor Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Stats is
   Generic(QWidth, Acc, SampleSize, WMoment1,
          WMoment2, WMoment3, WMoment4, WMoment5,
          WMoment6, WMoment7, WMoment8,
          WMoment9, WMoment10 : Integer);
   Port (clk : in std_logic;
        clk_IO : in std_logic;
        SKIP : in std_logic_vector(15 downto 0);
        PARK : out std_logic;
        feed : in std_logic_vector(QWidth -1 downto 0)
        );
end Stats;

architecture Behavioral of Stats is

--regional divided sampling clock
--we do this instead of passing a global divided clock through the switching fabric
--as it would add tremendous skew and latency to be used as a common clock
between
--the statistical engine and the synchronizer
signal sclk : std_logic := '0';

--regional PSPX harmonized clock
signal pclk : std_logic := '0';
--flag to denote post-processed output register is live with first data
signal PsPxOverheadDealt : std_logic := '0';

--post processor input registers
```

```
signal uBit0, uBit1, uBit2, uBit3, uBit4, uBit5, uBit6, uBit7, uBit8, uBit9,
uBit10, uBit11, uBit12, uBit13, uBit14, uBit15 : std_logic_vector(15 downto 0) :=
(others => '0');
--post processor output registers
signal pBit0, pBit1, pBit2, pBit3, pBit4, pBit5, pBit6, pBit7, pBit8, pBit9,
pBit10, pBit11, pBit12, pBit13, pBit14, pBit15 : std_logic_vector(7 downto 0);

--actual post processed feed
signal PFeed : std_logic_vector(15 downto 0) := (others => '0');

signal bStart : std_logic := '0';
signal bSet : std_logic := '0';
signal reserved : std_logic := '0';

signal bReady : std_logic := '0';
type ram1024 is array(0 to 31) of Unsigned(Acc -1 downto 0);
type bitstoregroup is array(0 to 15) of Unsigned(2**Acc -1 downto 0);

signal bitstore, pstore : Unsigned(15 downto 0) := (others => '0');
signal distance0, distance1, distance2, distance3, distance4, distance5, dis-
tance6, distance7,
        distance8, distance9, distance10, distance11, distance12, distance13, dis-
tance14, distance15 : ram1024 := (others => (others => '0'));
signal RamElements : ram1024 := (others => (others => '0'));
signal SelectBit : std_logic_vector(3 downto 0) := (others => '0');
signal trackbit : Unsigned(0 downto 0);


   begin


   CLOCK_DIVIDER : process(clk)
   variable i : Integer range 0 to 2**16 -1 := 0;

   begin
      if (rising_edge(clk)) then
         i := i + 1;
         if (i = to_integer(UNSIGNED(SKIP)) -1) then
            i := 0;
            sclk <= NOT sclk;
         end if;
      end if;
   end process;
```

```
   Moments_Processing : process(sclk)

   constant NStages : Integer := 12;
   constant MaxSampleLabel : Integer := SampleSize -1;
   constant FinalStageLabel : Integer := NStages -1;
   constant MaxCount : Integer := MaxSampleLabel + NStages;
   variable count : Integer range 0 to MaxCount := 0;
   variable uIndex : Integer range 0 to 15 := 0;
   variable pIndex : Integer range 0 to 7 := 0;
    variable ind0, ind1, ind2, ind3, ind4, ind5, ind6, ind7, ind8, ind9, ind10, ind11,
ind12, ind13, ind14, ind15 : Integer range 0 to 31 := 0;

   begin
    --allow for proper state of Ring Oscillators to emerge by waiting for an unpark
signal
     --however now we can have a signal race between the PARK signal and the
actual deactivation of the Ring Oscillator
     --care should be taken so that bSample is set to True only on a properly un-
parked Ring and set to false NOT AFTER
   --the Rings are Parked



      --synchronous reset so that we can merge all the regiesters in the dsp block
      if (rising_edge(sclk)) then
         if(bset = '1') then
         --reset general and PostProcessor Counts
            count := 0;
            uIndex := 0;
            pIndex := 0;
            PsPxOverheadDealt <= '0';

            --flush registers
               --warning: failure to flush a stage register will result in erroneous re-
sults
            --in reruns of the statistical engine

            bReady <= '0';
            distance0 <= (others =>(others => '0'));
```

```
    distance1 <= (others =>(others => '0'));
    distance2 <= (others =>(others => '0'));
    distance3 <= (others =>(others => '0'));
    distance4 <= (others =>(others => '0'));
    distance5 <= (others =>(others => '0'));
    distance6 <= (others =>(others => '0'));
    distance7 <= (others =>(others => '0'));
    distance8 <= (others =>(others => '0'));
    distance9 <= (others =>(others => '0'));
    distance10 <= (others =>(others => '0'));
    distance11 <= (others =>(others => '0'));
    distance12 <= (others =>(others => '0'));
    distance13 <= (others =>(others => '0'));
    distance14 <= (others =>(others => '0'));
    distance15 <= (others =>(others => '0'));
    OnesDistance <= (others => '0');
    bitstore <= (others => '0');
    PFeed <= (Others => '0');
    pstore <= (others => '0');
    RamElements <= (others => (others => '0'));

end if;

if(bStart = '1' AND bSet = '0') then
--PSPX

    --feed input registers
    uBit0(uIndex) <= feed(0);
    uBit1(uIndex) <= feed(1);
    uBit2(uIndex) <= feed(2);
    uBit3(uIndex) <= feed(3);
    uBit4(uIndex) <= feed(4);
    uBit5(uIndex) <= feed(5);
    uBit6(uIndex) <= feed(6);
    uBit7(uIndex) <= feed(7);
    uBit8(uIndex) <= feed(8);
    uBit9(uIndex) <= feed(9);
    uBit10(uIndex) <= feed(10);
    uBit11(uIndex) <= feed(11);
    uBit12(uIndex) <= feed(12);
    uBit13(uIndex) <= feed(13);
    uBit14(uIndex) <= feed(14);
    uBit15(uIndex) <= feed(15);
```

```
if (uIndex < 15) then
uIndex := uIndex + 1;
end if;

if (uIndex = 15) then


        pBit0 <= uBit0(15 downto 8) XOR (uBit0(7 downto 0) XOR (uBit0(6 downto 0)&uBit0(7))) XOR ((uBit0(5 downto 0)&uBit0(7 downto 6))XOR (uBit0(3 downto 0)&uBit0(7 downto 4)));
        pBit1 <= uBit1(15 downto 8) XOR (uBit1(7 downto 0) XOR (uBit1(6 downto 0)&uBit1(7))) XOR ((uBit1(5 downto 0)&uBit1(7 downto 6))XOR (uBit1(3 downto 0)&uBit1(7 downto 4)));
        pBit2 <= uBit2(15 downto 8) XOR (uBit2(7 downto 0) XOR (uBit2(6 downto 0)&uBit2(7))) XOR ((uBit2(5 downto 0)&uBit2(7 downto 6))XOR (uBit2(3 downto 0)&uBit2(7 downto 4)));
        pBit3 <= uBit3(15 downto 8) XOR (uBit3(7 downto 0) XOR (uBit3(6 downto 0)&uBit3(7))) XOR ((uBit3(5 downto 0)&uBit3(7 downto 6))XOR (uBit3(3 downto 0)&uBit3(7 downto 4)));
        pBit4 <= uBit4(15 downto 8) XOR (uBit4(7 downto 0) XOR (uBit4(6 downto 0)&uBit4(7))) XOR ((uBit4(5 downto 0)&uBit4(7 downto 6))XOR (uBit4(3 downto 0)&uBit4(7 downto 4)));
        pBit5 <= uBit5(15 downto 8) XOR (uBit5(7 downto 0) XOR (uBit5(6 downto 0)&uBit5(7))) XOR ((uBit5(5 downto 0)&uBit5(7 downto 6))XOR (uBit5(3 downto 0)&uBit5(7 downto 4)));
        pBit6 <= uBit6(15 downto 8) XOR (uBit6(7 downto 0) XOR (uBit6(6 downto 0)&uBit6(7))) XOR ((uBit6(5 downto 0)&uBit6(7 downto 6))XOR (uBit6(3 downto 0)&uBit6(7 downto 4)));
        pBit7 <= uBit7(15 downto 8) XOR (uBit7(7 downto 0) XOR (uBit7(6 downto 0)&uBit7(7))) XOR ((uBit7(5 downto 0)&uBit7(7 downto 6))XOR (uBit7(3 downto 0)&uBit7(7 downto 4)));
        pBit8 <= uBit8(15 downto 8) XOR (uBit8(7 downto 0) XOR (uBit8(6 downto 0)&uBit8(7))) XOR ((uBit8(5 downto 0)&uBit8(7 downto 6))XOR (uBit8(3 downto 0)&uBit8(7 downto 4)));
        pBit9 <= uBit9(15 downto 8) XOR (uBit9(7 downto 0) XOR (uBit9(6 downto 0)&uBit9(7))) XOR ((uBit9(5 downto 0)&uBit9(7 downto 6))XOR (uBit9(3 downto 0)&uBit9(7 downto 4)));
          pBit10 <= uBit10(15 downto 8) XOR (uBit10(7 downto 0) XOR (uBit10(6 downto 0)&uBit10(7))) XOR ((uBit10(5 downto 0)&uBit10(7 downto 6))XOR (uBit10(3 downto 0)&uBit10(7 downto 4)));
pBit11 <= uBit11(15 downto 8) XOR (uBit11(7 downto 0) XOR (uBit11(6 downto 0)&uBit11(7))) XOR ((uBit11(5 downto 0)&uBit11(7 downto 6))XOR
```

```
(uBit11(3 downto 0)&uBit11(7 downto 4)));
                        pBit12 <= uBit12(15 downto 8) XOR (uBit12(7 downto 0) XOR
(uBit12(6 downto 0)&uBit12(7))) XOR ((uBit12(5 downto 0)&uBit12(7 downto
6))XOR (uBit12(3 downto 0)&uBit12(7 downto 4)));
                        pBit13 <= uBit13(15 downto 8) XOR (uBit13(7 downto 0) XOR
(uBit13(6 downto 0)&uBit13(7))) XOR ((uBit13(5 downto 0)&uBit13(7 downto
6))XOR (uBit13(3 downto 0)&uBit13(7 downto 4)));
                        pBit14 <= uBit14(15 downto 8) XOR (uBit14(7 downto 0) XOR
(uBit14(6 downto 0)&uBit14(7))) XOR ((uBit14(5 downto 0)&uBit14(7 downto
6))XOR (uBit14(3 downto 0)&uBit14(7 downto 4)));
                        pBit15 <= uBit15(15 downto 8) XOR (uBit15(7 downto 0) XOR
(uBit15(6 downto 0)&uBit15(7))) XOR ((uBit15(5 downto 0)&uBit15(7 downto
6))XOR (uBit15(3 downto 0)&uBit15(7 downto 4)));

                PsPxOverheadDealt <= '1';
                uIndex := 0;

                end if;

            end if;

             if(bStart = '1' AND bSet = '0' AND PsPxOverheadDealt = '1' AND (uIndex
mod 2 = 0)) then

                PFeed(0) <= pBit0(pIndex);
                PFeed(1) <= pBit1(pIndex);
                PFeed(2) <= pBit2(pIndex);
                PFeed(3) <= pBit3(pIndex);
                PFeed(4) <= pBit4(pIndex);
                PFeed(5) <= pBit5(pIndex);
                PFeed(6) <= pBit6(pIndex);
                PFeed(7) <= pBit7(pIndex);
                PFeed(8) <= pBit8(pIndex);
                PFeed(9) <= pBit9(pIndex);
                PFeed(10) <= pBit10(pIndex);
                PFeed(11) <= pBit11(pIndex);
                PFeed(12) <= pBit12(pIndex);
                PFeed(13) <= pBit13(pIndex);
                PFeed(14) <= pBit14(pIndex);
                PFeed(15) <= pBit15(pIndex);

                if(pIndex < 7) then
                pIndex := pIndex + 1;
```

```
        end if;
        if(pIndex = 7) then
        pIndex := 0;
        end if;


    --the ones count

        --stage 1
        if(count <= MaxSampleLabel) then
            bitstore <= Unsigned(PFeed);

--rest is the same as all others Stats instances--
```