ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάπτυξη Πλατφόρφας Ελέγχου Ολοκληρωμένων Κυκλωμάτων Υψηλών Ταχυτήτων σε FPGA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΠΑΝΤΕΛΗ Μ.ΣΑΡΑΗ

Επιβλέπων : Ιωάννης Παπανάνος
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάπτυξη Πλατφόρφας Ελέγχου Ολοκληρωμένων Κυκλωμάτων Υψηλών Ταχυτήτων σε FPGA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΠΑΝΤΕΛΗ Μ.ΣΑΡΑΗ

Επιβλέπων : Ιωάννης Παπανάνος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Οκτωβρίου 2015.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Ιωάννης Παπανάνος	Κιαμάλ Πεκμετζή	Γεώργιος Στασινόπουλος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2015

(Υπογραφή)

.....

ΠΑΝΤΕΛΗΣ Μ.ΣΑΡΑΗΣ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στην σύγχρονη εποχή η σημασία των ασύρματων τηλεπικοινωνιών αυξάνεται διαρκώς. Καθώς ο αριθμός των χρηστών μεγαλώνει, οι χρήστες μεταδίδουν αυξανόμενο όγκο δεδομένων κάθε χρόνο. Οι σημερινές ασύρματες συσκευές, όπως τα smart phones και τα tablet computers υποστηρίζουν πολλαπλά τηλεπικοινωνιακά πρότυπα τα οποία απαιτούν επίσης αυξανόμενο ρυθμό μετάδοσης. Η παραγωγή σημάτων ελέγχου για τις συσκευές αυτές γίνεται συνεχώς μεγαλύτερη πρόκληση αφού πρώτον αυξάνεται το πλήθος των σημάτων ελέγχου και δεύτερον πρέπει να μεταδίδονται σε υψηλότερες συχνότητες. Επιπλέον στις παράλληλες διεπαφές υπάρχει η απαίτηση τα σήματα ελέγχου να είναι συγχρονισμένα. Τα FPGA αποτελούν μια ιδανική επιλογή για την παραγωγή ψηφιακών σημάτων υψηλών ταχυτήτων καθώς παρέχουν κλιμακοσιμότητα και ευελιξία στην υλοποίηση του συστήματος.

Ο σκοπός της διπλωματικής εργασίας ήταν η ανάπτυξη μια πλατφόρμας ελέγχου ολοκληρωμένων κυκλωμάτων υψηλών ταχυτήτων. Για τον έλεγχο των ολοκληρωμένων κυκλωμάτων απαιτείται η παράγωγη παράλληλων διανυσμάτων υψηλής ταχύτητας καθώς και ανάγνωση και αποθήκευση ειδικών σημάτων ελέγχου που παρέχουν τα ολοκληρωμένα κυκλώματα.

Η πλατφόρμα έχει υλοποιηθεί στο υψηλών επιδόσεων Virtex-7 FPGA αναπτυξιακό χαρακτηρισμού VC7215 της Xilinx. Η πλατφόρμα δέχεται εντολές και δεδομένα μέσω MATLAB. Τα δεδομένα αποθηκεύονται σε πολλαπλά μπλοκ μνήμης και αναπαράγονται μέσω σειριακών πομπών που διαθέτει το αναπτυξιακό. Η πλατφόρμα θα χρησιμοποιηθεί για την επαλήθευση ψηφιακού RF διαμορφωτή ASIC παρέχοντας του δεδομένα με ρυθμό 2.65GHz.

Η επιλογή του συγκεκριμένου αναπτυξιακού κάνει δυνατή την κλιμάκωση της πλατφόρμας ελέγχου με μέγιστο αριθμό καναλιών 80 και μέγιστο ρυθμό μετάδοσης 13.1Gbps.

Λέξεις Κλειδιά: FPGA, Xilinx, Multi-Gigabit Transceiver, High-Speed Serial I/O, PWM modulator, digital testing, alignment, Verilog, Hardware Description Language, VLSI, Synthesis, Place and Route.

Abstract

The importance of wireless communication technology is growing constantly. While the number of users is increasing, users also transmit larger amounts of data as applications becoming more data hungry. Modern wireless devices need to operate in higher frequencies and have larger parallel I/O interfaces in order to cope with demands of growing bandwidth consumption. As a result, the generation of test signals for these devices is an on-going challenge. FPGAs provide and ideal choice for generating high-speed, phase-aligned digital signals and they offer scalability, controllability and flexibility of implementation.

The scope of this thesis was the development of a high-speed FPGA based testing platform. This work is focused on generating high-speed digital signals as well as retrieving feedback from the device under test.

The testing platform was implemented on the Virtex-7 FPGA VC7215 Characterization Kit. The user of the platform can send instructions and data from a host-pc using MATLAB. The data are stored in multiple memory blocks and then transmitted through multi-gigabit transceivers phase-aligned. There is also the requirement of fine-tuning the phase of each signal individually and calibrating the voltage level. The system is used for testing and verification of prototype PWM modulator ASICs in 2.65GHz.

The choice of this Characterization Kit permits the scaling of the testing platform in 80 channels with maximum line rate 13.1Gbps.

Keywords: FPGA, Xilinx, Multi-Gigabit Transceiver, High-Speed Serial I/O, PWM modulator, digital testing, alignment, Verilog, Hardware Description Language, VLSI, Synthesis, Place and Route

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε κατά το ακαδημαϊκό έτος 2014-2015 υπό την επίβλεψη του Ιωάννη Παπανάνου, καθηγητή Ε.Μ.Π. της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών στον οποίο οφείλω ιδιαίτερες ευχαριστίες για την ανάθεσή της. Μέσα από την διαδικασία της υλοποίησης, αλλά και της συγγραφής της παρούσας διπλωματικής εργασίας, είχα την ευκαιρία να εμβαθύνω σε διάφορους ενδιαφέροντες τομείς καθώς και να αποκτήσω εμπειρία στην ανάπτυξη κώδικα περιγραφής υλικού και την τεχνολογίας των FPGA. Ιδιαίτερες ευχαριστίες θα ήθελα να απευθύνω στον υποψήφιο διδάκτορα κ. Κώστα Γαλανόπουλο για τις συμβουλές του σε καίρια σημεία της σχεδίασης. Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου για τη στήριξη και βοήθεια που μου παρείχαν καθ'όλη την διάρκεια των σπουδών μου.

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΙΝΑΚΑΣ ΣΥΝΤΟΜΟΓΡΑΦΙΩΝ

ASMD	**ALGORITHMIC STATE MACHINE WITH DATAPATH**
CLB	**CONFIGURABLE LOGIC BLOCK**
CPLL	**CHANNEL PLL**
CRC	**CYCLIC REDUNDANCY CHECK**
DSP	**DIGITAL SIGNAL PROCESSING**
FIFO	**FIRST IN, FIRST OUT**
FSM	**FINATE STATE MACHINE**
BER	**BIT ERROR RATE/RATIO**
EOF	**END OF FRAME**
FMC	**FPGA MEZZANINE CARD**
FPGA	**FIELD PROGRAMMABLE GATE ARRAY**
GPIO	**GENERAL PURPOSE INPUT/OUTPUT**
IP	**INTELLECTUAL PROPERTY**
LUT	**LOOK-UP TABLE**
MGT	**MULTI GIGABIT TRANSCEIVER**
PCB	**PRINTED CIRCUIT BOARD**
PER	**PACKET ERROR RATE/RATIO**
PISO	**PARALLEL INPUT SERIAL OUTPUT**
PLL	**PHASE LOCKED LOOP**
PRBS	**PSEUDO RANDOM BINARY SEQUENCE**
QPLL	**QUAD PLL**
SIPO	**SERIAL INPUT PARALLEL OUTPUT**
SMA	**SUBMINIATURE VERSION A**
SOF	**START OF FRAME**
UART	**UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER**
VCO	**VOLTAGE-CONTROLLED OSCILLATOR**

1

Εισαγωγή

1.1 Συστήματα Ελέγχου Ολοκληρωμένων Κυκλωμάτων Υψηλών Ταχυτήτων

Στην σύγχρονη εποχή η σημασία των ασύρματων τηλεπικοινωνιών αυξάνεται διαρκώς. Καθώς ο αριθμός των χρηστών μεγαλώνει, οι χρήστες μεταδίδουν αυξανόμενο όγκο δεδομένων κάθε χρόνο. Οι σημερινές ασύρματες συσκευές, όπως τα smart phones και τα tablet computers υποστηρίζουν πολλαπλά τηλεπικοινωνιακά πρότυπα τα οποία απαιτούν επίσης αυξανόμενο ρυθμό μετάδοσης. Η παραγωγή σημάτων ελέγχου για τις συσκευές αυτές γίνεται συνεχώς μεγαλύτερη πρόκληση αφού πρώτον αυξάνεται το πλήθος των σημάτων ελέγχου και δεύτερον πρέπει να μεταδίδονται σε μεγαλύτερες συχνότητες. Επιπλέον υπάρχει η απαίτηση τα δεδομένα να είναι συγχρονισμένα στις παράλληλες διεπαφές. Τα FPGA αποτελούν μια ιδανική επιλογή για την παραγωγή ψηφιακών σημάτων υψηλών ταχυτήτων καθώς παρέχουν κλιμακοσιμότητα και ευελιξία στην υλοποίηση του συστήματος.

1.2 Αντικείμενο διπλωματικής

Ο σκοπός της διπλωματικής εργασίας ήταν η ανάπτυξη μια πλατφόρμας ελέγχου ολοκληρωμένων κυκλωμάτων υψηλών ταχυτήτων. Για τον έλεγχο των ολοκληρωμένων κυκλωμάτων απαιτείται η παράγωγη παράλληλων διανυσμάτων υψηλής ταχύτητας καθώς και ανάγνωση και αποθήκευση ειδικών σημάτων ελέγχου που παρέχουν τα ολοκληρωμένα κυκλώματα.

Η πλατφόρμα έχει υλοποιηθεί στο υψηλών επιδόσεων Virtex-7 FPGA αναπτυξιακό χαρακτηρισμού VC7215 της Xilinx. Η πλατφόρμα δέχεται εντολές και δεδομένα μέσω MATLAB. Τα δεδομένα αποθηκεύονται σε πολλαπλά μπλοκ μνήμης και αναπαράγονται μέσω σειριακών πομπών που διαθέτει το αναπτυξιακό. Η πλατφόρμα θα χρησιμοποιηθεί για την επαλήθευση ψηφιακού RF διαμορφωτή ASIC παρέχοντας του δεδομένα με ρυθμό 2.65GHz.

1.2.1 Συνεισφορά

Η συνεισφορά της διπλωματικής συνοψίζεται ως εξής:

1. Αναπαραγωγή παράλληλων διανυσμάτων υψηλών ταχυτήτων.
2. Ευθυγράμμιση δεδομένων.
3. Υλοποίηση ελεγκτή εντολών.
4. Υλοποίηση διασύνδεσης υπολογιστή και FPGA.
5. Συλλογή δεδομένων εισόδου από την συσκευή υπό έλεγχο και μετάδοση τους στον υπολογιστή.
6. Κατασκευή βιβλιοθήκης χειρισμού της πλατφόρμας ελέγχου.
7. Ρύθμιση του επιπέδου τάσης των καναλιών
8. Ρύθμιση της φάσης κάθε καναλιού ξεχωριστά

1.3 Οργάνωση κειμένου

Στο Κεφάλαιο 2 γίνεται μια εισαγωγή στην τεχνολογία και την αρχιτεκτονική των FPGAs. Στο Κεφάλαιο 3 παρουσιάζονται οι πομποδέκτες υψηλών ταχυτήτων και η οργάνωση τους. Στο Κεφάλαιο 4 αναπτύσσεται το κατασκευαστικό κομμάτι της εργασία και παρατίθενται οι λεπτομέρειες της υλοποίησης. Στο Κεφάλαιο 5 παρουσιάζεται η βιβλιοθήκη συναρτήσεων χειρισμού της πλατφόρμας. Στο Κεφάλαιο 6 γίνεται σύνοψη των αποτελεσμάτων και αναφέρονται μελλοντικές επεκτάσεις. Στο Παράρτημα Α, δίνεται ο κώδικας περιγραφής υλικού σε Verilog. Στο Παράρτημα Β, παρατίθεται ο κώδικας της βιβλιοθήκης σε MATLAB.

2

Θεωρητικό υπόβαθρο

Σε αυτό το εισαγωγικό κεφάλαιο γίνεται μια εισαγωγή και περιγραφή της συστοιχίας επιτόπια προγραμματιζόμενων πυλών (Field Programmable Gate Array FPGA). Στη συνέχεια δίνεται μια σύντομη ιστορική αναδρομή των FPGA και ακολουθεί η περιγραφή της αρχιτεκτονικής τους.

2.1 FPGA

Ένα FPGA είναι μια συσκευή ημιαγωγών αποτελούμενη από ένα πλέγμα προγραμματιζόμενων λογικών μπλοκ (CLB - Configurable Logic Blocks) που συνδέονται μεταξύ τους μέσω προγραμματιζόμενων διασυνδέσεων. Ένα FPGA μπορεί να επαναπρογραμματιστεί απεριόριστες φορές έτσι ώστε να υλοποιεί την επιθυμητή λειτουργικότητα. Αυτό το χαρακτηριστικό διαφοροποιεί τα FPGA από τα ολοκληρωμένα κυκλώματα ειδικού σκοπού (ASIC) που κατασκευάζονται έτσι ώστε να εκτελούν ειδικές λειτουργίες και απαιτήσεις.

Τα FPGA είναι κατάλληλα για εφαρμογές που οι παράμετροι λειτουργίας αλλάζουν συχνά ή για μικρές ποσότητες παραγωγής. Το γεγονός αυτό τα κάνει ιδανικά για testing platforms, όπου χρησιμοποιούνται για να επιβεβαιώσουν την ορθή λειτουργία πολλών συστημάτων με παραμέτρους που αντιστοιχούν σε διαφορετικά πρωτόκολλα και απαιτήσεις. Αντίθετα τα ASIC, στα οποία η λειτουργία είναι αυστηρά καθορισμένη, αν και έχουν πολύ

μεγαλύτερο κόστος σχεδίασης και κατασκευής, καταλήγουν με πολύ μικρότερο κόστος ανά τεμάχιο διότι κατασκευάζονται σε μεγάλες ποσότητες.

2.2 Ιστορική Αναδρομή

Η τεχνολογία FPGA έχει ως πρόγονο τόσο τις προγραμματιζόμενες μνήμες (PROM) όσο και τις προγραμματιζόμενες λογικές διατάξεις (PLDs). Και οι δύο είχαν την δυνατότητα να προγραμματιστούν στο πεδίο λειτουργίας (field programmable), παρόλα αυτά οι λογικές πύλες ήταν μόνιμα συνδεδεμένες με την προγραμματιζόμενη λογική.

Στα τέλη της δεκαετίας του 1980 το Πολεμικό Ναυτικό των ΗΠΑ χρηματοδότησε την ιδέα του Steve Casselman για σχεδίαση ενός υπολογιστή αποτελούμενο από 600.000 επαναπρογραμματιζόμενες πύλες. Το εγχείρημα του Casselman ήταν επιτυχές και η ευρεσιτεχνία κατοχυρώθηκε το 1992.

Η πρώτη εμπορικά διαθέσιμη συστοιχία επαναπρογραμματιζόμενων λογικών πυλών, η XC2064, εμφανίστηκε το 1985 από τους Ross Freeman και Bernard Vonderschmitt. Το XC2064 είχε 64 προγραμματιζόμενα λογικά μπλοκ που περιείχαν LUTs (Look-Up Tables) τριών εισόδων που διασυνδέονταν μεταξύ τους μέσω προγραμματιζόμενων διασυνδέσεων.

Η δεκαετία του 1990 ήταν μια εκρηκτική περίοδος για την τεχνολογία των FPGA, τόσο για την αύξηση της πολυπλοκότητας, όσο και την αύξηση των πωλήσεων. Από την χρήση στις τηλεπικοινωνίες και στα δίκτυα, η τεχνολογία των FPGA σιγά σιγά κατέκτησε και άλλους τομείς όπως η αυτοκινητοβιομηχανία, οι βιομηχανικές εφαρμογές αλλά και καταναλωτικά προϊόντα.

2.3 Αρχιτεκτονική FPGA

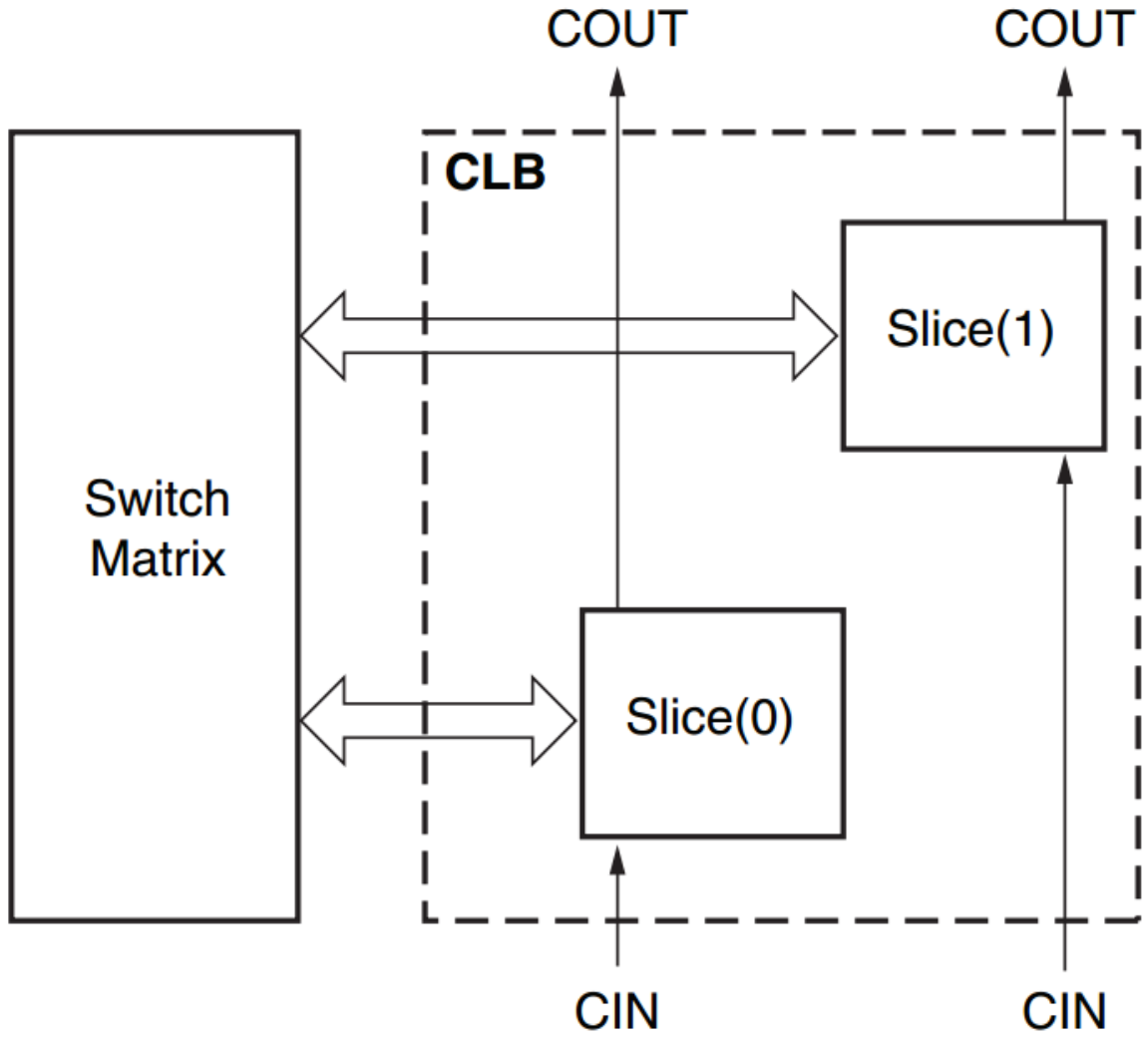
Η κλασσική αρχιτεκτονική των FPGA αποτελείται από μια συστοιχία προγραμματιζόμενων λογικών μπλοκ (Configurable Logic Block), κανάλια διασύνδεσης και ακίδες εισόδου εξόδου. Στην σειρά Virtex-7 κάθε CLB αποτελείται από:

- 6 εισόδων look-up table (LUT)
- Δυνατότητα κατανεμημένης μνήμης και καταχωρητή ολίσθησης
- Ειδική λογική κρατουμένου για αριθμητικές πράξεις
- Πολυπλέκτες για αποδοτική χρήση

Τα CLBs είναι οι βασικοί λογικοί πόροι για την υλοποίηση ακολουθιακής και συνδυαστικής λογικής. Κάθε CLB περιέχει δύο slices που συνδέονται σε ένα πίνακα διακοπτών για πρόσβαση στο γενικό δίκτυο διασύνδεσης. Τα LUTs μπορούν να ρυθμιστούν είτε ως 6 εισόδων με μια έξοδο, είτε με 5 εισόδους με ξεχωριστές εξόδους αλλά κοινές

διευθύνσεις ή εισόδους. Κάθε έξοδος ενός LUT 5-εισόδων μπορεί προαιρετικά να συνδεθεί με ένα flip-flop. Τέσσερα LUTs 6-εισόδων και τα οχτώ flip-flop τους καθώς και οι πολυπλέκτες και η λογική κρατουμένου σχηματίζουν ένα slice. Δύο slices σχηματίζουν ένα CLB. Τέσσερα flip-flop ανά slice (ένα σε κάθε LUT) μπορούν προαιρετικά να ρυθμιστούν ως μανδαλωτές. Σε αυτή την περίπτωση τα υπολειπόμενα τέσσερα flip-flops μένουν αχρησιμοποίητα. Προσεγγιστικά δύο τρίτα των slices είναι SLICEL (λογικά slices) και τα υπόλοιπα είναι SLICEM, τα οποία μπορούν επίσης να χρησιμοποιήσουν τα LUTs ως κατανεμημένη μνήμη των 64bit RAM ή ως 32bit καταχωρητές ολίσθησης (SRL32) ή ως δύο SRL16. Τα χαρακτηρίστηκα του FPGA που χρησιμοποιήθηκε (7VX690T) φαίνονται στον παρακάτω πίνακα:

Slices	SLICEL	SLICEM	6-input LUTs	Distributed RAM (Kb)	Shift Register(Kb)	Flip-Flops
108300	64750	43550	433200	10888	5444	866400

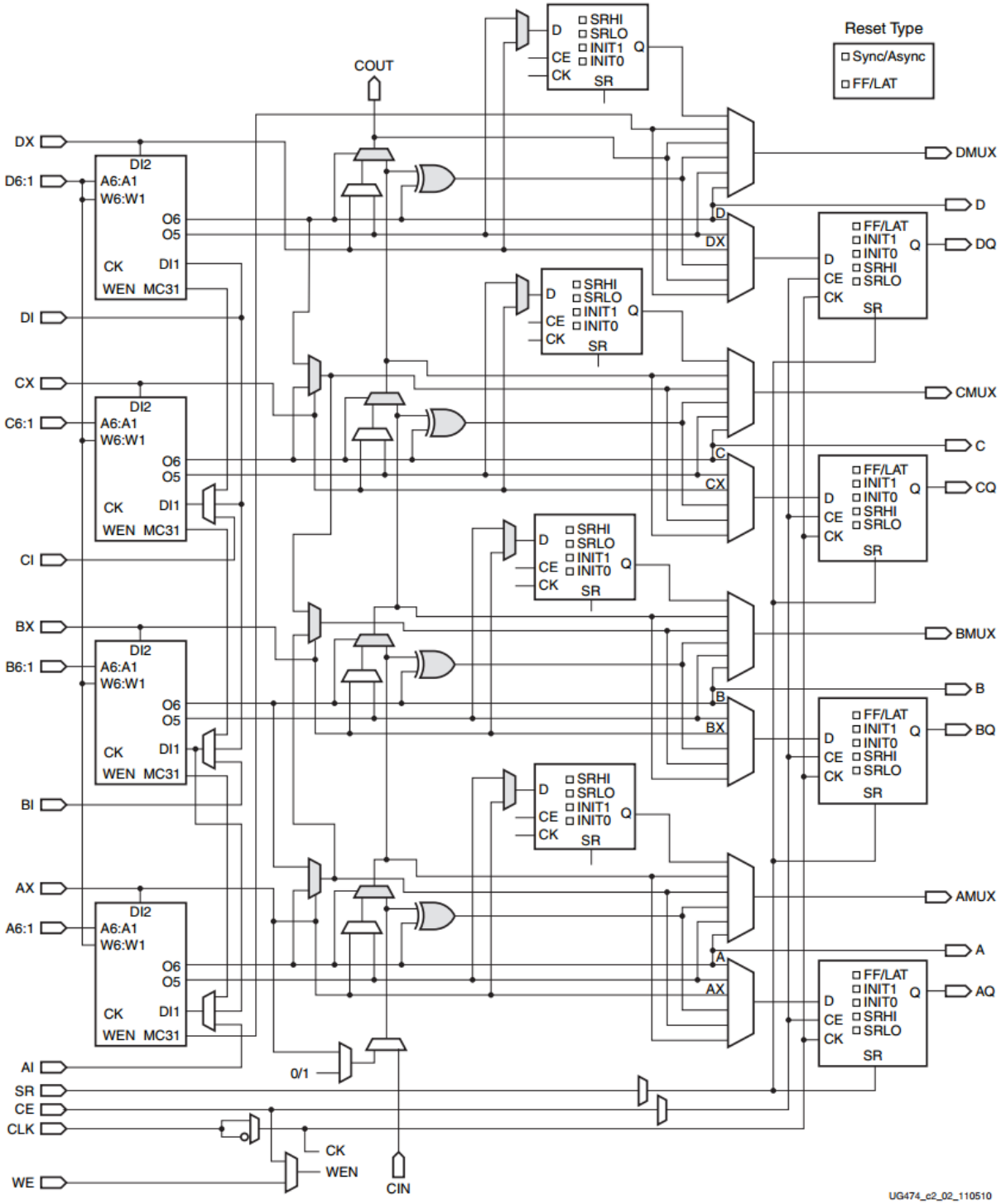


2.3.1 Περιγραφή των Slices

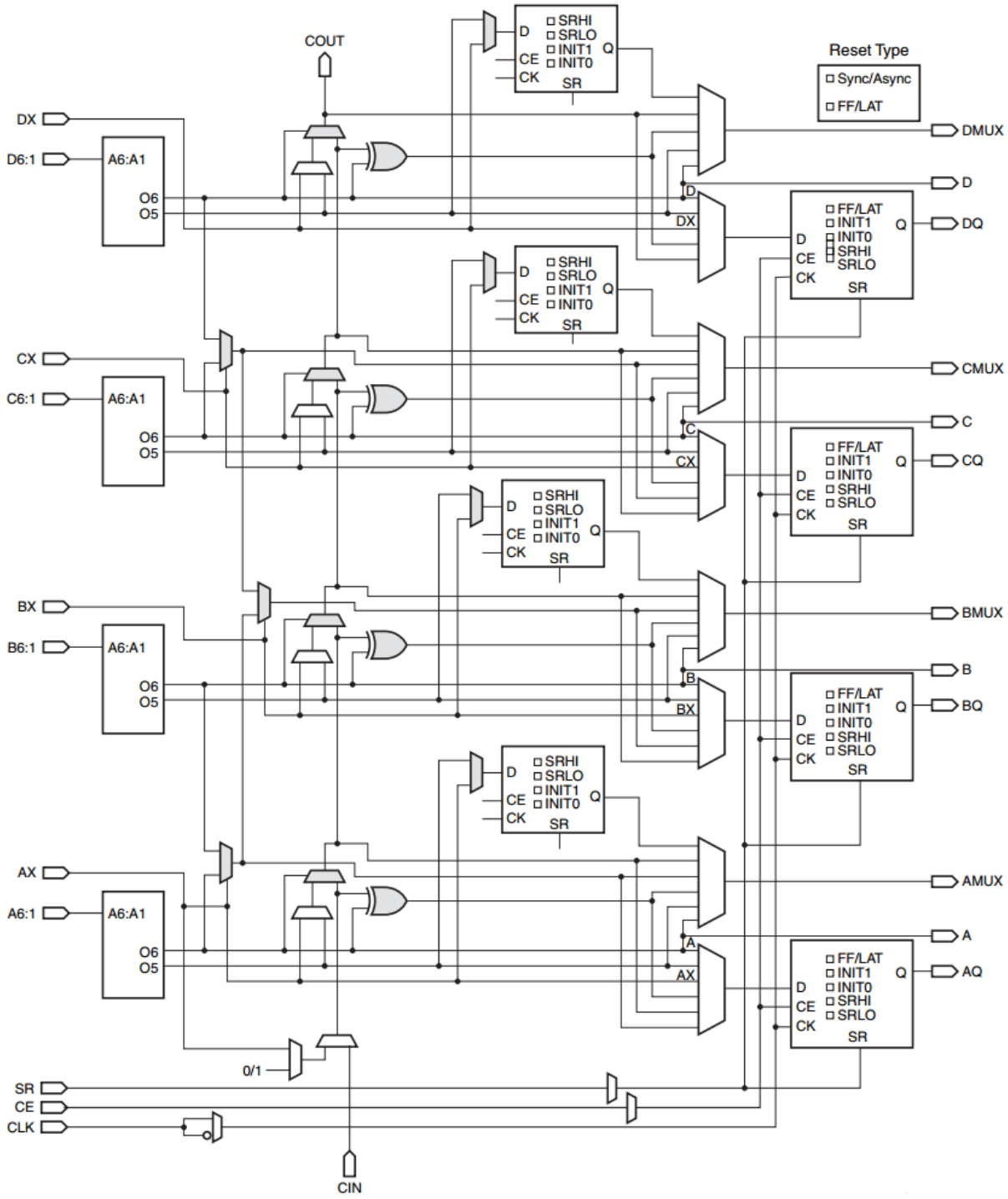
Κάθε slice περιέχει:

- Τέσσερις γεννήτριες λογικών συναρτήσεων (LUTs)
- Οχτώ στοιχεία μνήμης (Flip-Flops)
- Πολυπλέκτες
- Λογική Κρατουμένου

Αυτά τα στοιχεία χρησιμοποιούνται από όλα τα slices για λογικές και αριθμητικές συναρτήσεις καθώς και λειτουργίες ROM. Επιπρόσθετα, ορισμένα slices υποστηρίζουν επιπλέον δύο λειτουργίες: αποθήκευση δεδομένων μέσω κατανεμημένων RAM και ολίσθηση δεδομένων με 32bit καταχωρητών. Τα slices που υποστηρίζουν αυτές τις επιπλέον λειτουργίες καλούνται SLICEM, ενώ τα υπόλοιπα λέγονται SLICEL. Κάθε CLB μπορεί να περιέχει είτε δύο SLICEL είτε ένα SLICEL και ένα SLICEM. Στα παρακάτω δύο διαγράμματα φαίνεται η αρχιτεκτονική των slices:


Reset Type
Sync/Async
FF/LAT
COUT
DX
D6:1
DI
CX
C6:1
CI
BX
B6:1
BI
AX
A6:1
AI
SR
CE
CLK
WE
DI2
A6:A1
W6:W1
O6
O5
CK
DI1
WEN
MC31
SRHI
SRLO
INIT1
INIT0
D
CE
CK
SR
Q
FF/LAT
DMUX
D
DQ
CMUX
C
CQ
BMUX
B
BQ
AMUX
A
AQ
0/1
CK
WEN
CIN
UG474_c2_02_110510

SLICEM


COUT
Reset Type
Sync/Async
FF/LAT
SRHI
SRLO
INIT1
INIT0
D
CE
CK
SR
Q
DX
D6:1
A6:A1
O6
O5
DMUX
D
DQ
FF/LAT
CX
C6:1
CMUX
C
CQ
BX
B6:1
BMUX
B
BQ
AX
A6:1
AMUX
A
AQ
0/1
SR
CE
CLK
CIN

SLICEL

2.3.1.1 Περιγραφή Look-Up Table (LUT)

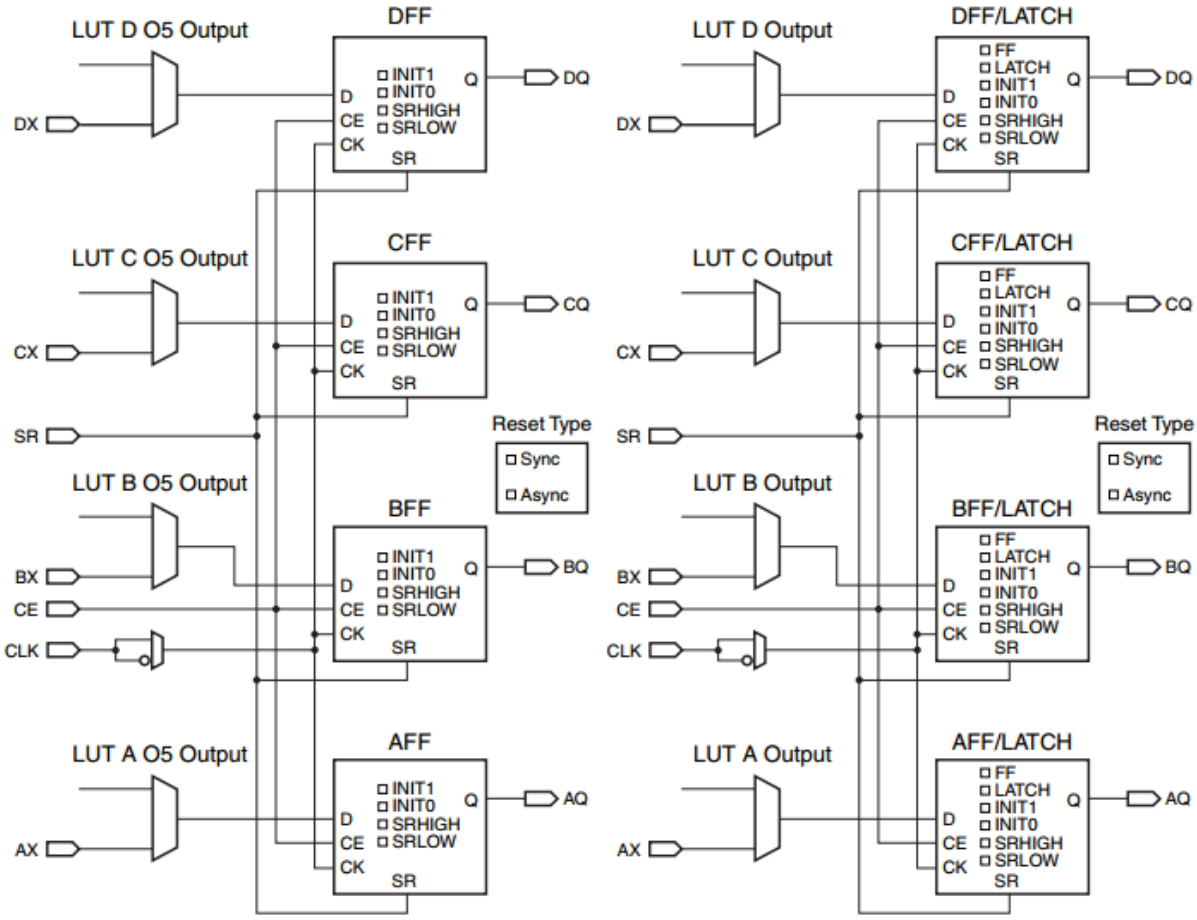
Οι γεννήτριες συναρτήσεων υλοποιούνται με LUT 6-εισόδων. Υπάρχουν έξι ανεξάρτητες είσοδοι και δύο ανεξάρτητες έξοδοι για κάθε μια από τις τέσσερις γεννήτριες συναρτήσεων σε ένα slice. Η καθυστέρηση διάδοσης είναι ανεξάρτητη της συνάρτησης που υλοποιείται. Οι γεννήτριες συναρτήσεων μπορούν να υλοποιήσουν:

- Μια λογική συνάρτηση 6-εισόδων
- Δύο λογικές συναρτήσεις 5-εισόδων, στην περίπτωση που αυτές οι δύο συναρτήσεις έχουν κοινές εισόδους.
- Δύο λογικές συναρτήσεις τριών και δύο εισόδων

Τα slices, πέρα από τα βασικά LUTs, περιέχουν τρεις πολυπλέκτες οι οποίοι συνδυάζονται για την υλοποίηση λογικών συναρτήσεων 7 ή 8-εισόδων. Για την υλοποίηση συναρτήσεων με περισσότερες από 8-εισόδους συνδυάζονται πολλαπλά slices.

2.3.2 Στοιχεία Αποθήκευσης

Υπάρχουν οχτώ στοιχεία αποθήκευσης σε κάθε slice. Τέσσερα μπορούν να διαμορφωθούν είτε ως ακμοπυροδότητα D flip-flops είτε ως μανδαλωτές ευαίσθητοι σε επίπεδο. Ένας μανδαλωτής παρουσιάζεται διαφανής όταν το σήμα ρολογιού CLK είναι Low. Οι D είσοδοι μπορούν να οδηγηθούν κατευθείαν από ένα LUT μέσω ενός πολυπλέκτη ή από τις εισόδους του slice παρακάμπτοντας τις γεννήτριες συναρτήσεων.



2.3.3 Διασύνδεση Πόρων

Ένα πλέγμα διατάξεων διακοπτών τοποθετούνται ως επίστρωμα στην αρχιτεκτονική των CLB, ώστε να παρέχει διασύνδεση γενικού σκοπού για διακλάδωση διαδρόμων δεδομένων και δρομολόγηση σημάτων σε όλη την συσκευή. Το σύστημα διασύνδεσης παρέχει τρεις τύπους συνδέσεων γενικού σκοπού: γραμμές μονού μήκους, γραμμές διπλού μήκους και μακρές γραμμές, Ένα πλέγμα από οριζόντιες και κατακόρυφες γραμμές μονού μήκους διασυνδέει μια διάταξη από σύνολα διακοπτών (switch boxes), οι οποίοι προσφέρουν μικρό αριθμό συνδέσεων μεταξύ διαδρομών σήματος μέσα σε κάθε σύνολο, αλλά κανένα καθολικό διακόπτη. Κάθε CLB διαθέτει ένα ζεύγος από απομονωτές τριών καταστάσεων που μπορούν να οδηγήσουν σήματα στις κοντινότερες οριζόντιες γραμμές πάνω ή κάτω από το CLB.

Οι (ειδικού σκοπού) προγραμματίσιμες, απευθείας γραμμές διασύνδεσης παρέχουν δρομολόγηση μεταξύ γειτονικών κατακόρυφων και οριζόντιων CLB που ανήκουν στην ίδια γραμμή ή στήλη. Πρόκειται για τοπικές συνδέσεις με σχετικά υψηλή ταχύτητα, που

υλοποιούνται με μεταλλικά στοιχεία. Οι γραμμές, όμως είναι πιο αργές από τις απλές μεταλλικές συνδέσεις ανάμεσα σε ζεύγη σημείων του ολοκληρωμένου, λόγω της καθυστέρησης που προκύπτει από την παρεμβολή των πυλών μετάδοσης, οι οποίες χρησιμοποιούνται για να προγραμματιστούν οι συγκεκριμένες διαδρομές. Οι απευθείας γραμμές διασύνδεσης δεν χρησιμοποιούν τις διατάξεις διακοπτών, εξαλείφοντας έτσι την αντίστοιχη καθυστέρηση.

Οι γραμμές διπλού μήκους διασχίζουν την απόσταση δύο CLB πριν εισέλθουν σε έναν πίνακα διακοπτών, παρακάμπτοντας τα ενδιάμεσα CLB.Αυτές οι γραμμές παρέχουν μια πιο αποτελεσματική υλοποίηση διασυνδέσεων ενδιάμεσου μήκους, αγνοώντας διατάξεις διακοπτών της διαδρομής και , κατά συνέπεια, μειώνοντας την καθυστέρηση διαδρομής.

Οι μακρές γραμμές διασχίζουν τη συνολική διάταξη των CLB κατακόρυφα και οριζόντια. Οι γραμμές αυτές χρησιμοποιούνται για την οδήγηση σημάτων ελέγχου, με τα οποία εξασφαλίζεται μικρή χρονική απόκλιση και δυνατότητα οδήγησης σημαντικού αριθμού λογικών εισόδων.

2.3.4 Μπλοκ Εισόδου/Εξόδου (IOB)

Κάθε προγραμματίσιμος ακροδέκτης I/O έχει ένα προγραμματίσιμο IOB με απομονωτές που επιτρέπουν συμβατότητα με τα επίπεδα σήματος. Το IOB μπορεί να χρησιμοποιηθεί ως είσοδος, ως έξοδος ή και ως αμφίδρομη θύρα. Ένα IOB που διαμορφώνεται ως είσοδος είναι δυνατό να παίρνει εξωτερικά δεδομένα ή απευθείας ή μέσω ενός μανδαλωτή ή μέσω ενός καταχωρητή. Όταν είναι διαμορφωμένο σαν έξοδος, το IOB μπορεί να παίρνει δεδομένα είτε απευθείας ή μέσω ενός καταχωρητή. Ο απομονωτής εξόδου ενός IOB διαθέτει μηχανισμούς ελέγχου της χρονικής απόκλισης (skew) και του ρυθμού μεταβολής του σήματος (slew). Οι καταχωρητές που υπάρχουν στη διαδρομή εισόδου και στην διαδρομή εξόδου ενός IOB οδηγούνται από ξεχωριστά, αντιστρέψιμα ρολόγια, ενώ υπάρχει και ένας καθολικός μηχανισμός θέσης/μηδενισμού.

Τα εσωτερικά στοιχεία καθυστέρησης αντισταθμίζουν την καθυστέρηση που παρουσιάζεται όταν ένα σήμα ρολογιού διέρχεται από ένα καθολικό απομονωτή, πριν φθάσει στο IOB. Αυτή η στρατηγική εξαλείφει την απαίτηση διατήρησης αναλλοίωτων των δεδομένων για ένα χρονικό διάστημα σε έναν εξωτερικό ακροδέκτη. Η έξοδος τριών καταστάσεων ενός IOB μπορεί, αν χρειαστεί, να καταστήσει τη σύνθετη αντίσταση εξόδου του απομονωτή εξόδου πολύ υψηλή (high-z). Οι τιμές της εξόδου και της επίτρεψης εξόδου είναι δυνατόν να αντιστραφούν. Ο ρυθμός μεταβολής του σήματος εξόδου του απομονωτή εξόδου μπορεί να ελεγχθεί, ώστε να ελαχιστοποιηθούν οι αιφνίδιες μεταβολές ρεύματος στο δίαυλο παροχής ισχύος, όταν επιλέγονται μη κρίσιμα σήματα. Ο ακροδέκτης IOB είναι

δυνατό να προγραμματιστεί, ώστε η στάθμη του σήματος να ανεβαίνει στο 1 ή να κατεβαίνει στο 0 με τέτοιο τρόπο ώστε να αποτρέπονται η περιττή κατανάλωση ρεύματος και ο θόρυβος.

3

Πομποδέκτες Υψηλών Ταχυτήτων (Multi-Gigabit Transceivers)

3.1 Εισαγωγή

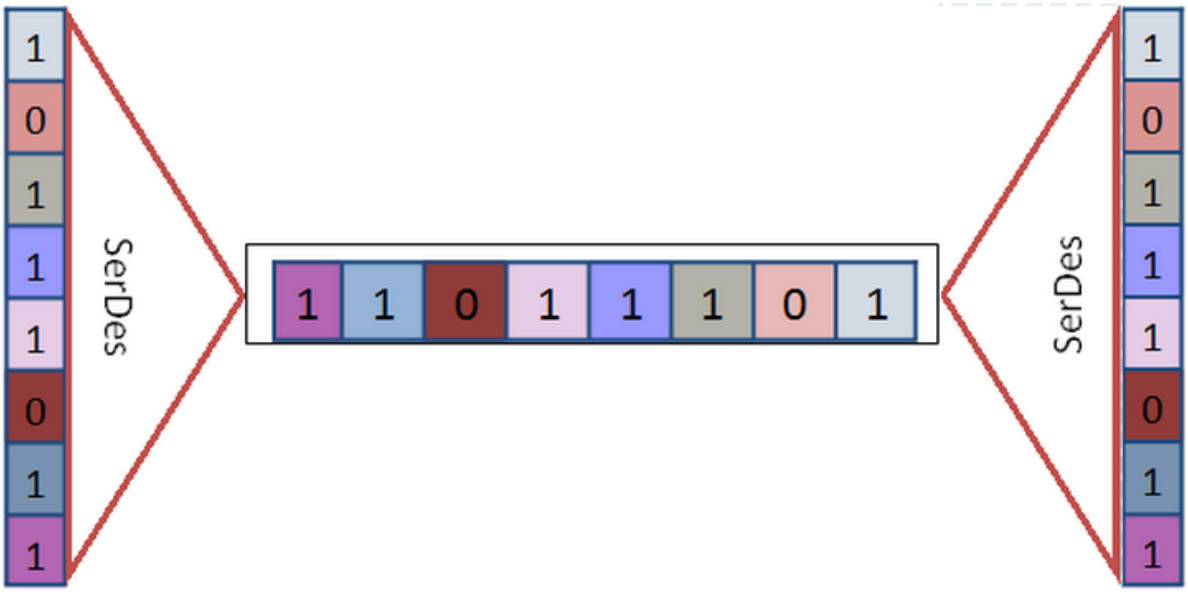
Ένας πομποδέκτης υψηλών ταχυτήτων MGT είναι ένα serializer/deserializer (SerDes) ο οποίος λειτουργεί σε ρυθμούς μετάδοσης άνω του ενός Gigabit/second. Οι MGTs χρησιμοποιούνται ευρύτατα στις τηλεπικοινωνίες διότι μπορούν να μεταδώσουν σε μεγαλύτερες αποστάσεις, χρησιμοποιούν λιγότερα καλώδια και συνεπώς έχουν μικρότερο κόστος από τις παράλληλες διεπαφές με την ίδια ικανότητα διαβίβασης δεδομένων.

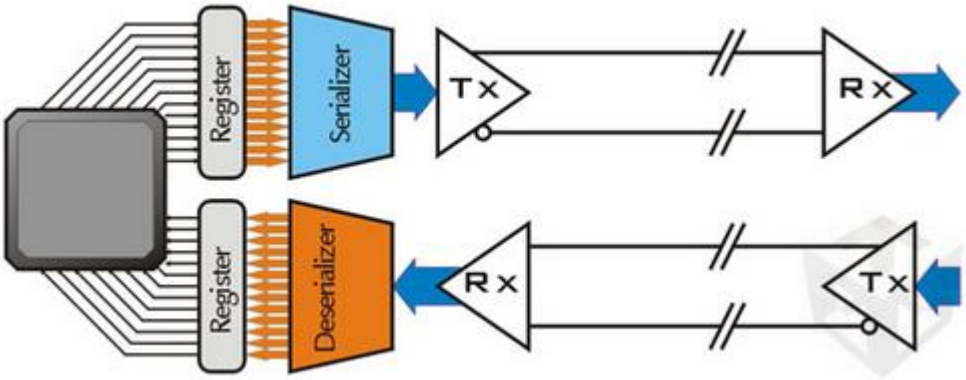
Η βασική λειτουργία ενός SerDes συνίσταται σε δύο λειτουργικά μπλοκ:

1. Parallel In Serial Out(PISO)
2. Serial In Parallel Out (SIPO)

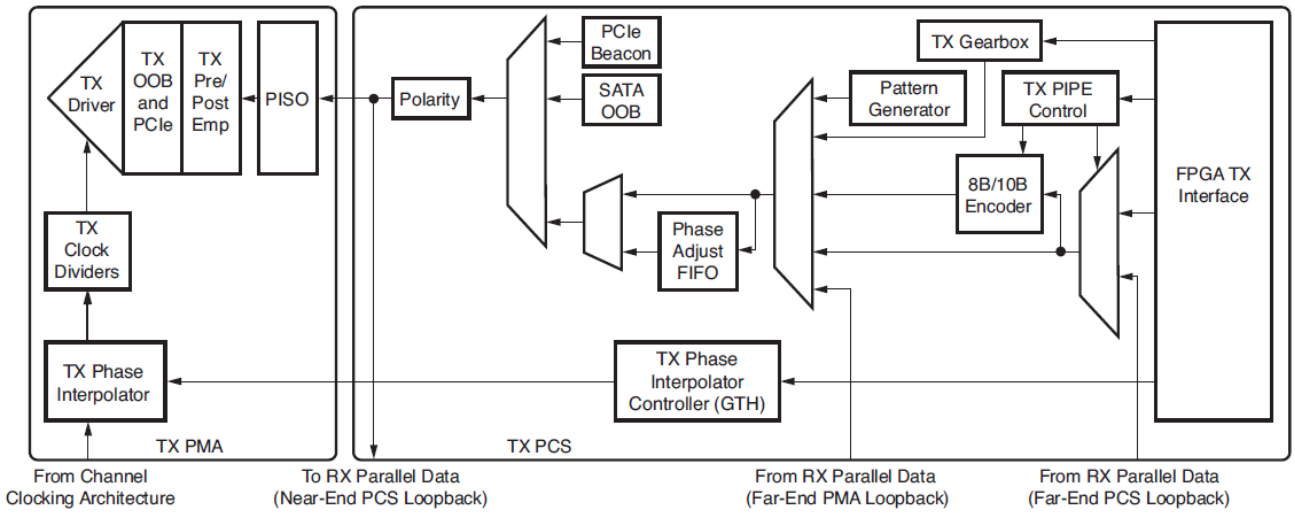
Το μπλοκ PISO έχει ένα παράλληλο ρολόι εισόδου, ένα σύνολο από γραμμές δεδομένων εισόδου και μανδαλωτές δεδομένων εισόδου. Ένας εξωτερικός βρόχος κλειδωμένης φάσης (PLL) χρησιμοποιείται για να πολλαπλασιάσει την συχνότητα του εισερχόμενου παράλληλου ρολογιού στην επιθυμητή σειριακή συχνότητα. Η απλούστερη μορφή ενός PISO είναι ένας καταχωρητής ολίσθησης με παράλληλη φόρτωση, στον οποίο φορτώνονται τα παράλληλα δεδομένα εισόδου σε κάθε κύκλο του παράλληλου ρολογιού, και τα ολισθαίνει προς την έξοδο με τον υψηλότερο ρυθμό μετάδοσης του σειριακού ρολογιού.

Από την άλλη πλευρά, ένα μπλοκ SIPO έχει ένα ρολόι λήψης ως έξοδο, ένα σύνολο γραμμών από γραμμές εξόδου και μανδαλωτές δεδομένων εξόδου. Το ρολόι λήψης ανακτάται από τα δεδομένα με την τεχνική clock recovery και έπειτα διαιρείται με τον αριθμό των γραμμών για να προκύψει το παράλληλο ρολόι εξόδου. Τυπικές υλοποιήσεις ενός SerDes έχουν δύο καταχωρητές συνδεδεμένους σαν ένα διπλό απομονωτή.



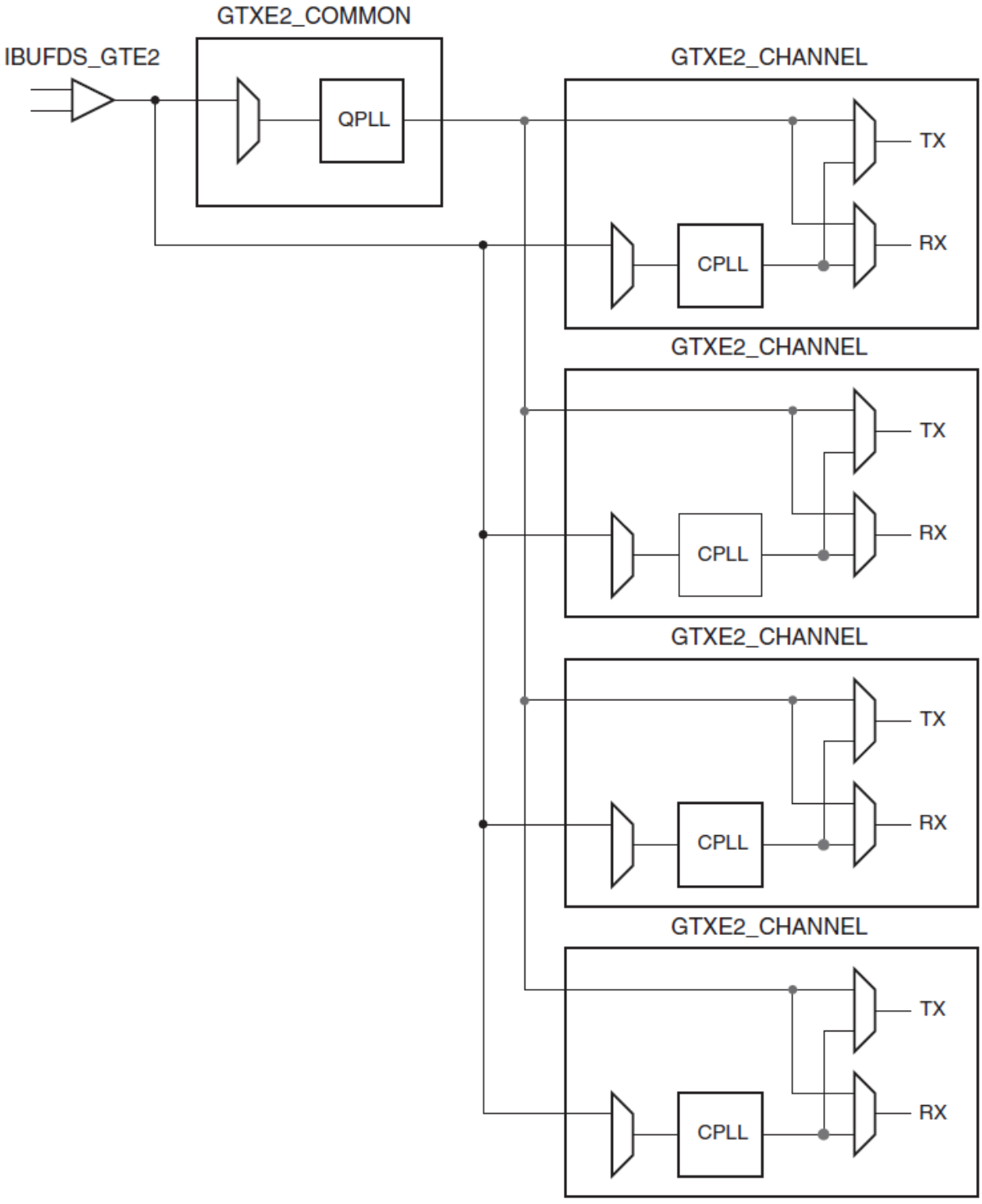


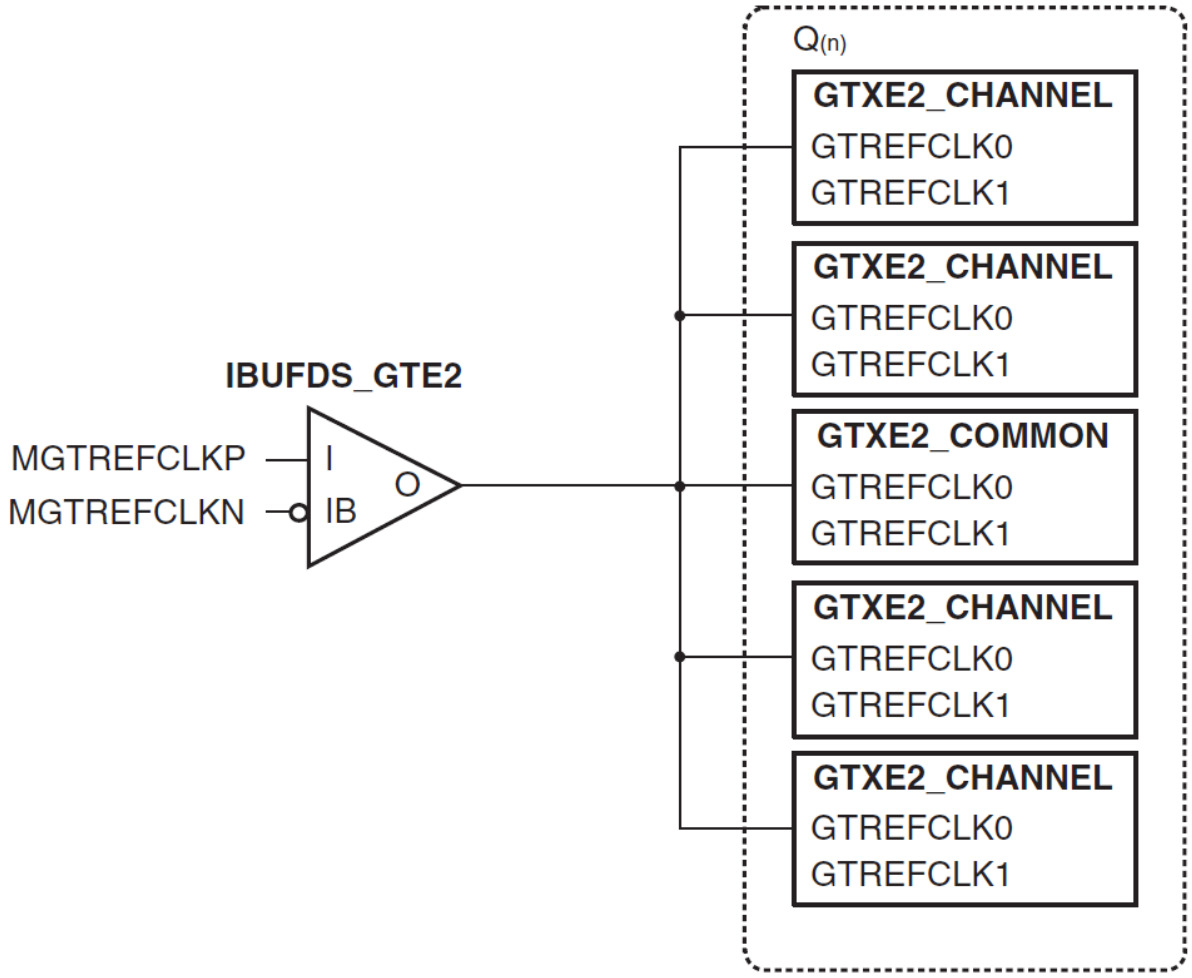
Στο παρακάτω σχήμα φαίνεται το μπλοκ διάγραμμα των πομποδεκτών GTH που διαθέτει το FPGA:



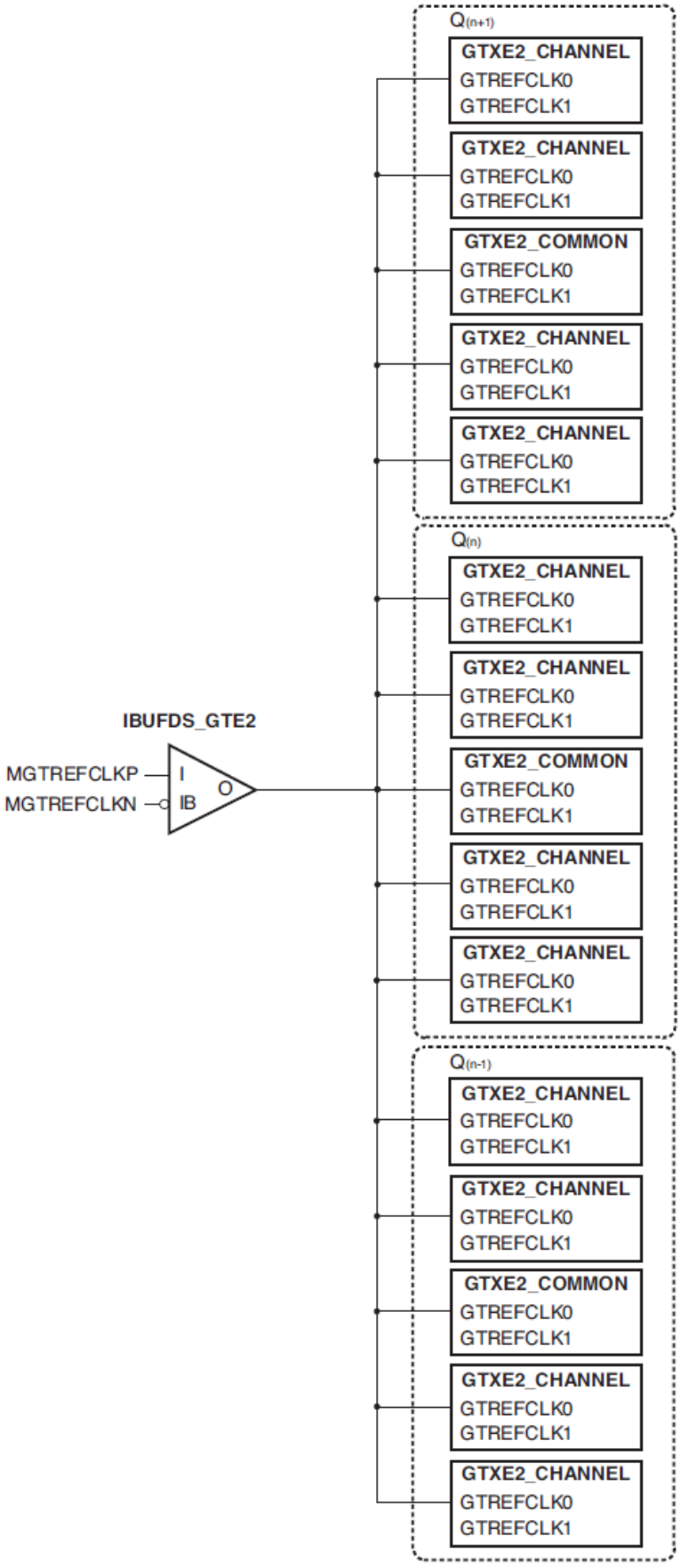
3.2 Διανομή ρολογιού αναφοράς

Οι transceivers οργανώνονται σε τετράδες οι οποίες στο εξής θα καλούμε QUAD. Κάθε QUAD περιέχει ένα QPLL (QUAD PLL) το οποίο δέχεται ένα ρολόι αναφοράς (παράλληλο ρολόι) με συχνότητα 265MHz. Στα παρακάτω σχήματα φαίνεται η διανομή του ρολογιού αναφοράς στους τέσσερις transceivers ενός QUAD, το οποίο αρχικά εισέρχεται σε ένα διαφορικό απομονωτή IBUFDS_GTE2 και στη συνέχεια τροφοδοτεί το κοινό QPLL. Σε αυτή την εφαρμογή τα CPLL (Channel PLL) είναι ανενεργά.


GTXE2_COMMON
IBUFDS_GTE2
QPLL
GTXE2_CHANNEL
TX
RX
CPLL
GTXE2_CHANNEL
TX
RX
CPLL
GTXE2_CHANNEL
TX
RX
CPLL
GTXE2_CHANNEL
TX
RX
CPLL



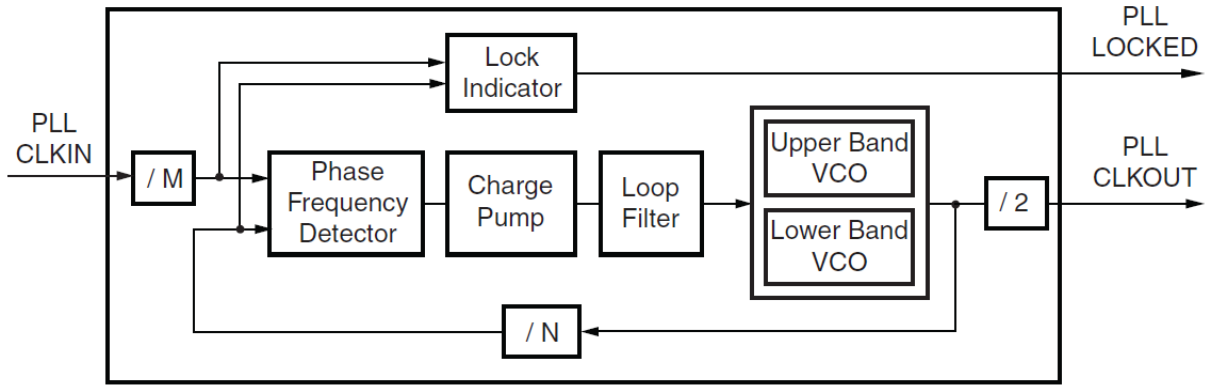
Προκειμένου να τηρηθούν τα περιθώρια jitter για σχεδιάσεις υψηλών ταχυτήτων είναι απαραίτητο ένα κοινό εξωτερικό διαφορικό ρολόι αναφοράς MGTREFCLKP/N να τροφοδοτεί μέχρι 12 transceivers ή 3 QUADS. Σχεδιάσεις με περισσότερους από 12 transceivers απαιτούν την χρήση πολλαπλών εξωτερικών ρολογιών. Στο παρακάτω σχήμα απεικονίζεται η χρήση ενός κοινού εξωτερικού ρολογιού αναφοράς για την οδήγηση 12 transceivers που ανήκουν σε 3 γειτονικά QUADS. Στην παρούσα σχεδίαση χρησιμοποιούνται 24 transceivers, οπότε χρησιμοποιήθηκαν δύο συμφασικά εξωτερικά ρολόγια αναφοράς στα 265MHz, τα οποία παράχθηκαν από το SuperClock-2 Module που βρίσκεται πάνω στο αναπτυξιακό.


Q(n+1)
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_COMMON
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
Q(n)
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_COMMON
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
IBUFDS_GTE2
MGTREFCLKP
MGTREFCLKN
I
IB
O
Q(n-1)
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_COMMON
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1
GTXE2_CHANNEL
GTREFCLK0
GTREFCLK1

3.3 QUAD PLL

Κάθε QUAD περιέχει ένα PLL βασισμένο σε LC. Ένα QPLL μπορεί να τροφοδοτήσει τους transceivers που βρίσκονται στο ίδιο QUAD. Οι έξοδοι το QPLL τροφοδοτούν τους διαιρέτες ρολογιού των TX και RX κάθε σειριακού καναλιού μέσα στο ίδιο QUAD. Στο παρακάτω σχήμα φαίνεται η αρχιτεκτονική ενός QPLL. Το εισερχόμενο σήμα ρολογιού PLL CLKIN μπορεί να διαιρεθεί με ένα παράγοντα M πριν εισέλθει στον ανιχνευτή φάσης συχνότητας (Phase Frequency Detector). Ο διαιρέτης ανάδρασης N καθορίζει τον λόγο πολλαπλασιασμού του VCO. Η συχνότητα εξόδου του QPLL είναι η μίση της συχνότητας του VCO. Το μπλοκ ανίχνευσης κλειδώματος (Lock Indicator) συγκρίνει το ρολόι αναφοράς και το σήμα ανάδρασης που βγαίνει από τον VCO για να προσδιορίσει εάν έχει επιτευχθεί κλείδωμα της συχνότητας. Η συχνότητα εξόδου υπολογίζεται από την παρακάτω σχέση σε GHz:

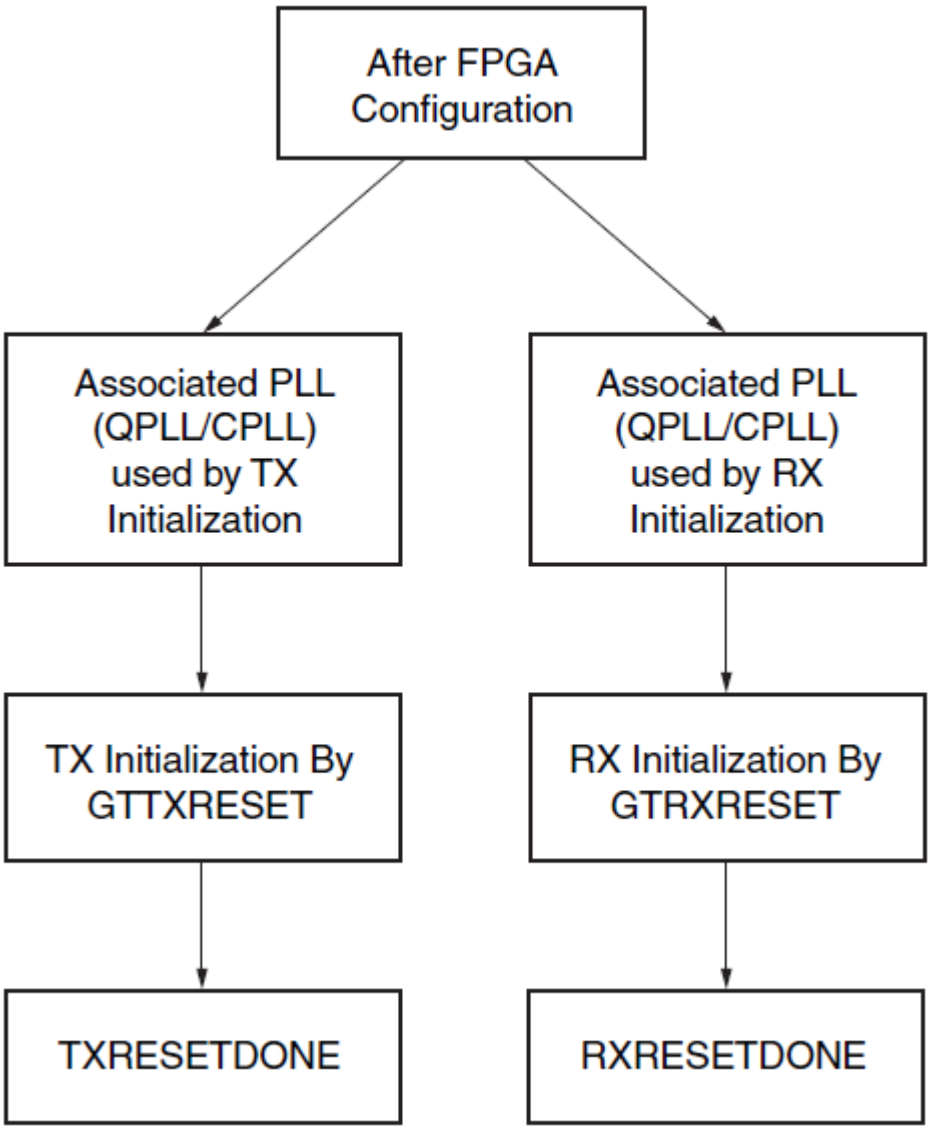
$$f_{PLLCLKout} = f_{PLLCLKin} \frac{N}{2M}$$



3.4 Αρχικοποίηση και Επαναφορά

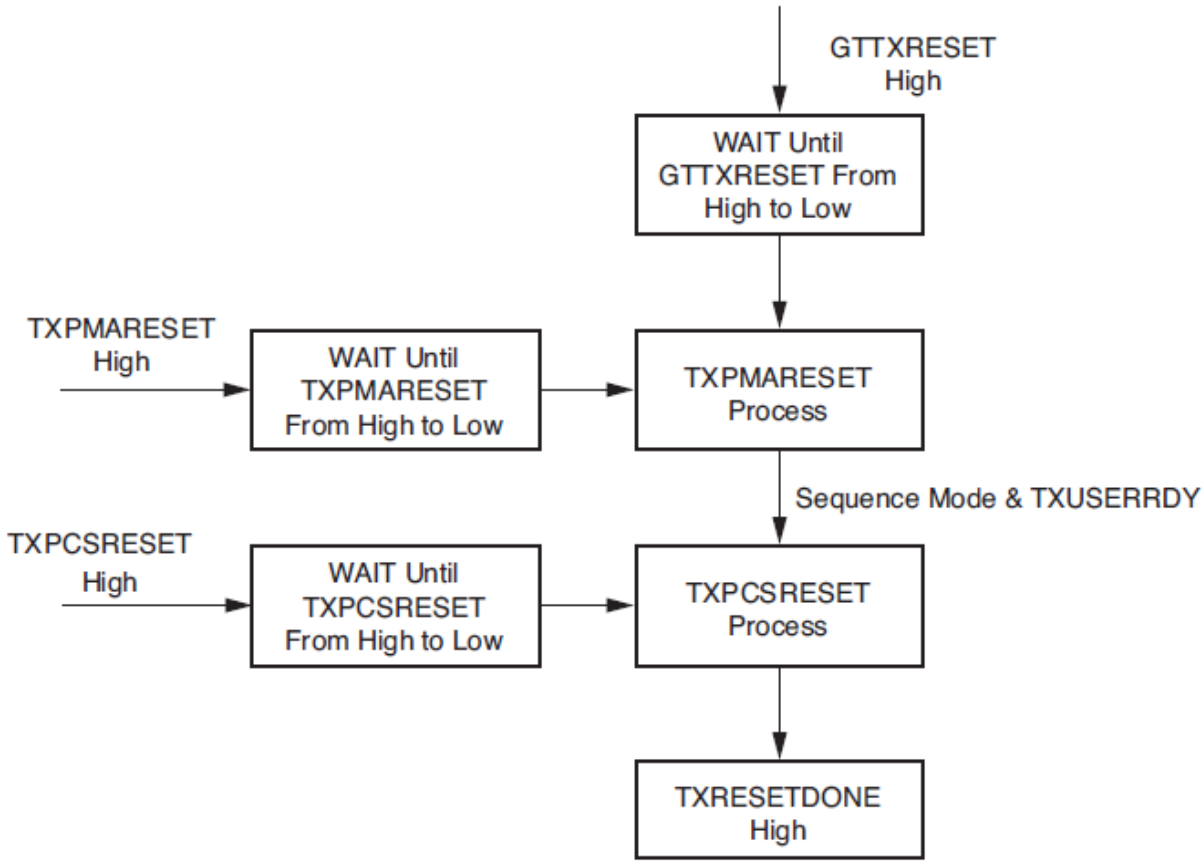
Για να χρησιμοποιηθούν οι transceivers απαιτείται μια διαδικασία αρχικοποίησης μετά την έναρξη λειτουργίας του FPGA. Ο πομπός TX και ο δέκτης RX μπορούν να αρχικοποιηθούν ανεξάρτητα και παράλληλα όπως φαίνεται στο παρακάτω σχήμα. Η διαδικασία αρχικοποίησης συνίσταται στα εξής δύο βήματα:

1. Αρχικοποίηση των αντίστοιχων PLL για την οδήγηση των TX/RX.
2. Αρχικοποίηση των μονοπατιών δεδομένων (datapaths).

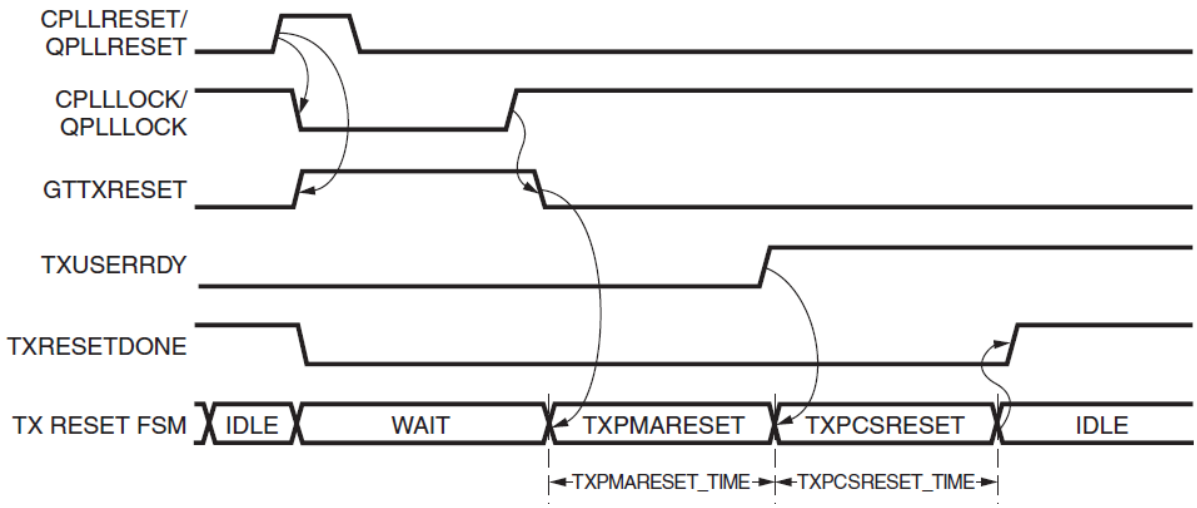


Για τον έλεγχο της διαδικασία αρχικοποίησης του TX χρησιμοποιείται μια μηχανή πεπερασμένων καταστάσεων, που φαίνεται στο παρακάτω σχήμα. Οι συνθήκες που πρέπει να ικανοποιηθούν είναι οι εξής:

1. Το σήμα GTRESETSEL πρέπει να είναι αρχικά χαμηλά μέχρι να κλειδώσει το PLL.
2. Το σήμα GTTXRESET πρέπει να χρησιμοποιηθεί.
3. Τα σήματα TXPMARESET και TXPCSRESET πρέπει να κρατηθούν χαμηλά κατά την διάρκεια της διαδικασίας μέχρι να ανιχνευτεί ότι το σήμα TXRESETDONE είναι ψηλά.



Το αντίστοιχο διάγραμμα χρονισμού για την αρχικοποίηση του TX είναι το εξής:



3.5 Διεπαφή FPGA TX

Η διεπαφή FPGA TX είναι η δίοδος για το μονοπάτι δεδομένων του TX. Οι εφαρμογές μεταδίδουν δεδομένα μέσω των πομπών γράφοντας δεδομένα στην πόρτα TXDATA στην

θετική ακμή του ρολογιού TXUSRCLK2. Το πλάτος της πόρτας ορίζεται στα 64bit. Η διεπαφή περιλαμβάνει δύο παράλληλα ρολόγια: τα TXUSRCLK και TXUSRCLK2. Ο απαιτούμενος ρυθμός του TXUSRCLK εξαρτάται από το πλάτος του εσωτερικού διαύλου δεδομένων (Internal Datapath Width), από τον ρυθμό μετάδοσης της γραμμής του trasmitter (Line Rate) και υπολογίζεται από την σχέση:

$$TXURSCLK\ Rate = \frac{Line\ Rate}{Internal\ Datapath\ Width}$$

Για Line Rate = 5.3Gbps και Internal Datapath Width = 32:

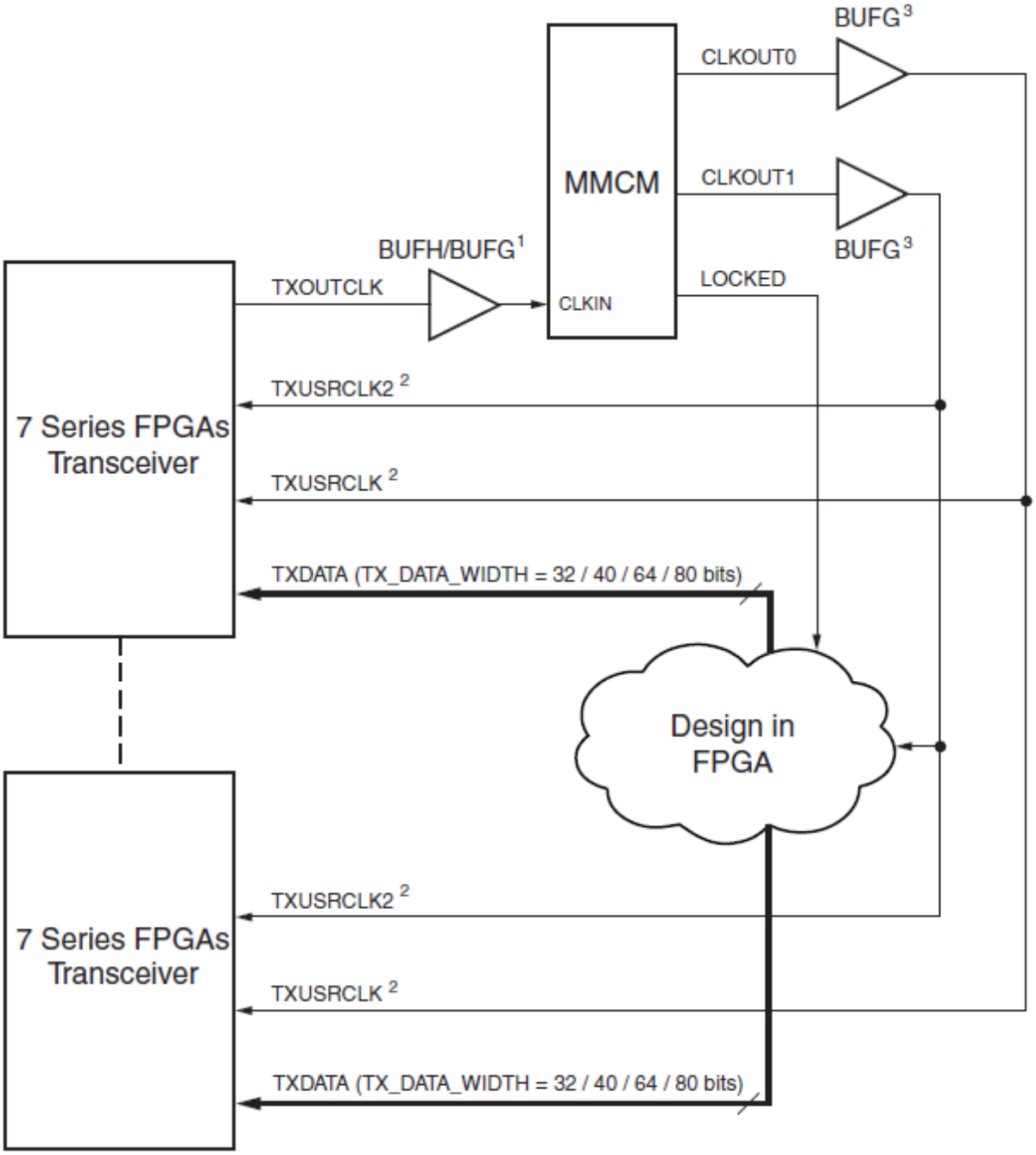
$$TXURSCLK\ Rate = \frac{5.3GHz}{32} = 165.625MHz$$

Το σήμα TXUSRCLK2 είναι το βασικό ρολόι συγχρονισμού για όλα τα σήματα της διεπαφής του transceiver. Για πλάτος δεδομένων 64bit (TX_DATA_WIDTH = 64) υπολογίζεται:

$$TXURSCLK2\ Rate = TXURSCLK\ Rate/2 = 82.8125MHz$$

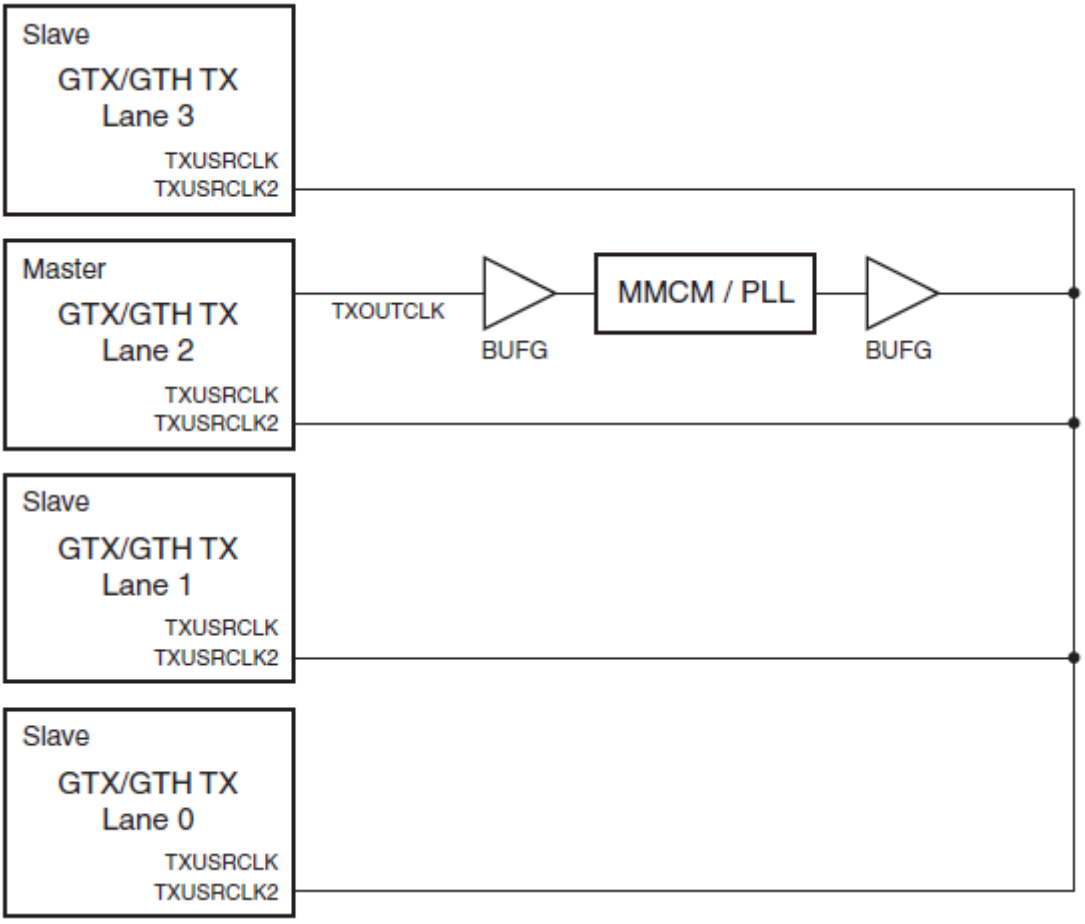
Τα TXUSRCLK και TXUSRCLK2 πρέπει να είναι ευθυγραμμισμένα, με όσο το δυνατό λιγότερο skew μεταξύ τους. Για τον λόγο αυτό οδηγούνται από τους ειδικούς απομονωτές BUFG (Global Buffer είναι απομονωτές για την διανομή σημάτων με μεγάλο fanout) και έχουν κοινή πηγή.

Στην συγκεκριμένη εφαρμογή όπου απαιτείται η χρήση πολλαπλών συγχρονισμένων γραμμών τα δύο αυτά ρολόγια παράγονται από μια μονάδα MMCM (Mixed-Mode Clock Manager Module) με ρολόι αναφοράς το TXOUTCLK της διεπαφής. Το παρακάτω σχήμα παρουσιάζει την διανομή των παραπάνω ρολογιών καθώς και του διαύλου δεδομένων TXDATA με πλάτος 64bit.

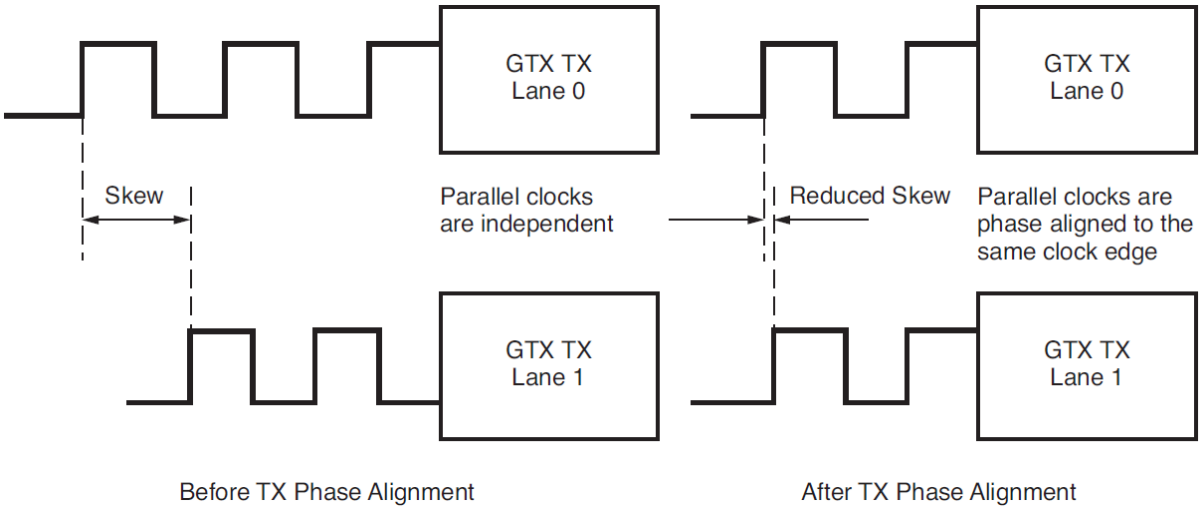


3.6 Ευθυγράμμιση Φάσης των Γραμμών Δεδομένων

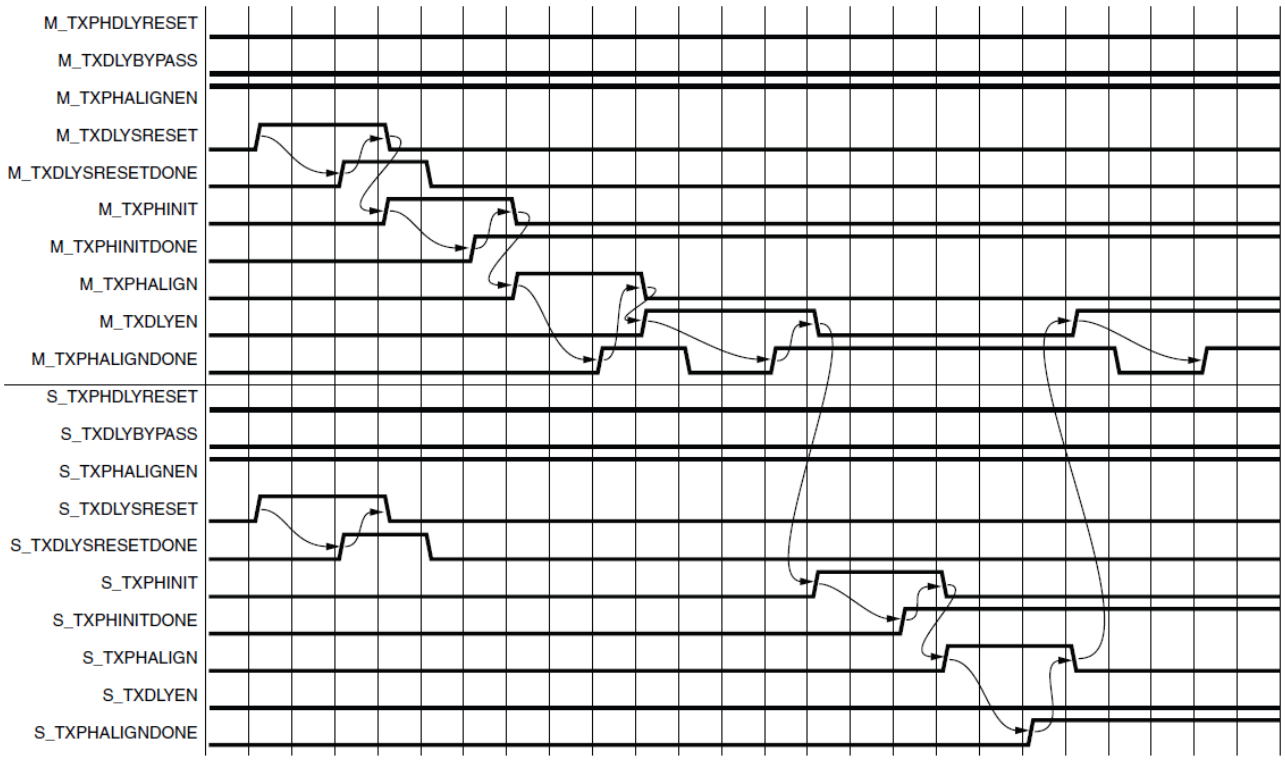
Σε αυτή την ενότητα περιγράφεται η διαδικασία που ακολουθείται για την ευθυγράμμιση – συγχρονισμό των γραμμών των πομπών. Σε εφαρμογές με πολλαπλές γραμμές μια γραμμή επιλέγεται ως master και είναι αυτή που παρέχει το ρολόι TXOUTCLK σε όλες τις slave γραμμές, όπως φαίνεται στο παρακάτω σχήμα:



Ακόμη για τον συγχρονισμό των γραμμών γίνεται χρήση του ειδικού κυκλώματος ευθυγράμμισης φάσης. Όπως φαίνεται στο παρακάτω σχήμα το κύκλωμα αυτό μπορεί να μειώσει το skew μεταξύ των γραμμών ευθυγραμμίζοντας τις περιοχές πολλαπλών ρολογιών (clock domains) σε ένα κοινό ρολόι. Το σχήμα δείχνει δύο γραμμές πομπών πριν και μετά την χρήση του κυκλώματος ευθυγράμμισης φάσης σε ένα κοινό ρολόι. Πριν την ευθυγράμμιση τα ρολόγια έχουν αυθαίρετες διαφορές φάσης. Μετά την ευθυγράμμιση, η μόνη διαφορά στην φάση οφείλεται στο skew του κοινού ρολογιού και όλες οι γραμμές μεταδίδουν ταυτόχρονα.



Για τον έλεγχο της αρχικοποίησης του κυκλώματος ευθυγράμμισης υλοποιείται FSM που διατρέχει μια ακολουθία καταστάσεων ελέγχοντας την ενεργοποίηση σημάτων έλεγχου μέχρις ότου φτάσει στην τελική κατάσταση στην οποίο παραμένει επ' αόριστον μέχρι να γίνει reset. Παρακάτω φαίνεται το διάγραμμα χρονισμού με τα βέλη να δείχνουν την ακολουθία των μεταβάσεων του FSM.



3.7 Ρύθμιση Φάσης Πομπού (TX Phase Interpolator)

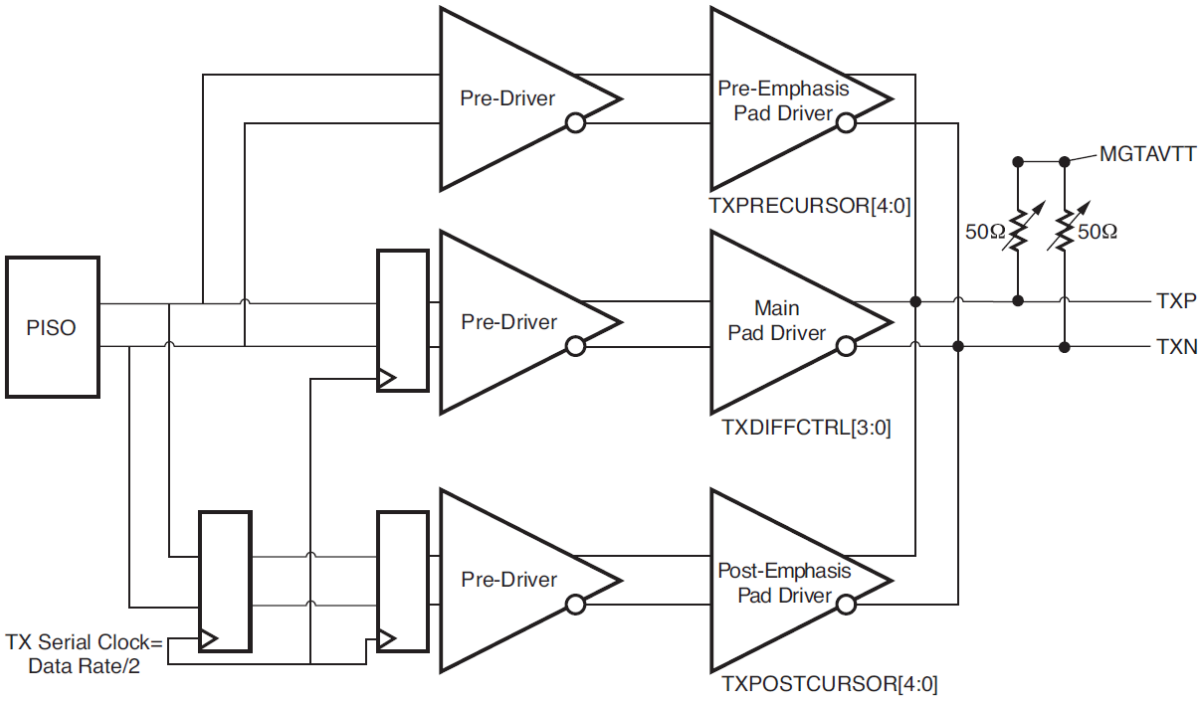
Πέρα από την ευθυγράμμιση της φάσης μεταξύ των γραμμών, η εφαρμογή παρέχει την δυνατότητα μικρορύθμισης της φάσης ξεχωριστά για κάθε γραμμή. Αυτή η δυνατότητα είναι ιδιαίτερη χρήσιμη στην περίπτωση που χρησιμοποιηθούν καλώδια με διαφορετικό μέγεθος ή αγώγιμα μονοπατια σε ένα PCB με διαφορετικό μήκος. Ο μονάδα ελέγχου του TX Phase Interpolator υποστηρίζει δυναμικό έλεγχο της φάσης με τον προσδιορισμό των εξής παραμέτρων:

- TXPIPPMEN: Όταν είναι στο λογικό 1 ενεργοποιείται ο ελεγκτής του TX Phase Interpolator επιτρέποντας την μεταβολή της φάσης.
- TXPIPPMSTEPSIZE[4:0]: Το πιο σημαντικό bit TXPIPPMSTEPSIZE[4] καθορίζει την αύξηση ή την μείωση της φάσης. Τα υπόλοιπα τέσσερα bit καθορίζουν το πόσο θα μεταβληθεί η φάση.

3.8 Ρύθμιση Επιπέδου Τάσης Πομπού (TX Configurable Driver)

Ο οδηγός του TX είναι ένας υψηλής ταχύτητας διαφορικός απομονωτής. Για την εξασφάλιση της ακεραιότητας των δεδομένων περιλαμβάνει τα εξής χαρακτηριστικά:

- Διαφορικό Έλεγχο της τάσης
- Pre-cursor and post-cursor transmit pre-emphasis
- Ρυθμιζόμενους τερματικούς αντιστάτες.



Ο διαφορικός έλεγχος της τάσης γίνεται δίνοντας παρέχοντας κατάλληλες τιμές στην θύρα TXDIFFCTRL[3:0]. Στον παρακάτω πίνακα φαίνεται η αντιστοιχία μεταξύ των τιμών του σήματος και της διαφορικής τάσης από κορυφή σε κορυφή:

[3:0]	V_{PPD}
4'b0000	0.269
4'b0001	0.336
4'b0010	0.407
4'b0011	0.474
4'b0100	0.543
4'b0101	0.609
4'b0110	0.677
4'b0111	0.741
4'b1000	0.807
4'b1001	0.866
4'b1010	0.924
4'b1011	0.973
4'b1100	1.018
4'b1101	1.056
4'b1110	1.092
4'b1111	1.119

4

Υλοποίηση της πλατφόρμας ελέγχου σε FPGA

Εδώ περιγράφεται το κύριο κομμάτι της διπλωματικής εργασίας, που είναι η σχεδίαση και υλοποίηση των υποσυστημάτων που αποτελούν την πλατφόρμα ελέγχου.

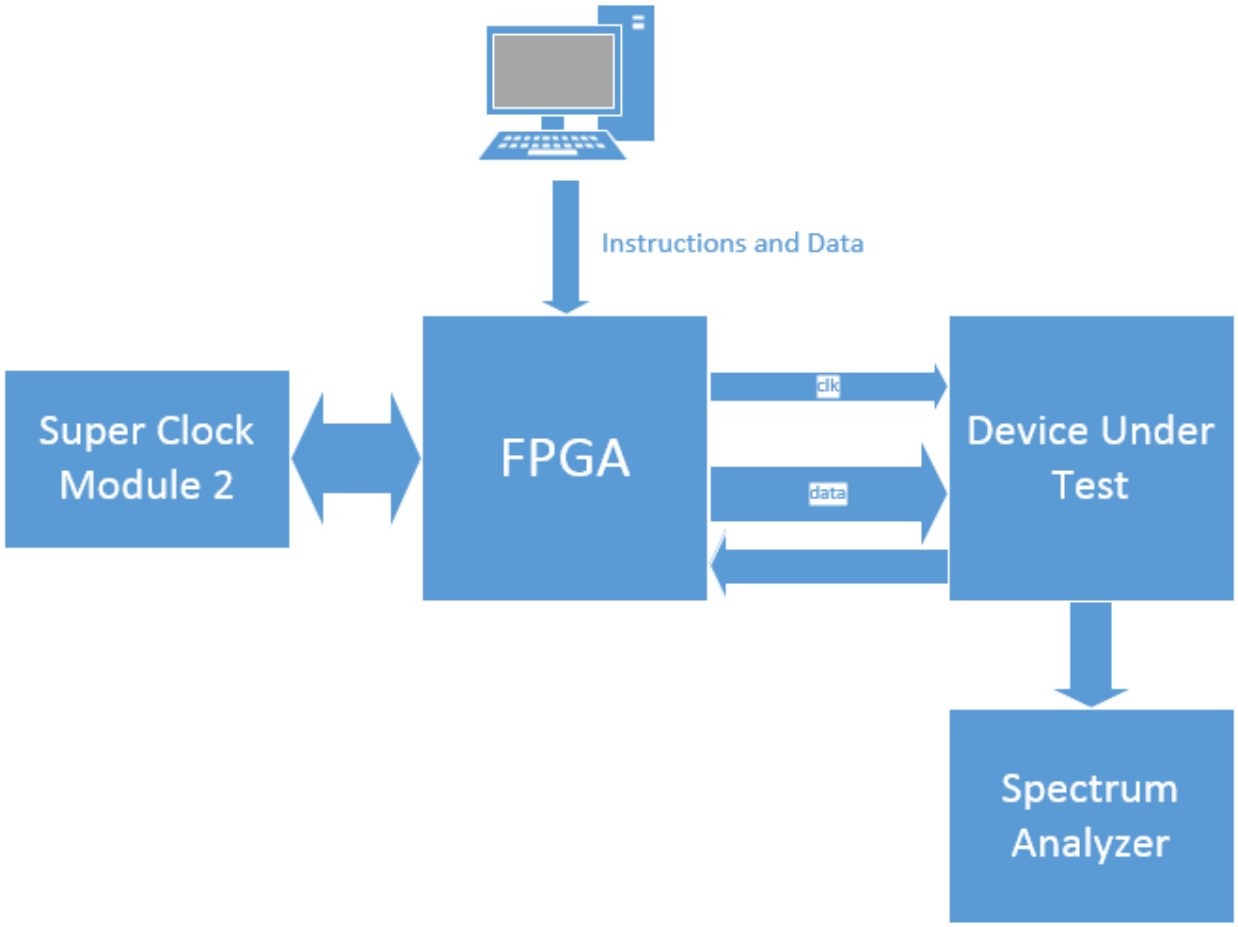
4.1 Περισκόπηση Συστήματος

Ο στόχος του project είναι η κατασκευή πλατφόρμας για των έλεγχο και την επαλήθευση της ορθής λειτουργίας μιας οικογένειας RF διαμορφωτών ASIC παρέχοντας δεδομένα με ρυθμό 2.65GHz. Η πλατφόρμα δέχεται εντολές και δεδομένα από ένα host PC μέσω MATLAB, αποθηκεύει τα δεδομένα στην εσωτερική μπλοκ μνήμη του FPGA. Έπειτα τα δεδομένα αναπαράγονται παράλληλα και συγχρονισμένα μέσω των πολλαπλών σειριακών καναλιών υψηλής ταχύτητας που διαθέτει το αναπτυξιακό. Ακόμη με τις εντολές μπορούν να ρυθμιστούν οι εξής παράμετροι:

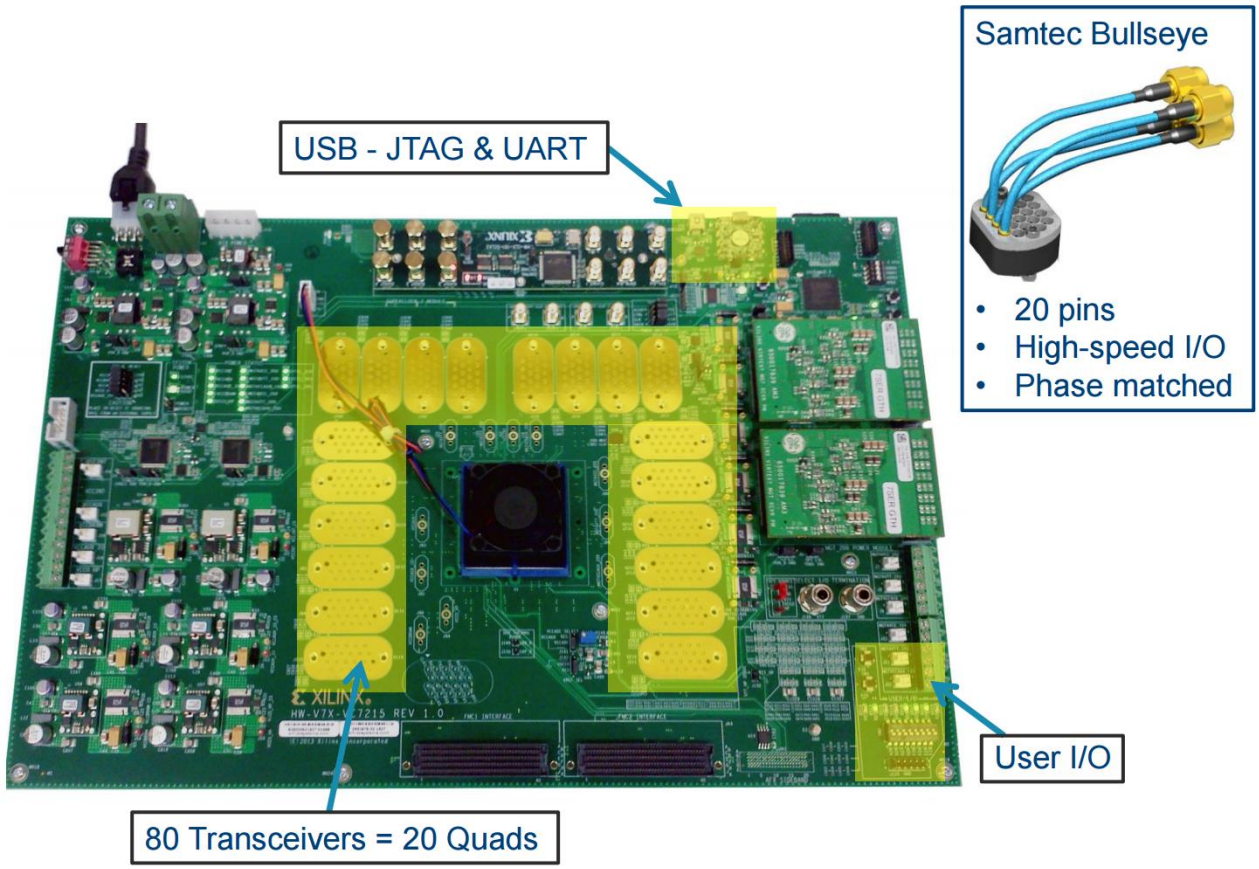
1. Πλήθος Καναλιών που θα φορτωθούν με δεδομένα
2. Πλήθος Δειγμάτων που θα αποθηκευτούν.
3. Πλήθος Δειγμάτων που θα αναπαραχθούν και θα διαβαστούν.
4. Επίπεδο τάσης των πομπών.
5. Ολίσθηση Φάσης σε κάθε κανάλι ξεχωριστά.

Το σύστημα έχει την δυνατότητα να λειτουργήσει σε κανονική λειτουργία (normal mode) και σε λειτουργία ελέγχου (testing mode). Στην κανονική λειτουργία τα δεδομένα αφού σταλούν από τον υπολογιστή και αποθηκευτούν στο FPGA αναπαράγονται κυκλικά. Στην λειτουργία

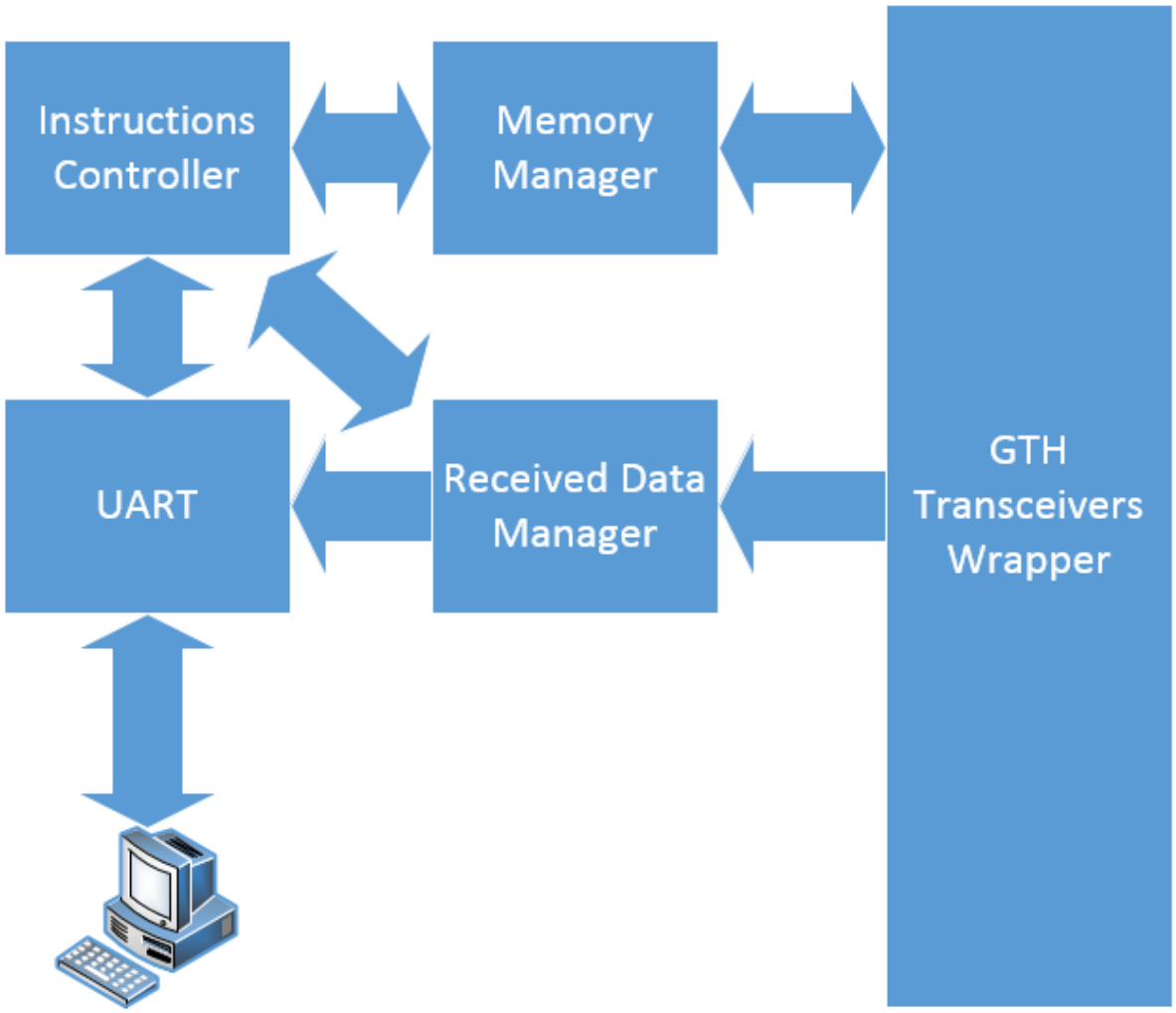
ελέγχου τα δεδομένα αναπαράγονται μια μόνο φορά, ενώ ταυτόχρονα αποθηκεύονται τα δεδομένα που λαμβάνει ο δέκτης σε μια ξεχωριστή μνήμη. Όταν τελειώσει η αναπαραγωγή των δεδομένων και ταυτόχρονα η λήψη δεδομένων από τον δέκτη, τα δεδομένα λήψης στέλνονται από το FPGA στον υπολογιστή προκειμένου να εφαρμοστεί η μεθοδολογία scan chain και να επαληθευθεί η ορθή λειτουργία της συσκευής υπό έλεγχο (Device Under Test).



Στην παρακάτω εικόνα φαίνεται το αναπτυξιακό VC7215. Στο κέντρο του αναπτυξιακού βρίσκεται το FPGA XC7VX690T FFG1927-3. Στο πάνω μέρος βρίσκεται η υπομονάδα SuperClock2-Module, που παρέχει τα δύο εξωτερικά ρολόγια αναφοράς στους πομποδέκτες υψηλών ταχυτήτων. Η υπομονάδα αυτή συνδέεται με το FPGA για τον προγραμματισμό του ολοκληρωμένου Si5368 (Clock Multiplier/ Jitter Attenuator). Στα δεξιά απεικονίζονται τα καλώδια που συνδέονται με τους MGTs. Κάθε QUAD αντιστοιχεί σε μια υποδοχή Bullseye.



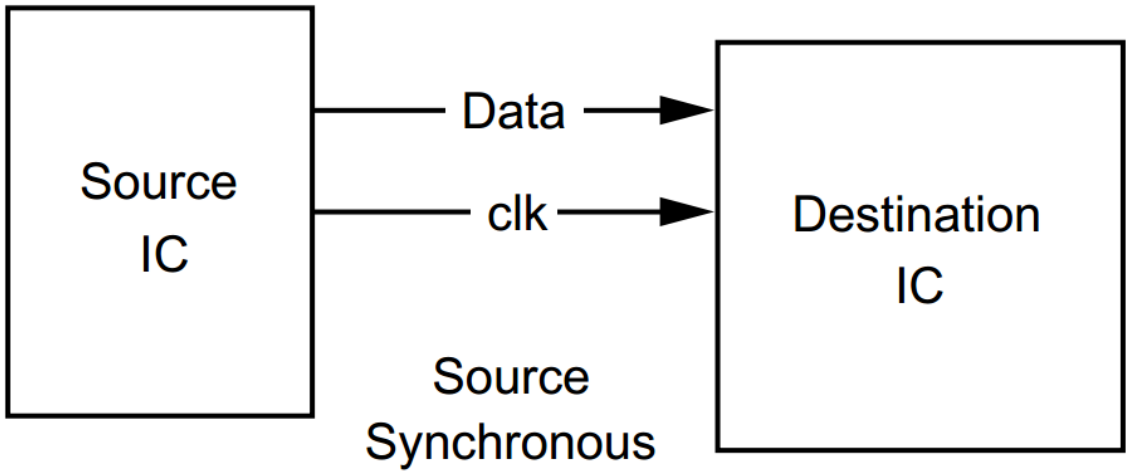
4.2 Μπλοκ Διαγραμμα Ανώτατου Επιπέδου (Top-Level)



4.3 Επικοινωνία με Σύγχρονη Πηγή (Source Synchronous)

Η επικοινωνία μεταξύ δύο ολοκληρωμένων κυκλωμάτων όπου το εκπέμπον IC δημιουργεί ένα ρολόι που συνοδεύει τα δεδομένα ονομάζεται επικοινωνία με σύγχρονη πηγή. Το ολοκληρωμένο που λαμβάνει χρησιμοποιεί αυτό το ρολόι για την λήψη των δεδομένων. Σε κάθε θετική ακμή του ρολογιού, ένα νέο δείγμα λαμβάνεται από την συσκευή υπό έλεγχο.

Στην παρούσα εργασία μια από τις γραμμές δεδομένων (κανάλι 0) χρησιμοποιείται για την δημιουργία ρολογιού και μεταβίβασή του στην συσκευή υπό έλεγχο. Για τον σκοπό αυτή η γραμμή λειτουργεί σε διπλάσια συχνότητα από τις υπόλοιπες γραμμές δεδομένων και σε αυτήν φορτώνεται μια σταθερά επαναλαμβανόμενη ακολουθία από 0101.

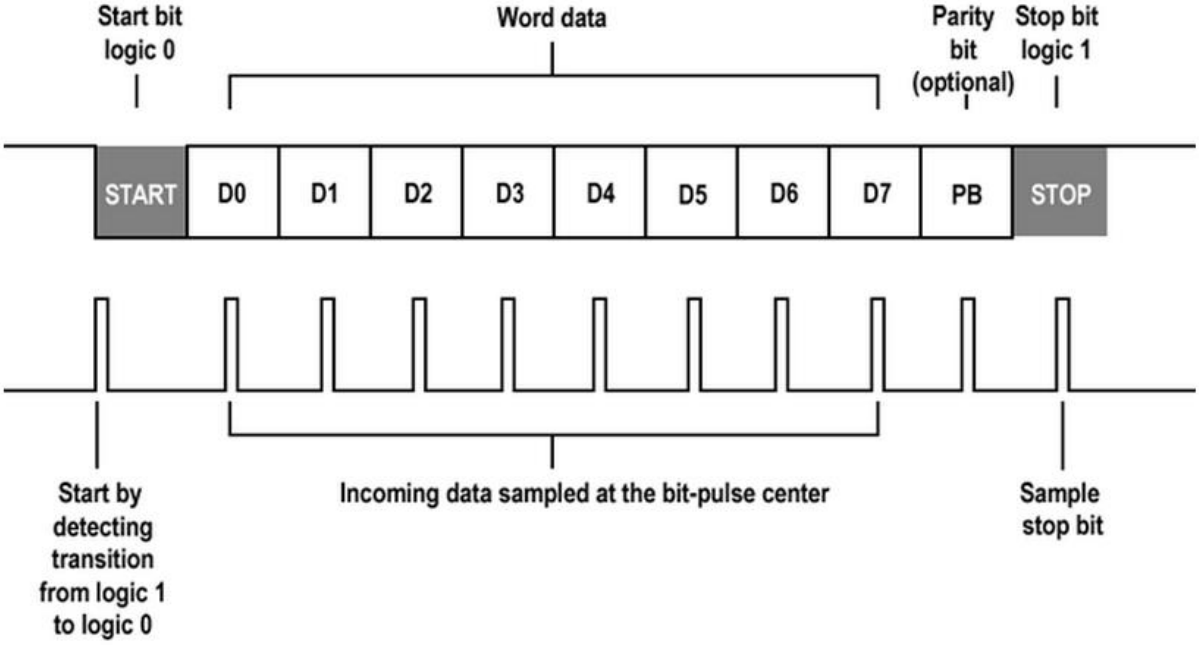


4.4 UART Πομποδέκτης

UART είναι η συντομογραφία του universal asynchronous receiver/transmitter και είναι ένα κύκλωμα που στέλνει παράλληλα δεδομένα μέσω μιας σειριακής γραμμής. Το αναπτυξιακό VC7215 διαθέτει γέφυρα USB-to-UART (CP2103) το οποίο παρέχει την δυνατότητα σειριακής επικοινωνίας μεταξύ ενός υπολογιστή και του FPGA μέσω ενός USB καλωδίου. Το FPGA υποστηρίζει την γέφυρα USB-to-UART με τα παρακάτω τέσσερα σήματα:

- Αποστολή (TX)
- Λήψη (RX)
- Αίτημα για αποστολή (RTS)
- Έτοιμο για αποστολή (CTS)

Η UART περιλαμβάνει πομπό και δέκτη. Ο πομπός είναι στην ουσία ένας καταχωρητής ολίσθησης ο οποίος φορτώνει παράλληλα δεδομένα και τα ολισθαίνει προς τα έξω ανά bit με έναν ορισμένο ρυθμό. Από την άλλη πλευρά ο δέκτης, ολισθαίνει τα δεδομένα ανά bit και τα συλλέγει σε μια λέξη δεδομένων. Η σειριακή γραμμή είναι αδρανής όταν είναι στο λογικό 1. Η μετάδοση αρχίζει με το bit εκκίνησης, το οποίο είναι 0, ακολουθούμενο από τα bit δεδομένων και ένα προαιρετικό bit ισοτιμίας, και λήγει με ένα bit λήξης, το οποίο είναι 1. Ο αριθμός των bit δεδομένων μπορεί να είναι 6,7 ή 8. Το προαιρετικό bit ισοτιμίας χρησιμοποιείται για ανίχνευση λαθών.



Μέσω της σειριακής γραμμής δεν μεταφέρεται πληροφορία για το ρολόι, οπότε πριν την έναρξη της μετάδοσης, ο πομπός και ο δέκτης πρέπει να έχουν προσυμφωνήσει σε ένα σύνολο παραμέτρων οι οποίες περιλαμβάνουν τον ρυθμό μετάδοσης (baud rate σε bit ανά δευτερόλεπτο) , το πλήθος των bit δεδομένων και των bit λήξης, την χρήση ή όχι bit ισοτιμίας. Στην παρούσα εφαρμογή χρησιμοποιήθηκε ο μέγιστος ρυθμός μετάδοσης που υποστηρίζεται από το CP2103, που είναι 921600bps. Ακόμη ορίστηκαν 8 bit δεδομένων και ένα bit λήξης, χωρίς bit ισοτιμίας.

4.4.1 Υποσύστημα Λήψης UART

Αφού δεν μεταδίδεται πληροφορία για τον συγχρονισμό των δεδομένων, ο δέκτης λαμβάνει τα bit δεδομένων μόνο με βάση τις προκαθορισμένες παραμέτρους. Εδώ χρησιμοποιείται ένα σχήμα υπερδειγματοληψίας για τον προσεγγιστικό υπολογισμό του μεσαίου σημείου των μεταδιδόμενων bits.

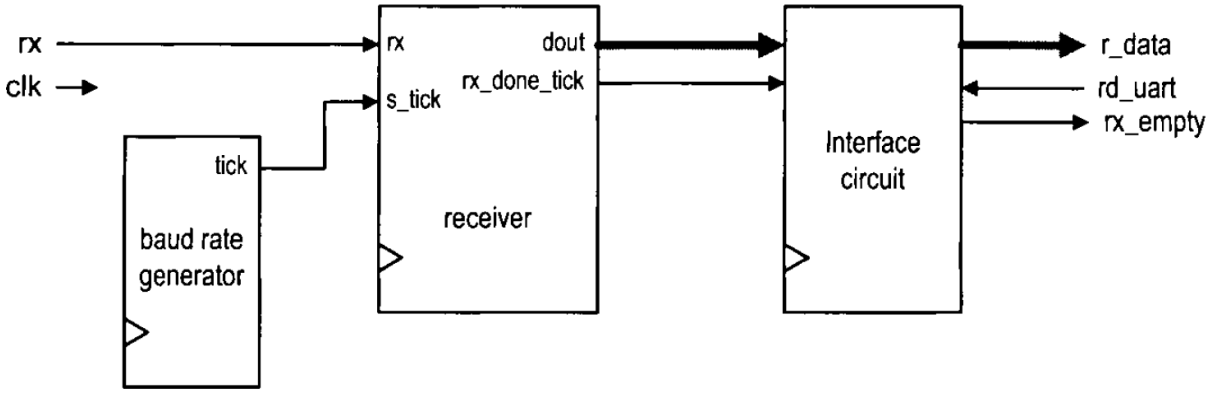
4.4.1.1 Διαδικασία Υπερδειγματοληψίας

Ο πιο κοινός ρυθμός δειγματοληψίας είναι 16 φορές ο ρυθμός μετάδοσης (baud rate), το οποίο σημαίνει ότι κάθε bit δειγματοληπτείται 16 φορές. Υποθέτουμε ότι η επικοινωνία απαιτεί N bit δεδομένων και M bit λήξης. Το σχήμα υπερδειγματοληψίας ακολουθεί τα παρακάτω βήματα:

1. Περιμένει μέχρι το σήμα εισόδου (RX) γίνει 0, το οποίο σηματοδοτεί την έναρξη του bit εκκίνησης, στο σημείο αυτό ξεκινάει ο μετρητής δειγματοληψίας.
2. Όταν ο μετρητής φτάσει στο 7, το εισερχόμενο σήμα έχει φτάσει στο μεσαίο σημείο του bit εκκίνησης. Μηδενίζει ο αθροιστής και επανεκκινεί.
3. Όταν ο μετρητής φτάσει στο 15, το εισερχόμενο σήμα έχει φτάσει στην μέση του πρώτου bit δεδομένων. Συλλέγεται η τιμή του και ολισθαίνει στον καταχωρητή ολίσθησης κατά μια θέση. Ο μετρητής επανεκκινεί.
4. Το βήμα 3 επαναλαμβάνεται για N-1 φορές, μέχρι να ανακτηθούν όλα τα bit δεδομένων.
5. Το βήμα 3 επαναλαμβάνεται για M φορές, για τα M bit λήξης.

Το σχήμα υπερδειγματοληψία στην ουσία εκτελεί την λειτουργία ενός σήματος ρολογιού. Αντί να χρησιμοποιείται η θετική ακμή για να σηματοδοτήσει πότε ένα σήμα είναι έγκυρο, χρησιμοποιεί ticks δειγματοληψίας για να προσεγγίσει το μεσαίο σημείο του κάθε bit. Καθώς ο δέκτης δεν έχει ακριβή πληροφορία για την ακριβή χρονική στιγμή του bit εκκίνησης, η προσέγγιση αυτή μπορεί να είναι το πολύ 1/16 άστοχη. Τα ακόλουθα bit μπορεί επίσης να είναι το πολύ 1/16 μακριά από το μεσαίο σημείο. Στο παρακάτω block διάγραμμα φαίνονται τα τρία υποσυστήματα που συνθέτουν τον δέκτη UART:

- **UART receiver**: κύκλωμα που ανακτά μια 8bit λέξη δεδομένων.
- **Baud Rate Generator**: κύκλωμα που δημιουργεί τον επιθυμητό ρυθμό μετάδοσης.
- **Interface circuit**: προαιρετικό κύκλωμα διεπαφής με το σύστημα που χρησιμοποιεί την UART. Παρέχει έναν απομονωτή και την κατάσταση του **UART receiver.**



4.4.1.2 Γεννήτρια Ρυθμού Μετάδοσης (Baud Rate Generator)

Ο Baud Rate Generator δημιουργεί ένα σήμα δειγματοληψίας του οποίου η συχνότητα είναι ακριβώς 16 φορές ο ονομαστικός ρυθμός μετάδοσης. Προκειμένου να μην δημιουργήσουμε μια νέα περιοχή ρολογιού (clock domain), παραβιάζοντας την αρχή της σύγχρονης σχεδίασης, το σήμα δειγματοληψίας λειτουργεί περισσότερο ως σήμα επίτρεψης για τον UART receiver παρά ως ένα σήμα ρολογιού.

Για ρυθμό μετάδοσης 921600bps, ο ρυθμός δειγματοληψίας είναι:

$$921600 * 16 = 14745600$$

Το ρολόι συστήματος έχει συχνότητα 200MHz. Επομένως ο baud rate generator μπορεί να υλοποιηθεί με ένα mod-14 μετρητή, καθώς

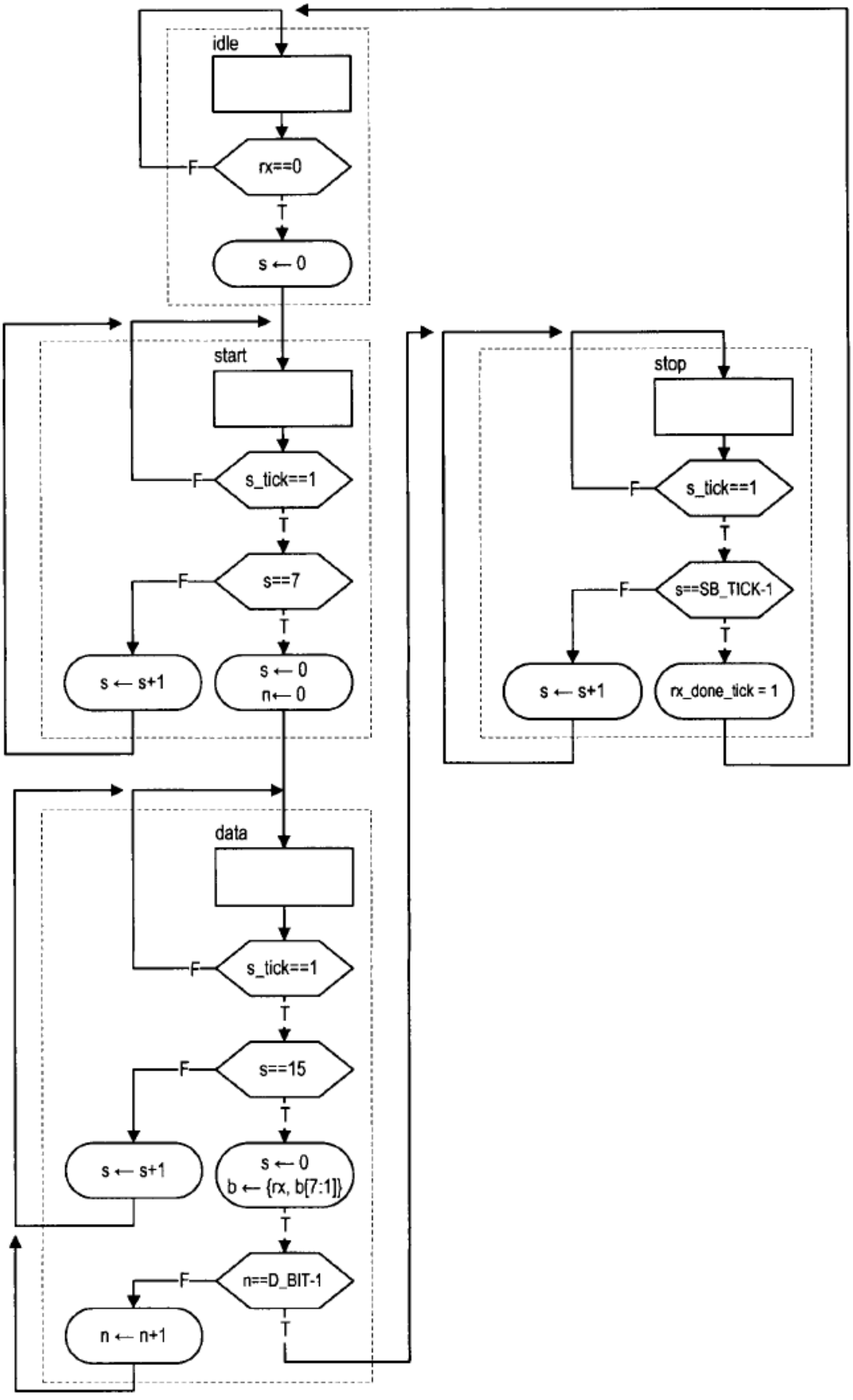
$$\frac{200 * 10^6}{14745600} = 13.56$$

στον οποίο ενεργοποιείται ένα παλμός πλάτους ενός κύκλου κάθε 14 κύκλους ρολογιού.

4.4.1.3 Δέκτης UART (UART Receiver)

Με βάση το σχήμα υπερδειγματοληψίας που περιγράφηκε παραπάνω, κατασκευάζεται το διάγραμμα αλγοριθμικής μηχανής καταστάσεων με μονοπάτι δεδομένων (ASMD algorithmic state machine diagramm).

Το ASMD έχει τρεις βασικές καταστάσεις: start, data και stop οι οποίες αναπαριστούν την επεξεργασία του bit έναρξης, των bit δεδομένων και του bit λήξης αντίστοιχα. Το σήμα s_tick είναι ο παλμός επίτρεψης της γεννήτριας του ρυθμού μετάδοσης (baud rate generator) και υπάρχουν 16 παλμοί στο διάστημα ενός bit. Υπάρχουν δύο μετρητές για την αρίθμηση των παλμών και την αρίθμηση των bit δεδομένων. Τα bit της εισόδου ολισθαίνουν στον καταχωρητή ολίσθησης b. Τέλος, συμπεριλαμβάνεται ένα σήμα κατάστασης, rx_done_tick, το οποίο τίθεται στο λογικό 1 για ένα κύκλο ρολογιού κάθε φορά που η διαδικασία ολοκληρώνεται.


idle
rx==0
F
T
s ← 0
start
s_tick==1
F
T
s==7
F
T
s ← s+1
s ← 0
n ← 0
data
s_tick==1
F
T
s==15
F
T
s ← s+1
s ← 0
b ← {rx, b[7:1]}
n==D_BIT-1
F
T
n ← n+1
stop
s_tick==1
F
T
s==SB_TICK-1
F
T
s ← s+1
rx_done_tick = 1

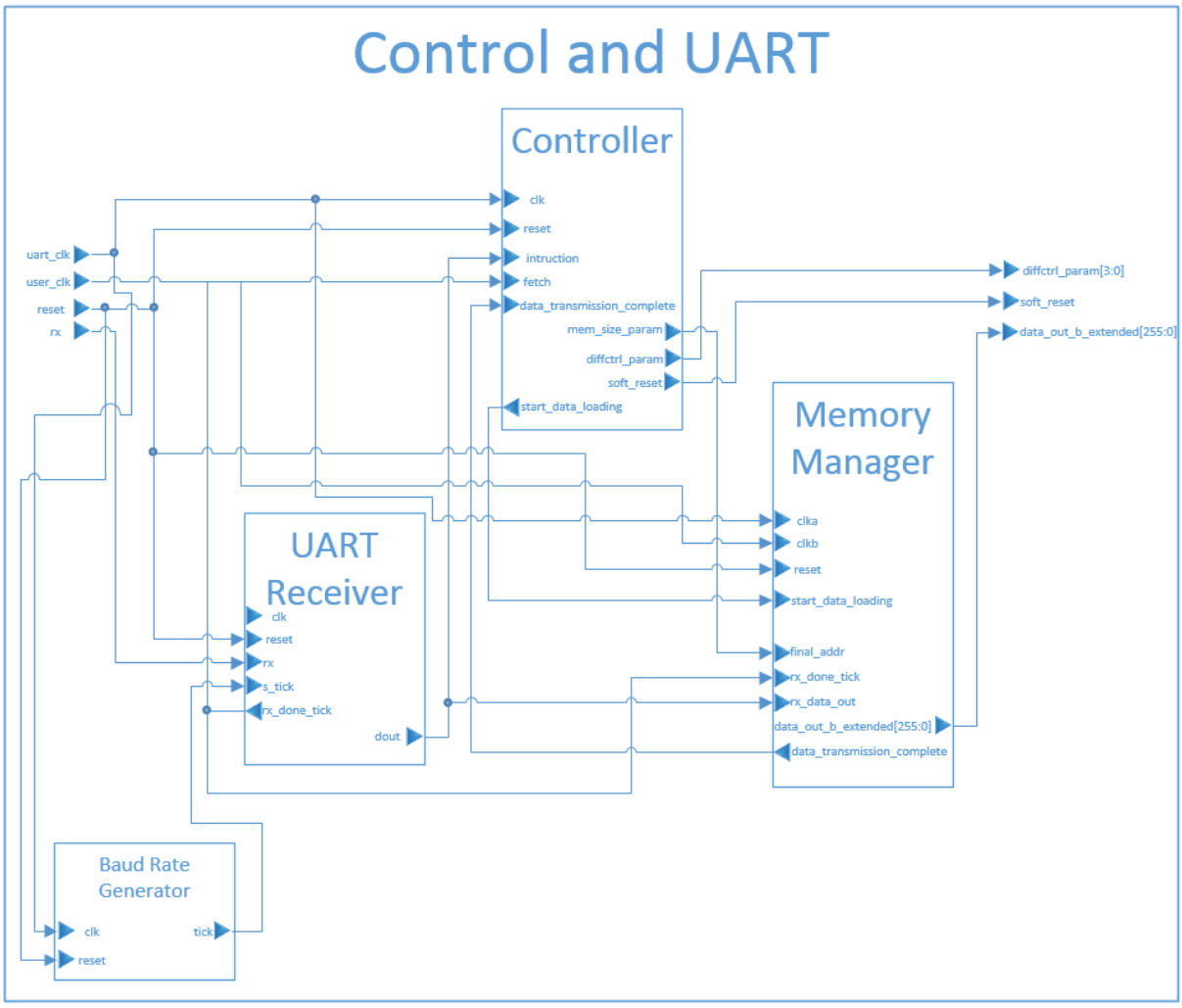
4.4.2 Υποσύστημα Εκπομπής UART

Η οργάνωση του υποσυστήματος εκπομπής UART είναι παρόμοιο με αυτό της λήψης. Αποτελείται από έναν πομπό UART, γεννήτρια ρυθμού μετάδοσης και ένα κύκλωμα διεπαφής. Ο πομπός είναι στην ουσία ένας καταχωρητής ολίσθησης ο οποίος ολισθαίνει προς τα έξω τα δεδομένα με ένα συγκεκριμένο ρυθμό. Ο ρυθμός αυτός δημιουργείται από την γεννήτρια ρυθμού μετάδοσης. Ένα bit ολισθαίνει προς τα έξω κάθε 14 παλμούς επίτρεψης. Το διάγραμμα της αλγοριθμικής μηχανής καταστάσεων είναι παρόμοιο με αυτό του δέκτη. Αφού το σήμα tx_start τεθεί στο λογικό 1, το FSMD φορτώνει την λέξη δεδομένων και σταδιακά διατρέχει τις καταστάσεις start, data και stop ολισθαίνοντας τα αντίστοιχα bits. Το σήμα tx_done_tick σηματοδοτεί την ολοκλήρωση της εκπομπής.

4.5 Υποσύστημα Φόρτωσης Δεδομένων και Ρύθμισης Παραμέτρων

Ο ελεγκτής εντολών είναι το κύκλωμα που κατευθύνει τις εντολές του συστήματος. Κατά κύριο λόγο απευθύνεται στον διαχειριστή μνήμης (Memory Manager) για την φόρτωση των δεδομένων στη μνήμη και στους transceivers για την ρύθμιση παραμέτρων και κάτα δεύτερο λόγο στο υποσύστημα συλλογής και αποστολής δεδομένων εισόδου. Στο παρακάτω block διάγραμμα φαίνεται η διασύνδεση του ελεγκτή, του υποσυστήματος λήψης UART και του διαχειριστή μνήμης.

Ο ελεγκτής δέχεται εντολές και δεδομένα από τον UART Receiver και είτε ενημερώνει το περιεχόμενο των καταχωρητών για την ρύθμιση των παραμέτρων των transceivers είτε προωθεί τα δεδομένα στον διαχειριστή μνήμης.



4.5.1 Ελεγκτής Εντολών (Instruction Controller)

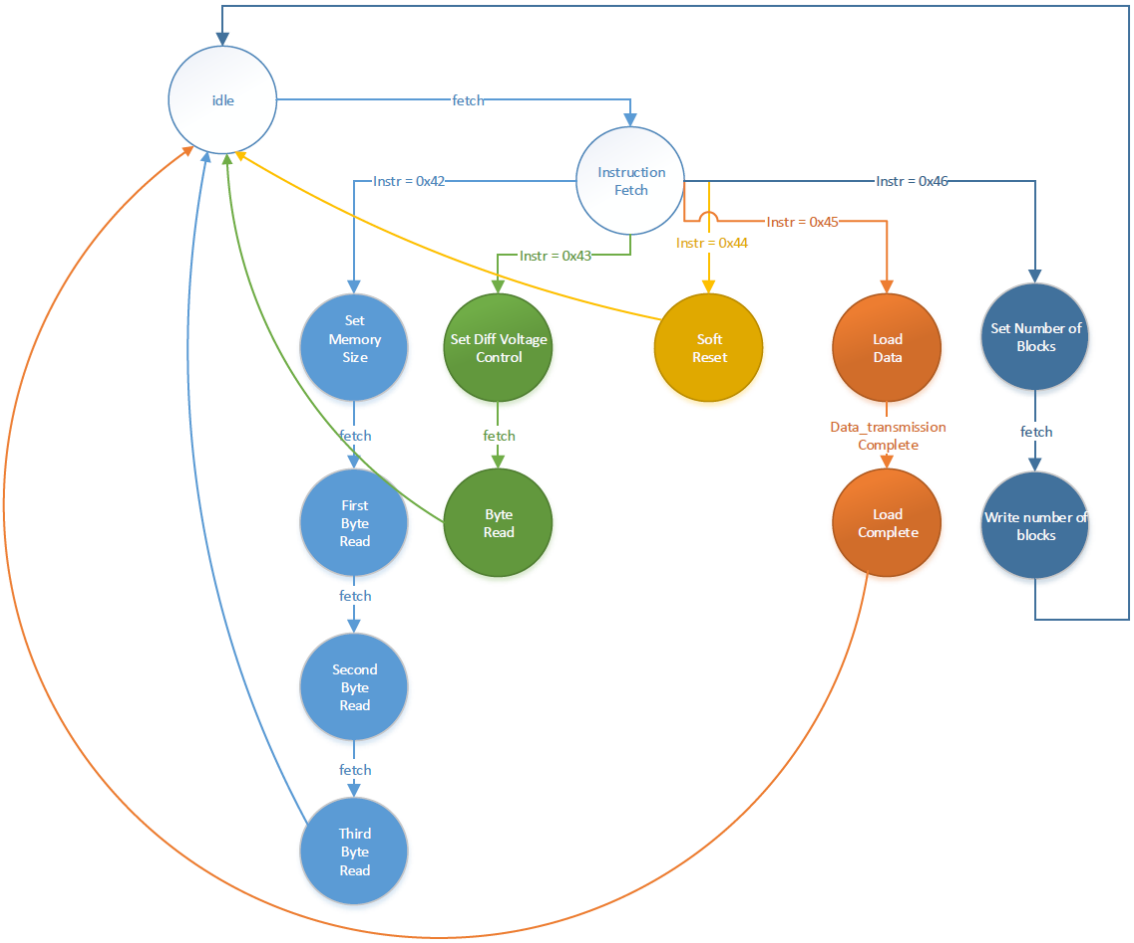
Το σύνολο εντολών που δέχεται ο ελεγκτής είναι το εξής:

1. Set_Memory_Size
2. Set_Final_Address_Read
3. Enable_Phase_Interpolator
4. Set_Driver_Swing_Control
5. Set_Number_of_Blocks
6. Set_Soft_Reset
7. Load_Data
8. Set_Test_Mode_ON
9. Set_Test_Mode_OFF

Οι εντολές είναι μεταβλητού μεγέθους, 1 έως 4 byte. Όλες έχουν opcode των 8bit το οποίο προσδιορίζει την εντολή, ακολουθούμενο από προαιρετικά byte που ορίζουν την τιμή των παραμέτρων:

Εντολή	Opcode	1ο Byte	2ο Byte	3ο Byte
Set_Memory_Size	0x42	LSB Address	Middle Byte Address	MSB Address
Set_Final_Address_Read	0x48	LSB Address	Middle Byte Address	MSB Address
Enable_Phase_Interpolator	0x47	Number of Transceiver	txpippmsize	
Set_Driver_Swing_Control	0x43	TXDIFFCTRL		
Set_Number_of_Blocks	0x46	Number of Blocks		
Set_Soft_Reset	0x44			
Load_Data	0x45			
Set_Test_Mode_ON	0x49			
Set_Test_Mode_OFF	0x4a			

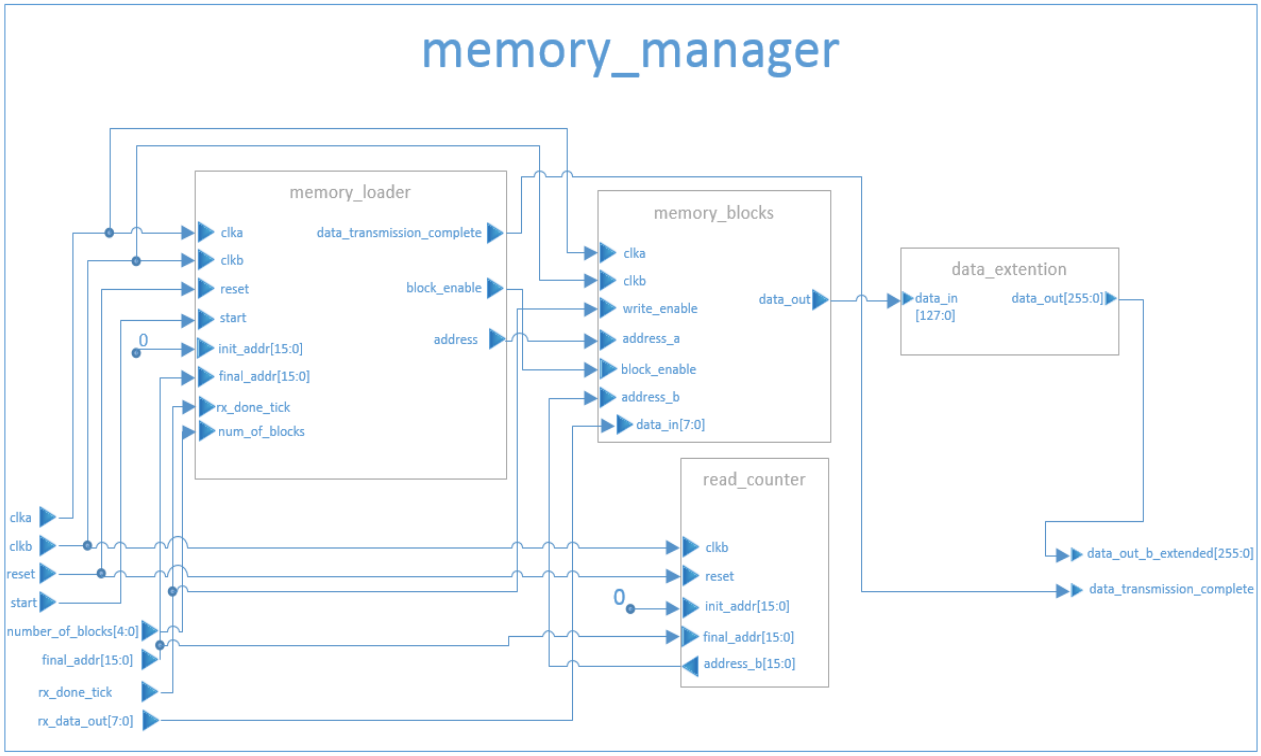
Ο ελεγκτής υλοποιείται με την παρακάτω μηχανή πεπερασμένων καταστάσεων:



4.5.2 Διαχείριση Μνήμης

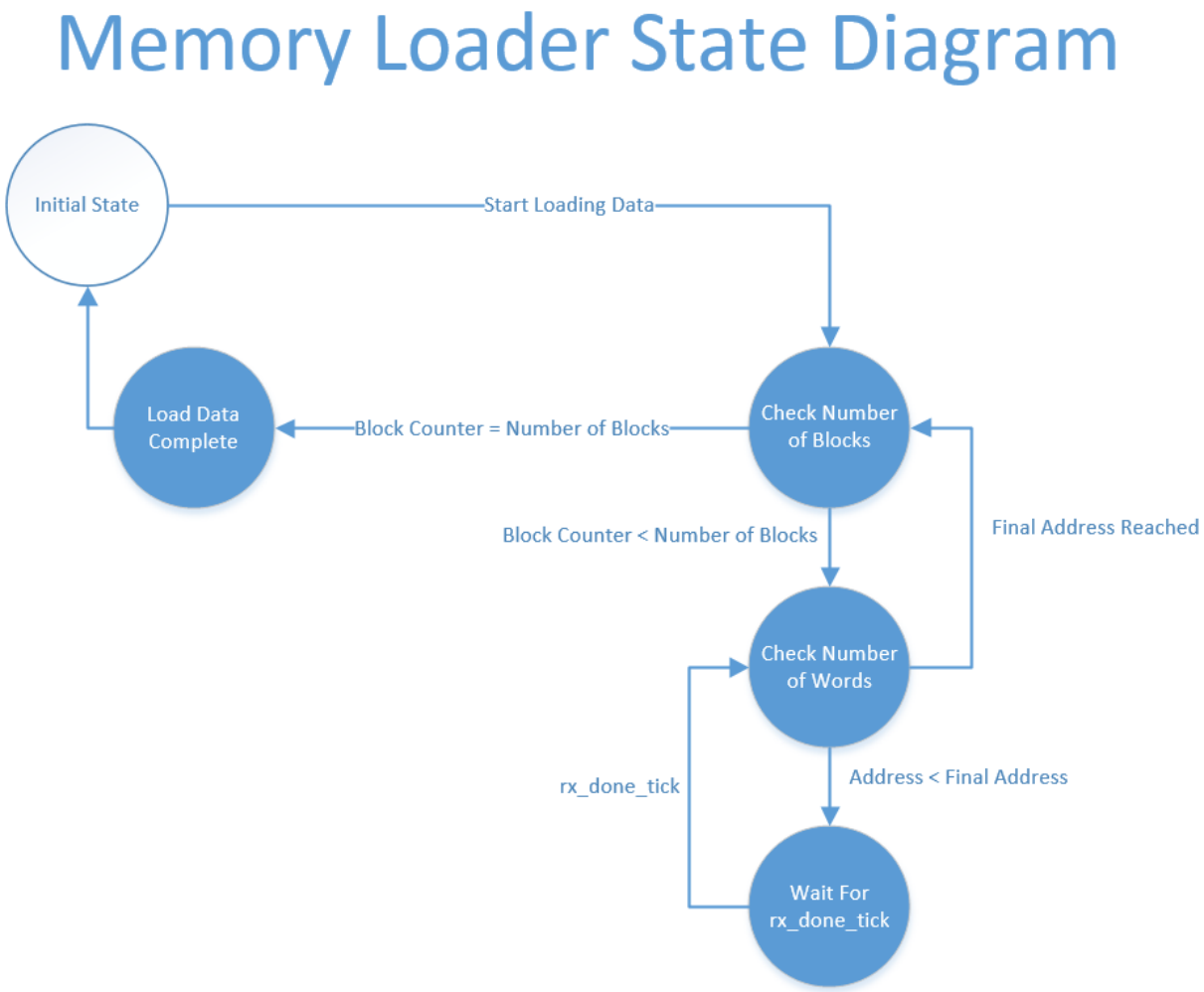
Ο διαχειριστής μνήμης αποτελείται από τις εξής τέσσερις οντότητες:

1. Φορτωτής Δεδομένων
2. Μπλοκ Μνήμης
3. Μετρητής Ανάγνωσης
4. Επέκταση Δεδομένων



4.5.2.1 Φορτωτής Δεδομένων (memory_loader)

Ο φορτωτής δεδομένων αναλαμβάνει να φορτώσει με δεδομένα την μνήμη λαμβάνοντας υπόψη τον αριθμό των μπλοκ μνήμης που θα φορτωθούν και το πλήθος των δεδομένων (ή την τελική διεύθυνση μνήμης). Όταν δοθεί λογικό 1 στο σήμα εκκίνησης φόρτωσης **start** φορτώνεται κάθε μπλοκ μνήμης με τη σειρά αυξάνοντας την τιμή της διεύθυνσης **address** κατά 1, από το 0 μέχρι την δοσμένη τελική διεύθυνση **final_address**. Αυτό επιτυγχάνεται ενεργοποιώντας διαφορετικό μπλοκ μνήμης κάθε φορά με το σήμα **block_enable**, το οποίο λειτουργεί ως one-hot αποκωδικοποιητής. Ο φορτωτής δεδομένων υλοποιείται με βάση την παρακάτω μηχανή πεπερασμένων καταστάσεων:



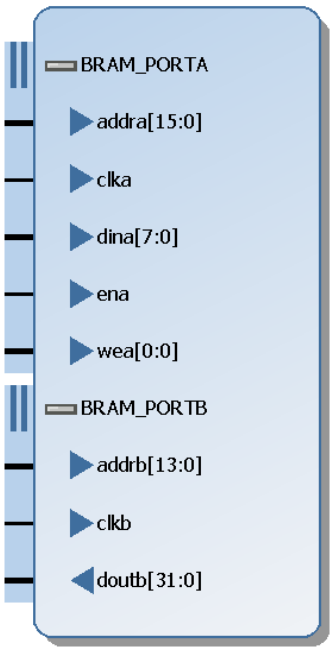
4.5.2.2 Μπλοκ Μνήμης (memory_blocks)

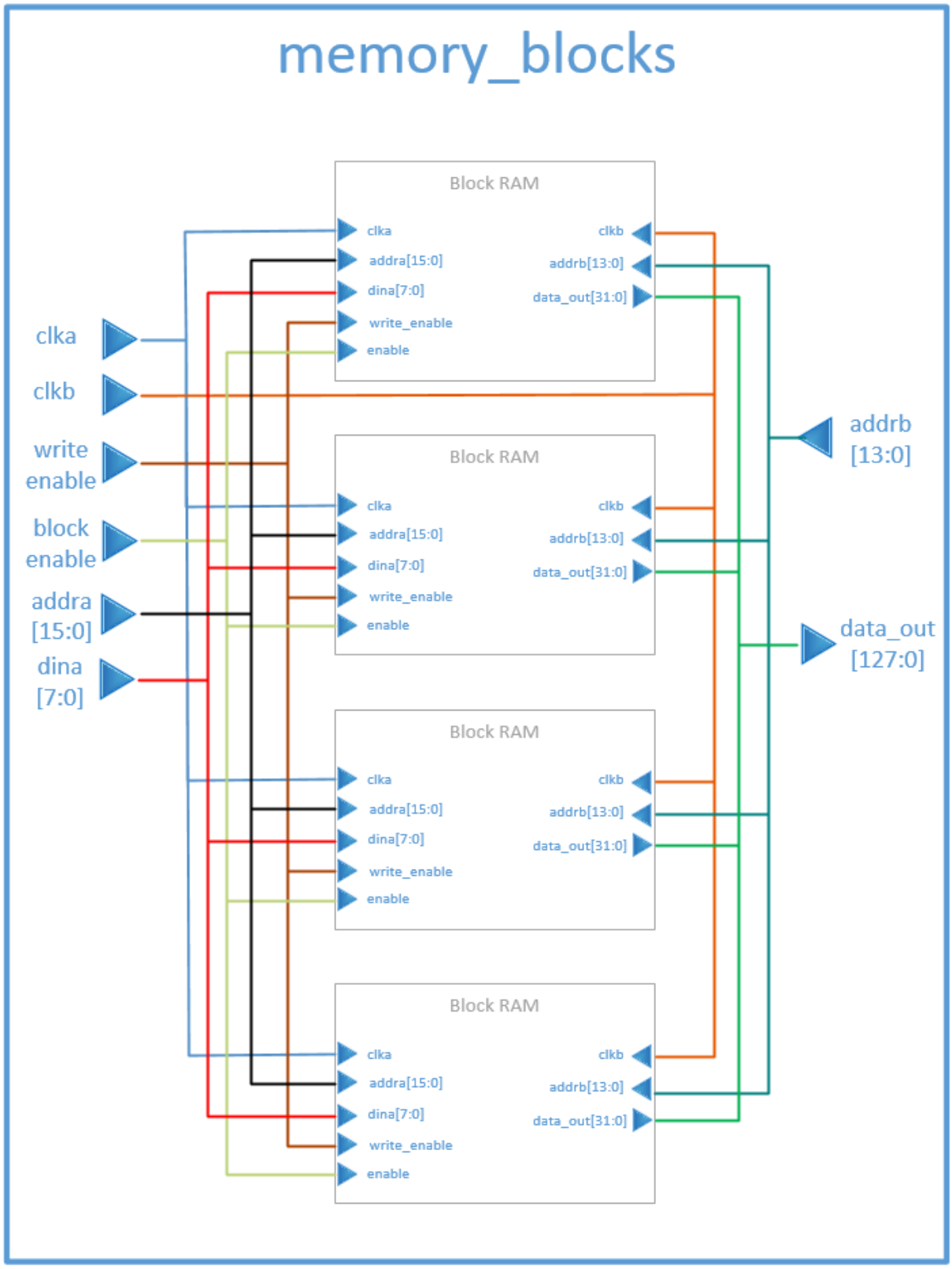
Τα μπλοκ μνήμης περικλείονται σε μια οντότητα έτσι ώστε να έχουν κοινές εισόδους διεύθυνσης εγγραφής, διεύθυνσης ανάγνωσης, ρολογιών και επίτρεψης εγγραφής καθώς και μια κοινή έξοδο η οποία είναι η συνένωση όλων των εξόδων των μπλοκ μνήμης. Όλα τα μπλοκ μνήμης είναι μνήμες RAM με δύο πόρτες (simple dual port RAM).

Το πλάτος της θύρας εγγραφής (BRAM_PORTA) είναι 8bit για να ταιριάζει με το πλάτος της λέξης δεδομένων του UART receiver. Το πλήθος των λέξεων είναι **65536** = 2^{16}, επομένως το πλάτος της διεύθυνσης εγγραφής (addra[15:0]) είναι 16.

Το πλάτος της θύρας ανάγνωσης (BRAM_PORTB) είναι 32bit και το πλήθος των λέξεων δεδομένων είναι 65536 / 4 = **16384** = 2^{14}. Άρα το πλάτος το διεύθυνσης ανάγνωσης είναι 14 (addrb[13:0]).

Το σήμα επίτρεψης **ena** εξασφαλίζει ότι γίνεται εγγραφή μόνο σε ένα μπλοκ μνήμης κάθε φορά, ενώ το αντίστοιχο σήμα επίτρεψης για ανάγνωση είναι πάντοτε ενεργοποιημένο.


BRAM_PORTA
addra[15:0]
clka
dina[7:0]
ena
wea[0:0]
BRAM_PORTB
addrb[13:0]
clkb
doutb[31:0]


memory_blocks
Block RAM
clka
addra[15:0]
dina[7:0]
write_enable
enable
clkb
addrb[13:0]
data_out[31:0]
clka
clkb
write enable
block enable
addra [15:0]
dina [7:0]
addrb [13:0]
data_out [127:0]

4.5.2.3 Μετρητής Ανάγνωσης

Ο μετρητής ανάγνωσης ξεκινάει να μετράει από την αρχική διεύθυνση **init_addr,** η οποία τίθεται στο μηδέν, και αυξάνει διαδοχικά σε κάθε κύκλο του ρολογιού **TXUSRCLK2** μέχρι την τελική διεύθυνση **final_addr**. Στη συνέχεια ξαναμηδενίζεται και επαναλαμβάνεται η μέτρηση κυκλικά.

4.5.2.4 Επέκταση Δεδομένων

Καθώς η σχεδίαση είναι **source synchronous** είναι απαραίτητο να παράγεται ένα σήμα ρολογιού για τον συγχρονισμό των δεδομένων που λαμβάνει ο δέκτης. Για αυτό τον λόγο περιλαμβάνει ένα ξεχωριστό κανάλι για την μετάδοση του ρολογιού, το οποίο έχει διπλάσιο ρυθμό μετάδοσης **5.3Gbps** σε σχέση με τα κανάλια δεδομένων που έχουν ρυθμό μετάδοσης **2.65Gbps**. Προκειμένου να τηρηθεί η αρχή της σύγχρονης σχεδίασης και να μπορούμε να ευθυγραμμίσουμε όλα τα σήματα δεδομένων και ρολογιού, ορίζεται ως ρυθμός μετάδοσης για όλα τα κανάλια **5.3Gbps** και επαναλαμβάνουμε κάθε bit δεδομένων μια φορά έτσι ώστε σε κάθε θετική ακμή του ρολογιού να εμφανίζεται ένα νέο bit δεδομένων. Για παράδειγμα η ακολουθία 1011 γίνεται 11001111. Το κύκλωμα επέκτασης δεδομένων δέχεται ως είσοδο 32bit δεδομένων από την μνήμη και τα μετατρέπει σε δεδομένα πλάτους 64bit σύμφωνα με την παραπάνω διαδικασία.

4.6 Συλλογή και Αποστολή Δεδομένων Εισόδου

Στη σύγχρονη σχεδίαση ολοκληρωμένων κυκλωμάτων ένα σημαντικό μέρος είναι ο σχεδιασμός για ελεγξιμότητα (design for testability) Στην παρούσα εργασία υλοποιήθηκε μια υπομονάδα για την παροχή και συλλογή δεδομένων που απαιτείται σε μια αλυσίδα σάρωσης για τον έλεγχο της ορθής λειτουργίας του ολοκληρωμένου.

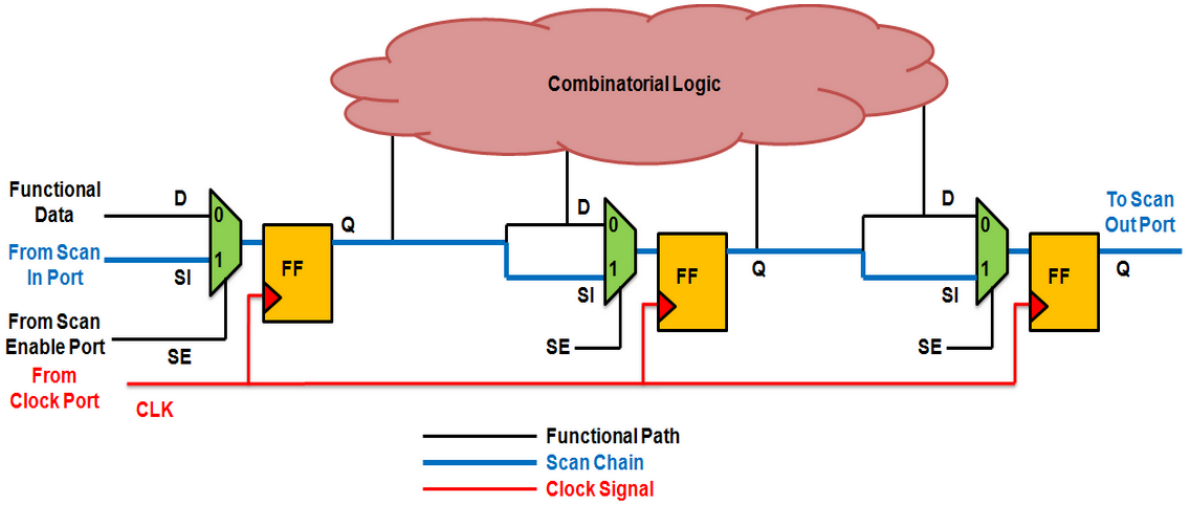
4.6.1 Αλυσίδα Σάρωσης (Scan Chain)

Αλυσίδα σάρωσης (Scan chain) είναι μια τεχνική που εφαρμόζεται στην σχεδίαση ολοκληρωμένων κυκλωμάτων για τον έλεγχό τους. Ο σκοπός της είναι να καταστήσει πιο εύκολη την διαδικασία ελέγχου παρέχοντας ένα απλό τρόπο για την παρατήρηση κάθε flip-flop ενός ολοκληρωμένου κυκλώματος. Όλα τα flip-flop συνδέονται μεταξύ τους για να σχηματίσουν ένα καταχωρητή ολίσθησης. Η βασική δομή της αλυσίδας περιλαμβάνει το

ακόλουθο σύνολο σημάτων έτσι ώστε να ελεγχθεί και να παρατηρηθεί ο μηχανισμός σάρωσης:

1. Scan_in και scan_out ορίζονται ως η είσοδος και η έξοδος της αλυσίδας αντίστοιχα.
2. Scan_enable είναι ένα σήμα επίτρεψης, το οποίο όταν τίθεται στο λογικό 1, κάθε flip-flop του ολοκληρωμένου κυκλώματος συνδέεται σε διάταξη καταχωρητή ολίσθησης.
3. Ένα σήμα ρολογιού για τον έλεγχο των flip-flop στην αλυσίδα κατά την φάση ολίσθησης. Μπορεί να εισαχθεί μια οποιαδήποτε αυθαίρετη ακολουθία παλμών και να διαβαστεί η κατάσταση κάθε flip-flop

Το πρώτο flip-flop της αλυσίδας συνδέεται με την θύρα scan_in και το τελευταίο με την θύρα scan_out. Ο έλεγχος σάρωσης (scan testing) πραγματοποιείται ούτως ώστε να ανιχνευθούν λάθη κατασκευής στο συνδυαστικό μέρος του κυκλώματος.



4.6.2 Υποσύστημα συλλογής και αποστολής δεδομένων εισόδου.

Το υποσύστημα συλλογής και αποστολής δεδομένων αποτελείται από :

1. Μια μπλοκ μνήμη
2. Υποσύστημα Διευθυνσιοδότησης της Μνήμης
3. UART πομπό.

Η δομή και λειτουργία των παραπάνω υποσυστημάτων έχει ήδη παρουσιαστεί στις προηγούμενες ενότητες του κεφαλαίου.

5

Χειρισμός της πλατφόρμας ελέγχου μέσω MATLAB

Ο χειρισμός της πλατφόρμας μπορεί να γίνει αποκλειστικά μέσω MATLAB, γεγονός που επιτρέπει στον χρήστη να αγνοήσει τις λεπτομέρειες χαμηλότερων επιπέδων. Ακόμη είναι δυνατή η απομακρυσμένη πρόσβαση. Για τον σκοπό αυτό κατασκευάστηκε μια βιβλιοθήκη συναρτήσεων.

5.1 Εγκαθίδρυση Σειριακής Σύνδεσης μεταξύ FPGA και υπολογιστή

Όλες οι συναρτήσεις έχουν ως ένα από τα ορίσματα τους ένα σειριακό αντικείμενο (Serial port object). Αυτό το αντικείμενο δημιουργείται με την συνάρτηση ορίζοντας την σειριακή θύρα και μια λίστα παραμέτρων:

```
obj = serial('port','PropertyName',PropertyValue,...)
```

Οι παράμετροι είναι οι εξής:

- Baudrate = 921600bps
- ParityBit = None

- DataBits = 8
- StopBits = 1
- Timeout = 100s - μέγιστος χρόνος αναμονής για λειτουργιά ανάγνωσης ή εγγραφής.
- FlowControl = none – δεν χρησιμοποιείται handshaking.
- InputBufferSize = 5000000bytes – συνολικός αριθμός bytes που μπορούν να αποθηκευτούν στον buffer εισόδου.

Το παρακάτω script δημιουργεί ένα σειριακό αντικείμενο, ορίζει τις παραπάνω παραμέτρους και στην περίπτωση που διαβαστούν δεδομένα τυπώνονται στο Command Window.

```
s = serial('COM5','FlowControl','hardware', 'Baudrate', 921600,'Terminator',
'')


set(s, 'InputBufferSize',5000000);
set(s, 'FlowControl', 'none');
set(s, 'BaudRate', 921600);
set(s, 'Parity', 'none');
set(s, 'DataBits', 8);
set(s, 'StopBits', 1);
set(s, 'Timeout', 100);
s.BytesAvailableFcnCount = 1;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @(src, evt)fprintf(2, '%02x ', fread(src, 1) );

fopen(s);
```

5.2 *Ορισμός Πλήθους Καναλιών που θα χρησιμοποιηθούν*

```
function [ ] = set_number_of_blocks( serial_object, num_of_blocks )
```

Με την συνάρτηση `set_number_of_blocks()` ορίζεται το πλήθος των καναλιών που θα χρησιμοποιηθούν. Πρακτικά περιορίζεται ο αριθμός των μνημών BRAMs που θα φορτωθούν με δεδομένα. Η τιμή του num_of_blocks πρέπει να είναι μια δεκαδική τιμή από 1 έως 24.

5.3 *Ορισμός Πλήθους byte που θα φορτωθούν στην μνήμη*

```
function [ ] = set_memory_size( serial_object, mem_size )
```

Το πλήθος των byte ορίζεται μέσω της παραμέτρου *mem_size* και η τιμή που μπορεί να πάρει είναι από 0 έως 65535.

5.4 Soft Reset

```
function [ ] = set_soft_reset( serial_object )
```

Με την κλήση της set_soft_reset γίνεται soft reset στο σύστημα. Η συνάρτηση έχει ως μοναδικό όρισμα το σειριακό αντικείμενο.

5.5 Ορισμός τελικής διεύθυνσης ανάγνωσης

```
function [  ] = set_final_addr_read( serial_object, final_addr_read )
```

Αυτή η συνάρτηση δίνει την δυνατότητα να αναπαραχθεί ένα μικρότερο σύνολο δειγμάτων από αυτά που έχουν ήδη φορτωθεί. Επειδή η διεύθυνση ανάγνωσης κατά δύο Bit μικρότερη από την διεύθυνση εγγραφής οι τιμές που μπορεί να πάρει η παράμετρος final_addr_read είναι μεταξύ 0 και 16383.

5.6 Ορισμός της κατάστασης λειτουργίας

```
function [ ] = set_test_mode( serial_object, flag )
```

Όταν η παράμετρος flag παίρνει την τιμή 0, τότε το σύστημα τίθεται σε κατάσταση κανονικής λειτουργίας και τα δεδομένα που είναι αποθηκευμένα στις μνήμες αναπαράγονται κυκλικά. Για οποιαδήποτε άλλη τιμή το σύστημα τίθεται σε testing mode, οπότε τα δεδομένα αναπαράγονται μια μόνο φορά και ταυτόχρονα γίνεται αποθήκευση των δεδομένων εισόδου. Μόλις τελειώσει η αναπαραγωγή, τα δεδομένα στέλνονται αυτόματα μέσω της σειριακής σύνδεσης στον υπολογιστή.

5.7 Ορισμός του επιπέδου τάσης των καναλιών

Η παράμετρος value μπορεί να πάρει τιμές από 0 έως 15 που αντιστοιχούν σε διαφορική τάση από κορυφή-σε-κορυφή που φαίνεται στον παρακάτω πίνακα:

[3:0]	V_{PPD}
4'b0000	0.269
4'b0001	0.336
4'b0010	0.407
4'b0011	0.474
4'b0100	0.543
4'b0101	0.609
4'b0110	0.677
4'b0111	0.741
4'b1000	0.807
4'b1001	0.866
4'b1010	0.924
4'b1011	0.973
4'b1100	1.018
4'b1101	1.056
4'b1110	1.092
4'b1111	1.119

5.8 Ενεργοποίηση Ολίσθηση φάσης

Με την παράμετρο transceiver_number ορίζεται ο αριθμός του καναλιού στο οποίο θα εφαρμοσθεί ολίσθηση φάσης. Η τιμή της μπορεί να είναι από 0 έως 23. Η παράμετρος txpippmstepsize μπορεί να είναι μια δυαδική τιμή πλάτος 5bit με δεκαεξαδική αναπαράσταση. Το πιο σημαντικό bit txpippmstepsize [4] καθορίζει την αύξηση ή την μείωση της φάσης. Τα υπόλοιπα τέσσερα bit καθορίζουν το πόσο θα μεταβληθεί η φάση.

```
function [  ] = txpi_enable( serial_object, transceiver_number,
txpippmstepsize)
```

5.9 Φόρτωση Δεδομένων

```
function [ ] = set_given_data(serial_object, final_address,
number_of_blocks, Data)
```

Η συνάρτηση set_given_data κάνει χρήση συναρτήσεων χαμηλότερου επιπέδου για να θέσει τον αριθμό των καναλιών και το πλήθος των byte που θα φορτωθούν. Η παράμετρος

final_address προσδιορίζει το πλήθος των δεδομένων που θα αποθηκευτούν ενώ η number_of_blocks θέτει τον αριθμό των καναλιών που θα φορτωθούν με δεδομένα. Στον πίνακα Data πρέπει να είναι καταχωρημένα που θα φορτωθούν. Το μέγεθος του πίνακα πρέπει να είναι ίσο με το γινόμενο των παραμέτρων `final_address * number_of_blocks`.

6

Μετρήσεις, Αποτελέσματα και Μελλοντικές Επεκτάσεις

Σε αυτό το κεφάλαιο συνοψίζονται οι πειραματικές μετρήσεις και τα αποτελέσματα της υλοποίησης. Επίσης προτείνονται μελλοντικές επεκτάσεις του παρόντος συστήματος.

6.1 Σύνοψη

Στόχος της παρούσας διπλωματικής ήταν η ανάπτυξη ενός αυτοματοποιημένου συστήματος ελέγχου ολοκληρωμένων κυκλωμάτων υψηλών ταχυτήτων. Συγκεκριμένα υλοποιήθηκε η λειτουργικότητα της αναπαραγωγής παράλληλων διανυσμάτων, της ευθυγράμμιση φάσης των δεδομένων, της ρύθμισης παραμέτρων, της φόρτωσης δεδομένων και της συλλογής δεδομένων και αποστολής τους στον υπολογιστή για επιβεβαίωση της ορθής λειτουργίας της συσκευή υπό έλεγχο. Η καινοτομία στη σχεδίαση έγκειται στο γεγονός ότι εφαρμόζεται αυτόματη ευθυγράμμιση των δεδομένων.

Στα πρώτα κεφάλαια δίνονται οι απαραίτητες θεωρητικές βάσεις που χρειάζονται για την σχεδίαση και υλοποίηση αλγορίθμων σε FPGA. Ακολούθως παρουσιάζονται οι πομποδέκτες υψηλών ταχυτήτων, η οργάνωση τους και τα υποσύστημα που τους αποτελούν.

Για την μελέτη, σχεδίαση και υλοποίηση του συστήματος ελέγχου ακολουθήθηκε η εξής διαδικασία:

1. Αρχικά μελετήθηκαν οι πομποδέκτες υψηλών ταχυτήτων και ορίστηκαν οι περιορισμοί του συστήματος, π.χ. συχνότητα μετάδοσης, δημιουργία ρολογιού αναφοράς.
2. Ακολούθησε η σχεδίαση της αρχιτεκτονικής του συστήματος με τον ορισμό των υπομονάδων που το συνθέτουν και η συγγραφή του κώδικα σε γλώσσα Verilog για κάθε μια από τις υπομονάδες.
3. Στη συνέχεια για κάθε μια από τις υπομονάδες (modules) κατασκευάστηκαν testbenches και εκτελέστηκαν προσομοιώσεις για την επιβεβαίωση της ορθής λειτουργίας όλων των υπομονάδων ξεχωριστά και του όλου συστήματος.
4. Ορίστηκαν περιορισμοί (constraints) για την υλοποίηση των μονάδων στο FPGA.
5. Ακολούθησε η σύνθεση και υλοποίηση του συστήματος σε FPGA κάνοντας χρήση της πλατφόρμας Vivado 15.2.
6. Εφαρμόστηκαν τεχνικές time closure για να επιλυθούν προβλήματα χρονισμού.
7. Το bitstream που παράγεται μετά την υλοποίηση χρησιμοποιήθηκε για τον προγραμματισμό του FPGA.
8. Έγινε συγγραφή κώδικα σε MATLAB για την δημιουργία συνάρτησης χειρισμού της πλατφόρμας.
9. Τέλος έγινε έλεγχος της σωστής λειτουργίας του συστήματος συνδέοντας καθένα από τα κανάλια σε παλμογράφο και επικοινωνία με ηλεκτρονικό υπολογιστή για αποστολή εντολών, και αποστολή και λήψη δεδομένων. Ελέγχθηκαν και οι δύο καταστάσεις λειτουργίας.

6.2 Μετρήσεις

6.2.1 Επιβεβαίωση της αξιοπιστίας της σύνδεσης με ηλεκτρονικό υπολογιστή

Μια βασική απαίτηση του συστήματος είναι να μην υπάρχουν σφάλματα κατά την μετάδοση και λήψη δεδομένων μεταξύ ηλεκτρονικού υπολογιστή και FPGA. Για τον λόγο αυτό συντάχθηκε MATLAB script το οποίο κατασκευάζει αρχείο με τυχαία δεδομένα, έπειτα τα στέλνει στο FPGA. Από την πλευρά του FPGA τα δεδομένα λαμβάνονται, αποθηκεύονται σε μπλοκ μνήμης και ύστερα αποστέλλονται πίσω στον υπολογιστή και αποθηκεύονται σε

ένα αρχείο ανάγνωσης. Τα δύο αρχεία συγκρίνονται μεταξύ τους για να επιβεβαιωθεί ότι δεν υπάρχουν σφάλματα στην επικοινωνία του FPGA με τον υπολογιστή.

6.2.2 Επιβεβαίωση της αυτόματης ευθυγράμμισης φάσης

Τα δεδομένα είναι απαραίτητο να είναι συγχρονισμένα για να ληφθούν χωρίς λάθη από την συσκευή υπό έλεγχο. Στις παρακάτω εικόνες φαίνονται στιγμιότυπα από μετρήσεις σε παλμογράφο που επιβεβαιώνουν ότι έχει γίνει ευθυγράμμιση. Στην πρώτη εικόνα η πάνω γραμμή είναι το ρολόι αναφοράς και οι άλλες δύο γραμμές έχουν σταθερά επαναλαμβανόμενα pattern 00001111 (μεσαία γραμμή) και 11001100 (κάτω γραμμή). Στην δεύτερη εικόνα κρατώντας σταθερές τις δύο κάτω γραμμές, για αναφορά, εφαρμόζουμε τυχαία δεδομένα στην πρώτη και παρατηρούμε και πάλι ευθυγράμμιση φάσης.

6.2.3 Ρύθμιση του επιπέδου τάσης

Εκτελώντας την συνάρτηση την συνάρτηση `set_driver_swing_control` με παράμετρο:

- TXDIFFCTRL = 1:

  ```
  set_driver_swing_control(s, 1);
  ```

  Παρατηρείται τάση από κορυφή σε κορυφή V_{pp} = 200mV

- TXDIFFCTRL = 1:

  ```
  set_driver_swing_control(s, 1);
  ```

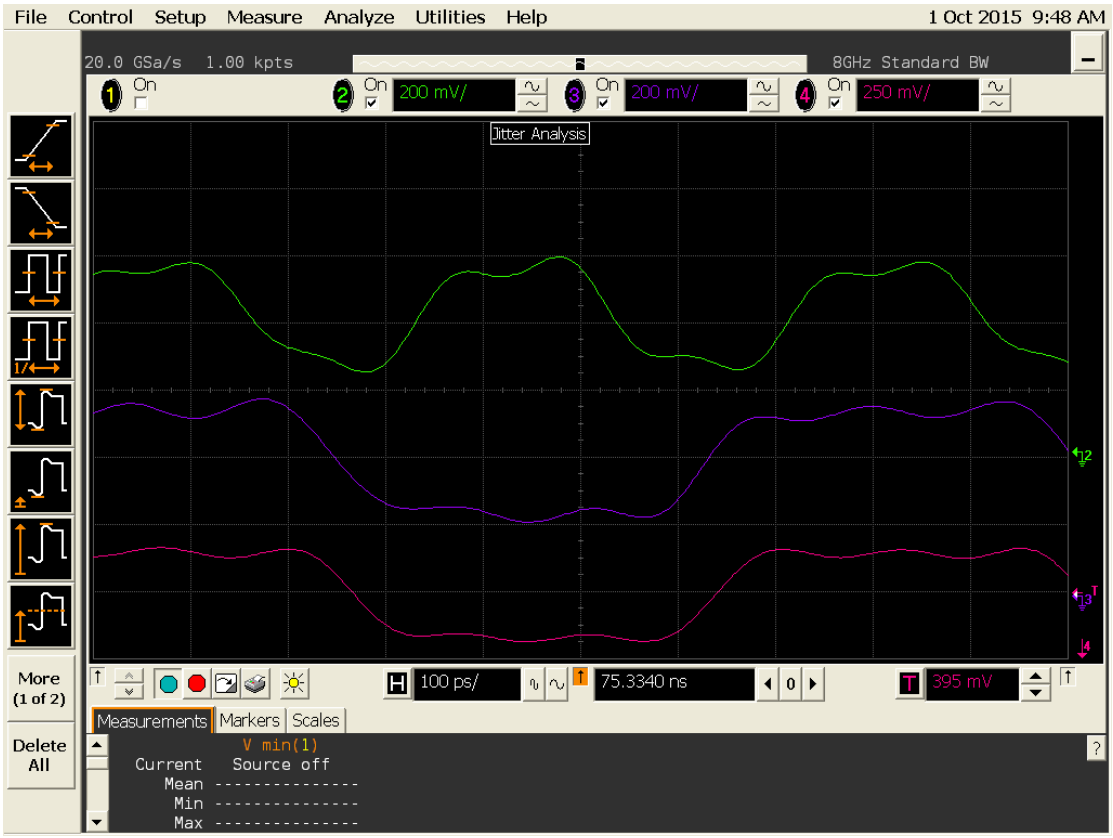
  Παρατηρείται τάση από κορυφή σε κορυφή V_{pp} = 440mV

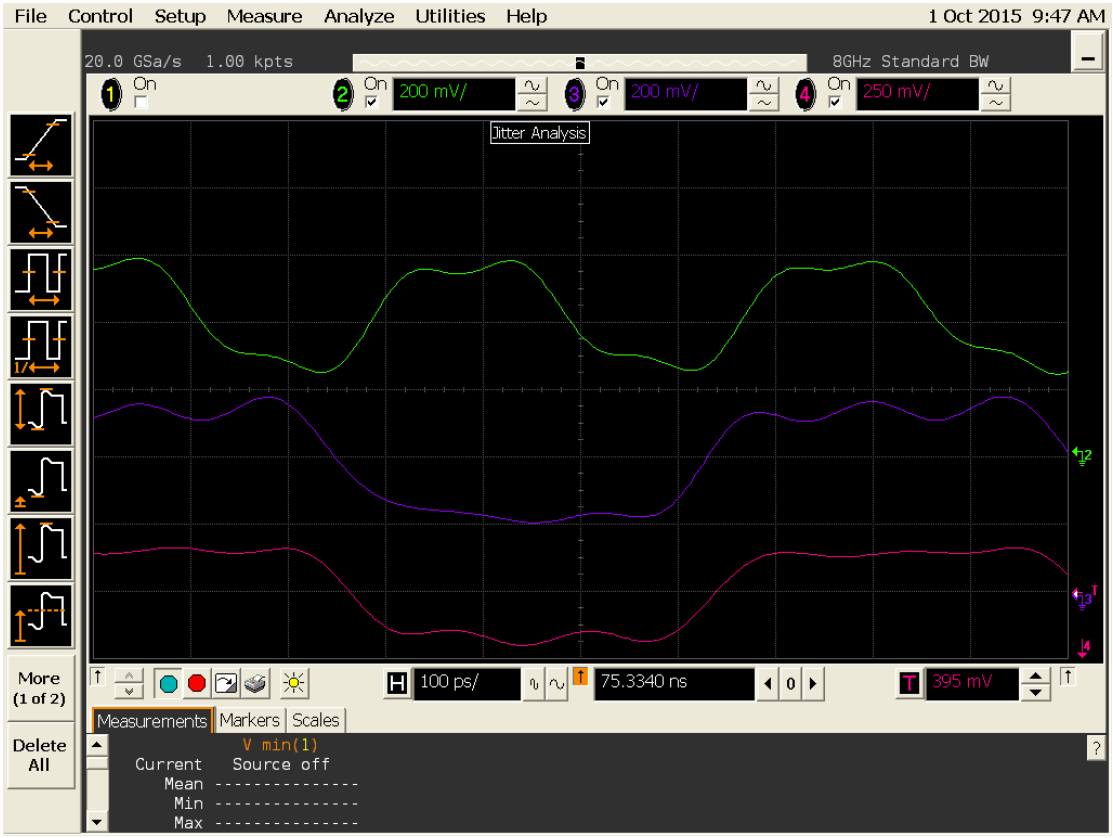

File Control Setup Measure Analyze Utilities Help
1 Oct 2015 3:02 PM
20.0 GSa/s 1.00 kpts
8GHz Standard BW
On 200 mV/
On 200 mV/
On 250 mV/
Jitter Analysis
200 ps/
75.3340 ns
395 mV
More (1 of 2)
Delete All
Measurements
Markers
Scales
V p-p(4) V p-p(3) V p-p(2) V min(1)
Current 205 mV 201 mV 184 mV Source off
Mean 207.4 mV 203.1 mV 178.1 mV
Min 198 mV 194 mV 171 mV
Max 223 mV 215 mV 190 mV

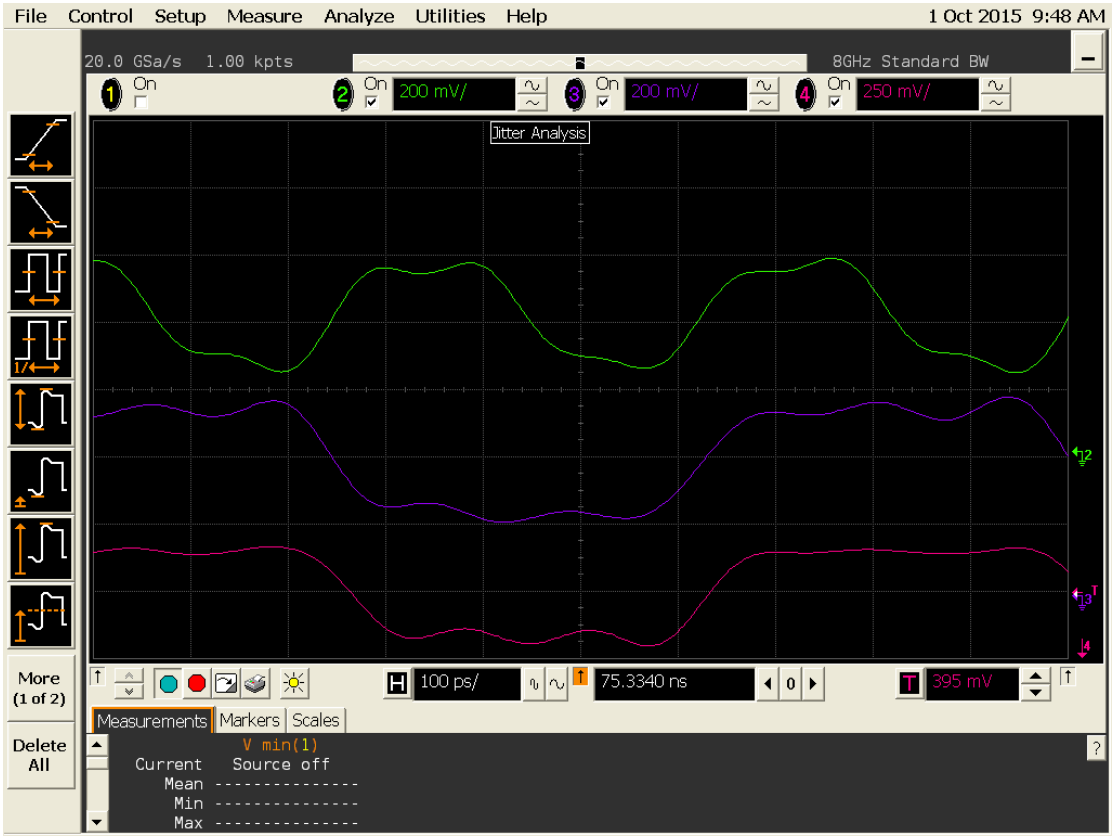

File Control Setup Measure Analyze Utilities Help
1 Oct 2015 3:03 PM
20.0 GSa/s 1.00 kpts
8GHz Standard BW
On 200 mV/
On 200 mV/
On 250 mV/
Jitter Analysis
200 ps/
75.3340 ns
395 mV
More (1 of 2)
Delete All
Measurements
Markers
Scales
V p-p(4) V p-p(3) V p-p(2) V min(1)
Current ? 421 mV 438 mV 404 mV Source off
Mean ? 343.86 mV 350.86 mV 321.41 mV
Min ? 197 mV 194 mV 171 mV
Max ? 443 mV 451 mV 422 mV

6.2.4 Ρύθμιση της φάσης.

Με κλήση της συνάρτησης `txpi_enable(s, '17', '01')` δίνοντας στο δεύτερο όρισμα τον αριθμό του καναλιού και ως τρίτο όρισμα την τιμή της παραμέτρου TXPIPPMSTEPSIZE[4:0] όπως έχει περιγραφεί στο κεφάλαιο 3 μπορούμε να μετατοπίσουμε την φάση κάθε καναλιού ξεχωριστά. Στα παρακάτω στιγμιότυπα παρατηρούμε την μείωση της φάσης (μετατόπιση προς τα αριστερά) του πάνω (πράσινου) καναλιού με διαδοχικές κλήσεις της συνάρτησης.




File Control Setup Measure Analyze Utilities Help
1 Oct 2015 9:47 AM
20.0 GSa/s 1.00 kpts
8GHz Standard BW
On
200 mV/
200 mV/
250 mV/
Jitter Analysis
More (1 of 2)
Delete All
100 ps/
75.3340 ns
395 mV
Measurements
Markers
Scales
V min(1)
Current Source off
Mean
Min
Max


File Control Setup Measure Analyze Utilities Help
1 Oct 2015 9:48 AM
20.0 GSa/s 1.00 kpts
8GHz Standard BW
On
200 mV/
200 mV/
250 mV/
Jitter Analysis
More (1 of 2)
Delete All
100 ps/
75.3340 ns
395 mV
Measurements
Markers
Scales
V min(1)
Current Source off
Mean
Min
Max

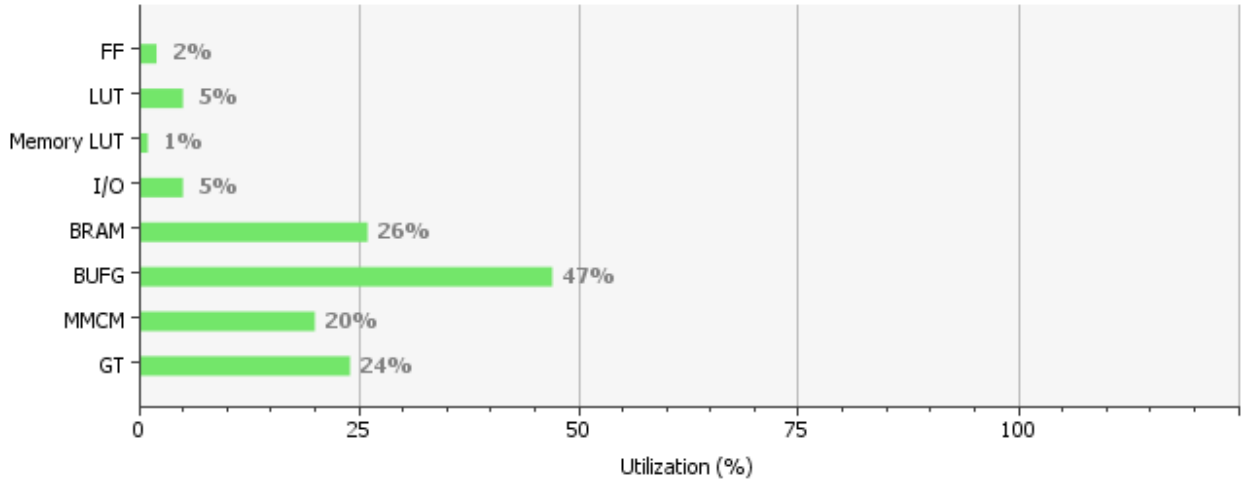
6.3 *Αποτελέσματα Υλοποιημένης Σχεδίασης*

Σε αυτή την ενότητα παρουσιάζονται τα αποτελέσματα χρήσης πόρων του FPGA μετά την σύνθεση και υλοποίηση του συστήματος

6.3.1 *Χρήση Πόρων*

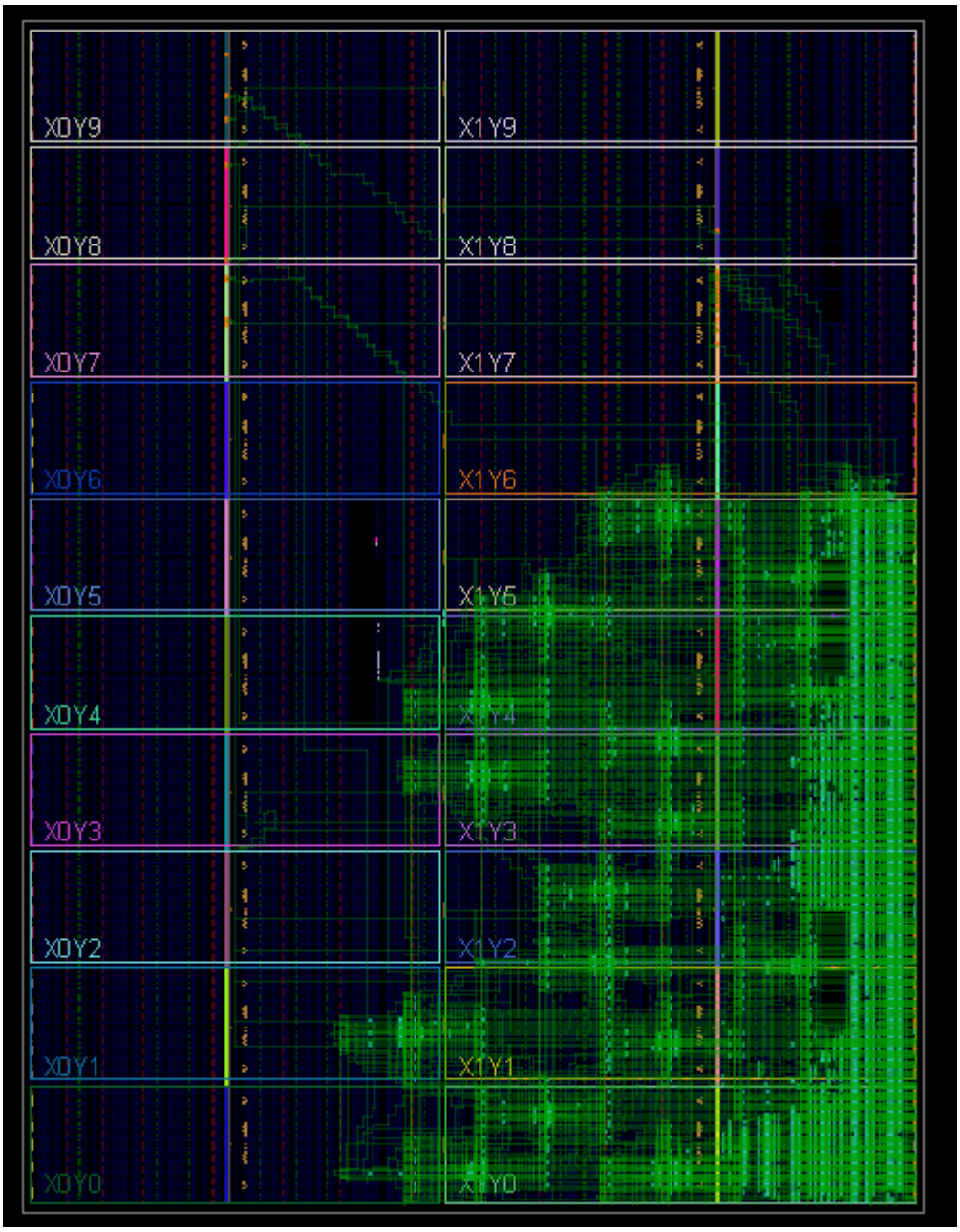
Παρακάτω φαίνονται τα ποσοστά χρήσης των διαθέσιμων πόρων που διαθέτει η συσκευή XC7VX690T-3FFG1927E

Resource	Utilization	Available	Utilization %
FF	18392	866400	2.12
LUT	20481	433200	4.73
Memory LUT	792	174200	0.45
I/O	32	600	5.33
BRAM	385	1470	26.19
BUFG	15	32	46.88
MMCM	4	20	20.00
GT	24	100	24.00



6.3.2 *Δίκτυο διασύνδεσης του FPGA*

Στην παρακάτω εικόνα φαίνεται εποπτικά το δίκτυο διασύνδεσης των πόρων του FPGA. Όπως φαίνεται η συσκευή είναι χωρισμένη σε περιοχές όπου κάθε περιοχή έχει το δικό της δίκτυο διάδοσης ρολογιού. Στις κατακόρυφες στήλες βρίσκονται τοποθετημένα τα μπλοκ μνήμης.



6.3.3 Αποτελέσματα Χρονισμού

6.3.3.1 Setup Time

Worst Negative Slack (WNS):	0.548 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	22828

6.3.3.2 Hold Time

Worst Hold Slack (WHS):	0.021 ns
Total Hold Slack (THS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	22828

6.3.3.3 Pulse Width

Worst Pulse Width Slack (WPWS):	0.72 ns
Total Pulse Width Negative Slack (TPWS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	12356

6.4 Αποτελέσματα μέτρησης κατανάλωσης ισχύος

Total On-Chip Power:	**7.589 W**
Junction Temperature:	**31,4 °C**
Thermal Margin:	68,6 °C (76,9 W)
Effective ϑJA:	0,8 °C/W
Power supplied to off-chip devices:	0.004 W

6.5 Μελλοντικές Επεκτάσεις

Μερικές από τις μελλοντικές επεκτάσεις του υπάρχοντος συστήματος είναι οι εξής:

1. Υλοποίηση αλγορίθμου Cyclic Redundancy Check (CRC) για τον εντοπισμό σφαλμάτων κατά την μετάδοση δεδομένων από και προς τον υπολογιστή.
2. Ταχύτερη επικοινωνία FPGA με υπολογιστή μέσω PCI Express.
3. Επέκταση της μνήμης με εξωτερική μνήμη DDR3 SDRAM.
4. Υλοποίηση αλγορίθμων ψηφιακής επεξεργασίας σήματος για τηλεπικοινωνιακές εφαρμογές.

7

Κώδικας Περιγραφής Υλικού σε Verilog.

7.1 Controller UART

```
module controller_uart #
(
    // Memory Blocks
    parameter ADDR_WIDTH = 16,
    parameter NUMBER_OF_BLOCKS = 24,
    // UART Receiver
    parameter DBIT    = 8,          // Number of data bits transmitted and received by
UART
    parameter SB_TICK = 16,         // Number of ticks for stop bits
    parameter DVSR    = 14          // baud rate divisor
                                   // DVSR = 200MHz / (16 * baud rate)
)
(
    input uart_clk,
    input user_clk,
    input reset,

    input wire rx,

    output wire [64 * (NUMBER_OF_BLOCKS - 1)- 1:0] data_out_b_extended,
```

```
    output wire [63:0] clk_data_out,
    output wire soft_reset,
    output wire [3:0]  diffctrl_param,
    // TX Phase Interpolator
    output wire [2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0] txpi_selector,
    output wire [4:0] txpippmstepsize,
    output wire       txpippmsel,

    // Write Receiver Data
    output wire start_writing,
    output wire test_completed
);
    // Signal Declaration
    localparam NUMBER_OF_BITS_IN_BLOCK_COUNTER = log2(NUMBER_OF_BLOCKS);

    wire [7:0] instruction;
    wire       test_mode;
    wire fetch;
    wire data_transmission_complete;

    // VIO ouputs
    wire [ADDR_WIDTH - 1 : 0] final_addr_write;
    wire [ADDR_WIDTH - 3 : 0] final_addr_read;
    wire [10:0] txpi_param, txpi_param_hold;
    wire [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0] num_blocks_param;

    wire start_data_loading;
    wire [4:0] contr_current_state;
    wire [ADDR_WIDTH - 1:0] address_a;

    // UART
    wire tick;
    instruction_controller  #
    (
        .ADDR_WIDTH(ADDR_WIDTH),
        .NUMBER_OF_BLOCKS(NUMBER_OF_BLOCKS)
    )
    instr_controller
    (
        .clk(uart_clk),
        .reset(reset),
        .instruction_code(instruction),
```

```
    .fetch_instruction(fetch),
    .data_transmission_complete(data_transmission_complete),

    .mem_size_param(final_addr_write),
    .final_address_read_param(final_addr_read),
    .diffctrl_param(diffctrl_param),
    .num_blocks_param(num_blocks_param),
    .soft_reset(soft_reset),
    .start_data_loading(start_data_loading),
    .test_mode(test_mode),
    .txpi_param(txpi_param),

    // Debugging
    .current_state(contr_current_state)
);
data_hold #
(
    .WIDTH(11),
    .HOLD_CYCLES(1000)
)
data_holder
(
    .clk(uart_clk),
    .reset(reset),
    .async_in(txpi_param),
    .out(txpi_param_hold)
);

assign txpippmstepsize = txpi_param_hold[5:1];
assign txpippmsel      = txpi_param_hold[0];


decoder #
(
    .NUMBER_OF_BLOCKS(NUMBER_OF_BLOCKS)
)
txpi_decoder
(
    .select(txpi_param_hold[10:6]),
    .out(txpi_selector)
);
// Instatiation of UART Receiver
```

```
uart_rx #
(
    .DBIT(DBIT),
    .SB_TICK(SB_TICK)
)
uart_rx_unit
(
    .clk(uart_clk),
    .reset(reset),
    .rx(rx),
    .s_tick(tick),
    .rx_done_tick(fetch),
    .dout(instruction)
);

// Baud Rate Genarator
mod_m_counter #
(
    .M(DVSR) // 14
)
baud_gen_unit
(
    .clk(uart_clk),
    .reset(reset),
    .q(),
    .max_tick(tick)
);

vio_0 vio_controller
(
  .clk(uart_clk),                // input wire clk
  .probe_in0(final_addr_write),   // input wire [19 : 0] probe_in0
  .probe_in1(diffctrl_param),   // input wire [3 : 0] probe_in1
  .probe_in2(contr_current_state),    // input wire [3 : 0] probe_in2
  .probe_in3(address_a),        // input wire [15 : 0] probe_in3
  .probe_in4(instruction),      // input wire [7 : 0] probe_in4
  .probe_in5(num_blocks_param)    // input wire [4 : 0] probe_in5
);



memory_manager #
(
```

```
        // Memory Blocks
        .ADDR_WIDTH(ADDR_WIDTH),
        .NUMBER_OF_BLOCKS(NUMBER_OF_BLOCKS)
    )
    mem_manager
    (
        .clka(uart_clk),
        .clkb(user_clk),
        .reset(reset),
        .start(start_data_loading),
        // limits
        .number_of_blocks(num_blocks_param),
        .final_addr_write(final_addr_write),
        .final_addr_read(final_addr_read),
        // uart
        .rx_done_tick(fetch),
        .rx_data_out(instruction),
        // Write Receiver Data Control Signals
        .test_mode(test_mode),
        .start_writing(start_writing),
        .test_completed(test_completed),
        // output
        .data_out_b_extended(data_out_b_extended),
        .clk_data_out(clk_data_out),
        .data_transmission_complete(data_transmission_complete),
        .address_a(address_a)
    );

    // log2 constant function
    function integer log2(input integer n);
        integer i;
    begin
        log2 = 1;
        for (i = 0; 2 ** i < n; i = i + 1)
            log2 = i + 1;
    end
    endfunction

endmodule
```

7.1.1 Instruction Controller

```
module instruction_controller #
(
    parameter ADDR_WIDTH = 16,
    parameter NUMBER_OF_BLOCKS = 24,
    // Default memory size = 56 = 0x38
    parameter default_mem_size_first_byte  = 8'h38,
    parameter default_mem_size_second_byte = 8'h00,
    parameter default_final_addr_read      = 8'h0f,
    // The default value of the driver swing control corresponds to Vppd = 0.973V
    parameter default_diff_ctrl            = 4'hB,
    parameter default_num_of_blocks        = 5'h18
)
(
    input wire         clk,  reset,
    input wire  [7:0]  instruction_code,
    input wire         fetch_instruction,
    input wire         data_transmission_complete,

    output wire [ADDR_WIDTH - 1:0]                      mem_size_param,
    output wire [ADDR_WIDTH - 3:0]                      final_address_read_param,
    output wire [3:0]                                   diffctrl_param,
    output wire [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0] num_blocks_param,
    output wire [10:0]                                  txpi_param,
    output wire                                         soft_reset,
    output wire                                         start_data_loading,
    output wire                                         test_mode,

    // Just For Debugging
    output wire [4:0]  current_state
);

    localparam NUMBER_OF_BITS_IN_BLOCK_COUNTER = log2(NUMBER_OF_BLOCKS);



    // ------------------------------------------------------------------------//
    //---------------------------- Instructions ----------------------------------//
    // ------------------------------------------------------------------------//
    localparam[7:0]
        Set_Memory_Size             = 8'h42,
        Set_Driver_Swing_Control    = 8'h43,
        Set_Soft_Reset              = 8'h44,
        Load_Data                   = 8'h45,
        Set_Number_of_Blocks        = 8'h46,
```

```
    Enable_Phase_Inerpolator    = 8'h47,
    Set_Final_Address_Read      = 8'h48,
    Set_Test_Mode_ON            = 8'h49,
    Set_Test_Mode_OFF           = 8'h4a;

// ---------------------------------------------------------------------------//
//----------------------------------- States ---------------------------------//
// ---------------------------------------------------------------------------//
localparam[4:0]
    idle                            = 5'b00000,
    instruction_fetch               = 5'b00001,
    // Instruction: Set_Memory_Size
    mem_size                        = 5'b00010,
    first_mem_size_byte             = 5'b00011,
    second_mem_size_byte            = 5'b00100,
    third_mem_size_byte             = 5'b00101,
    // Instruction: Set_Driver_Swing_Control
    diff_ctrl                       = 5'b00110,
    write_diff_ctrl_param           = 5'b00111,
    // Instruction: Set_Soft_Reset
    set_soft_reset                  = 5'b01000,
    // Instruction: Load_Data
    data_transmission               = 5'b01001,
    end_of_data_transmission        = 5'b01010,
    // Instruction: Set_Number_of_Blocks
    set_number_of_blocks            = 5'b01011,
    write_num_of_blocks             = 5'b01100,
    // Instruction: Enable_Phase_Interpolator
    phase_interpolator_enable       = 5'b01101,
    phase_interpolator_num          = 5'b01110,
    txpippmstepsize                 = 5'b01111,
    phase_interpolator_send         = 5'b10000,
    // Instruction: Set_Final_Address_Read
    final_address_read              = 5'b10001,
    first_final_address_read_byte   = 5'b10010,
    second_final_address_read_byte  = 5'b10011,
    third_final_address_read_byte   = 5'b10100,
    // Instruction: Set_Test_Mode_ON
    set_test_mode_on                = 5'b10101,
    // Instruction: Set_Test_Mode_OFF
    set_test_mode_off               = 5'b10110;



// ---------------------------------------------------------------------------//
//--------------------------- Signal Declaration -----------------------------//
// ---------------------------------------------------------------------------//
```

```
    reg [4:0]  state_reg, state_next;
    reg        test_mode_reg, test_mode_next;
    // Memory Size
    reg [7:0]               mem_size_first_byte,  mem_size_first_byte_next;
    reg [7:0]               mem_size_second_byte, mem_size_second_byte_next;
    reg [3:0]               mem_size_third_byte,  mem_size_third_byte_next;
// not used for 16bit address
    reg [ADDR_WIDTH - 1:0]  mem_size_param_reg,   mem_size_param_reg_next;
    // Set_Final_Address_Read
    reg   [7:0]                                                 final_address_read_first_byte_reg,
final_address_read_first_byte_next;
    reg   [4:0]                                                final_address_read_second_byte_reg,
final_address_read_second_byte_next;
    reg   [3:0]                                                 final_address_read_third_byte_reg,
final_address_read_third_byte_next;         // not used for 16bit address
    reg [ADDR_WIDTH - 3:0]  final_address_read_param_reg,   final_address_read_param_next;
    // Differential Voltage Swing Control
    reg [3:0]  diffctrl_param_reg,   diffctrl_param_reg_next;
    reg [3:0]  diff_ctrl_reg, diff_ctrl_reg_next;
    // Number of Blocks
    reg [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0]  num_of_blocks_reg,   num_of_blocks_next;
    reg      [NUMBER_OF_BITS_IN_BLOCK_COUNTER      -      1:0]            num_of_blocks_param_reg,
num_of_blocks_param_next;
    // TX Phase Interpolator
    reg [5:0]  phase_interpolator_num_reg, phase_interpolator_num_next;     // Choose 1 between 64
(2 ^ 6) transmitters
    reg [4:0]  txpippmstepsize_reg,  txpippmstepsize_next;
    reg [11:0] txpi_param_reg, txpi_param_next;


    // ------------------------------------------------------------//
    //-------------------- State Register ---------------------------//
    // ------------------------------------------------------------//
    always @ (posedge clk, posedge reset)
    begin
        if (reset)
        begin
            state_reg                  <= idle;
            test_mode_reg              <= 1'b0;
            // Memory Size
            mem_size_param_reg                                  <=   {default_mem_size_second_byte,
default_mem_size_first_byte};  // For more than 16bit address change this line
            mem_size_first_byte        <= default_mem_size_first_byte;
            mem_size_second_byte       <= default_mem_size_second_byte;
            mem_size_third_byte        <= 4'h0;
            // Set_Final_Address_Read
            final_address_read_first_byte_reg   <= default_final_addr_read;
            final_address_read_second_byte_reg  <= 4'h0;
            final_address_read_third_byte_reg   <= 4'h0;
```

```
        final_address_read_param_reg        <= {4'h0, default_final_addr_read};

        diffctrl_param_reg          <= default_diff_ctrl;
        diff_ctrl_reg               <= default_diff_ctrl;
        num_of_blocks_reg           <= default_num_of_blocks;
        num_of_blocks_param_reg     <= default_num_of_blocks;
        phase_interpolator_num_reg  <=  6'b000000;
        txpippmstepsize_reg         <=  5'b00000;
        txpi_param_reg              <= 11'b00000000000;
    end
    else
    begin
        state_reg                   <= state_next;
        test_mode_reg               <= test_mode_next;
        // Memory Size
        mem_size_param_reg          <= mem_size_param_reg_next;
        mem_size_first_byte         <= mem_size_first_byte_next;
        mem_size_second_byte        <= mem_size_second_byte_next;
        mem_size_third_byte         <= mem_size_third_byte_next;
        // Set_Final_Address_Read
        final_address_read_first_byte_reg   <= final_address_read_first_byte_next;
        final_address_read_second_byte_reg  <= final_address_read_second_byte_next;
        final_address_read_third_byte_reg   <= final_address_read_third_byte_next;
        final_address_read_param_reg        <= final_address_read_param_next;

        diffctrl_param_reg          <= diffctrl_param_reg_next;
        diff_ctrl_reg               <= diff_ctrl_reg_next;
        num_of_blocks_reg           <= num_of_blocks_next;
        num_of_blocks_param_reg     <= num_of_blocks_param_next;

        phase_interpolator_num_reg  <= phase_interpolator_num_next;
        txpippmstepsize_reg         <= txpippmstepsize_next;
        txpi_param_reg              <= txpi_param_next;
    end
end
// -------------------------------------------------------------------//
//------------------------ Next-State Logic  ----------------------//
// -------------------------------------------------------------------//
always @ *
begin
    state_next                  = state_reg;
    test_mode_next              = test_mode_reg;
    // Memory Size
    mem_size_param_reg_next     = mem_size_param_reg;
    mem_size_first_byte_next    = mem_size_first_byte;
    mem_size_second_byte_next   = mem_size_second_byte;
    mem_size_third_byte_next    = mem_size_third_byte;
```

```
// Set_Final_Address_Read
final_address_read_first_byte_next  = final_address_read_first_byte_reg;
final_address_read_second_byte_next = final_address_read_second_byte_reg;
final_address_read_third_byte_next  = final_address_read_third_byte_reg;
final_address_read_param_next   = final_address_read_param_reg;
// Differential Voltage Swing Control
diff_ctrl_reg_next          = diff_ctrl_reg;
diffctrl_param_reg_next     = diffctrl_param_reg;
// Number of Blocks
num_of_blocks_next          = num_of_blocks_reg;
num_of_blocks_param_next    = num_of_blocks_param_reg;
// TX Phase Interpolator
phase_interpolator_num_next = phase_interpolator_num_reg;
txpippmstepsize_next        = txpippmstepsize_reg;
txpi_param_next             = txpi_param_reg;


case (state_reg)
    idle:
        begin
            txpi_param_next = 11'b000000000000;
            if (fetch_instruction == 1'b1)
                state_next = instruction_fetch;
            else
                state_next = idle;
        end


    instruction_fetch:
        begin
            case (instruction_code)
                // Instruction: Set_Memory_Size
                Set_Memory_Size:
                    state_next = mem_size;

                // Instruction: Set_Final_Address_Read
                Set_Final_Address_Read:
                    state_next = final_address_read;

                // Instruction: Set_Driver_Swing_Control
                Set_Driver_Swing_Control:
                    state_next = diff_ctrl;

                // Instruction: Set_Soft_Reset
                Set_Soft_Reset:
                    state_next = set_soft_reset;
```

```
            // Instruction: Load_Data
            Load_Data:
                state_next = data_transmission;

            // Instruction: Set_Number_of_Blocks
            Set_Number_of_Blocks:
                state_next = set_number_of_blocks;

            // Instruction: Enable_Phase_Interpolator
            Enable_Phase_Inerpolator:
                state_next = phase_interpolator_enable;

            // Instruction: Set_Test_Mode_ON
            Set_Test_Mode_ON:
                state_next = set_test_mode_on;

            // Instruction: Set_Test_Mode_OFF
            Set_Test_Mode_OFF:
                state_next = set_test_mode_off;

            default:
                state_next = idle;
        endcase
    end

// ------------------------------------------------------------------//
//-------------- Instruction: Set_Test_Mode_ON  --------------------//
// ------------------------------------------------------------------//
set_test_mode_on:
    begin
        test_mode_next = 1'b1;
        state_next = idle;
    end

// ------------------------------------------------------------------//
//---------------- Instruction: Set_Test_Mode_OFF  ----------------//
// ------------------------------------------------------------------//
set_test_mode_off:
    begin
        test_mode_next = 1'b0;
        state_next = idle;
    end

// ------------------------------------------------------------------//
//----------- Instruction: Set_Final_Address_Read  -----------------//
// ------------------------------------------------------------------//
final_address_read:
```

```
            if (fetch_instruction == 1'b1)
            begin
                state_next = first_final_address_read_byte;
                final_address_read_first_byte_next = instruction_code;
            end

        first_final_address_read_byte:
            if (fetch_instruction == 1'b1)
            begin
                state_next = second_final_address_read_byte;
                final_address_read_second_byte_next = instruction_code[3:0];
            end

        second_final_address_read_byte:
            if (fetch_instruction == 1'b1)
            begin
                state_next = third_final_address_read_byte;
                final_address_read_third_byte_next = instruction_code[3:0];
            end

        third_final_address_read_byte:
            if (fetch_instruction == 1'b1)
            begin
                final_address_read_param_next      =      {final_address_read_second_byte_next,
final_address_read_first_byte_next};
                state_next = idle;
            end

        // -------------------------------------------------------------//
        //----------------- Instruction: Set_Memory_Size  ----------------//
        // -------------------------------------------------------------//
        mem_size:
            if (fetch_instruction == 1'b1)
            begin
                state_next = first_mem_size_byte;
                mem_size_first_byte_next = instruction_code;
            end
            else
                state_next = mem_size;

        first_mem_size_byte:
            if (fetch_instruction == 1'b1)
            begin
                state_next = second_mem_size_byte;
                mem_size_second_byte_next = instruction_code;
            end
            else
```

```
                state_next = first_mem_size_byte;

            second_mem_size_byte:
                if (fetch_instruction == 1'b1)
                begin
                    state_next = third_mem_size_byte;
                    mem_size_third_byte_next = instruction_code[3:0];
                end
                else
                    state_next = second_mem_size_byte;

            third_mem_size_byte:
                if (fetch_instruction == 1'b1)
                begin
                    // For more than 16bit address change this line to: mem_size_param_reg_next =
{mem_size_third_byte, mem_size_second_byte, mem_size_first_byte};
                    mem_size_param_reg_next = {mem_size_second_byte, mem_size_first_byte};
                    state_next = idle;
                end
                else
                    state_next = third_mem_size_byte;
            // ------------------------------------------------------------------//
            //---------------- Instruction: Set_Driver_Swing_Control  --------------//
            // ------------------------------------------------------------------//
            diff_ctrl:
                if (fetch_instruction == 1'b1)
                begin
                    state_next = write_diff_ctrl_param;
                    diff_ctrl_reg_next = instruction_code[3:0];
                end
                else
                    state_next = diff_ctrl;

            write_diff_ctrl_param:
                begin
                    diffctrl_param_reg_next = diff_ctrl_reg;
                    state_next = idle;
                end
            // ------------------------------------------------------------//
            //------------------- Instruction: Set_Number_of_Blocks   -------//
            // ------------------------------------------------------------//
            set_number_of_blocks:
                if (fetch_instruction == 1'b1)
                begin
                    state_next = write_num_of_blocks;
                    num_of_blocks_next = instruction_code[NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0];
                end
```

```
    else
        state_next = set_number_of_blocks;

write_num_of_blocks:
    begin
        num_of_blocks_param_next = num_of_blocks_reg;
        state_next = idle;
    end
// ------------------------------------------------------------------------------//
//--------------------------- Instruction: Set_Soft_Reset   ----------------------//
// ------------------------------------------------------------------------------//
set_soft_reset:
    state_next = idle;
// ------------------------------------------------------------------------------//
//--------------------------- Instruction: Load_Data   -----------------------------//
// ------------------------------------------------------------------------------//
data_transmission:
    state_next = end_of_data_transmission;

end_of_data_transmission:
    begin
        if (data_transmission_complete == 1'b1)
        begin
            state_next = idle;
        end
        else
            state_next = end_of_data_transmission;
    end

// ------------------------------------------------------------------------------//
//---------------- Instruction: Enable_Phase_Interpolator   ----------------------//
// ------------------------------------------------------------------------------//
phase_interpolator_enable:
    if (fetch_instruction == 1'b1)
    begin
        state_next = phase_interpolator_num;
        phase_interpolator_num_next = instruction_code[4:0];
    end
    else
        state_next = phase_interpolator_enable;


phase_interpolator_num:
    if (fetch_instruction == 1'b1)
    begin
        state_next = txpippmstepsize;
        txpippmstepsize_next = instruction_code[4:0];
```

```
                end
                else
                    state_next = phase_interpolator_num;


           txpippmstepsize:
                begin
                    txpi_param_next = {phase_interpolator_num_reg, txpippmstepsize_reg, 1'b1};
                    state_next = idle;
                end
            // Default Next State is idle.
            default:
                state_next = idle;
        endcase
    end
    // -------------------------------------------------------------------------------//
    //---------------------------------- Output Logic ---------------------------------//
    // -------------------------------------------------------------------------------//
    assign mem_size_param               = mem_size_param_reg;
    assign final_address_read_param     = final_address_read_param_reg;
    assign test_mode            = test_mode_reg;
    assign diffctrl_param       = diffctrl_param_reg;
    assign num_blocks_param     = num_of_blocks_param_reg;
    assign txpi_param           = txpi_param_next;

    assign soft_reset           = (state_reg == set_soft_reset);
    assign start_data_loading   = (state_reg == data_transmission);

    // Debugging
    assign current_state = state_reg;
    assign first_byte    = mem_size_first_byte;
    assign second_byte   = mem_size_second_byte;
    assign third_byte    = mem_size_third_byte;


    // log2 constant function
    function integer log2(input integer n);
        integer i;
    begin
        log2 = 1;
        for (i = 0; 2 ** i < n; i = i + 1)
            log2 = i + 1;
    end
    endfunction
endmodule
```

7.1.2 Data_Holder

```
module data_hold #
(
        parameter WIDTH = 11,
        parameter HOLD_CYCLES = 100
)
(
        input wire clk, reset,
        input wire  [WIDTH - 1 : 0] async_in,
        output wire [WIDTH - 1 : 0] out
);

        localparam[1:0]
                idle             = 2'b00,
                start            = 2'b01,
                count_up = 2'b11;

        reg [1:0]                        state_reg, state_next;
        reg [10:0]                       count_reg, count_next;
        reg [WIDTH - 1 : 0] out_reg,   out_next;

        always @ (posedge clk, posedge reset)
                if (reset)
                begin
                        state_reg <= idle;
                        count_reg <= 0;
                        out_reg   <= 0;
                end
                else
                begin
                        count_reg    <= count_next;
                        out_reg      <= out_next;
                        state_reg    <= state_next;
                end

        // Next-State Logic
        always @ *
        begin
                count_next = count_reg;
                out_next   = out_reg;
                state_next = state_reg;

                case(state_reg)
                        idle:
                                if (async_in[0] == 1'b0)
                                begin
```

```
                                        out_next = 0;
                                        state_next = start;
                                        count_next = 0;
                                end

                        start:
                                if (async_in[0] == 1'b1)
                                begin
                                        out_next = async_in;
                                        state_next = count_up;
                                        count_next = 0;
                                end

                        count_up:
                                if (count_reg == HOLD_CYCLES)
                                begin
                                        count_next = 0;
                                        state_next = idle;
                                end
                                else
                                        count_next = count_reg + 1;
                endcase
        end

        assign out = (state_reg == count_up) ? out_reg : 0;
endmodule
```

7.1.3 Txpi_decoder

```
module decoder #
(
    parameter NUMBER_OF_BLOCKS = 24
)
(
    input  [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1: 0]         select,
    output [2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1: 0]    out
);
    // Signal Declaration
    localparam NUMBER_OF_BITS_IN_BLOCK_COUNTER = log2(NUMBER_OF_BLOCKS);

    wire [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1: 0] not_select;
    wire [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1: 0] t [2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1 :
0];


    genvar  i,j;
```

```
    for (i = 0; i < NUMBER_OF_BITS_IN_BLOCK_COUNTER; i = i + 1)
        not (not_select[i], select[i]);


    for (i = 0; i < 2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER; i = i + 1)
    begin
        for (j = 0; j < NUMBER_OF_BITS_IN_BLOCK_COUNTER;  j = j + 1)
        begin
            if ((i >> j) % 2 == 0)
                assign t[i][j] = not_select[j];
            else
                assign t[i][j] = select[j];
        end
        assign out[i] = &t[i];
    end

    // log2 constant function
    function integer log2(input integer n);
        integer i;
    begin
        log2 = 1;
        for (i = 0; 2 ** i < n; i = i + 1)
            log2 = i + 1;
    end
    endfunction
endmodule
```

7.1.4 UART_RX

```
module uart_rx #
(
    // Number of data bits
    parameter DBIT = 8,
    // Number of ticks for stop bits
    parameter SB_TICK = 16
)
(
    input wire clk, reset,
    input wire rx, s_tick,
    output reg rx_done_tick,
    output wire [7:0] dout
);
```

```
// Symbolic State Declaration
localparam [1:0]
    idle  = 2'b00,
    start = 2'b01,
    data  = 2'b10,
    stop  = 2'b11;


// Signal Declaration
reg [1:0] state_reg, state_next;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;


// body
// FSMD state & data registers
always @ (posedge clk, posedge reset)
    if (reset)
    begin
        state_reg <= idle;
        s_reg <= 0;
        n_reg <= 0;
        b_reg <= 0;
    end
    else
    begin
        state_reg <= state_next;
        s_reg <= s_next;
        n_reg <= n_next;
        b_reg <= b_next;
    end


// FSMD next-state logic
always @ *
begin
    state_next = state_reg;
    rx_done_tick = 1'b0;
    s_next = s_reg;
    n_next = n_reg;
    b_next = b_reg;

    case (state_reg)
        idle:
            if (~rx)
            begin
```

```
                state_next = start;
                s_next = 0;
            end

        start:
            if (s_tick)
                if (s_reg == 7)
                begin
                    state_next = data;
                    s_next = 0;
                    n_next = 0;
                end
            else
                s_next = s_reg + 1;

        data:
            if (s_tick)
                if (s_reg == 15)
                    begin
                        s_next = 0;
                        b_next = {rx, b_reg[7:1]};
                        if (n_reg == (DBIT - 1))
                            state_next = stop;
                        else
                            n_next = n_reg + 1;
                    end
                else
                    s_next = s_reg + 1;

        stop:
            if (s_tick)
                if (s_reg == (SB_TICK - 1))
                begin
                    state_next = idle;
                    rx_done_tick = 1'b1;
                end
                else
                    s_next = s_reg + 1;
        endcase
    end

    // output
    assign dout = b_reg;
endmodule
```

7.1.5 Baud Rate Generator

`timescale 1ns / 1ps

```
/*
    The baud rate generator generates a sampling signal whose frequency is exactly 16 times
    the UART's designated baud rate. To avoid creating a new clock domain and violating the
    synchronous design principle, the sampling signal should function as enable ticks rather
    than the clock signal to the UART receiver.

    For the 19200 baud rate, the sampling rate has to be 19200 * 16 = 307200 ticks per second.
    Since the system clock rate is 50MHz, the baud rate generator needs a mod-163 counter
    (50 * 10^6) / 307200 = 162., in which a one-clock-cycle tick is asserted once every 163
    clock cycles. The parameterized mom-m counter can be used for this purpose by setting the
    M parameter to 163.
*/

module mod_m_counter #
(
    // M parameter specifies the limit
    parameter M = 14    // Mod-M
)
(
    input wire clk, reset,
    output wire max_tick,
    output  wire [log2(M) - 1:0] q
);

    // Signal Declaration
    localparam N = log2(M);
    reg  [N-1:0] r_reg;
    wire [N-1:0] r_next;

    // Regiser
    always @ (posedge clk, posedge reset)
        if (reset)
            r_reg <= 0;
        else
            r_reg <= r_next;

    // Next-State Logic
    assign r_next = (r_reg == (M - 1)) ? 0 : r_reg + 1;

    // Output Logic
    assign q = r_reg;
    assign max_tick = (r_reg == (M - 1)) ? 1'b1 : 1'b0;
```

```
    // log2 constant function
    function integer log2(input integer n);
        integer i;
    begin
        log2 = i;
        for (i = 0; 2 ** i < n; i = i + 1)
            log2 = i + 1;
    end
    endfunction
endmodule
```

7.1.6 Memory Manager

```
module memory_manager #
(
    // Memory Blocks
    parameter ADDR_WIDTH = 16,
    parameter NUMBER_OF_BLOCKS = 24
)
(
    input wire clka, clkb,
    input wire reset,
    input wire start,
    // limits
    input wire [4:0]  number_of_blocks,
    input wire [ADDR_WIDTH - 1:0] final_addr_write,
    input wire [ADDR_WIDTH - 3:0] final_addr_read,
    // uart
    input wire        rx_done_tick,
    input wire [7:0]  rx_data_out,
    // output
    output wire [63:0] clk_data_out,
    output wire [64 * (NUMBER_OF_BLOCKS - 1) - 1:0] data_out_b_extended,
    output wire        data_transmission_complete,

    // Receiver Data Write Control Signals
    input  wire test_mode,
    output wire start_writing,
    output wire test_completed,

    // Testing
    output wire [ADDR_WIDTH - 1:0] address_a
);

    localparam NUMBER_OF_BITS_IN_BLOCK_COUNTER = log2(NUMBER_OF_BLOCKS);
```

```
// Signal Declaration
wire [(2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER) - 1:0] block_enable;

wire [ADDR_WIDTH - 3:0] address_b;
wire [ADDR_WIDTH - 4:0] address_b_clk;

wire [2:0] current_state_db;
wire [(NUMBER_OF_BLOCKS - 1) * 32 - 1:0] data_out_b;



// Memory Loader Instantiation
mem_loader2 #
(
    .ADDR_WIDTH(ADDR_WIDTH),
    .NUMBER_OF_BLOCKS(NUMBER_OF_BLOCKS)
)
memory_loader
(
    .clk(clka),
    .reset(reset),

    .start(start),

    .init_addr(0),
    .final_addr(final_addr_write),
    .rx_done_tick(rx_done_tick),
    .num_of_blocks(number_of_blocks),

    .address(address_a),
    .data_transmission_complete(data_transmission_complete),
    .block_enable(block_enable),

    // Testing Only
    .current_state_db(current_state_db)
);

// Memory Blocks Instantiation
memory_blocks #
(
    .ADDR_WIDTH(ADDR_WIDTH),
    .NUMBER_OF_BLOCKS(NUMBER_OF_BLOCKS)
)
mem_blocks
(
    .clka(clka),
    .clkb(clkb),
```

```
    .write_enable(rx_done_tick),
    .data_in(rx_data_out),
    .address_a(address_a),
    .block_enable(block_enable),
    .addrb(address_b),
    .address_b_clk(address_b_clk),
    .clk_data_out(clk_data_out),
    .data_out(data_out_b)
);


// Instantiation of Data Extention
data_extention #
(
    .NUMBER_OF_BLOCKS(NUMBER_OF_BLOCKS)
)
data_extention_32_to_64
(
    .data_in(data_out_b),
    .data_out(data_out_b_extended)
);

read_counter #
(
    .ADDR_WIDTH(ADDR_WIDTH)
)
read_counter_unit
(
    // Inputs
    .clk(clkb),
    .reset(reset),
    .test_mode(test_mode),
    .final_addr_read(final_addr_read),
    // Outputs
    .address_b(address_b),
    .address_b_clk(address_b_clk),
    .start_writing(start_writing),
    .test_completed(test_completed)
);

// log2 constant function
function integer log2(input integer n);
    integer i;
begin
    log2 = 1;
    for (i = 0; 2 ** i < n; i = i + 1)
        log2 = i + 1;
```

```
    end
    endfunction
endmodule
```

7.1.6.1 Memory Loader

```
module mem_loader2 #
(
    parameter ADDR_WIDTH = 16,
    parameter NUMBER_OF_BLOCKS = 24
)
(
    input wire clk, reset,

    input wire start,

    input wire [ADDR_WIDTH - 1:0] init_addr,
    input wire [ADDR_WIDTH - 1:0] final_addr,

    input wire          rx_done_tick,
    input wire  [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0]   num_of_blocks,

    output wire [ADDR_WIDTH - 1:0] address,
    output wire data_transmission_complete,
    output reg  [(2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER) - 1:0] block_enable,

    // Testing Only
    output wire [2:0] current_state_db
);

    localparam NUMBER_OF_BITS_IN_BLOCK_COUNTER = log2(NUMBER_OF_BLOCKS);

    // Symbolic State Declaration
    localparam[2:0]
        initial_state           = 3'b000,
        check_num_of_blocks     = 3'b001,
        check_num_of_words      = 3'b010,
        wait_for_rx_done_tick   = 3'b011,
        load_complete           = 3'b100;

    // Registers
    reg [2:0] state_reg, state_next;

    reg [ADDR_WIDTH - 1:0] address_reg;
    reg [ADDR_WIDTH - 1:0] address_next;
```

```
reg [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0] block_count_reg;
reg [NUMBER_OF_BITS_IN_BLOCK_COUNTER - 1:0] block_count_next;


always @ (posedge clk, posedge reset)
    if (reset)
    begin
        state_reg <= initial_state;
        address_reg <= init_addr;
        block_count_reg <= 0;
    end
    else
    begin
        state_reg <= state_next;
        address_reg <= address_next;
        block_count_reg <= block_count_next;
    end


always @ *
    begin
        state_next          = state_reg;
        address_next        = address_reg;
        block_count_next    = block_count_reg;

        case(state_reg)
            initial_state:
                begin
                    address_next = init_addr;
                    block_count_next = 0;
                    if (start == 1'b1)
                        state_next = check_num_of_blocks;
                    else
                        state_next = initial_state;
                end

            check_num_of_blocks:
                begin
                    address_next = init_addr;
                    if (block_count_reg < num_of_blocks)
                    begin
                        state_next = check_num_of_words;
                    end

                    else
                        state_next = load_complete;
```

```
                end

            check_num_of_words:
                begin
                    if (address < final_addr)
                    begin
                        state_next = wait_for_rx_done_tick;
                    end

                    else
                    begin
                        state_next = check_num_of_blocks;
                        block_count_next = block_count_reg + 1;
                    end
                end

            wait_for_rx_done_tick:
                begin
                    if (rx_done_tick == 1'b1)
                    begin
                        state_next = check_num_of_words;
                        address_next = address_reg + 1;
                    end
                    else
                        state_next = wait_for_rx_done_tick;
                end

            load_complete:
                state_next = initial_state;

            default:
                state_next = initial_state;
        endcase
    end


// One-Hot Decoder
always @ *
    begin
        // In normal operation, after loading, all memory block enables are deasserted.
        if (state_reg == initial_state)
            block_enable = 0;
        else
        begin
            block_enable = 0;
            block_enable[block_count_reg] = 1'b1;
        end
```

```
        end

    // Output Logic
    assign address = address_reg;
    assign data_transmission_complete = (state_reg == load_complete) ? 1'b1 : 1'b0;



    // Debugging
    assign current_state_db = state_reg;



    // log2 constant function
    function integer log2(input integer n);
       integer i;
    begin
       log2 = 1;
       for (i = 0; 2 ** i < n; i = i + 1)
           log2 = i + 1;
    end
    endfunction
endmodule
```

7.1.6.2 Memory Blocks

```
module memory_blocks #
(
    parameter ADDR_WIDTH = 16,
    parameter NUMBER_OF_BLOCKS = 24
)
(
    // Inputs
    input  wire clka, clkb,
    input  wire write_enable,
    input  wire [7:0] data_in,
    input  wire [ADDR_WIDTH - 1:0] address_a,
    input  wire [(2 ** NUMBER_OF_BITS_IN_BLOCK_COUNTER) - 1:0] block_enable,
    input  wire [ADDR_WIDTH - 3:0] addrb,
    input  wire [ADDR_WIDTH - 4:0] address_b_clk,
    // Data Outputs
    output wire [63:0] clk_data_out,
    output wire [(NUMBER_OF_BLOCKS - 1) * 32 - 1:0] data_out
);

    // Signal Declaration
    localparam NUMBER_OF_BITS_IN_BLOCK_COUNTER = log2(NUMBER_OF_BLOCKS);
    genvar i;
```

```
    blk_mem_gen_clock   block_memory_clock
    (
        .clka(clka),
        .ena(block_enable[0]),
        .wea(write_enable),
        .addra(address_a),
        .dina(data_in),
        .clkb(clkb),
        // The address space of block_memory_clock is half of block_memory.
        .addrb(address_b_clk),
        .doutb(clk_data_out)
    );

    generate
        for (i = 1; i < NUMBER_OF_BLOCKS; i = i + 1)
        begin: mem_block
            blk_mem_gen_0 block_memory
            (
                .clka(clka),                                        // input wire clka
                .ena(block_enable[i]),                              // input wire ena
                .wea(write_enable),                                 // input wire [0 : 0] wea
                .addra(address_a),                                  // input wire [19 : 0] addra
                .dina(data_in),                                     // input wire [7 : 0] dina
                .clkb(clkb),                                        // input wire clkb
                .addrb(addrb),                                      // input wire [17 : 0] addrb
                .doutb(data_out[(32 * (i - 1)) + 31:(32 * (i - 1))])   // output wire [31 : 0]
doutb
            );
        end
    endgenerate




    // log2 constant function
    function integer log2(input integer n);
        integer i;
    begin
        log2 = 1;
        for (i = 0; 2 ** i < n; i = i + 1)
            log2 = i + 1;
    end
    endfunction
endmodule
```

7.1.6.3 Data Extention

```
module data_extention #
(
    parameter NUMBER_OF_BLOCKS = 24
)
(
         input  wire [((NUMBER_OF_BLOCKS - 1) * 32) - 1:0] data_in,
         output wire [((NUMBER_OF_BLOCKS - 1) * 64) - 1:0] data_out
);

    genvar i;

    generate
        for (i = 0; i <= (((NUMBER_OF_BLOCKS - 1) * 32) - 1); i = i + 1)
        begin: Verkabelung
           assign data_out[2*i] = data_in[i];
           assign data_out[2*i + 1] = data_in[i];
        end
    endgenerate
endmodule
```

7.1.6.4 Read Counter

```
module read_counter #
(
         parameter ADDR_WIDTH = 16
)
(
         input  wire clk, reset,
         input  wire test_mode,
         input  wire [ADDR_WIDTH - 3:0] final_addr_read,

         output wire [ADDR_WIDTH - 3:0] address_b,
    output wire [ADDR_WIDTH - 4:0] address_b_clk,

         output wire                    start_writing,
         output wire                    test_completed
);
    // Symbolic State Declaration
    localparam[2:0]
        idle                = 3'b000,
        normal_mode         = 3'b001,
        testing_mode        = 3'b010,
        start_test          = 3'b011,
        assert_test_completed = 3'b100,
```

```
    end_of_test           = 3'b101;

wire [ADDR_WIDTH - 3:0] div2_test;
wire [ADDR_WIDTH - 3:0] final_adrr_read_even;
wire [ADDR_WIDTH - 4:0] final_addr_read_clk;

// Signal Declaration
reg  [2:0]              state_reg, state_next;
     reg  [ADDR_WIDTH - 3:0] count_reg, count_next;
     reg  [ADDR_WIDTH - 3:0] count_clk_reg, count_clk_next;

assign final_adrr_read_even = make_odd(final_addr_read);
assign final_addr_read_clk = final_adrr_read_even / 2;

     // FSMD state & data registers
     always @ (posedge clk, posedge reset)
        if (reset)
        begin
            state_reg <= idle;
            count_reg <= 0;
            count_clk_reg <= 0;
        end
        else
        begin
            state_reg     <= state_next;
            count_reg     <= count_next;
            count_clk_reg <= count_clk_next;
   end

// FSMD next-state logic
always @ *
begin
    state_next = state_reg;
    count_next = count_reg;

    case (state_reg)
        idle:
            if (~test_mode)
            begin
                state_next = normal_mode;
                count_next = 0;
                count_clk_next = 0;
            end
            else
            begin
                state_next = start_test;
                count_next = 0;
```

```
        count_clk_next = 0;
    end

normal_mode:
    if (test_mode)
    begin
        state_next = start_test;
        count_next = 0;
        count_clk_next = 0;
    end
    else
    begin
        if (count_clk_reg == final_addr_read_clk)
            count_clk_next = 0;
        else
            count_clk_next = count_clk_reg + 1;

        if (count_reg == final_adrr_read_even)
            count_next = 0;
        else
            count_next = count_reg + 1;
    end
start_test:
    begin
        state_next = testing_mode;
        count_next = count_reg + 1;
        count_clk_next = count_clk_reg + 1;
    end
testing_mode:
    begin
        if (count_clk_reg == final_addr_read_clk)
            count_clk_next = 0;
        else
            count_clk_next = count_clk_reg + 1;

        if (count_reg == final_adrr_read_even)
            state_next = assert_test_completed;
        else
            count_next = count_reg + 1;
    end


assert_test_completed:
    begin
        count_next = 0;
        count_clk_next = 0;
        state_next = end_of_test;
```

```
                end

            end_of_test:
                begin
                    if (test_mode)
                    begin
                        state_next = end_of_test;
                    end
                    else
                        state_next = idle;
                end
        endcase
    end

    // Output Logic
    assign address_b     = count_reg;
    assign address_b_clk = count_clk_reg;

    assign start_writing  = (state_reg == start_test) ? 1'b1 : 1'b0;
    assign test_completed = (state_reg == assert_test_completed) ? 1'b1 : 1'b0;


    // log2 constant function
    function integer log2(input integer n);
        integer i;
    begin
        log2 = 1;
        for (i = 0; 2 ** i < n; i = i + 1)
            log2 = i + 1;
    end
    endfunction

    function integer make_odd(input integer n);
    begin
        if (n % 2 == 0)
            make_odd = n - 1;
        else
            make_odd = n;
    end
    endfunction
endmodule
```

7.2 Write and Transmit Received Data

```
module write_receiver_data #
(
    parameter WRITE_ADDR_WIDTH = 9,
    parameter READ_ADDR_WIDTH = WRITE_ADDR_WIDTH + 3,
    parameter DBIT = 8,
    parameter SB_TICK = 16,
    parameter DVSR = 14
)
(
    input clk, reset,
    input gt0_rxusrclk2_i,
    input start_writting,
    input start_transmission,
    input [63:0] gt_rx_data,
    input [WRITE_ADDR_WIDTH - 1:0] final_addr_write,
    input [READ_ADDR_WIDTH - 1:0]  final_addr_read,

    output tx
);

    // Block RAM Write Enable
    wire write_enable;
    // Block RAM PORTA addra
    wire [WRITE_ADDR_WIDTH - 1:0] addra;
    // Start UART_TX (send an 8bit word)
    wire tx_start;
    // Every time an 8bit word is transmitted by UART_TX this signal is asseted
    wire continue_count;
    // Start Address Counter of PORTB for transmission of data to the PC
    //wire start_address_counter;
    // Block RAM PORTB addrb
    wire [READ_ADDR_WIDTH - 1:0] read_addr;
    // Baud Rate Generator
    wire tick;
    // Start Read Counter
    // wire start_writting_pulse;
    // 8bit word to be transmitted by UART_TX. BRAM -> uart_tx
    wire [7:0] tx_data_in;



    write_receiver_data_counter #
    (
        .ADDR_WIDTH(WRITE_ADDR_WIDTH)
    )
```

```
write_rx_data_counter_unit
(
    // Inputs
    .clk(gt0_rxusrclk2_i),
    .reset(reset),
    .start_writting(start_writting),
    .init_addr_write(0),
    .final_addr_write(final_addr_write),
    // Outputs
    .write_enable(write_enable),
    .addra(addra)
);

blk_mem_receiver bram_receiver
(
    // PortA - Write Port.
    .clka(gt0_rxusrclk2_i),
    .wea(write_enable),
    .addra(addra),
    .dina(gt_rx_data),
    // PortB - Read Port.
    .clkb(clk),
    .addrb(read_addr),
    .doutb(tx_data_in)
);



uart_tx #
(
    .DBIT(DBIT),
    .SB_TICK(SB_TICK)
)
uart_tx_unit
(
    .clk(clk),
    .reset(reset),
    .tx_start(tx_start),
    .s_tick(tick),
    .data_in(tx_data_in),
    .tx(tx),
    .tx_done_tick(continue_count)
);

// TX address counter
address_counter #
(
    .ADDR_WIDTH(READ_ADDR_WIDTH)
```

```
    )
    addr_counter
    (
        .clk(clk),
        .reset(reset),
        .start(start_transmission),
        .init_addr(0),
        .final_addr(final_addr_read),
        .continue_count(continue_count),

        .tx_start_tick(tx_start),
        .count(read_addr),

        // Testing Only
        .current_state(),
        .count_next_i()
    );


     mod_m_counter #
    (
        .M(DVSR)
    )
    baud_gen_unit_tx
    (
        .clk(clk),
        .reset(reset),
        .q(),
        .max_tick(tick)
    );


endmodule
```

7.2.1 Received Data Counter

```
`timescale 1ns / 1ps


module write_receiver_data_counter #
(
    parameter ADDR_WIDTH = 9
)
(
    input wire clk, reset,
    input wire start_writting,
```

```
    input wire [ADDR_WIDTH - 1:0] init_addr_write,
    input wire [ADDR_WIDTH - 1:0] final_addr_write,

    output wire write_enable,
    output wire [ADDR_WIDTH - 1:0] addra
);

    // Symbolic State Declaration
    localparam [0:0]
        idle  = 1'b0,
        write = 1'b1;

    reg state_reg, state_next;
    reg [ADDR_WIDTH - 1 : 0] addra_reg, addra_next;

    always @ (posedge clk, posedge reset)
    begin
        if (reset)
        begin
            state_reg <= idle;
            addra_reg <= init_addr_write;
        end
        else
        begin
            state_reg <= state_next;
            addra_reg <= addra_next;
        end
    end

    // Next-State Logic
    always @ *
        begin
            state_next = state_reg;
            addra_next = addra_reg;

            case (state_reg)
                idle:
                    begin
                        addra_next = init_addr_write;
                        if (start_writting == 1'b1)
                        begin
                            state_next = write;
                        end
                        else
                        begin
                            state_next = idle;
                        end
```

```
                end

            write:
                begin
                    if (addra_reg < final_addr_write)
                    begin
                        state_next = write;
                        addra_next = addra_reg + 1;
                    end
                    else
                    begin
                        state_next = idle;
                    end
                end

            default:
                state_next = idle;
        endcase
    end

    // Output Logic
    assign write_enable = (state_reg == write);
    assign addra = addra_reg;

endmodule
```

7.2.2 UART TX

```
module uart_tx #
(
    parameter DBIT    = 8,          // Number of data bits
    parameter SB_TICK = 16          // Number of ticks for stop bits
)
(
    input wire clk, reset,
    input wire tx_start, s_tick,
    input wire [7:0] data_in,

    output reg tx_done_tick,
    output wire tx
);

    // Symbolic State Declaration
    localparam [1:0]
        idle  = 2'b00,
        start = 2'b01,
```

```
    data  = 2'b10,
    stop  = 2'b11;


// Signal Declaration
reg [1:0] state_reg, state_next;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;
reg tx_reg, tx_next;


// Body
// FSMD state & data registers
always @ (posedge clk, posedge reset)
    if (reset)
    begin
        state_reg <= idle;
        s_reg <= 0;
        n_reg <= 0;
        b_reg <= 0;
        tx_reg <= 1'b1;
    end
    else
    begin
        state_reg <= state_next;
        s_reg  <= s_next;
        n_reg  <= n_next;
        b_reg  <= b_next;
        tx_reg <= tx_next;
    end


// FSMD next-state logic & functional units
always @ *
begin
    state_next   = state_reg;
    tx_done_tick = 1'b0;
    s_next  = s_reg;
    n_next  = n_reg;
    b_next  = b_reg;
    tx_next = tx_reg;

    case (state_reg)
        idle:
            begin
                tx_next = 1'b1;
```

```
        if (tx_start)
        begin
            state_next = start;
            s_next = 0;
            b_next = data_in;
        end
    end

start:
    begin
        tx_next = 1'b0;
        if (s_tick)
            if (s_reg == 15)
            begin
                state_next = data;
                s_next = 0;
                n_next = 0;
            end
            else
                s_next = s_reg + 1;
    end

data:
    begin
        tx_next = b_reg[0];
        if (s_tick)
            if (s_reg == 15)
            begin
                s_next = 0;
                b_next = b_reg >> 1;

                if (n_reg == 7)
                    state_next = stop;
                else
                    n_next = n_reg + 1;
            end
        else
            s_next = s_reg + 1;
    end

stop:
    begin
        tx_next = 1'b1;
        if (s_tick)
            if (s_reg == 15)
            begin
                state_next = idle;
```

```
                        tx_done_tick = 1'b1;
                    end
                else
                    s_next = s_reg + 1;
            end
        endcase
    end

    // Output
    assign tx = tx_reg;


endmodule
```

7.2.3 Address Counter

```
module address_counter #
(
    parameter ADDR_WIDTH = 12
)
(
    input wire clk, reset,
    input wire start,
    input wire [ADDR_WIDTH - 1:0] init_addr,
    input wire [ADDR_WIDTH - 1:0] final_addr,
    input wire continue_count,

    output wire [ADDR_WIDTH - 1:0] count,
    output tx_start_tick,

    // Testing Only
    output wire [1:0] current_state,
    output wire [ADDR_WIDTH - 1:0] count_next_i
);

    // Symbolic State Declaration
    localparam [1:0]
        idle       = 2'b00,
        set_tick   = 2'b01,
        hand_shake = 2'b10;

    // Signal Declarationh
    reg [1:0] state_reg, state_next;
    reg [ADDR_WIDTH - 1:0] count_reg;
    reg [ADDR_WIDTH - 1:0] count_next;
```

```
// State Register
always @ (posedge clk, posedge reset)
begin
    if (reset)
    begin
        state_reg <= idle;
        count_reg <= init_addr;
    end
    else
    begin
        state_reg <= state_next;
        count_reg <= count_next;
    end
end

// Next-State Logic
always @ *
    begin
        state_next = state_reg;
        count_next = count_reg;

        case (state_reg)
            idle:
                begin
                    count_next = init_addr;
                    if (start == 1'b1)
                    begin
                        state_next = set_tick;
                    end
                    else
                        state_next = idle;
                end

            set_tick:
                if (count_reg < final_addr)
                begin
                    state_next = hand_shake;
                    count_next = count_reg + 1;
                end
                else
                    state_next = idle;

            hand_shake:
                if (continue_count == 1'b1)
                    state_next = set_tick;
                else
                    state_next = hand_shake;
```

```
                default:
                    state_next = idle;
            endcase
        end

    // Output Logic
    assign tx_start_tick = (state_reg == set_tick);
    assign count = count_reg;

    // Only for testing
    assign current_state = state_reg;
    assign count_next_i = count_next;
endmodule
```

7.3 Tcl Script για τον προγραμματισμό του SuperClock Module2

```
# This is the frequency the SCM2 will be configured to
################################################################################
set freq 265.00
################################################################################

#
# get_valid_code - table lookup to find proper ROM code give the frequency (in MHz)
#
proc get_valid_code { freq } {
        set freqs [list 161.13 81.25 162.50 325.00 650.00 173.37 61.44 122.88 153.63 245.76 \
                    491.52 67.50 81.00 135.00 162.00 106.25 212.50 425.00 62.50 125.00 \
                    250.00 500.00 187.50 132.81 195.31 265.63 390.63 531.25 76.80 153.60 \
                    307.20 614.40 19.33 77.76 155.52 311.04 622.08 166.63 333.26 666.51 \
                    666.75 167.33 669.31 168.05 174.69 100.00 125.00 250.00 75.00 150.00 \
                    300.00 600.00 74.25 148.50 297.00 594.00 167.06 334.13 668.25 78.13 \
                    156.25 312.50 625.00 66.67 133.33 166.67 266.67 333.33 533.33 644.00 \
                    666.67 205.00 210.00 215.00 220.00 225.00 230.00 235.00 240.00 245.00 \
                    250.00 255.00 260.00 265.00 270.00 275.00 280.00 285.00 290.00 295.00 \
                    300.00 305.00 310.00 315.00 320.00 325.00 330.00 335.00 340.00 345.00 \
                    350.00 355.00 360.00 365.00 370.00 375.00 380.00 385.00 390.00 395.00 \
                    400.00 405.00 410.00 415.00 420.00 425.00 430.00 435.00 440.00 445.00 \
                    450.00 455.00 460.00 465.00 470.00 475.00 480.00 485.00]
  if {$freq == -1} {
        return -1
  } else {
        set freq_code [lsearch -exact $freqs $freq]
```

```
        if {$freq_code == -1} {
                puts "No valid code for frequency: $freq!"
        } else {
                puts "found $freq, will use code $freq_code"
        }
  }
        return $freq_code
}

#
# configure_scm2 - configures the VIO 2.0 core in the design to the specified
# frequency, and starts both the Si570 and Si5368 to that frequency.  It also
# checks the frequency counters to make sure the proper frequency is being
# detected.
#
proc configure_scm2 { freq } {
  set freq_code [get_valid_code $freq]
  if {$freq_code == -1} {
        puts "invalid frequency code!  Exiting..."
        exit
  }
        set_property             OUTPUT_VALUE_RADIX             UNSIGNED             [get_hw_probes
{super_clock/u_vio_sclk2_control/si570_addr}]
        set_property             OUTPUT_VALUE_RADIX             UNSIGNED             [get_hw_probes
{super_clock/u_vio_sclk2_control/si5368_addr}]
        set_property              INPUT_VALUE_RADIX             UNSIGNED             [get_hw_probes
{super_clock/u_vio_sclk2_control/si570_freq}]
        set_property              INPUT_VALUE_RADIX             UNSIGNED             [get_hw_probes
{super_clock/u_vio_sclk2_control/si5368_freq}]

        set_property               OUTPUT_VALUE              $freq_code              [get_hw_probes
{super_clock/u_vio_sclk2_control/si5368_addr}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_addr}]

        set_property OUTPUT_VALUE 45 [get_hw_probes {super_clock/u_vio_sclk2_control/si570_addr}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si570_addr}]

        set_property OUTPUT_VALUE 08 [get_hw_probes {super_clock/u_vio_sclk2_control/sclk_out_w}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/sclk_out_w}]

        set_property OUTPUT_VALUE 1 [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_clkin}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_clkin}]

        set_property OUTPUT_VALUE 55 [get_hw_probes super_clock/u_vio_sclk2_control/si570_idcode]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si570_idcode}]

        set_property OUTPUT_VALUE 1 [get_hw_probes {super_clock/u_vio_sclk2_control/si570_start}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si570_start}]
```

```
        set_property OUTPUT_VALUE 0 [get_hw_probes {super_clock/u_vio_sclk2_control/si570_start}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si570_start}]

        set_property OUTPUT_VALUE 1 [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_start}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_start}]

        set_property OUTPUT_VALUE 0 [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_start}]
        commit_hw_vio [get_hw_probes {super_clock/u_vio_sclk2_control/si5368_start}]

        refresh_hw_vio [get_hw_vios {hw_vio_1}]

        # vio input doesn't have decimal places
        set expected_freq [expr $freq * 1000]

        set          si570_freq          [get_property          INPUT_VALUE          [get_hw_probes
{super_clock/u_vio_sclk2_control/si570_freq}]]
        set          si5368_freq          [get_property          INPUT_VALUE          [get_hw_probes
{super_clock/u_vio_sclk2_control/si5368_freq}]]

         # The script input has 5 digits of precision and the VIO console has 6.  Round off the
VIO value
    set si570_freq [expr round([expr $si570_freq/10])]
    set si5368_freq [expr round([expr $si5368_freq/10])]
    set si570_freq [expr $si570_freq * 10]
    set si5368_freq [expr $si5368_freq * 10]

        if {$expected_freq == $si570_freq} {
                puts "Si570 has expected frequency: $expected_freq"
        } else {
                puts "Si570 does not have expected frequency (${expected_freq}) : $si570_freq"
        }

        if {$expected_freq == $si5368_freq} {
                puts "Si5368 has expected frequency: $expected_freq"
        } else {
                puts "Si5368 does not have expected frequency (${expected_freq}) : $si5368_freq"
        }

}

##################################################################################
#
#  MAIN
#
##################################################################################
```

```
refresh_hw_device [lindex [get_hw_devices] 1]
refresh_hw_vio -update_output_values [get_hw_vios {hw_vio_1}]
configure_scm2 $freq

refresh_hw_vio [get_hw_vios {hw_vio_1}]
```

8

Κώδικας Συναρτήσεων Βιβλιοθήκης σε MATLAB

8.1 Εγκαδίδρυση Σειριακής Σύνδεσης μεταξύ FPGA και υπολογιστή

```
clear all;
clc;

s = serial('COM5','FlowControl','hardware', 'Baudrate', 921600,'Terminator',
'')


set(s, 'InputBufferSize',5000000);
set(s, 'FlowControl', 'none');
set(s, 'BaudRate', 921600);
set(s, 'Parity', 'none');
set(s, 'DataBits', 8);
set(s, 'StopBits', 1);
set(s, 'Timeout', 100);
s.BytesAvailableFcnCount = 1;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @(src, evt)fprintf(2, '%02x ', fread(src, 1) );

fopen(s);
```

8.2 Ορισμός Πλήθους Καναλιών που θα χρησιμοποιηθούν

```
function [ ] = set_number_of_blocks( serial_object, num_of_blocks )
```

```
    % Precondition: 1 <= num_of_blocks <= 24
    if ((num_of_blocks > 24) && (num_of_blocks < 1))
        error('The value of num_of_blocks must be between 1 and 24.')
    end


    % Set the number of memory blocs. The size is a 5bit unsigned number.
    % 2byte instruction: (opcode = 0x43)(Byte)
    % If not used the default number of blocks is 4.
    % Example of use: Set TXDIFFCTRL  to 0x03.
    fwrite(serial_object, hex2dec('46'))           % opcode 0x46
    fwrite(serial_object, num_of_blocks)           % Byte = 0x03
end
```

8.3 Ορισμός Πλήθους byte που θα φορτωθούν στην μνήμη

```
function [ ] = set_memory_size( serial_object, mem_size )


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%% Instruction: Set_Memory_Size %%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Set the size of memory. The size is a 20bit unsigned number.
    % 4byte instruction: (opcode = 0x42)(1st Byte)(2nd Byte)(3rd Byte)
    % If not used the default memory size is 0x001FF = 511.
    % Example of use: Set the memory size to 0x40302.


    most_significant_byte = mem_size(1:2);
    middle_byte = mem_size(3:4);
    least_significant_byte = mem_size(5:6);


    fwrite(serial_object, hex2dec('42'))                              % opcode
0x42 = 66
    fwrite(serial_object, hex2dec(least_significant_byte))             % Least
Significant Byte = 0x02
    fwrite(serial_object, hex2dec(middle_byte))                       % Middle
Byte            = 0x03
    fwrite(serial_object, hex2dec(most_significant_byte))             % Most
Significant Byte  = 0x04
end
```

8.4 Soft Reset

```
function [ ] = set_soft_reset( serial_object )

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%% Instruction: Set_Soft_Reset %%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Assert the soft_reset for one clock period.
    % 1byte instruction: (opcode = 0x44)

    fwrite(serial_object, hex2dec('44'))           % opcode 0x44
end
```

8.5 Ορισμός τελικής διεύθυνσης ανάγνωσης

```
function [  ] = set_final_addr_read( serial_object, final_addr_read )

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%% Instruction: Set_Final_Address_Read %%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Set the size of memory. The size is a 14bit unsigned number.
    % 4byte instruction: (opcode = 0x48)(1st Byte)(2nd Byte)(3rd Byte)
    % If not used the default final address  is 0x0F.
    % Example of use: Set the final address to 0x40302.


    most_significant_byte = final_addr_read(1:2);
    middle_byte = final_addr_read(3:4);
    least_significant_byte = final_addr_read(5:6);


    fwrite(serial_object, hex2dec('48'))                                % opcode
0x48
    fwrite(serial_object, hex2dec(least_significant_byte))          % Least
Significant Byte = 0x02
    fwrite(serial_object, hex2dec(middle_byte))                         % Middle
Byte             = 0x03
    fwrite(serial_object, hex2dec(most_significant_byte))           % Most
Significant Byte  = 0x04
    fwrite(serial_object, hex2dec('00'))
end
```

8.6 Ορισμός της κατάστασης λειτουργίας

```
function [ ] = set_test_mode( serial_object, flag )

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%% Instruction: Set_Final_Address_Read %%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fwrite(serial_object, hex2dec('4a'));

    if (flag == 0)
        fwrite(serial_object, hex2dec('4a'));
    else
        fwrite(serial_object, hex2dec('49'));
    end
end
```

8.7 Ορισμός του επιπέδου τάσης των καναλιών

```
function [ ] = set_driver_swing_control( serial_object, value )


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%% Instruction: Set_Driver_Swing_Control %%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    % TXDIFFCTRL[3:0]   Vppd(V)
    % 4'b0000           0.269
    % 4'b0001           0.336
    % 4'b0010           0.407
    % 4'b0011           0.474
    % 4'b0100           0.543
    % 4'b0101           0.609
    % 4'b0110           0.677
    % 4'b0111           0.741
    % 4'b1000           0.807
    % 4'b1001           0.866
    % 4'b1010           0.924
    % 4'b1011           0.973
    % 4'b1100           1.018
    % 4'b1101           1.056
    % 4'b1110           1.092
    % 4'b1111           1.119

    % Set the TXDIFFCTRL[3:0] port of GTH. The size is a 4bit unsigned
number.
    % 2byte instruction: (opcode = 0x43)(Byte)
    % If not used the default value is 0x0B = 11
    % Example of use: Set TXDIFFCTRL  to 0x0A.

    if (value < 0) || (value > 15)
        error('The value TXDIFFCTRL should be in the range (0, 15)');
    end

    fwrite(serial_object, hex2dec('43'))           % opcode 0x43
    fwrite(serial_object, value)          % Byte = 0x0A


end
```

8.8 *Ενεργοποίηση Ολίσθηση φάσης*

```
function [  ] = txpi_enable( serial_object, transceiver_number,
txpippmstepsize)

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%% Instruction: Enable_Phase_Interpolator %%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Enable the phase interpolator for a single channel.
    % The size of the parameter is a 11bit unsigned number.
    % 3byte instruction: (opcode = 0x47)(1st Byte)(2nd Byte)(3rd Byte)
    % In normal operation default value of the parameter is 11'b00000000000.
    % Example of use: Enable the txpi of gt0:

    % For gt0 use transceiver_number = 0x1E
    fwrite(serial_object, hex2dec('47'));                % opcode 0x42 = 66
    fwrite(serial_object, hex2dec(transceiver_number));  % Least Significant
Byte = 0x02
    fwrite(serial_object, hex2dec(txpippmstepsize));     % Most Significant
Byte  = 0x04
end
```

8.9 *Φόρτωση Δεδομένων*

```
function [ ] = set_given_data(serial_object, final_address,
number_of_blocks, Data)

    % Precondition: final_address >= 4
    if (final_address < 4)
        error('The final address must be at least 4.')
    end


    % Number of 8bit words to be written
    number_of_words = final_address * number_of_blocks;


    % Check if the array Data has the right size.
    if (length(Data) ~= number_of_words)
        error('The size of the given Data array is not right.')
    end

    % Check the type of data. The type must be uint8.
    if (isa(Data, 'uint8') ~= 1)
        error('The given data must be of type uint8.')
    end

    % Set number of blocks of memory
    set_number_of_blocks(serial_object, number_of_blocks);


    % Set final address
    final_address_hex_str = dec2hex(final_address);
    switch length(final_address_hex_str)
        case 1
            fin_address = strcat('00000', final_address_hex_str);
        case 2
            fin_address = strcat('0000', final_address_hex_str);
        case 3
            fin_address = strcat('000', final_address_hex_str);
        case 4
            fin_address = strcat('00', final_address_hex_str);
        case 5
            fin_address = strcat('0', final_address_hex_str);
        case 6
            fin_address = final_address_hex_str;
        otherwise
            fin_address = '001FF';
    end
    set_memory_size(serial_object, fin_address);

    % Set number of blocks of memory
    set_number_of_blocks(serial_object, number_of_blocks);


    % Call Load Data Function to enable writting in the memory blocks
    load_data(serial_object);

    % Send Data
    for i = 1:number_of_words
        fwrite(serial_object, Data(i));
    end
end
```

8.10 Επαλήθευση Συστήματος UART.

Με το παρακάτω script φορτώνονται τυχαία δεδομένα στην μνήμη του FPGA, μεταδίδονται πάλι πίσω στον υπολογιστή και γίνεται σύγγριση μεταξύ τους για επιβεβαίωση της ορθής λειτουργίας του συστήματος σειριακής εκπομπής και λήψης UART.

```
% This script is used for testing the UART serial communication.
%  1. Creates a txt file with random 8bit data(hex), named
'rand_hex.data.txt',
%     the size of of which is determined by the num_of_words parameter.
%  2. Sends the random data from the 'rand_hex_data.txt' via the serial
%     port to the FPGA.
%  3. Reads back the data from the FPGA and stores it to the
%     'data_from_serial.txt' file.
%  4. Optionaly compare the two files with visdiff(f1, f2).


%%

clear;
clc;

% Number of (8bit) words to be written in file.
num_of_words = 512;

% 1. Creates a txt file with random 8bit data(hex), named
'rand_hex.data.txt',
%     the size of of which is determined by the num_of_words parameter.
fileID = fopen('rand_hex_data.txt','w');

for i = 1:num_of_words
    x = randsample('0123456789abcdef', 2, true);
    fprintf(fileID, x);
    fprintf(fileID, '\n');
end

fclose(fileID);


% Read the hex data.
fileID_read = fopen('rand_hex_data.txt', 'r');

if (fileID_read ~= -1)
    formatSpec = '%x';
    A = fscanf(fileID_read, formatSpec);

    fclose(fileID_read);
end


%%
%  2. Sends the random data from the 'rand_hex_data.txt' via the serial
%     port to the FPGA.

%  Open a new file to write data from the serial port
```

```
fileID_serial = fopen('data_from_serial.txt','w');

% Create serial port object.
s = serial('COM5', 'Baudrate', 921600,'Terminator', '')


set(s, 'InputBufferSize', 200000);
set(s, 'FlowControl', 'none');
set(s, 'BaudRate', 921600);
set(s, 'Parity', 'none');
set(s, 'DataBits', 8);
set(s, 'StopBits', 1);
set(s, 'Timeout', 100);
s.BytesAvailableFcnCount = 1;
s.BytesAvailableFcnMode = 'byte';
%   3. Reads back the data from the FPGA and stores it to the
%      'data_from_serial.txt' file.
s.BytesAvailableFcn = @(src, evt)fprintf(fileID_serial, '%02x\n', fread(src,
1) );


% Connect serial port object to device.
fopen(s);

%%
for i = 1:num_of_words
    fwrite(s, A(i));
end


%%
% 4. Check If the two files are identical
visdiff('rand_hex_data.txt', 'data_from_serial.txt')

%%
% Close Serial Object.
fclose(s);
% Close Open File.
fclose(fileID_serial);


%%
delete(instrfindall)
```

9

Βιβλιογραφία

[1] High-Speed Serial I/OMade Simple, A Designers' Guide, with FPGA Applications, 2005

[2] 7 Series FPGAs GTX/GTH Transceivers, User Guide, UG476 (v1.11) February 23, 2015

[2] Vivado Design Suite Tutorial, Design Flows Overview
UG888 (v2015.1) April 1, 2015

[3] 7 Series FPGAs Transceivers Wizard v3.5
LogiCORE IP Product Guide, Vivado Design Suite
PG168 April 1, 2015

[4] Vivado Design Suite Tutorial
Using Constraints
UG945 (v2015.1) May 26, 2015

[5] FPGA PROTOTYPING BY VERILOG EXAMPLES
Xilinx SpartanTM-3 Version, Pong P. Chu, 2008

[6] Verification Methodology Manual for SystemVerilog by Janick Bergeron, Eduard Cerny, Alan Hunter, Springer 2005

[7] Synthesis and Simulation Design Guide
UG626 (v 13.1) March 1, 2011

[8] Overview and Dynamics of Scan Chain Testing,
http://anysilicon.com/overview-and-dynamics-of-scan-testing

[9] Scan Chain Wikipedia,
https://en.wikipedia.org/wiki/Scan_chain

[10] Field-programmable gate array Wikipedia,
https://en.wikipedia.org/wiki/Field-programmable_gate_array

[11] FPGA Implementation of Single Bit Error Correction using CRC, Pramod S P, Rajagopal A, Akshay S Kotain Department of E&C, DSCE, VTU,Bangalore, INDIA,
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.258.7991&rep=rep1&type=pdf

[12] HW-CLK-101-SCLK2 SuperClock-2 Module User Guide UG770 (v1.0) September 23, 2010

[13] Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics, DS183 (v1.24) September 24, 2015

[14] CMOS VLSI Design
A Circuits and Systems Perspective, Neil H. E. Weste, David Money Harris, Addison-Wesley 2011

[15] UART Explained, Support the standard serial bus,
https://electricimp.com/docs/resources/uart/

[16] VC7215 Virtex-7 FPGA GTH Transceiver Characterization Board User Guide UG972 (v1.3) October 17, 2014

[19] Virtex-7 FPGA VC7215 Characterization Kit IBERT Getting Started Guide UG970 (Vivado Design Suite v2015.1) April 27, 2015

[20] VC7215 Schematics PDF

[21] CP2103, SINGLE-CHIP USB TO UART BRIDGE Datasheet

[22] Digital Design, M.Morris Mano, Michael D.Ciletti, Fourth Edition