ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη Δυναμικού Επαναπρογραμματισμού Μονάδας FPGA για Εφαρμογή σε Συστήματα Οπτικών Επικοινωνιών Πολύ Υψηλής Απόδοσης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΔΗΜΗΤΡΙΟΥ ΑΠΟΣΤΟΛΑΚΗ

Επιβλέπων : Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Study of Dynamic Reconfiguration for High-Performance Optical Systems using FPGAs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΔΗΜΗΤΡΙΟΥ ΑΠΟΣΤΟΛΑΚΗ

Επιβλέπων : Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17^{η} Ιουλίου 2020.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Δ. Σούντρης Καθηγητής Ε.Μ.Π.	Η. Αβραμόπουλος Καθηγητής Ε.Μ.Π.	Π. Τσανάκας Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2020

(Υπογραφή)

....

ΔΗΜΗΤΡΙΟΣ ΑΠΟΣΤΟΛΑΚΗΣ

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

Περίληψη

Στην εποχή μας, υπάρχει πραγματική ανάγκη για μετάβαση σε τηλεπικοινωνιακά δίκτυα επόμενης γενιάς, τα οποία βασίζονται σε οπτικές ίνες για την μεταφορά δεδομένων σε ταχύτητες ανώτερες των 100Gbps (N x100G). Τα FPGAs που είναι εξοπλισμένα με σειριακούς πομποδέκτες πολλαπλών γιγαδυφίων (serial multi-gigabit transceivers) έχουν βρεθεί στο επίκεντρο του ενδιαφέροντος σε σχέση με την σχεδίαση και κατασκευή δικτυακών υποδομών υψηλών ταχυτήτων, καθώς φαίνεται να είναι η πιο συμφέρουσα επιλογή για συστήματα δικτύωσης που αναζητούν υψηλό εύρος ζώνης (bandwidth), υψηλή πυκνότητα, υψηλές επιδόσεις, ευελιξία σχεδιασμού, και πολύ καλή σχέση κόστους-αποτελεσματικότητας. Όπως υποδηλώνει το όνομά τους, τα FPGAs είναι προγραμματιζόμενα «στο πεδίο», με την έννοια ότι το εσωτερικό κύκλωμα μπορεί να διαμορφωθεί μετά την κατασκευή τους, καθώς και να τροποποιηθεί χωρίς να χρειάζεται η ανακατασκευή τους, όπως στα παραδοσιακά ASICs. Η Μερική Αναδιαμόρφωση (Partial Reconfiguration) οδηγεί αυτήν τους την ευελιξία ένα βήμα παραπέρα, δίνοντας τη δυνατότητα σε ένα FPGA που είναι ενεργό να τροποποιήσει ένα κομμάτι του όσο το υπόλοιπο σύστημα συνεχίζει να λειτουργεί κανονικά, χωρίς να βάζει σε κίνδυνο την ακεραιότητα των υπολογισμών που εκτελούνται στα τμήματα της συσκευής που δεν αναδιαμορφώνονται. Αυτή η τεχνική οδηγεί στην μείωση των πόρων που χρειάζονται για να υλοποιηθεί μια δεδομένη λειτουργία, με επακόλουθη μείωση στο κόστος και την ενεργειακή κατανάλωση, παρέχει ευελιξία στους αλγόριθμους/πρωτόκολλα που είναι διαθέσιμα σε μία εφαρμογή και επιταχύνει την υπολογιστική διαδικασία επιτρέποντας σε ένα σύστημα να είναι έτοιμο να ανταποκριθεί σε νέες απαιτήσεις γρηγορότερα. Σε αυτή την εργασία προσπαθήσαμε να εξερευνήσουμε την τεχνολογία της Μερικής Αναδιαμόρφωσης σε FPGAs και να εφαρμόσουμε τη γνώση που αποκτήθηκε για να υλοποιήσουμε ένα τηλεπικοινωνιακό σύστημα υψηλού εύρους ζώνης στη συσκευή Virtex®-7 H580T της Xilinx. Αυτή η συσκευή προκύπτει από την συνένωση δύο ενοτήτων προγραμματιζόμενης λογικής (δηλ. δύο FPGAs) και μιας ενότητας πομποδεκτών 28Gbps με 8 κανάλια (γνωστοί ως σειριακοί πομποδέκτες GTZ) σε ένα ενιαίο τσιπ, επιτρέποντας την γρήγορη διασύνδεση μεταξύ αυτών των τριών ενοτήτων και την χαμηλή κατανάλωση ισχύος. Κάνοντας χρήση και των οχτώ καναλιών της ενότητας με τους GTZ πομποδέκτες που διαθέτει η συσκευή Virtex®-7 H580T, κατασκευάσαμε ένα απλό αναδιαμορφώσιμο σύστημα που προσφέρει 2x100G εύρος ζώνης. Φυσικά, όταν πρόκειται για αναδιαμορφώσιμα συστήματα που διαχειρίζονται και επεξεργάζονται δεδομένα σε τόσο υψηλούς ρυθμούς, ο χρόνος αναδιαμόρφωσης μπορεί να επηρεάσει σημαντικά την συνολική τους απόδοση. Για να βελτιώσουμε την ταχύτητα αναδιαμόρφωσης του συστήματός μας, εξετάσαμε διάφορες αρχιτεκτονικές και σχήματα αναδιαμόρφωσης. Η αναδιαμορφώσιμη αρχιτεκτονική στην οποία βασιστήκαμε τελικά, χρησιμοποιεί την υψηλής ταχύτητας μνήμη block RAM (BRAM) του FPGA και μια μονάδα που αναπτύχθηκε «από το μηδέν» σε γλώσσα περιγραφής υλικού για να ελέγχει την θύρα ICAP του FPGA και την μερική αναδιαμόρφωση της συσκευής μας μέσω της θύρας αυτής. Αυτή η αρχιτεκτονική μας επέτρεψε να εκμεταλλευτούμε πλήρως την υψηλή ρυθμαπόδοση (high throughput) της θύρας ICAP, και έτσι να μειώσουμε σημαντικά τον χρόνο αναδιαμόρφωσης. Η τελευταία πινελιά στο σύστημά μας, δόθηκε με την υλοποίηση ενός μικροεπεξεργαστή στην προγραμματιζόμενη λογική (δηλ. τα FPGAs) της συσκευής, ώστε να διευκολυνθεί η διαχείριση του συστήματος από τον «έξω» κόσμο και έτσι να επιτραπεί ο απομακρυσμένος έλεγχος της μερικής αναδιαμόρφωσης της συσκευής. Το αποτέλεσμα ήταν ένα υβριδικό σύστημα υλικού-λογισμικού (hardware-software) στο οποίο η 2x100G εφαρμογή που θα υλοποιούνταν βέλτιστα κάθε φορά στο υλικό, μπορούσε εύκολα να καθοριστεί από το λογισμικό (η έννοια του «βοηθούμενου από το υλικό, προσδιοριζόμενου από το λογισμικό»).

Λέξεις Κλειδιά: FPGA, Xilinx Virtex-7 H580T, Μερική Αναδιαμόρφωση, σειριακοί πομποδέκτες πολλαπλών γιγαδυφίων, GTZ, BRAM, ICAP, MicroBlaze, Αναδιαμορφώσιμες Αρχιτεκτονικές, Ετερογενές τρισδιάστατο FPGA, τεχνολογία SSI, N x100G εφαρμογές, οπτικά δίκτυα επόμενης γενιάς

Abstract

Nowadays, the market need for Nx100G networking line cards and next-generation optics is real. Field-Programmable Gate Arrays (FPGAs) equipped with high-speed, serial, Multi-Gigabit Transceivers (MGTs) have gained the interest of network developers, as they appear to be the most advantageous choice for networking systems looking for high bandwidth, high density, high performance, design flexibility and cost effectiveness. As their name denotes, FPGAs are programmable "in the field", meaning that their functionality can be defined after the fabrication process and modified, if needed, without going to re-fabrication process, as common ASICs. Partial Reconfiguration (PR) takes this advantage one step further, by allowing an operating FPGA design to modify a part of itself, while the rest of the system continues to function normally, without compromising the integrity of the computation running on those parts of the device that are not being reconfigured. This technique leads to reduction of the amount of resources required to implement a given function, with consequent reductions in cost and power consumption, provides flexibility in the algorithms/protocols available to an application and accelerates computing by enabling a design to be ready to correspond to new computation requirements much faster. In this thesis, we tried to explore the PR technology on FPGAs and apply the knowledge acquired to implement a high-bandwidth telecom system on a Xilinx Virtex®-7 H580T device. This device, described by Xilinx as *"The world's first 3D heterogeneous all programmable product"*, combines two FPGA dices and an 8-channel 28Gbps transceiver die (GTZ serial transceivers) into a single package, while enabling fast interconnection between them and power efficiency. By using all eight GTZ transceiver channels available on the Virtex®-7 H580T device we built a simple reconfigurable system that offers 2x100G bandwidth. Of course, for reconfigurable systems that manage and process data at such a high rate, reconfiguration time has a deep impact on their overall performance. So, in order to improve our system's reconfiguration throughput, we examined several architectures and reconfiguration schemes. The reconfigurable architecture on which we were eventually based, utilizes the FPGA's high-speed block RAM (BRAM) resources and a hardware module developed from scratch in RTL to control the FPGA's internal configuration access port (ICAP) and the entire PR process through it. This architecture has allowed us to fully exploit the ICAP's high throughput capabilities, and thus significantly reduce the reconfiguration time. The finishing touch to our reconfigurable system, was the implementation of a microprocessor entirely within the device general-purpose memory and logic fabric in order to facilitate the management of our system from the outside world and thus enable a user to remotely control the Partial Reconfiguration of the device. The result was a hybrid hardware-software system where the 2x100G application optimally implemented in the hardware each time, could easily be defined by the software (the "hardware-enabled, software-defined" concept).

Keywords: FPGA, Xilinx Virtex-7 H580T, Partial Reconfiguration, serial Multi-Gigabit Transceivers, GTZ, BRAM, ICAP, MicroBlaze, Reconfigurable Computing, Heterogeneous 2.5D FPGA, SSI technology, Nx100G applications, next-generation optics

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον καθηγητή κ. Δημήτριο Σούντρη για την δυνατότητα που μου έδωσε να ασχοληθώ με το σύγχρονο και ενδιαφέρον αντικείμενο της παρούσης διπλωματικής. Επίσης, θα ήθελα να ευχαριστήσω τους μεταδιδακτορικούς ερευνητές Κωνσταντίνο Μαραγκό, Χρήστο Σπαθαράκη και Γεώργιο Λεντάρη για τη συνεχή παρακολούθηση, την υποστήριξη και τον χρόνο που αφιέρωσαν για την εκπόνηση της εργασίας αυτής. Ακόμα, ευχαριστώ θερμά τον καθηγητή κ. Ηρακλή Αβραμόπουλο ο ρόλος του οποίου ήταν καθοριστικός στην πραγματοποίηση της μελέτης αυτής.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς και τους ανθρώπους που ήταν κοντά μου, για την απεριόριστη στήριξη που μου παρείχαν σε καλές και δύσκολες στιγμές κατά την διάρκεια των σπουδών μου.

Δημήτρης Αποστολάκης,

Αθήνα, 17η Ιουλίου 2020

Contents

Ευρετήριο Σχημάτων και Πινάκων

List of Figures

List of Tables

0 *Περιληπτική Απόδοση*

Στην εποχή μας, η χρήση του Internet και των διαδικτυακών πόρων (cloud services, e-commerce, e-Health, streaming video, mobile internet, online gaming) παρουσιάζει εντυπωσιακή αύξηση. Το γεγονός αυτό, καθιστά απαραίτητη την χρήση σύγχρονων οπτικών δικτύων που μπορούν να προσφέρουν πολύ μεγαλύτερες ταχύτητες μεταφοράς δεδομένων (Nx100 Gbit/s) και αποδοτικότερη χρήση του διαθέσιμου εύρους ζώνης (bandwidth) σε σχέση με τις συμβατικές τηλεπικοινωνιακές ζεύξεις (ενσύρματα, ασύρματα). Την ίδια στιγμή, παρατηρείται μια αλλαγή στη φύση της κίνησης τα τελευταία χρόνια, καθώς τόσο η απαίτηση για γρηγορότερη πρόσβαση όσο και η χρήση φορητών συσκευών την έχουν καταστήσει πολύ πιο δυναμική και απρόβλεπτη. Σε αυτό το πλαίσιο, γίνεται αναγκαία η υιοθέτηση καινοτόμων τεχνολογιών και αρχιτεκτονικών στα σύγχρονα οπτικά δίκτυα, που θα τους επιτρέψουν να είναι ευέλικτα και ικανά να διαχειριστούν την δυναμική και αυξανόμενη δικτυακή κίνηση. Βασικό στοιχείο ενός ευέλικτου οπτικού δικτύου είναι οι ευέλικτοι, επαναρυθμιζόμενοι οπτικοί πομποδέκτες (flexible optical transceivers), ικανοί να διαχειριστούν δυναμικά τους διαθέσιμους δικτυακούς πόρους ανάλογα με τις απαιτήσεις της τηλεπικοινωνιακής κίνησης. Αυτή η ευελιξία συμβάλει επίσης σημαντικά στην οικονομική και αποτελεσματική κλιμάκωση / αναβάθμιση των δικτυακών υποδομών, καθώς μας επιτρέπει να εφαρμόζουμε τις τελευταίες τεχνολογίες οπτικής διασύνδεσης (ή εκείνες που ταιριάζουν καλύτερα στις ανάγκες μας κάθε φορά), έχοντας πάντα ως στόχο το υψηλό εύρος ζώνης, την χαμηλή κατανάλωση ισχύος και το μειωμένο κόστος.

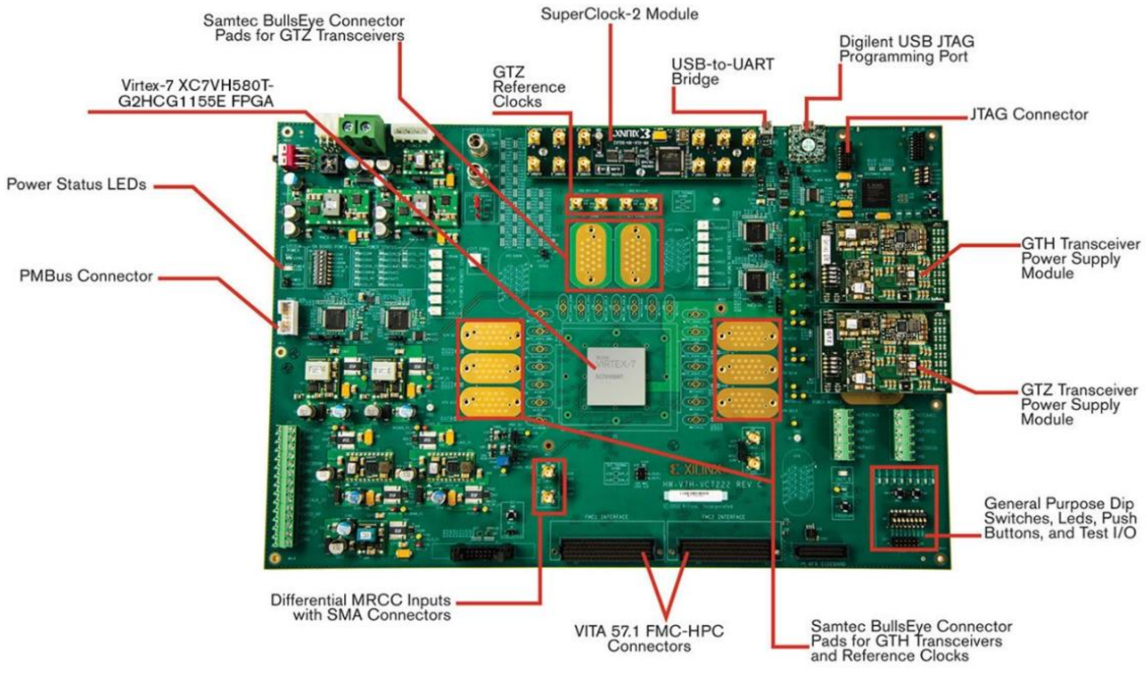
Τα σύγχρονα και πιο εξελιγμένα FPGAs που παρέχουν υψηλότερες επιδόσεις, δυνατότητες και χωρητικότητα, σε συνδυασμό με πολύ πιο ισχυρούς σειριακούς πομποδέκτες μπορούν να ικανοποιήσουν όλες αυτές τις ανάγκες των οπτικών δικτύων επόμενης γενιάς. Τα FPGAs έχουν σημαντικό ρόλο στον εξοπλισμό των δικτύων, όχι μόνο επειδή μπορούν να συνδυάζουν υψηλή επεξεργαστική ισχύ με χαμηλή κατανάλωση ενέργειας και μικρό φυσικό μέγεθος, αλλά και λόγω της ευελιξίας και της ικανότητάς τους να επαναπρογραμματίζονται πολλές φορές μετά την κατασκευή τους ή την τοποθέτησή τους σε κάποια συσκευή. Επομένως, για συστήματα δικτύωσης που αναζητούν υψηλό εύρος ζώνης, υψηλή πυκνότητα, υψηλές επιδόσεις, ευελιξία σχεδιασμού, και πολύ καλή σχέση κόστους-αποτελεσματικότητας, τα FPGAs που είναι εξοπλισμένα με σειριακούς πομποδέκτες πολλαπλών γιγαδυφίων (serial Multi-Gigabit Transceivers) είναι η προφανής επιλογή.

Η αναδιαμορφωσιμότητα των FPGAs τα καθιστά μια ευέλικτη -με πλήθος εφαρμογών- πλατφόρμα και μια βιώσιμη λύση για γρήγορη υλοποίηση και προτυποποίηση νέων συστημάτων. Επιπλέον, τα τελευταία χρόνια είναι ιδιαίτερα διαδεδομένη η χρήση των FPGAs για την υλοποίηση αναδιαμορφώσιμων υπολογιστικών συστημάτων (Reconfigurable Computing systems). Πρόκειται για ετερογενείς πλατφόρμες που συνδυάζουν την ευελιξία του λογισμικού που τρέχει σε επεξεργαστές γενικού σκοπού, με την αποδοτικότητα μονάδων υψηλής επεξεργαστικής ισχύος, όπως τα FPGAs. Η κύρια διαφορά με τις «παραδοσιακές» αρχιτεκτονικές έγκειται στην ικανότητα του υλικού να προσαρμόζεται κατά τη διαδικασία εκτέλεσης και να μεταβάλλει τον εαυτό του «φορτώνοντας ένα καινούργιο κύκλωμα» κάθε φορά που μια συγκεκριμένη εργασία το απαιτεί. Η Μερική Αναδιαμόρφωση (Partial Reconfiguration) οδηγεί αυτήν την ευελιξία ένα βήμα παραπέρα, δίνοντας τη δυνατότητα σε ένα FPGA που είναι ενεργό να τροποποιήσει ένα κομμάτι του όσο το υπόλοιπο σύστημα συνεχίζει να λειτουργεί κανονικά, χωρίς να βάζει σε κίνδυνο την ακεραιότητα των υπολογισμών που εκτελούνται στα τμήματα της συσκευής που δεν αναδιαμορφώνονται. Τα βασικά συγκριτικά πλεονεκτήματα της μερικής αναδιαμόρφωσης ενός FPGA, έναντι του επαναπρογραμματισμού ολόκληρης της συσκευής, είναι ότι μας επιτρέπει να αλλάξουμε την λειτουργία που υλοποιείται στο FPGA δυναμικά (on-the-fly) και σε πολύ μικρότερο χρονικό διάστημα.

Στο πλαίσιο αυτής της διπλωματικής εργασίας, διερευνούμε τις προκλήσεις της υλοποίησης ενός αναδιαμορφώσιμου τηλεπικοινωνιακού συστήματος υψηλού εύρους ζώνης σε μια υψηλών επιδόσεων συσκευή FPGA. Εξερευνούμε την τεχνολογία της Μερικής Αναδιαμόρφωσης, εξετάζοντας διαφορετικές αρχιτεκτονικές και σχήματα αναδιαμόρφωσης προκειμένου να βελτιώσουμε τον χρόνο αναδιαμόρφωσης. Επιπλέον, αντιμετωπίζουμε τις προκλήσεις της υλοποίησης ενός μικροεπεξεργαστή χρησιμοποιώντας αποκλειστικά και μόνο πόρους του FPGA, ώστε να διευκολυνθεί η διαχείριση του συστήματος από τον «έξω» κόσμο (π.χ. επικοινωνία με έναν κεντρικό υπολογιστή) και έτσι να επιτραπεί ο απομακρυσμένος έλεγχος της μερικής αναδιαμόρφωσης της συσκευής. Για τη μελέτη μας χρησιμοποιήσαμε τη συσκευή XC7VH580T-G2HCG1155E της Xilinx, η οποία διαθέτει οχτώ ενσωματωμένους πομποδέκτες υψηλής ταχύτητας, καθένας από τους οποίους μπορεί να λειτουργεί σε ρυθμούς έως και 28,05 γιγαδυφίων ανά δευτερόλεπτο (Gbps). Στην εν λόγω συσκευή υλοποιήσαμε όλη αυτή την λογική που απαιτείται για να διαμορφώσουμε κατάλληλα και τους οχτώ πομποδέκτες αυτής, δημιουργώντας ένα 2x100G τηλεπικοινωνιακό σύστημα, του οποίου τα δομικά στοιχεία υλοποιούνταν πάντα βέλτιστα στο υλικό (hardware), αλλά η λειτουργικότητά του μπορούσε να καθοριστεί δυναμικά από το λογισμικό (software).

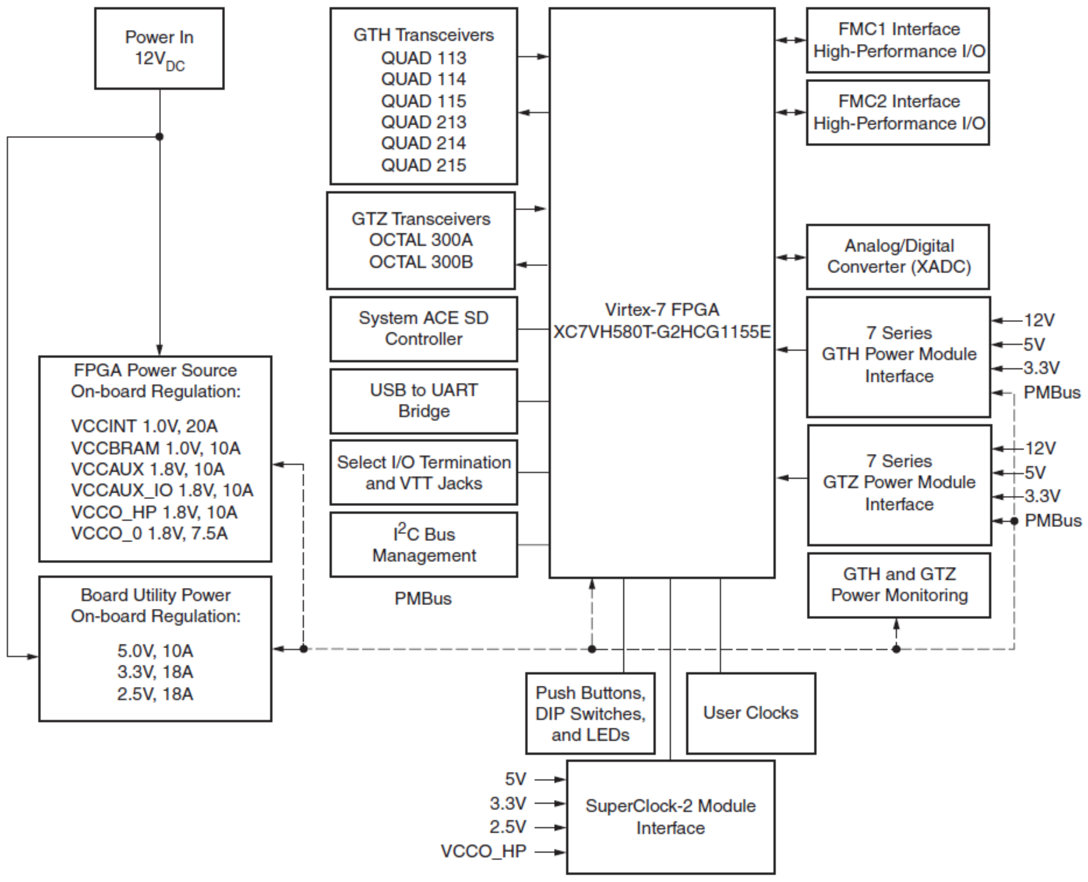
Η αναπτυξιακή πλακέτα VC7222

Η πλακέτα VC7222 (βλ. Σχήμα 1) παρέχει τον υλικό εξοπλισμό για τον χαρακτηρισμό και την αξιολόγηση των GTH και GTZ πομποδεκτών που διατίθενται στην συσκευή XC7VH580T-G2HCG1155E. Το XC7VH580T, μέλος της οικογένειας Xilinx Virtex®-7, δεν είναι απλώς ένα συνηθισμένο, μονολιθικό FPGA (όπου με τον όρο «μονολιθικό» εννοούμε ότι υπάρχει μία μοναδική ενότητα πυριτίου μέσα στο τσιπ). Το XC7VH580T είναι ένα από τα τρισδιάστατα ολοκληρωμένα κυκλώματα της Xilinx (Xilinx® 2.5D ICs) το οποίο έχει σχεδιαστεί με χρήση της τεχνολογίας Stacked Silicon Interconnect (SSI). Σε μία συσκευή SSI πολλαπλές ενότητες-μήτρες πυριτίου συνδέονται μεταξύ τους μέσω ενός πολύ μεγάλου δικτύου διασυνδέσεων πυριτίου και συσκευάζονται σε ένα ενιαίο τσιπ, επιτρέποντας έτσι την ενσωμάτωση τεράστιων ποσοτήτων λογικής διασύνδεσης, πομποδεκτών και μικροκυκλωματικών (on-chip) πόρων σε μία μόνο συσκευή. Κάθε ενότητα-μήτρα πυριτίου που περιέχεται σε μία συσκευή SSI αποτελεί μία ξεχωριστή περιοχή μέσα στην συσκευή που ονομάζεται Super Logic Region (SLR). Ένα μεγάλο δίκτυο πόρων δρομολόγησης, οι πόροι SLL (Super Long Line), χρησιμοποιείται για την σύνδεση της λογικής μεταξύ των διαφορετικών SLR-περιοχών της συσκευής. Η SSI-συσκευή XC7VH580T που θα μας απασχολήσει στην παρούσα εργασία, έχει την επιπλέον ιδιαιτερότητα να είναι ετερογενής, καθώς οι διαφορετικές ενότητες-μήτρες πυριτίου που περιλαμβάνει δεν είναι πανομοιότυπες. Για την ακρίβεια, το XC7VH580T αποτελείται από δύο ενότητες (SLR0 και SLR1) προγραμματιζόμενης λογικής (δηλ. δύο FPGAs) και μια ενότητα (SLR2) πομποδεκτών 28Gbps με 8 κανάλια (σειριακοί πομποδέκτες GTZ).


Samtec BullsEye Connector Pads for GTZ Transceivers
SuperClock-2 Module
Digilent USB JTAG Programming Port
USB-to-UART Bridge
GTZ Reference Clocks
Virtex-7 XC7VH580T-G2HCG1155E FPGA
JTAG Connector
Power Status LEDs
PMBus Connector
GTH Transceiver Power Supply Module
GTZ Transceiver Power Supply Module
General Purpose Dip Switches, Leds, Push Buttons, and Test I/O
Differential MRCC Inputs with SMA Connectors
VITA 57.1 FMC-HPC Connectors
Samtec BullsEye Connector Pads for GTH Transceivers and Reference Clocks

Σχήμα 1: Η αναπτυξιακή πλακέτα VC7222.

Το σχηματικό διάγραμμα της πλακέτας VC7222 φαίνεται στο Σχήμα 2.


Power In 12V DC
GTH Transceivers QUAD 113 QUAD 114 QUAD 115 QUAD 213 QUAD 214 QUAD 215
GTZ Transceivers OCTAL 300A OCTAL 300B
System ACE SD Controller
USB to UART Bridge
Select I/O Termination and VTT Jacks
I2C Bus Management
PMBus
Virtex-7 FPGA XC7VH580T-G2HCG1155E
FMC1 Interface High-Performance I/O
FMC2 Interface High-Performance I/O
Analog/Digital Converter (XADC)
7 Series GTH Power Module Interface
12V
5V
3.3V
PMBus
7 Series GTZ Power Module Interface
GTH and GTZ Power Monitoring
FPGA Power Source On-board Regulation:
VCCINT 1.0V, 20A
VCCBRAM 1.0V, 10A
VCCAUX 1.8V, 10A
VCCAUX_IO 1.8V, 10A
VCCO_HP 1.8V, 10A
VCCO_0 1.8V, 7.5A
Board Utility Power On-board Regulation:
5.0V, 10A
3.3V, 18A
2.5V, 18A
Push Buttons, DIP Switches, and LEDs
User Clocks
SuperClock-2 Module Interface
5V
3.3V
2.5V
VCCO_HP

Σχήμα 2: Σχηματικό διάγραμμα της VC7222.

Τα πιο σημαντικά χαρακτηριστικά της πλακέτας VC7222 είναι:

- Virtex-7 XC7VH580T-G2HCG1155E FPGA
- Onboard power supplies for all necessary voltages
- Terminal blocks for optional use of external power supplies
- Digilent USB JTAG programming port
- System ACE™ SD controller
- Power module supporting Virtex-7 FPGA GTH transceiver power requirements
- Power module supporting Virtex-7 FPGA GTZ transceiver power requirements
- A fixed, 200 MHz 2.5V LVDS oscillator wired to multi-region clock capable (MRCC) Inputs
- Two pairs of differential MRCC inputs with SMA connectors
- SuperClock-2 module supporting multiple frequencies
- Six Samtec BullsEye connector pads for the GTH transceivers and reference clocks
- Two Samtec BullsEye connector pads for the GTZ transceivers and two pairs of SMA connectors for GTZ transceiver reference clocks

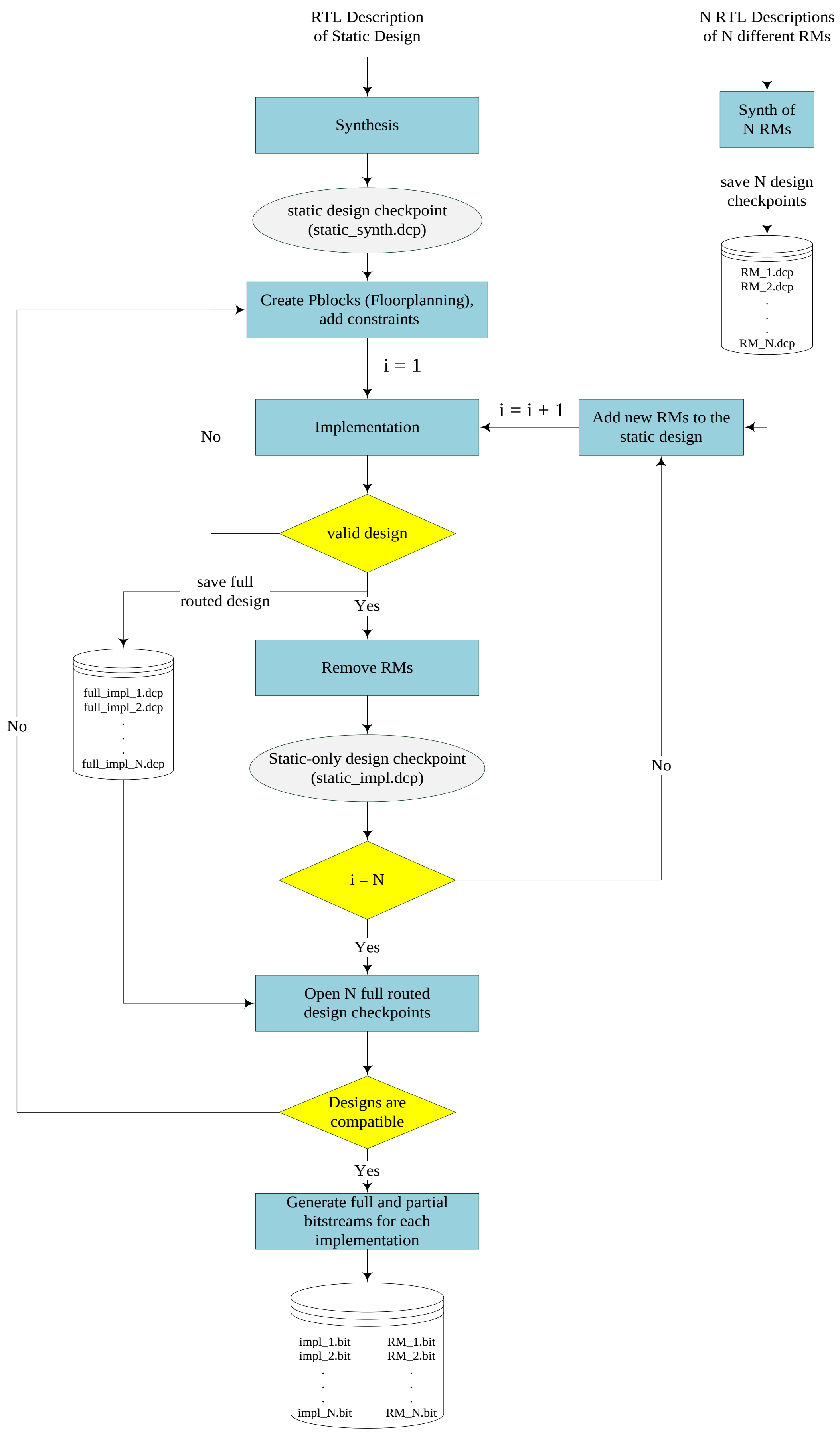
- Power status LEDs
- General purpose DIP switches, LEDs, pushbuttons, and test I/O
- Two VITA 57.1 FPGA mezzanine card (FMC) high pin count (HPC) connectors
- USB-to-UART bridge
- I2C bus
- PMBus connectivity to onboard digital power supplies
- Active cooling for the FPGA

Μερική Αναδιαμόρφωση (Partial Reconfiguration)

Ένα μεγάλο πλεονέκτημα των FPGAs, όπως υποδηλώνει το όνομά τους, είναι ότι είναι προγραμματιζόμενα «στο πεδίο» με την έννοια ότι το εσωτερικό κύκλωμα διαμορφώνεται μετά την κατασκευή, και μπορεί να τροποποιηθεί χωρίς να χρειαστεί η ανακατασκευή τους. Η Μερική Αναδιαμόρφωση προχωράει αυτό το σκεπτικό ένα βήμα παραπέρα επιτρέποντας τη μερική τροποποίηση ενός ενεργού FPGA, όσο το υπόλοιπο κύκλωμα συνεχίζει να λειτουργεί κανονικά, χωρίς να επηρεάζει την λειτουργία των μερών που δεν αναδιαμορφώνονται. Υπάρχουν πολλοί λόγοι που η Μερική Αναδιαμόρφωση μπορεί να είναι επωφελής. Μερικοί από αυτούς είναι:

- Μείωση του μεγέθους του FPGA που απαιτείται για να υλοποιηθεί μια λειτουργία, με αντίστοιχη μείωση στο κόστος και την κατανάλωση ενέργειας.
- Ευελιξία στην επιλογή αλγορίθμων και πρωτοκόλλων που είναι διαθέσιμα σε μια εφαρμογή.
- Βελτίωση στην ανοχή σφαλμάτων του FPGA.
- Επιτάχυνση του αναδιαμορφούμενου υπολογισμού.
- Επιτρέπει τη δημιουργία νέων εφαρμογών σε FPGA, που διαφορετικά θα ήταν αδύνατο να υλοποιηθούν.

Όπως απεικονίζεται και στο διάγραμμα ροής του παρακάτω σχήματος, η διαδικασία της Μερικής Αναδιαμόρφωσης απαιτεί την υλοποίηση πολλαπλών διαμορφώσεων (configurations) οι οποίες τελικά καταλήγουν σε ολικά bitstreams για κάθε configuration, και μερικά bitstreams για κάθε αναδιαμορφούμενο εργαλείο. Ο αριθμός των απαιτούμενων configurations ποικίλει ανάλογα με τον αριθμό των εργαλείων που χρειάζεται να υλοποιηθούν. Όμως, όλα τα configurations μοιράζονται την ίδια στατική λογική (λογική η οποία δεν αναδιαμορφώνεται). Η στατική λογική εξάγεται από το αρχικό configuration και εισάγεται σε όλα τα επακόλουθα configurations και έτσι, σε κάθε configuration διαφέρει μόνο η αναδιαμορφούμενη λογική, ενώ η στατική λογική μένει πανομοιότυπη.


RTL Description of Static Design
N RTL Descriptions of N different RMs
Synthesis
Synth of N RMs
save N design checkpoints
static design checkpoint (static_synth.dcp)
RM_1.dcp
RM_2.dcp
RM_N.dcp
Create Pblocks (Floorplanning), add constraints
i = 1
Implementation
i = i + 1
Add new RMs to the static design
No
valid design
save full routed design
Yes
full_impl_1.dcp
full_impl_2.dcp
full_impl_N.dcp
Remove RMs
No
Static-only design checkpoint (static_impl.dcp)
No
i = N
Yes
Open N full routed design checkpoints
Designs are compatible
Yes
Generate full and partial bitstreams for each implementation
impl_1.bit
impl_2.bit
impl_N.bit
RM_1.bit
RM_2.bit
RM_N.bit

Σχήμα 3: Διαδικασία Μερικής Αναδιαμόρφωσης.

Η πλειοψηφία των FPGA συσκευών, υποστηρίζουν αρκετούς διαφορετικούς τρόπους με τους οποίους μπορεί κάποιος να φορτώσει ένα μερικό bitstream στην ήδη προγραμματισμένη και ενεργή συσκευή για να την αναδιαμορφώσει μερικώς. Στην περίπτωση της συσκευής XC7VH580T-G2HCG1155E αυτοί οι τρόποι διαμόρφωσης (configuration modes) συνοψίζονται ως εξής:

Configuration Mode	Max Clock Rate	Max Data Width	Maximum Bandwidth
ICAP	100 MHz	32 bits	3,2 Gbits/s
SelectMAP (Slave)	100 MHz	32 bits	3,2 Gbits/s
Serial Mode (Slave)	100 MHz	1 bit	100 Mbits/s
JTAG	20 MHz	1 bit	20 Mbits/s

Πίνακας 1: Τρόποι μερικής αναδιαμόρφωσης της συσκευής XC7VH580T.

Κάθε τρόπος διαμόρφωσης στον παραπάνω πίνακα βασίζεται σε μία συγκεκριμένη διεπαφή διαμόρφωσης (αυτές είναι οι ICAP, SelectMAP, Serial, JTAG) και έχει μία μέγιστη θεωρητική τιμή ρυθμαπόδοσης ανάλογα με το μέγιστο πλάτος (σε bits) του αντίστοιχου datapath που χρησιμοποιείται από την εκάστοτε διεπαφή και την μέγιστη συχνότητα ρολογιού μέχρι την οποία μπορεί (θεωρητικά) αυτή η διεπαφή να είναι λειτουργική.

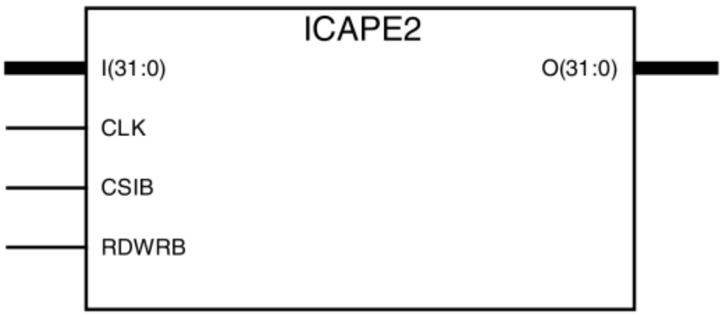
Από τον Πίνακα 1 γίνεται εμφανές ότι το χρονικό διάστημα που απαιτείται για να ολοκληρωθεί η μερική αναδιαμόρφωση μιας συσκευής επηρεάζεται άμεσα από τον τρόπο διαμόρφωσης που θα επιλέξουμε να χρησιμοποιήσουμε. Προφανώς, ο χρόνος αναδιαμόρφωσης σχετίζεται άμεσα με το μέγεθος του μερικού bitstream. Ωστόσο, όταν μιλάμε για ένα μερικό bitstream συγκεκριμένου μεγέθους, η ταχύτητα αναδιαμόρφωσης εξαρτάται άμεσα από την αρχιτεκτονική αναδιαμόρφωσης (Reconfigurable Architecture - RA) που χρησιμοποιείται. Ανάλογα με το μέσο που χρησιμοποιείται για την αποθήκευση του μερικού bitstream, την διεπαφή διαμόρφωσης που επιλέγεται για την μερική αναδιαμόρφωση του FPGA, και τον τρόπο με τον οποίο διαβάζεται το bitstream από το μέσο αποθήκευσης και αποστέλλεται στην επιλεγμένη διεπαφή διαμόρφωσης, μπορούμε να δημιουργήσουμε διάφορες αρχιτεκτονικές αναδιαμόρφωσης που καθεμία να προσφέρει διαφορετική ταχύτητα αναδιαμόρφωσης.

Σε περιπτώσεις σαν την δική μας, όπου ένα αναδιαμορφώσιμο σύστημα πρέπει να διαχειριστεί και να επεξεργαστεί δεδομένα σε πάρα πολύ υψηλούς ρυθμούς, ο χρόνος αναδιαμόρφωσης μπορεί να επηρεάσει σημαντικά την συνολική του απόδοση. Στο πλαίσιο αυτό, εξετάσαμε πέντε διαφορετικές αρχιτεκτονικές αναδιαμόρφωσης (RAs) των οποίων οι ταχύτητες αναδιαμόρφωσης παρουσιάζονται στον παρακάτω πίνακα (σημειώνεται ότι το μέγεθος του μερικού bitstream που χρησιμοποιήθηκε σε κάθε περίπτωση είναι ενδεικτικό και ως εκ τούτου περιμένουμε ανάλογα αποτελέσματα/διαφορές για bitstreams μεγαλύτερου ή μικρότερου μεγέθους):

Name of RA	Partial Bitstream Format	Partial Bitstream Size	Reconfiguration Time	Throughput
EXT_JTAG	.bit	681.660 Bytes	926,02 msec	5,889071 Mbits/s
EXT_HWICAP	.bin	681.532 Bytes	709 sec	7,690064 Kbits/s
INT_ HWICAP			37,599270 msec	145,009623 Mbits/s
INT_ HWICAP_ASYNC			19,669035 msec	277,199974 Mbits/s
INT_CUSTOM_ICAP			1,703860 msec	3,199943 Gbits/s

Πίνακας 2: Ταχύτητες αναδιαμόρφωσης των διαφόρων αρχιτεκτονικών.

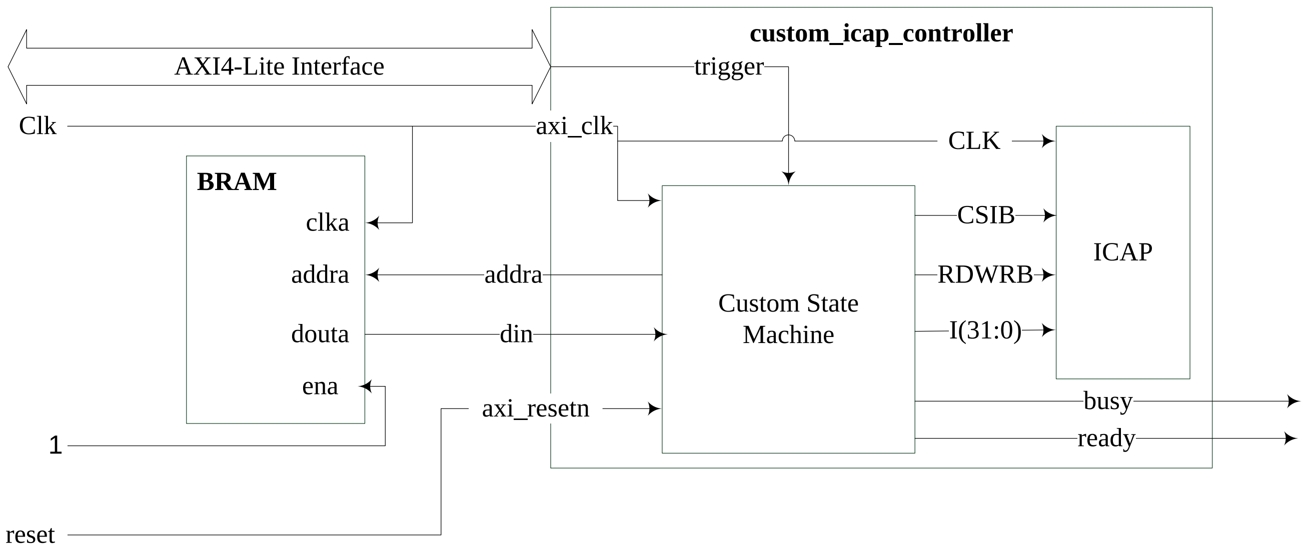
Η πρώτη από τις αρχιτεκτονικές αναδιαμόρφωσης του Πίνακα 2 με όνομα «EXT_JTAG», βασίζεται στην διεπαφή JTAG και έχει συμπεριληφθεί εδώ μόνο για λόγους σύγκρισης, καθώς αποτελεί την πιο απλή και συχνά χρησιμοποιούμενη μέθοδο μερικής αναδιαμόρφωσης μιας συσκευής FPGA. Οι υπόλοιπες τέσσερις αρχιτεκτονικές βασίζονται στην διεπαφή ICAP και μπορούν να θεωρηθούν οι κύριες αρχιτεκτονικές που δημιουργήσαμε στην προσπάθεια να βελτιώσουμε την ταχύτητα αναδιαμόρφωσης του συστήματος μας.
Η διεπαφή ICAP είναι εσωτερική, επιτρέπει την εγγραφή δεδομένων διαμόρφωσης στην συσκευή (κατά την μερική αναδιαμόρφωση αυτής) με τον μέγιστο δυνατό ρυθμό των 3,2 Gbps (βλ. Πίνακας 1) και είναι ιδανική για προσαρμοσμένες στις ανάγκες του χρήστη λύσεις. Η χρήση της ICAP βασίζεται στους εσωτερικούς προγραμματιζόμενους πόρους και τα δομικά κυκλωματικά στοιχεία (primitives) ενός FPGA και επομένως είναι μία διεπαφή που μπορεί πάντα να χρησιμοποιηθεί μετά από μία πρώτη διαμόρφωση ολόκληρης της συσκευής με το κατάλληλο ολικό bitstream.



Σχήμα 4: ICAPE2 primitive.

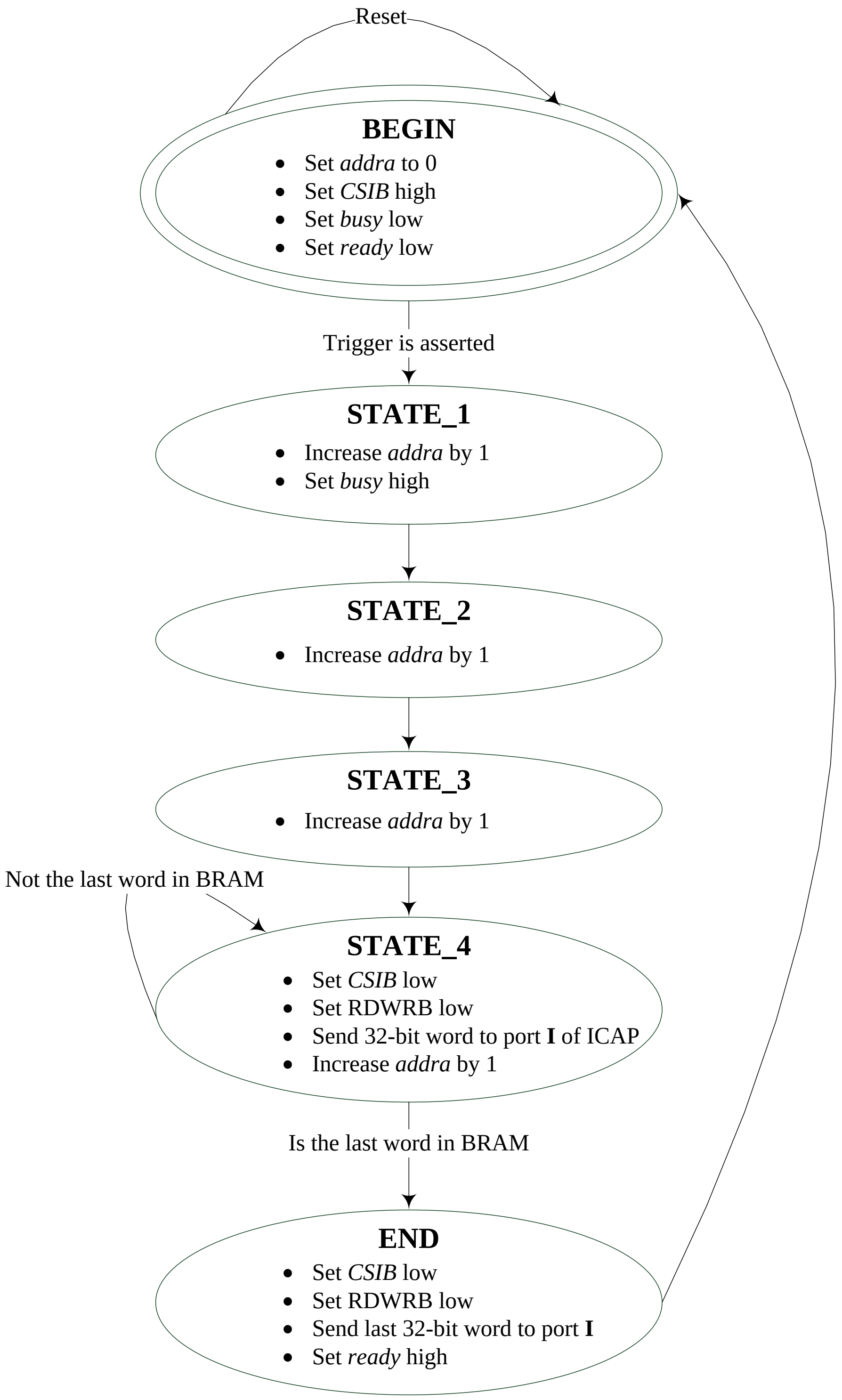
Για να αποκτήσουμε πρόσβαση (εγγραφή/ανάγνωση) στην εσωτερική μνήμη διαμόρφωσης της συσκευής XC7VH580T μέσω της διεπαφής ICAP, είναι απαραίτητη η χρήση του ICAPE2 primitive (βλ. Σχήμα 4) καθώς επίσης και ενός ελεγκτή για την οδήγηση των σημάτων ελέγχου αυτού (δηλ. CLK, CSIB and RDWRB). Ανάλογα με την λειτουργία που επιτελείται μέσω του ICAPE2 (εγγραφή/ανάγνωση), ο ελεγκτής είναι επίσης υπεύθυνος για την αποστολή δεδομένων διαμόρφωσης στην θύρα εγγραφής I [31:0] (εάν RDWRB=0) ή την παραλαβή δεδομένων διαμόρφωσης από την θύρα ανάγνωσης O [31:0] (εάν RDWRB=1).

Η Xilinx έχει δημιουργήσει και διαθέτει δωρεάν στο κοινό έναν τέτοιο ελεγκτή ICAP που ονομάζεται AXI HWICAP. Πρόκειται για μια έτοιμη προς χρήση μονάδα που υλοποιείται εξ ολοκλήρου στην προγραμματιζόμενη λογική και τους πόρους του FPGA και η οποία χρησιμοποιήθηκε στις τρεις πρώτες κύριες αρχιτεκτονικές μας (δηλ. «EXT_HWICAP», «INT_ HWICAP», «INT_ HWICAP_ASYNC»). Από τα στοιχεία του Πίνακα 2, παρατηρούμε ότι οι ταχύτητες αναδιαμόρφωσης αυτών των τριών αρχιτεκτονικών ήταν πολύ μικρότερες από την μέγιστη θεωρητική τιμή ρυθμαπόδοσης της διεπαφής ICAP (3,2 Gbps). Αποδείχθηκε ότι η χρήση διαύλων (buses) που βασίζονται σε καθιερωμένα πρωτόκολλα επικοινωνίας (π.χ. Advanced Microcontroller Bus Architecture – AMBA) για την μεταφορά δεδομένων διαμόρφωσης του μερικού bitstream από την μνήμη που είναι αποθηκευμένα, στην θύρα εγγραφής του ICAPE2, αποτελεί το μεγαλύτερο εμπόδιο στην ταχύτητα αναδιαμόρφωσης μιας αρχιτεκτονικής. Για να ξεπεράσουμε το εμπόδιο αυτό, αναπτύξαμε «από το μηδέν» σε Verilog HDL τον δικό μας ελεγκτή ICAP για την υλοποίηση της αναδιαμορφώσιμης αρχιτεκτονικής «INT_CUSTOM_ICAP» που φαίνεται στο παρακάτω σχήμα:



Σχήμα 5: Η αρχιτεκτονική αναδιαμόρφωσης «INT_CUSTOM_ICAP».

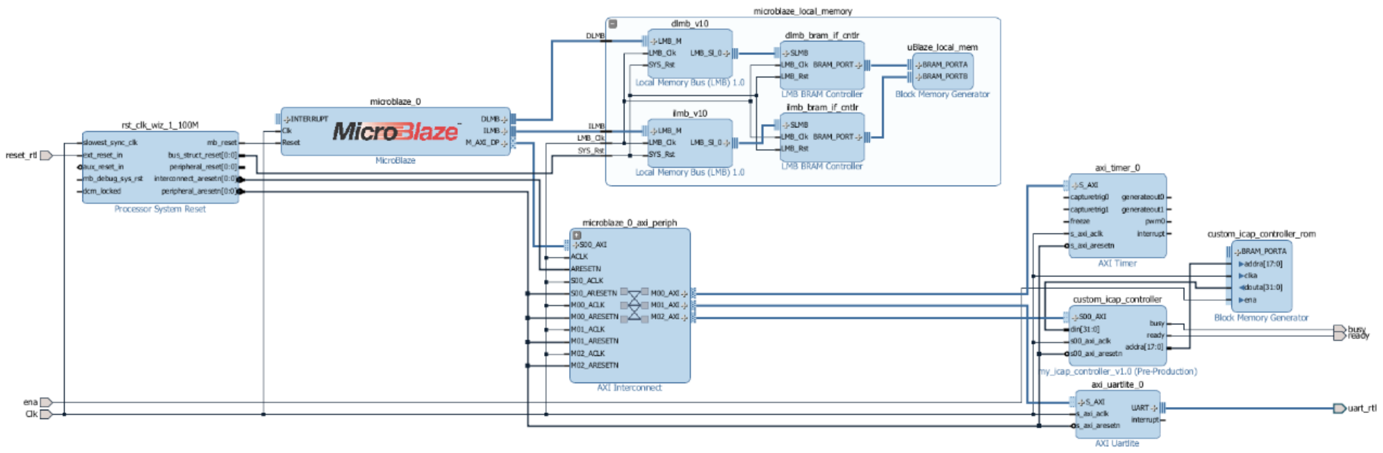
Το αναδιαμορφώσιμο σχήμα «INT_CUSTOM_ICAP» βασίζεται στην πολύ υψηλής ταχύτητας μνήμη block RAM (BRAM) του FPGA για την αποθήκευση των δεδομένων διαμόρφωσης του μερικού bitstream, καθώς και στον δικής μας κατασκευής ελεγκτή ICAP που διασυνδέει με άμεσο τρόπο (δηλ. χωρίς την χρήση διαύλων επικοινωνίας) την μνήμη BRAM με το ICAPE2 primitive. Ο ελεγκτής ICAP που κατασκευάσαμε, υλοποιεί, ουσιαστικά, στο υλικό (hardware) μία μηχανή καταστάσεων (state machine) μέσω της οποίας τα δεδομένα διαμόρφωσης που είναι αποθηκευμένα στην BRAM αποστέλλονται στην θύρα εγγραφής του ICAPE2 primitive κάθε φορά που πυροδοτούμε την μερική αναδιαμόρφωση της συσκευής μας. Στο Σχήμα 6 φαίνεται το διάγραμμα ροής της εν λόγω μηχανής καταστάσεων.



Σχήμα 6: Διάγραμμα ροής μηχανής καταστάσεων για την μερική αναδιαμόρφωση.

Ο μικροεπεξεργαστής MicroBlaze

Η αναδιαμορφώσιμη αρχιτεκτονική «INT_CUSTOM_ICAP» (βλ. Σχήμα 5) ενισχύθηκε περαιτέρω, με την ενσωμάτωση των δομικών της μονάδων σε ένα σύστημα βασισμένο σε μικροεπεξεργαστή (βλ. Σχήμα 7). Όλες οι μονάδες υλικού (hardware) αυτού του επεξεργαστικού συστήματος (δηλ. ο μικροεπεξεργαστής, η μνήμη και τα περιφερειακά του, κτλ.) υλοποιήθηκαν με αποκλειστική χρήση της προγραμματιζόμενης λογικής και των πόρων του FPGA. Ο μικροεπεξεργαστής μπορούσε να επικοινωνεί με έναν κεντρικό υπολογιστή μέσω σειριακής θύρας (USB-UART Bridge) και έτσι να διευκολύνει την διαχείριση του αναδιαμορφώσιμου συστήματός μας από τον «έξω» κόσμο. Για να επιτευχθεί αυτό, βασιστήκαμε στις έτοιμες προς χρήση μονάδες της Xilinx, MicroBlaze και AXI UART Lite.



Σχήμα 7: Το βασισμένο στον μικροεπεξεργαστή MicroBlaze σύστημα που υλοποιήθηκε στο FPGA.

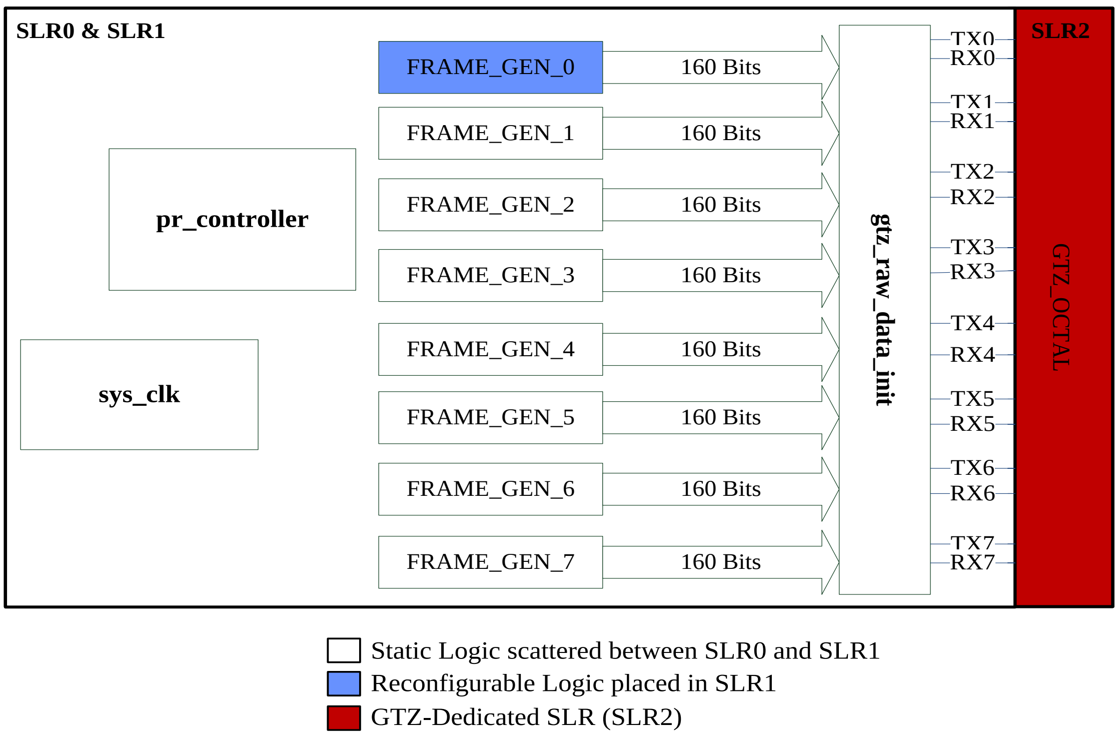
Μία bare-metal εφαρμογή που εκτελείται στον μικροεπεξεργαστή MicroBlaze είναι υπεύθυνη για την πυροδότηση της μερικής αναδιαμόρφωσης της συσκευής βάσει της εισόδου που δέχεται από τον χρήστη του κεντρικού υπολογιστή μέσω της σειριακής θύρας.

Το αναδιαμορφώσιμο 2x100G σύστημα

Στο Σχήμα 8 παρουσιάζεται απλοποιημένα ολόκληρο το σύστημα για το οποίο εφαρμόσαμε το παραπάνω σχήμα αναδιαμόρφωσης (βλ. Σχήμα 7). Βασιζόμενοι σε αυτήν την απλουστευμένη σχηματική απεικόνιση μπορούμε να φανταστούμε ότι το τελικό μας design συντίθεται από τις εξής έντεκα δομικές μονάδες (hardware modules):

- **sys_clk module:** χρησιμοποιείται για την παραγωγή σημάτων χρονισμού (clock signals) όλων των υπόλοιπων μονάδων και κυκλωματικών στοιχείων του hardware design μας.

- **pr_controller module:** πρόκειται για απλοποιημένη αναπαράσταση όλου του MicroBlaze συστήματος που φαίνεται στο Σχήμα 7. Επομένως, αυτή η μονάδα αποτελείται, ουσιαστικά, από πολλές μικρότερες υπομονάδες, που χάριν απλούστευσης όμως, δεν εμφανίζονται.
- **gtz_raw_data_init module:** όπως η μονάδα pr_controller, έτσι και αυτή η μονάδα αποτελείται από πολλές μικρότερες υπομονάδες που υλοποιήθηκαν με την βοήθεια του 7 series FPGAs Transceivers Wizard LogiCORE™ IP (Wizard) της Xilinx. Αυτή η μονάδα, μας δίνει τη δυνατότητα να διαμορφώσουμε όπως επιθυμούμε τους οχτώ επαναρυθμιζόμενους GTZ πομποδέκτες της συσκευής XC7VH580T, και να τους ενσωματώσουμε στο design μας. Μέσω αυτής της μονάδας γίνεται ιδιαίτερα απλή η διασύνδεση της λογικής του χρήστη (user logic) που υλοποιείται στο FPGA με τα κανάλια των GTZ πομποδεκτών για την αποστολή και λήψη δεδομένων.
- **8x FRAME_GEN modules:** οχτώ γεννήτριες δεδομένων (μία για κάθε πομπό GTZ) συνδεδεμένες στην μονάδα gtz_raw_data_init. Κάθε μία από τις οχτώ γεννήτριες (FRAME_GEN_0 - FRAME_GEN_7) τροφοδοτεί ένα από τα οχτώ κανάλια GTZ με δεδομένα προς μετάδοση.



Σχήμα 8: Απλοποιημένη σχηματική απεικόνιση του HW design που υλοποιήθηκε.

Για την περίπτωση της μονάδας FRAME_GEN_0 που αντιστοιχεί στον 1^{ο} πομποδέκτη (GT_0) δημιουργήσαμε δύο διαφορετικές γεννήτριες τις οποίες μπορούσαμε να εναλλάσσουμε δυναμικά (στη θέση της μονάδας αυτής) μέσω της μερικής αναδιαμόρφωσης της συσκευής μας. Έτσι, έχοντας προγραμματίσει την συσκευή μας και ενώ αυτή λειτουργούσε κανονικά, μπορούσαμε να αλλάζουμε την λειτουργία που υλοποιούνταν από την μονάδα FRAME_GEN_0, χωρίς να επηρεάζεται η λειτουργία του υπόλοιπου συστήματος. Με την χρήση ενός scope πολύ υψηλών επιδόσεων, μπορούσαμε να παρακολουθούμε τα δεδομένα που εξέπεμπαν οι πομποί GTZ και έτσι να παρατηρούμε τα αποτελέσματα της μερικής αναδιαμόρφωσης της συσκευής μας σε ένα πραγματικό περιβάλλον εργαστηρίου.

1 Introduction

Information Technologies (ITs) have turned out to be part and parcel of our society, having a deep impact on the modern social and economic way of living, and enriching our daily lives with a variety of services from media entertainment (e.g. video) to more sensitive and safety-critical applications (e.g. e-commerce, e-Health, first responder services, etc.). Mostly driven by streaming video, HD video, cloud computing and mobile networking (migrating from 4G to 5G), consumer market's demand for network bandwidth is steadily increasing. If specialists' predictions hold true, then over the next few decades almost every physical device and everyday object we see (e.g. clothes, cars, trains, etc.) will also be connected to Internet Protocol (IP) networks (Internet of Things - IoT). Also, according to Cisco's Visual Network Index (VNI), that tracks and forecasts bandwidth growth, global IP traffic is projected to reach 396 Exabytes (EB) per month in 2022 (Figure 1.1).

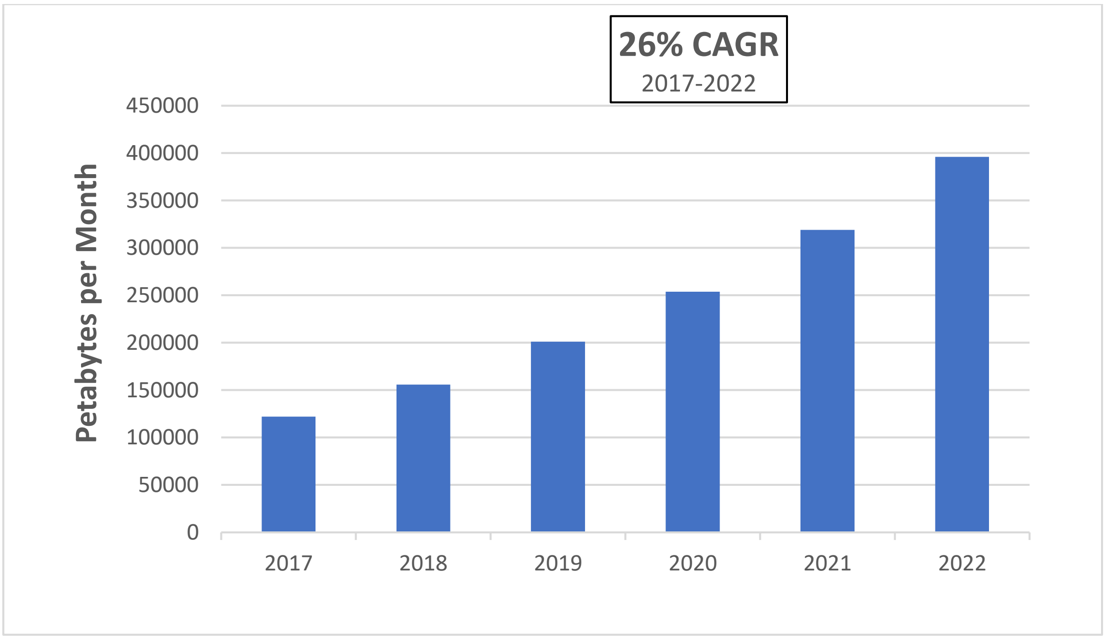


Figure 1.1: Global IP Traffic Forecast from Cisco VNI for 2017-2022.

Offering extremely high bandwidth, low power loss, low security risk and resilience to electromagnetic interference (EMI), small in size and light in weight, optical fibers are the clear choice in order to manage the dynamic and ever-increasing internet traffic. At this time, the majority of network infrastructure is connected via optical fiber. Optical fibers will keep replacing copper wire, as the communications industry is moving to the next-generation optics, developing Nx100G line cards for networking systems [1].

Optical fiber is, undoubtedly, the medium to be used in order to support the needs of today's and tomorrow's telecommunication networks. However, being able to transfer big amounts of data really fast, without the ability to manage and process them at a similar rate, would be useless. Obviously, electronic systems should also scale with this bandwidth demand. The challenge faced by system designers is not only processing the incoming data, but managing the data flow in and out of semiconductor devices. Because the number of pins on these devices does not scale proportionally with transistor count and logic capacity, each pin must allow the accommodation of more total traffic. With the advantages of fewer pins, simpler system clocking, lower cost and lower EMI, high-speed serial interfaces are ideal for managing traffic on- and off-chip [2].

New and more sophisticated Field-Programmable Gate Arrays (FPGAs) that deliver greater capacity, performance and features, along with more robust serial transceivers supporting higher line rates, can meet next-generation networking requirements. FPGAs play a critical role in networking equipment, not only because of their ability to combine high performance processing with low power consumption and small physical size, but also because of their flexibility and ability to rapidly implement the latest networking standards, even as these standards continue to evolve. Consequently, for systems that need high bandwidth, high density, high performance, design flexibility and low cost, FPGAs with high-speed serial I/O (Input/Output) are the clear choice [1], [2].

In addition to all these advantages mentioned, FPGAs can also support reconfigurable computing systems. Reconfigurable computing is the concept that bridges the gap between the separate worlds of hardware (HW) and software (SW) design, combining the flexibility of software running on general purpose processors with the efficiency of high-performance computing platforms, such as FPGAs. The key-feature of reconfigurable computing is the ability to adapt the hardware by "loading a new circuit" on the reconfigurable fabric upon the requirements of the system. Partial Reconfiguration (PR) technology, offered by FPGAs, takes this asset one step further, allowing designers to change hardware's functionality on the fly, by reconfiguring only a limited and predefined part of the FPGA while the remainder of the device continues to operate normally. The key advantages of PR are the reconfiguration "on-the-fly" stated above (without shutting down the system), as well as the reduced reconfiguration time compared to the reconfiguration of the entire fabric.

1.1 Motivation and Thesis Objectives

The continuous evolution of networks leads to ever-bigger network infrastructure(s) that is (are) very hard to be managed, with increased resource and energy demands. For this reason, along with the efforts to fulfill the insatiable need for high-speed and increased bandwidth networks, network service providers are also looking for ways to reduce capital expenses (CapEX) and operating expenses (OpEX). Responding to this need, some revolutionary technologies like Network Function Virtualization (NFV) and Software Defined Networks (SDN) have been introduced to simplify the network management and bring innovation through network programmability. Software-based networks are aiming to reduce the dependency on the underlying physical network (hardware), providing many benefits when compared with traditional hardware-based network architectures, such as faster service enablement, ease of deployment and management, reduced equipment costs, better scalability, availability, flexibility and fine-grain control of traffic. On the other hand, traditional networks that are mainly based and implemented on dedicated appliances (hardware) using Application Specific Integrated Circuits (ASICs), are known to offer significant performance and power efficiency advantages compared with virtual and software-based networks. Service providers have been brought face to face with the eternal problem of finding this architecture that best combines high efficiency with high flexibility.

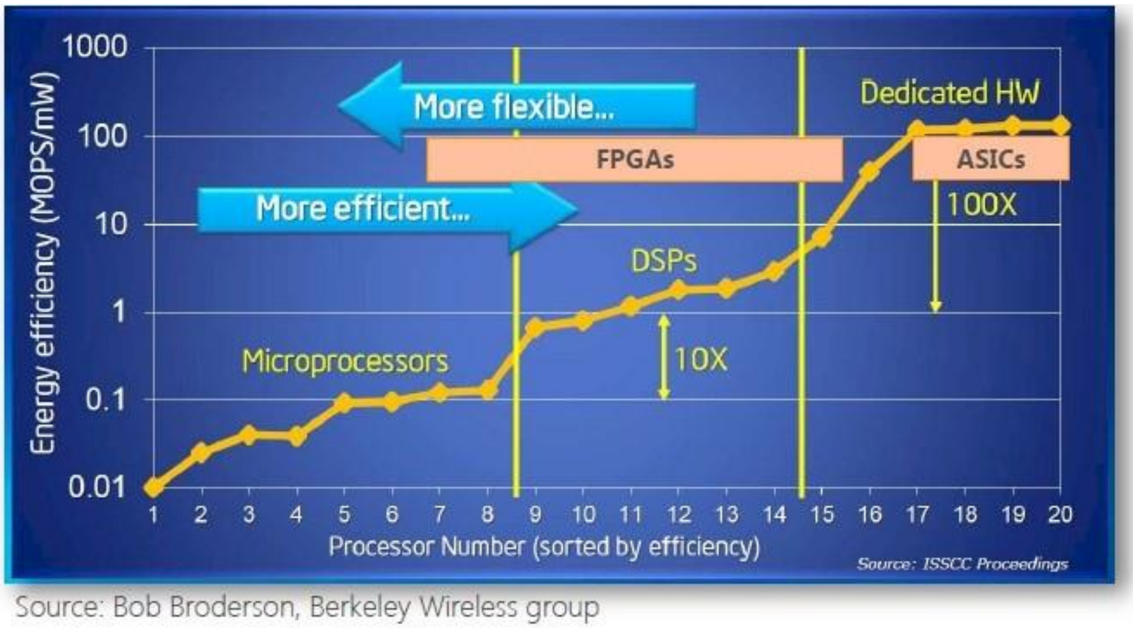


Figure 1.2: Processing efficiency vs flexibility.

As the above handy graph from Microsoft (Figure 1.2), FPGA seems to be the technology to bridge the gap between the efficiency of a hardware-based architecture and the flexibility offered by software. Especially, by using the technique of Partial Reconfiguration, just a single FPGA is able to support different network applications/protocols by loading different bit images at any time, really fast, and without disturbing networking flows and services. This flexibility, offered by FPGAs, through their programmability, is a significant characteristic that enables networks to support fast time to market for new services and efficient scaling, while preserving the performance and power efficiency of a hardware-centric approach [3]. Of course, when it comes to systems which are expected to manage and process data at extremely high rates, reconfiguration time has a deep impact on the system's overall performance.

In the context of this thesis, we investigate the challenges of implementing a reconfigurable telecom system in high-performance FPGAs. We are exploring the technique of Partial Reconfiguration, examining different architectures and reconfiguration schemes in order to improve the reconfiguration time. Additionally, we deal with the challenges of synthesizing an embedded soft processor core onto the FPGA to better control the partial reconfiguration flow ("on-the-fly" changes on hardware's functionality), and to facilitate the management of our system from the outside world (e.g. communication with another processor). For our study we employed Xilinx's XC7VH580T-G2HCG1155E FPGA device, hosting eight embedded high-speed transceivers, each of which is able to operate at serial bit rates up to 28.05 Gigabits per second (Gbps). We implement in the target-device all this functionality and logic needed to properly configure its eight transceivers, resulting in a telecom system with a total bandwidth of 2x100 Gbps, whose functionality can be dynamically defined by a software application running on the embedded soft processor core, while its components are efficiently implemented in hardware.

1.2 Chapter Overview

The remainder of this thesis is organized as follows:

- In Chapter 2, we focus on the theoretical background required for understanding the core concepts of FPGAs and serial multi-gigabit transceivers, thus also showing how these two technologies complement each other. In addition, we introduce the reader to our target-device (i.e. the XC7VH580T-G2HCG1155E) and the VC7222 evaluation board which provides the hardware environment for characterizing and evaluating the serial multi-gigabit transceivers available on the XC7VH580T device.
- In Chapter 3, we first deal with the technical background of this thesis; Partial Reconfiguration is discussed along with the XC7VH580T available interfaces to perform PR and limitations to the reconfigurable fabric. After that we present the

various reconfigurable architectures that were tested and compared in terms of reconfiguration time in the context of this thesis.

- In Chapter 4, we present in detail the workflow that was followed to implement our reconfigurable, high-bandwidth telecom system on the VC7222 evaluation board. We discuss the structure and role of each individual component in the hardware design, the PR workflow and the procedure for setting up the VC7222 board to test and demonstrate our design on real hardware.
- Finally, in Chapter 5, we summarize our work and refer to issues we did not have the chance to work on and which could be regarded as future work.

2 Background on FPGAs & Multi-Gigabit Transceivers

In the context of this thesis we are going to use the Xilinx's VC7222 Evaluation Board, featuring the high-end XC7VH580T FPGA device which include special hard-wired multi-gigabit transceiver blocks. In order to familiarize the reader with the main concepts concerning Field-Programmable Gate Arrays (FPGAs) and serial Multi-Gigabit Transceivers (MGTs), in this chapter we are going to provide the necessary background and theoretical knowledge, along with some brief historical references regarding the necessities and reasons that led to the emergence and establishment of these technologies. In Section 2.1 FPGAs are introduced. In Section 2.2 basic information about serial MGTs is provided. Finally, in Section 2.3 our target board (VC7222) is presented.

2.1 Field-Programmable Gate Arrays

For many years, the design of digital circuits used to be something that only big companies was able to do. It used to be a very time-consuming and costly process, since it was mainly based on creating Application Specific Integrated Circuits (ASICs), which involves the interconnection of large amounts of individual chips, in order to implement various complex logic functions. This situation changed by the introduction of Field Programmable Gate Arrays (FPGAs) which, undoubtedly, brought one of the greatest revolutions in the design of digital circuits. Unlike the ASICs, FPGAs are not custom manufactured to perform only one specific task. They are a particular type of digital Integrated Circuits (ICs) whose functionality can be defined at run time [4].
As it is indicated from its full name, FPGA is essentially an array of logic gates (Gate Array) which can be programmed and reconfigured over and over again, at any time and any place (Field Programmable), being able to implement almost any digital circuit that we can imagine.

Programming of these logic gates within an FPGA, involves the loading of a bit image, named *bitstream,* on the FPGA device. Several technologies can be used in order to store the configuration bitstream in an FPGA.

The majority of commercially available FPGAs is based on Static Random-Access Memory (SRAM) technology. Some vendors may also offer device families of FPGAs based on Flash memory or Antifuse technology. FPGAs that hold their configuration patterns in SRAM memory-cells (Figure 2.1) are more energy-intensive, but offer higher densities, more flexibility and better performance. SRAM memory cells are volatile. This means that SRAM-based FPGAs can’t keep configuration data without power source. For this reason, these devices always need to be programmed (configured) when power is first applied to the system. For the purpose of this thesis, we are only interested in FPGA devices that are based on SRAM technology, so any information provided for FPGAs, is referred to SRAM-based devices.

Since we have decided on the underlying technology we are interested in, we should take a closer look at the internal structure and the building blocks of an (SRAM-based) FPGA in order to better understand why and how this device can be so flexible.

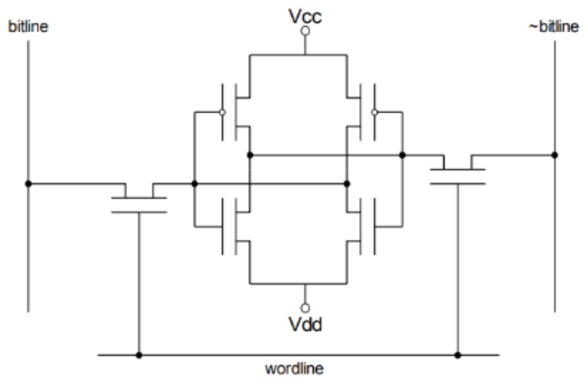


Figure 2.1: SRAM cell.

2.1.1 *Architectural structure of an FPGA*

The internal structure of an FPGA may be very complex and its structural blocks can vary from vendor to vendor or even from family to family (for FPGAs of the same vendor). In fact, different vendors, often, don’t even use the same terminology in order to refer to these building blocks (we are going to use Xilinx’s terminology when it is needed). However, typically, an FPGA consists of three basic components: a large array of general-purpose configurable Logic Cells (LCs), hard-wired blocks specialized for certain tasks (e.g. memory and arithmetic operations) and a network of flexible programmable interconnects for their linking. Combining configurable LCs with dedicated-functionality blocks reduces power consumption of an FPGA and increases its performance. LCs are combined together to form larger structures named Configurable Logic Blocks (CLBs). Along with CLBs, specific-functionality blocks are provided within an FPGA, like RAM memory blocks (Block RAMs), Digital Signal Processing blocks (DSPs), input/output blocks (IOBs) and even more complicated components such as Digital Clock Manager (DCM), DRAM memory controller, etc. [5].

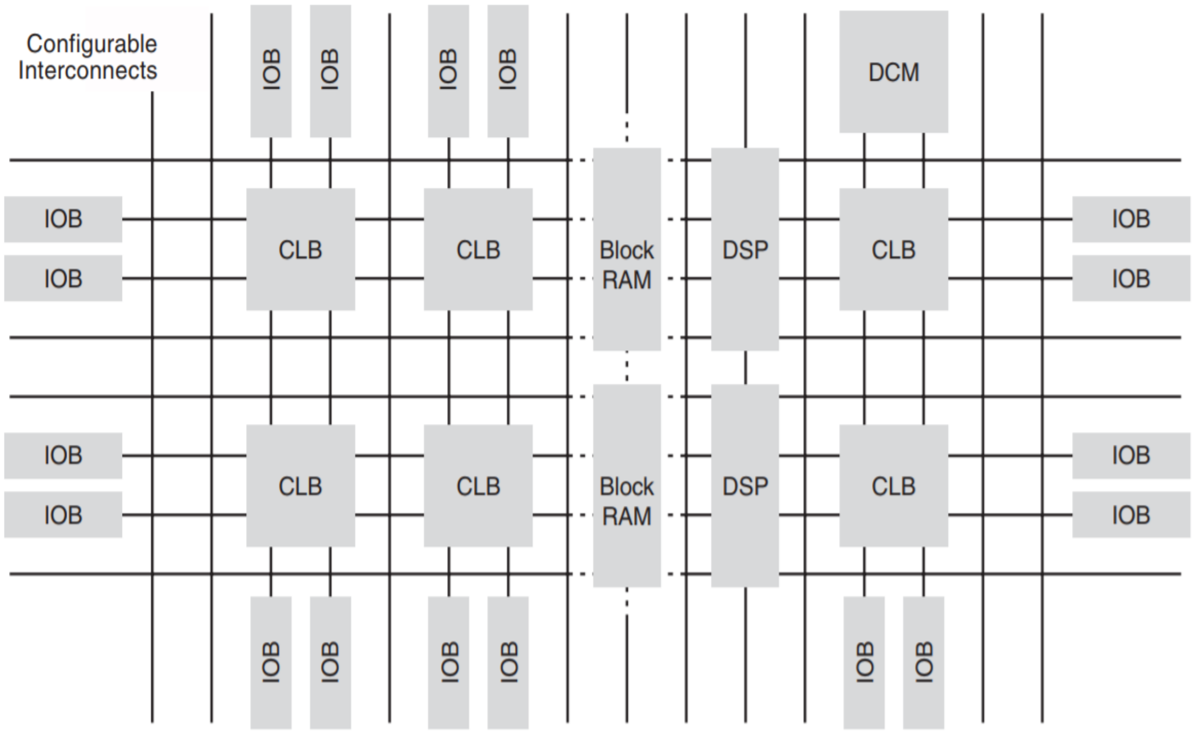


Figure 2.2: A highly simplified architectural layout of FPGA [5].

DSP blocks and CLBs can be configured to perform arithmetic and logic operations (add, multiply, compare, etc.), like the arithmetic logic unit (ALU) of a processor. Contrary to a processor's ALU whose architecture is fixed and designed in a general-purpose manner to execute various operations, the CLBs can be programmed with just the operations needed by the application, resulting in increased compute efficiency. Operations' results and data, are stored in registers present in CLBs, DSPs and block RAM, and are able to directly flow from the output of one block into the input of another using a network of flexible programmable interconnects. Thus, there are no inefficiencies like processor cache misses. FPGA data can be streamed between operators, and these operators can be fully pipelined, establishing links between them by using configurable interconnects. In a few words, FPGAs allow us to create a massive array of application-specific ALUs that enable both instruction and data-level parallelism [5]. In the following subsections, we make a quick reference to FPGA's building blocks.

Logic Cells (LCs) & Configurable Logic Blocks (CLBs)

Logic cell is the fundamental building block of an FPGA. LCs are the reason why FPGAs are able to take on many different hardware configurations. There is not a strict standard to the internal structure of an LC, but generally it comprises an N-input (typically 4-input or 6-input) Look-Up Table (LUT) being able to compute any N-input logic function, one or more registers that can be configured either as flip-flops (synchronous storage) or as latches (asynchronous storage) and an assortment of other elements, such as multiplexers (muxes) and XOR gates, that implement some special carry and control logic.

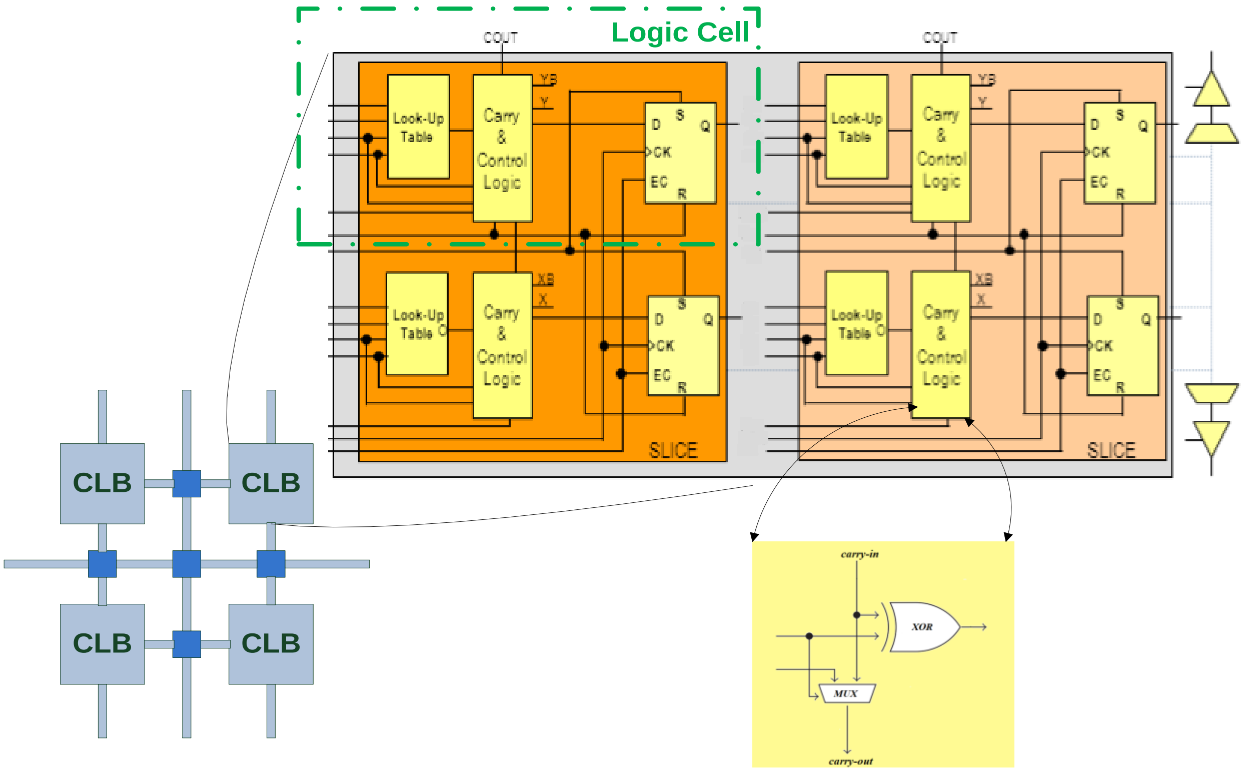


Figure 2.3: A simplified view of a CLB and its components.

As Figure 2.3 depicts, a number of LCs (typically two) are combined into a single unit named slice, and then several slices (typically two or four), linked by dedicated interconnect, are grouped together to form a CLB.

Specialized Blocks

CLBs enable us to implement an extremely wide range of circuits. However, there are some known basic functions, which are often required by several applications, and the implementation of which demands the interconnection of a large number of CLBs, resulting in complex and inefficient designs with excessively high area requirements. For this reason, FPGAs are further strengthened with ready-to-use silicon components specialized for certain tasks, such as BRAMs, multipliers, DSPs, IOBs and in some cases even more complicated components, like memory controllers, Ethernet MACs, high-speed MGTs or embedded microprocessors.

Configurable Interconnects

CLBs and the specialized blocks of an FPGA are able to communicate really fast with each other through a network of programmable interconnects. In fact, a big array of tracks running vertically and horizontally across the chip, creates a huge number of preexisting physical connections between FPGA's blocks. Eventually, programmable interconnects, located at the intersection points of the tracks, are to define which of the connections will be active.

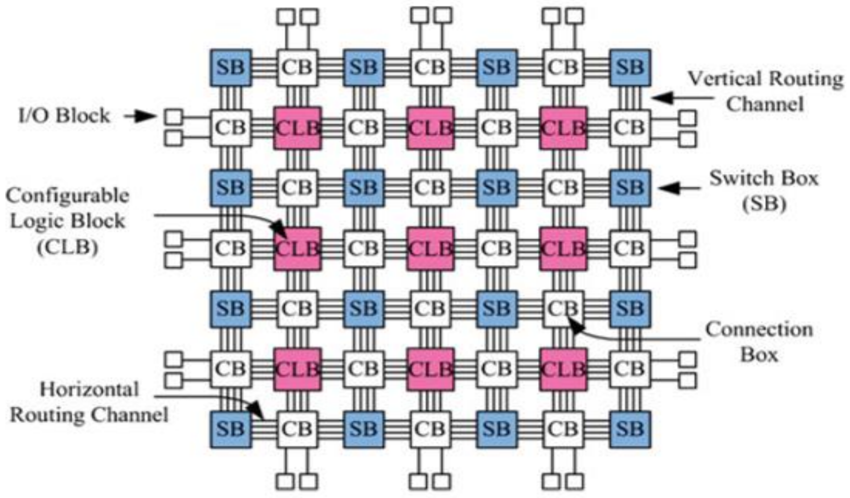


Figure 2.4: Configurable routing architecture within an FPGA [6].

2.1.2 Programming methods

Approaching the issue of programming an FPGA in a general and abstract way, we could say that our ultimate goal through this process is to properly use and configure FPGA's resources (building blocks and associated interconnections) in order to implement a digital circuit in it. So, programming an FPGA, essentially, involves the design of an electronic circuit. An obvious, but also very primitive way of designing an electronic circuit is based on creating schematics, in which all the circuit fundamental elements (e.g. logic gates) and the connections between them are drawn, by hand or using a Computer-Aided Design (CAD) tool, one by one. In the case of a modern FPGA, that requires an enormous number of logic to be correctly configured, it's impossible for a human designer to manually reprogram each element individually. Programming of an FPGA is, usually, done by using specific CAD tools that are able to receive the description of a digital circuit at a higher level of abstraction and undertake the hard work of “drawing” the respective schematic on behalf of us. Most of these tools allow us to describe the desired circuits in more than one way, each of which offers a different level of abstraction. In the following subsections, we make a quick reference to the three more dominant description ways.

Hardware Description Languages (HDLs)

The most common way to describe our design is by using a Hardware Description Language (HDL), like Verilog or VHDL. HDLs offer the Register-Transfer Level (RTL) abstraction, modeling a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals [7].

Intellectual Property (IP) cores

Today’s advanced CAD tools, supporting and encouraging the Electronic Design Automation (EDA) “trend”, provide us with a huge library of Intellectual Property (IP) cores. IP core is a reusable and pre-designed block of logic or data that, ideally, is able to easily be inserted into any vendor technology or design methodology. Universal Asynchronous Receiver-Transmitter (UART), Ethernet controllers, Peripheral Component Interconnect (PCI) interfaces, Dynamic Random-Access Memory (DRAM) controllers, Central Processing Units (CPUs) are some example-components that can be designed stand alone as IP cores. IP cores offer such a level of abstraction that allow us to describe very complex and sophisticated designs extremely fast and easily, without even having to know the details of how these pre-designed “sub-circuits” (IP cores) are described and implemented. The tool, using suitable libraries, can directly map each IP core to the corresponding RTL implementation, allowing us to use these cores as black boxes.

High-Level Synthesis (HLS)

High-Level Synthesis (HLS) is a design methodology that provides an even higher level of abstraction in digital circuit description than the methods mentioned above. HLS tools are able to take the description of a design in a High-Level Language, like C++, as input and turn it into an RTL implementation. HLS has made FPGAs accessible to a wider range of developers and has been very effective in reducing time to market.

2.1.3 Development flow

In most cases, the description of a design is not only based on one particular way from those mentioned in the previous subsection 2.1.2, but is a result of a combination of them. No matter which description way/ways will be used, we can finally make one, single description of our design at the RTL level. After that, we are able to map the design to a specific FPGA architecture through a fully automated process, offered by the CAD tools. As depicted in Figure 2.5 below, this process comprises the following steps: *Synthesis*, *Implementation* and *Bitstream Generation*.

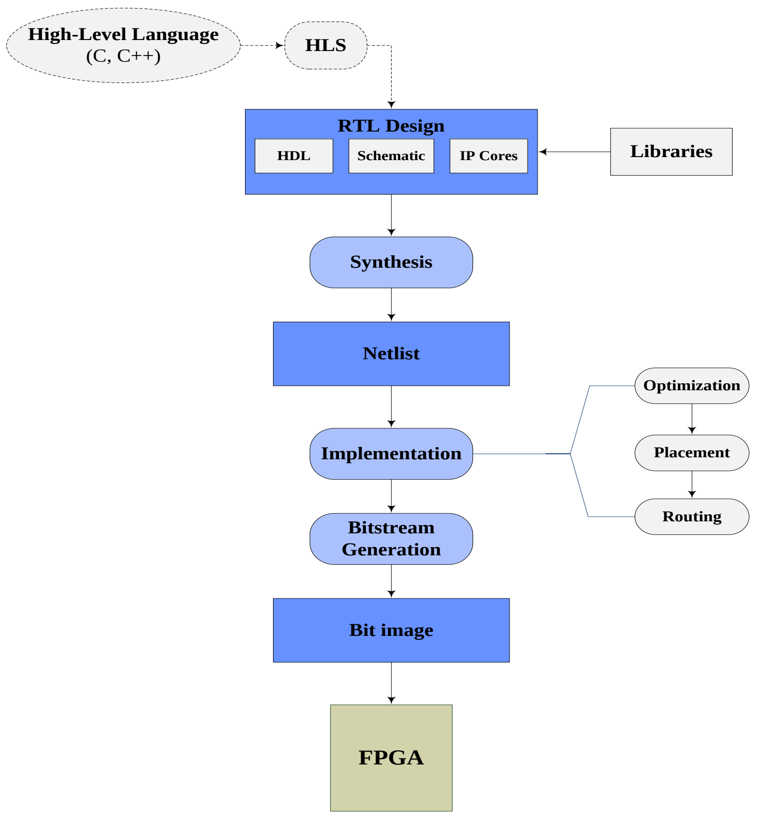


Figure 2.5: A typical FPGA design flow.

In the ***Synthesis*** stage, the tool, after analyzing the hierarchy of our design, recognizing the functionality described and optimizing our design for the device architecture we are targeting for, transforms the RTL-specified design into a netlist of logic gates. The next stage is called ***Implementation***. This stage is broken down to three main steps. First, the logical netlist is optimized to better fit into the available resources on the target FPGA device (*Optimization*) [8]. After that, the tool places logic blocks from the optimized netlist onto specific sites in the FPGA device we are targeting for (*Placement*) [8]. Finally, the proper programmable switches are activated in order to establish all the connections between the previously placed logic blocks (*Routing*). Once the design is successfully placed and routed, the tool, in the last stage called

Bitstream Generation, creates the final bitstream. This is a particular type of binary file that carries all the necessary information to specify the state of each individual element inside the FPGA. This file is all that an FPGA needs in order to be programmed. The bitstream can be downloaded on the FPGA device and configure it to perform the operations of the design we initially described.

2.2 Serial Multi-Gigabit Transceivers

The most common approach to transfer big amounts of data between two or more digital devices connected on a circuit, is using a bus [9]. Bus is a group of tracks (signal lines) that enables the communication between the devices attached to it in parallel mode, allowing multiple data bits to be transferred over multiple channels at the same time.

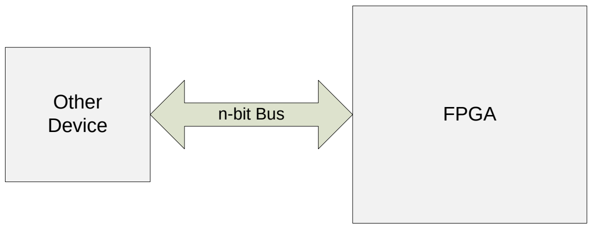


Figure 2.6: Use a bus to transfer data between devices [9].

As digital devices are continuously increasing in size and processing power, buses are also increasing in width (from 8-bit to 16-bit, from 16-bit to 32-bit, from 32-bit to 64-bit and so forth) in order for the devices to exchange data at rates comparable to those at which they process them.
Particularly in regards to FPGA devices, I/O resources and communication with outside world have become the main bottleneck to their vast capability in parallel data processing and their extremely high performance. So, finding a way to efficiently exchange data with outside devices is very important and crucial for their overall performance.
However, increasing buses' width is not a panacea. The problem is that this consumes a lot of pins on each device and requires a lot of tracks to connect the devices together, leading to high-cost solutions. Routing these tracks such that they are all the same length, impedance and so forth becomes really painful and difficult as devices grow in complexity [9]. Furthermore, increasing buses' size makes it really hard to deal with crosstalk effect, simultaneous switching output (SSO) noise, electromagnetic interference (EMI) problems and Signal Integrity (SI) issues generally. Gigabit serial I/O appears to be a more attractive solution (than the traditional data bus) to the complex problem of increasing the I/O bandwidth of a device while taking into account all the other parameters mentioned above.

For this reason, particularly in recent years, special attention was devoted by FPGAs' vendors to equip their devices with hard-wired serial multi-gigabit transceiver (MGT) blocks. And by transceiver we mean the outcome of integrating both a transmitter and a receiver in a single device.

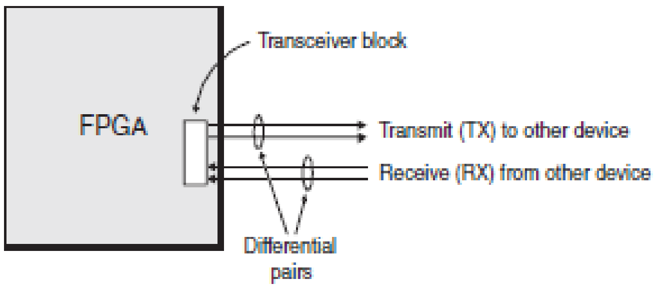


Figure 2.7: Use MGTs to transfer data between devices [9].

A Multi-Gigabit Transceiver is more or less a Serializer/Deserializer (SerDes) that is able of running at line rates of 1Gigabit/second and more. So, MGTs enable devices to transfer parallel data as stream of serial bits, making it easier and cheaper to transfer more data over longer distances by using less I/O pins and connections, compared to buses.

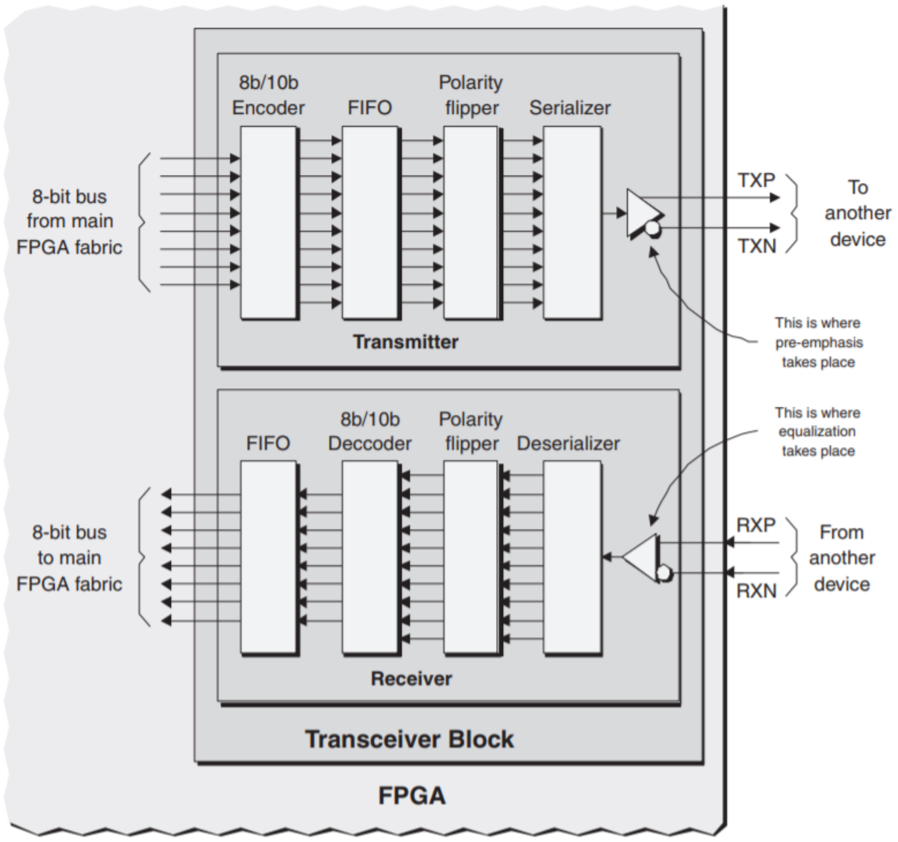


Figure 2.8: A highly simplified depiction of an MGT block [9].

As Figure 2.8 depicts, in case of transmission, bytes (8-bits) of data are delivered from FPGA fabric to the transceiver block using an 8-bit bus. Data are initially encoded using a proper encoding scheme (8b/10b in this simplified example), and then pass through a polarity flipper which turns zeros into aces and vice versa (the flipper may be bypassed if it is not required). A FIFO (First In – First Out) is used between the encoder and the polarity flipper in order to temporarily store data when bits are pulled out from the transmitter in a lower rate than that with which they are presented to it. Finally, a serializer converts the parallel data, coming out of the polarity flipper, into a stream of serial bits, and pass them to a particular buffer that transmits each one of them as a differential pair of signals [9]. Respectively, the receiver subblock converts an incoming stream of serial bits into 10-bits words and following the reverse procedure passes bytes (8-bits) of data to FPGA fabric for processing.
Figure 2.8 is an extremely simplified representation of an MGT block, and we use it just in order to make a smooth introduction to the basic elements that, typically, make up a Multi-Gigabit Transceiver. For the purposes of the above oversimplified example, we assumed an 8-bit width data bus and an 8b/10b encoder. However, the size of data bus, the encoding scheme and some other parameters are directly dependent on the type (family) of the transceiver that is used. In fact, the MGT blocks embedded in FPGAs typically have a number of configurable (programmable) features (e.g. comma detection, pre-emphasis, equalization, etc.), and thus, when programming the FPGA, we are able to define the size of data bus, the encoding scheme, the transmitting/receiving data rates, etc. by selecting between a few valid and permissible choices. This is why just a single FPGA device with embedded MGTs is able to support and implement not only multiple known and predefined interface standards (e.g. XAUI, PCI Express, Aurora, etc.) but custom protocols as well.

2.3 VC7222 Evaluation Board

The VC7222 board (Figure 2.9) provides the hardware environment for characterizing and evaluating the GTH and GTZ transceivers available on the XC7VH580T-G2HCG1155E FPGA device. The XC7VH580T, member of the Xilinx Virtex®-7 family, is not just an ordinary, monolithic FPGA (and by “monolithic” we mean that there is one silicon die in the package). XC7VH580T is one of Xilinx® 2.5D ICs that is designed using the Stacked Silicon Interconnect (SSI) technology. An SSI device is one in which multiple silicon dies are connected together via silicon interconnect, and packaged into a single device. An SSI device enables high-bandwidth connectivity between multiple dies by providing a much greater number of connections. It also imposes much lower latency and consumes dramatically lower power than either a multiple FPGA or a multi-chip module approach, while enabling the integration of massive quantities of interconnect logic, transceivers, and on-chip resources within a single package. Every single die slice contained in an SSI device is a Super Logic Region (SLR). Special routing resources running between SLRs, the Super Long Line (SLL) resources, are used to connect the logic from one region to the next [10]. Our target-device (i.e. the XC7VH580T) has the further peculiarity of being heterogeneous, since the SLR components it comprises are not identical. Actually, the XC7VH580T is made up of two FPGA die slices (SLR0 and SLR1) and an 8-channel 28Gbps transceiver die (SLR2).

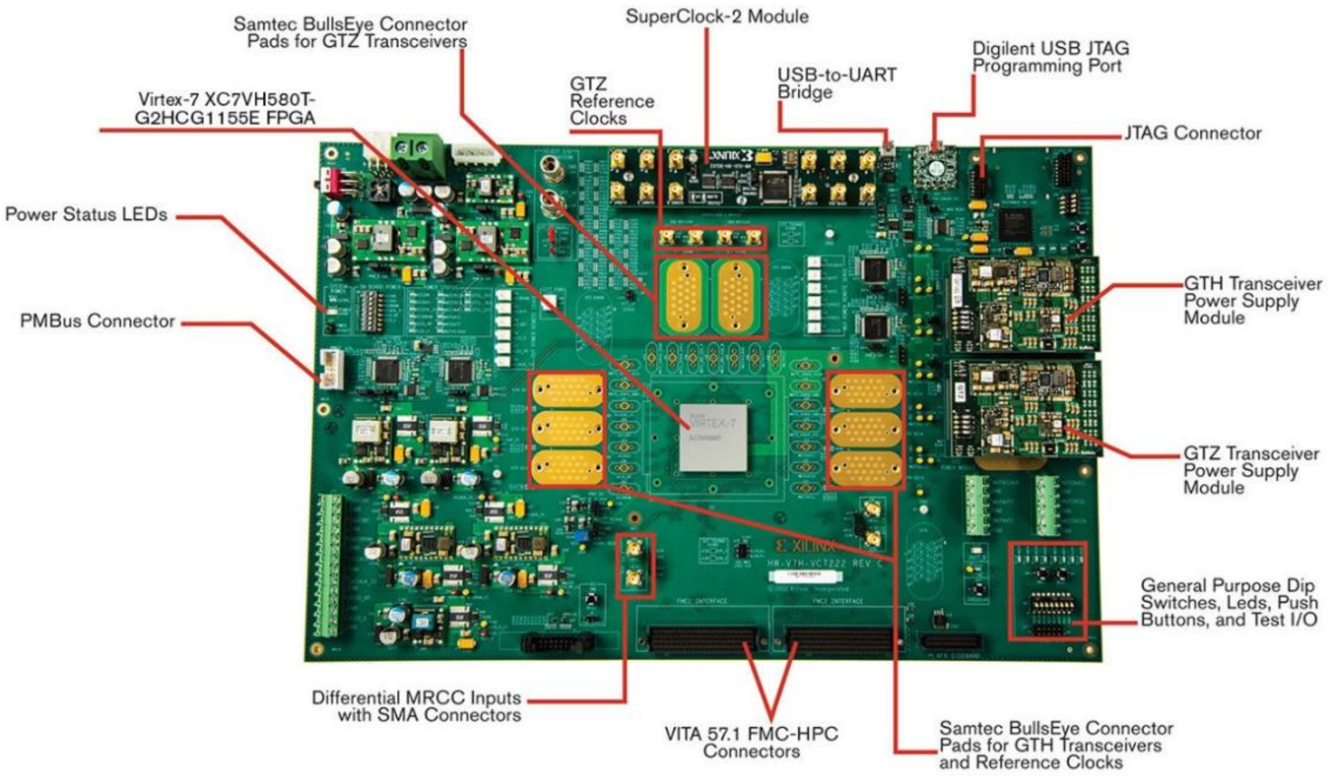

Samtec BullsEye Connector Pads for GTZ Transceivers
SuperClock-2 Module
Digilent USB JTAG Programming Port
USB-to-UART Bridge
GTZ Reference Clocks
Virtex-7 XC7VH580T-G2HCG1155E FPGA
JTAG Connector
Power Status LEDs
PMBus Connector
GTH Transceiver Power Supply Module
GTZ Transceiver Power Supply Module
General Purpose Dip Switches, Leds, Push Buttons, and Test I/O
Differential MRCC Inputs with SMA Connectors
VITA 57.1 FMC-HPC Connectors
Samtec BullsEye Connector Pads for GTH Transceivers and Reference Clocks

Figure 2.9: The VC7222 Evaluation Board [11].

The VC7222 board features can be summarized as follows [12]:

- Virtex-7 XC7VH580T-G2HCG1155E FPGA
- Onboard power supplies for all necessary voltages
- Terminal blocks for optional use of external power supplies
- Digilent USB JTAG programming port
- System ACE™ SD controller
- Power module supporting Virtex-7 FPGA GTH transceiver power requirements
- Power module supporting Virtex-7 FPGA GTZ transceiver power requirements

- A fixed, 200 MHz 2.5V LVDS oscillator wired to multi-region clock capable (MRCC) Inputs
- Two pairs of differential MRCC inputs with SMA connectors
- SuperClock-2 module supporting multiple frequencies
- Six Samtec BullsEye connector pads for the GTH transceivers and reference clocks
- Two Samtec BullsEye connector pads for the GTZ transceivers and two pairs of SMA connectors for GTZ transceiver reference clocks
- Power status LEDs
- General purpose DIP switches, LEDs, pushbuttons, and test I/O
- Two VITA 57.1 FPGA mezzanine card (FMC) high pin count (HPC) connectors
- USB-to-UART bridge
- I2C bus
- PMBus connectivity to onboard digital power supplies
- Active cooling for the FPGA

The VC7222 board block diagram is shown in the following figure [12]:

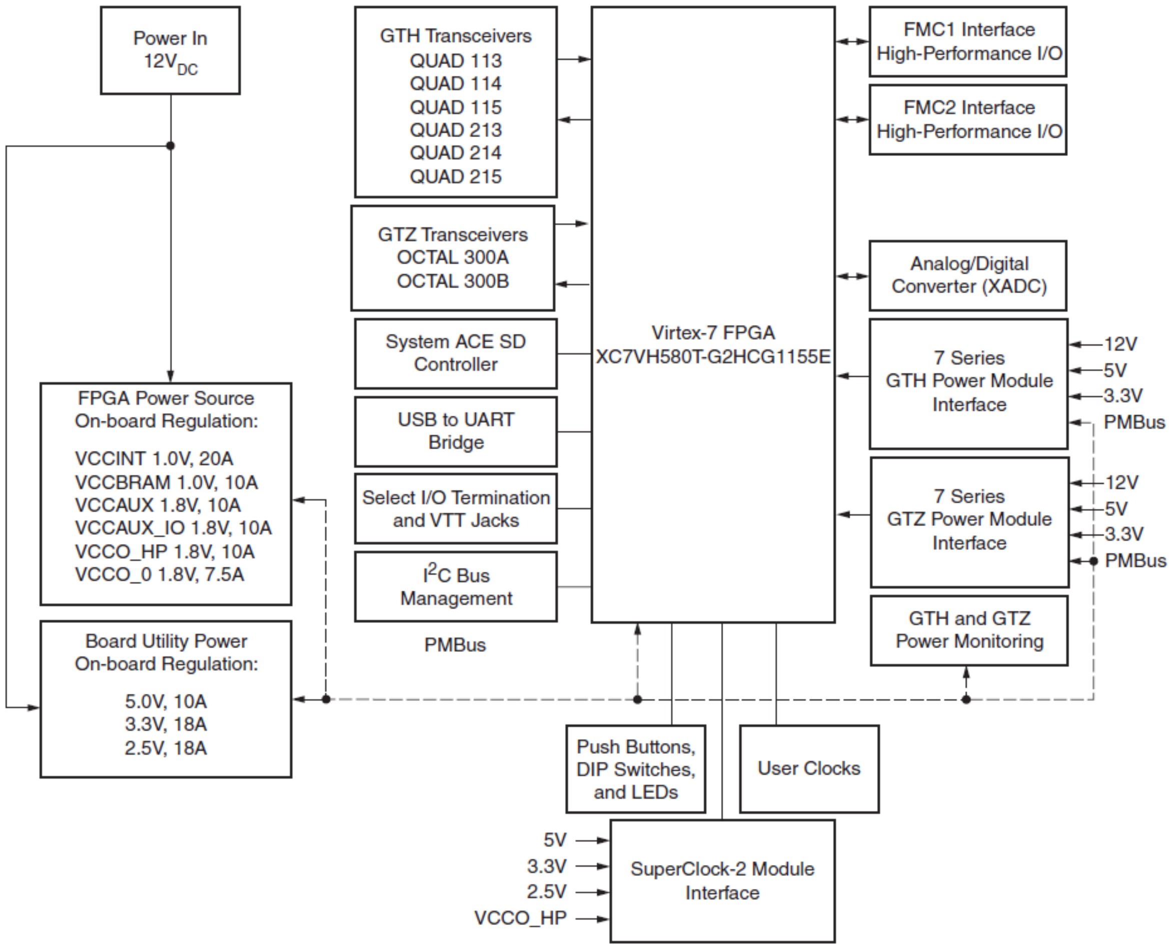


Figure 2.10: The VC7222 Board Block Diagram [12].

3 Partial Reconfiguration

In this chapter we will focus on the technique of Partial Reconfiguration (PR) in FPGA. First, we give a generic overview of this technique in Section 3.1. In Section 3.2 we describe the workflow that was followed to implement the PR through the Xilinx's Vivado® Design Suite. In Section 3.3 we discuss about the different modes and interfaces that can be used to download (full and partial) configuration files (i.e. bitstreams) to our target-device. Finally, in Section 3.4 we present the various reconfigurable architectures that were tested and compared in terms of reconfiguration time in the context of this thesis.

3.1 Overview

The key advantage of FPGAs is their flexibility to be programmed and re-programmed at any time and any place ("in the field"). Partial Reconfiguration (PR) takes this advantage one step further, by allowing partial modification of an operating FPGA design, while the rest of the design continues to function normally. This is done by loading a partial configuration file, usually a partial BIT file. After a full BIT file configures the FPGA, partial BIT files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured [13].

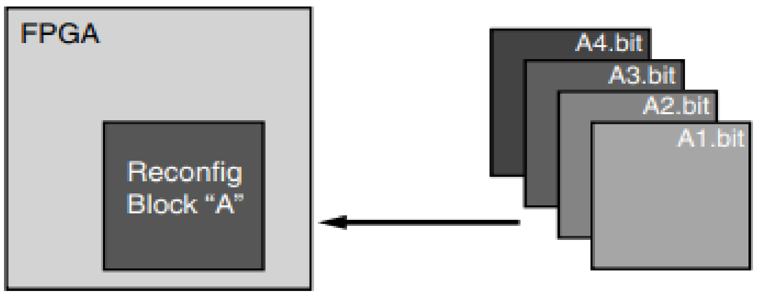


Figure 3.1: Basic premise of Partial Reconfiguration [13].

As shown above (Figure 3.1), the function implemented in Reconfig Block “A” is modified by downloading one of several partial BIT files, A1.bit, A2.bit, A3.bit or A4.bit. The logic in the FPGA design is divided into two different types, reconfigurable logic and static logic. The gray area of the FPGA block represents static logic and the block portion labeled Reconfig Block “A” represents reconfigurable logic. The static logic remains functioning and is unaffected by the loading of a partial BIT file. The reconfigurable logic is replaced by the contents of the partial BIT file. Usually, we refer to Reconfig Block “A” as Reconfigurable Partition (RP), while partial BIT files (A1.bit, A2.bit, etc.) constitute the Reconfigurable Modules (RMs) of this RP [13].

Partial bitstreams, that are delivered during normal device operation to replace functionality in a pre-defined device region, have the same structure as full bitstreams. They are fully self-contained, so they are delivered to an appropriate configuration port. Loading a partial bitstream into an FPGA does not require knowledge of the physical location of the RM. All addressing, header, and footer details are contained within partial bitstreams, just as they would be for full configuration bitstreams. The only difference is that partial bitstreams are limited to specific address sets to program only a specific part of the device. The size of a partial bitstream is directly proportional to the size of the region it is reconfiguring. For example, if the Reconfigurable Partition is composed of 20% of the device resources, the partial bitstream is roughly 20% the size of the full design bitstream. This is why partial reconfiguration is much faster than a standard full configuration of a device [13].

Apart from the module-based PR, where a limited, pre-defined part of the FPGA is completely reconfigured with a new (partial) design, FPGAs can also support a difference-based PR. Difference-based PR is only useful when the different partial designs described by two (or more) Reconfigurable Modules have common logic, and therefore the differences between them are very small. In this case, we can take the PR one step further by reconfiguring just the part of the Reconfigurable Partition that is related with these differences. A difference-based partial bitstream is generated based only on the differences between two Reconfigurable Modules, and, consequently, its size is even smaller than that of the equivalent module-based partial bitstream. In the context of this thesis, we will only deal with the module-based PR as it is the most widely used and considered one.

PR advantages

PR makes it possible to time multiplex hardware dynamically on a single FPGA device, offering multiple benefits, such as:

- Reducing the size of the FPGA required to implement a given function, with consequent reductions in cost and power consumption.
- Providing flexibility in the choices of algorithms or protocols available to an application without hindering overall application functionality (in terms of powering off the device)
- Enabling new techniques in design security
- Improving FPGA fault tolerance
- Accelerating configurable computing

In addition to reducing size, weight, power and cost, PR also enables new types of FPGA projects that would otherwise be impossible to implement [13].

PR limitations

Not all logic resources are permitted to be actively reconfigured. Global logic and clocking resources must be placed in the static region to not only remain operational during reconfiguration, but to benefit from the initialization sequence that occurs at the end of a full device configuration. The eligible resources that can be placed in a Reconfigurable Module in our device includes [13]:

- All logic components that are mapped to a CLB slice in the device. This includes LUTs (Look-Up Tables), FFs (flip-flops), SRLs (Shift Registers), RAMs (Random Access Memories) and ROMs (Read-Only Memories).
- Block Ram and FIFO
- DSP blocks: DSP48E1
- PCIe® (PCI Express): Entered using PCIe IP

All other logic must remain in static logic and must not be places in an RM, including [13]:

- Clocks and Clock Modifying Logic - Includes BUFG, BUFR, MMCM, PLL, and similar components
- I/O and I/O related components (ISERDES, OSERDES, IDELAYCTRL, etc.)
- Serial transceivers (MGTs) and related components
- Individual architecture feature components (such as BSCAN, STARTUP, XADC, etc.)

3.2 Xilinx Partial Reconfiguration Workflow

The Xilinx Vivado® Partial Reconfiguration design flow is similar to a standard design flow, with some notable differences. The implementation software automatically manages the low-level details to meet the silicon requirements. Our role is to define the size and location of the reconfigurable area via the floorplan utility. The following steps summarize the Vivado PR workflow [13]:

1. Synthesize the static and Reconfigurable Modules separately.
2. Create physical constraints (Pblocks) to define the reconfigurable regions.
3. Set the HD.RECONFIGURABLE property on each Reconfigurable Partition.
4. Implement a complete design (static and one Reconfigurable Module per Reconfigurable Partition) in context.
5. Save a design checkpoint for the full routed design.
6. Remove Reconfigurable Modules from this design and save a static-only design checkpoint.
7. Lock the static placement and routing.
8. Add new Reconfigurable Modules to the static design and implement this new configuration, saving a checkpoint for the full routed design.
9. Repeat Step 8 until all Reconfigurable Modules are implemented.
10. Run a verification utility (pr_verify) on all configurations.
11. Create bitstreams for each configuration.

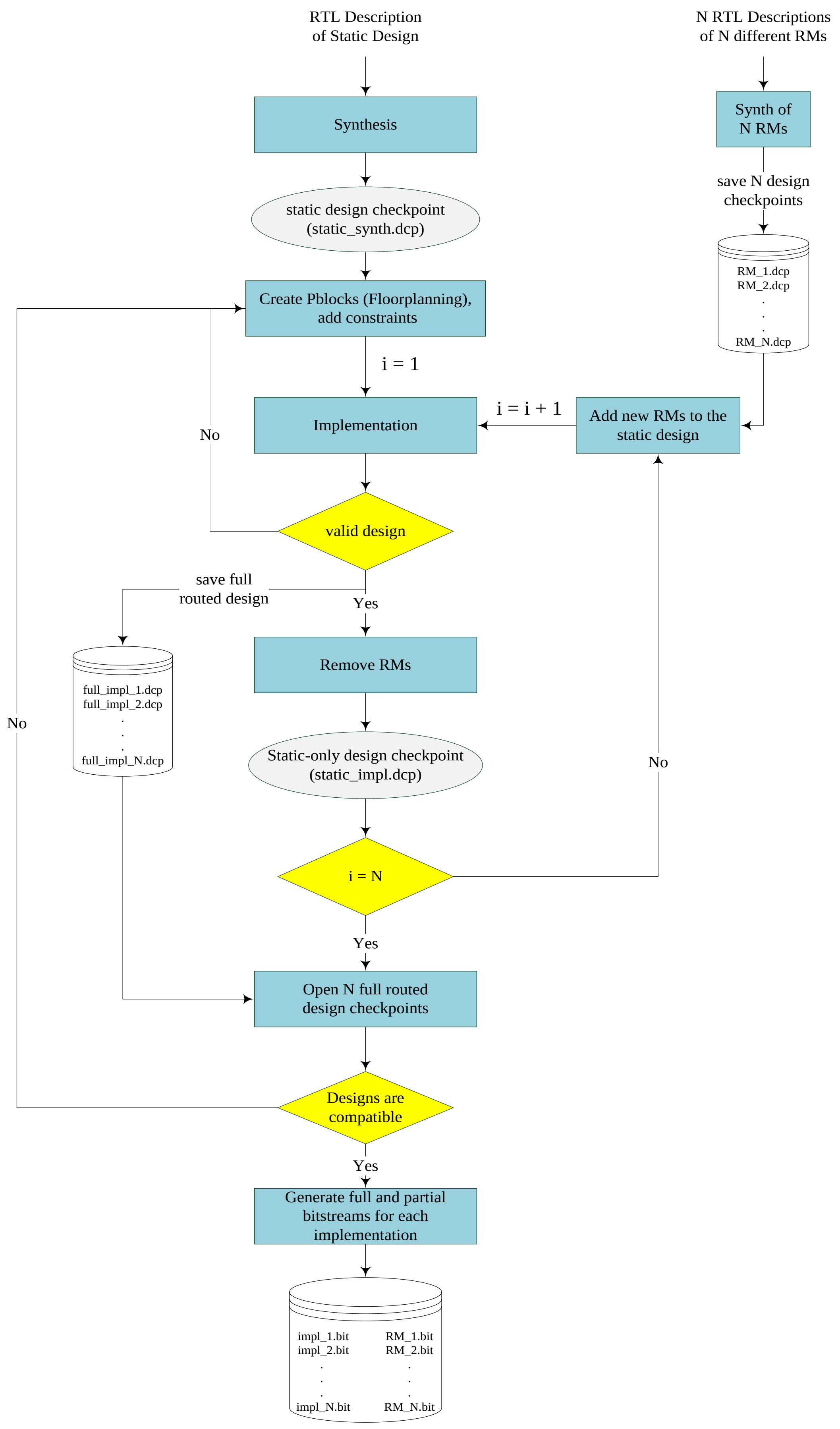

RTL Description of Static Design
N RTL Descriptions of N different RMs
Synthesis
Synth of N RMs
save N design checkpoints
static design checkpoint (static_synth.dcp)
RM_1.dcp
RM_2.dcp
RM_N.dcp
Create Pblocks (Floorplanning), add constraints
i = 1
i = i + 1
Implementation
Add new RMs to the static design
No
valid design
save full routed design
Yes
Remove RMs
full_impl_1.dcp
full_impl_2.dcp
full_impl_N.dcp
No
Static-only design checkpoint (static_impl.dcp)
No
i = N
Yes
Open N full routed design checkpoints
Designs are compatible
Yes
Generate full and partial bitstreams for each implementation
impl_1.bit
impl_2.bit
impl_N.bit
RM_1.bit
RM_2.bit
RM_N.bit

Figure 3.2: Partial Reconfiguration Workflow.

As depicted in the flowchart in Figure 3.2, PR flow requires the implementation of multiple configurations which ultimately results in full bitstreams for each configuration, and partial bitstreams for each Reconfigurable Module. The number of configurations required varies by the number of modules that need to be implemented. However, all configurations share the same top-level, or static, placement and routing results. These static results are exported from the initial configuration, and imported by all subsequent configurations using checkpoints. The static logic remains identical in every configuration and only the reconfigurable logic differs [13]. Therefore, although each Reconfigurable Module is free to use and configure the programmable logic within the corresponding Reconfigurable Partition, their interface with the unchanged static design must also be kept unchanged, so that they can properly fit into the reconfigurable region defined in the floorplanning step (Step 2). This is a very important constraint to remember when processing a PR design.

3.3 Configuration Modes in Virtex-7 VH580T

As already mentioned in the previous chapter, our target-device is a member of Virtex®-7 family. Virtex®-7 is one of the four families that Xilinx® 7 series FPGAs comprise. Almost all 7 Series devices support the same configuration interfaces and the same configuration modes. In the following subsections we are going to describe in detail how we can use these interfaces in order to configure our target-device, first, with a full and, later, with a partial configuration file.

3.3.1 External Configuration Interfaces

The simplest way to configure the XC7VH580T FPGA device with a full or partial bitstream is via JTAG, using one of the following options:

- USB JTAG connector
- System ACE SD
- JTAG cable connector

JTAG is a well-known standard, used by the industry for prototyping and debugging and is the default interface used to configure the device through Vivado Logic Analyzer, regardless of whether we use a full or a partial bitstream. The VC7222 board comes with an embedded USB-to-JTAG configuration module which allows a host computer to access the board JTAG chain using a standard A to micro-B USB cable. Alternately, the FPGA can be configured via system

ACE from a Secure Digital (SD) memory card. Finally, a JTAG connector is available to provide access to the JTAG chain using one of Xilinx's configuration cables—Platform Cable USB, Platform Cable USB II or Parallel Cable IV (PCIV) [12].

JTAG is one of the five external configuration interfaces available on Xilinx® 7 series FPGA devices. The other four external interfaces that can be used are the SelectMAP interface, the serial interface, the Serial Peripheral Interface (SPI) and the Byte Peripheral Interface (BPI).
When one of the SPI or BPI interfaces is used, the FPGA loads (or boots) itself from an external SPI or BPI nonvolatile flash memory device, respectively. In this case, the FPGA acts as a Master. The device itself controls the configuration process by driving the configuration clock (CCLK) from an internal oscillator. In the case of serial and SelectMAP interfaces, the FPGA is usually configured by an external smart source (such as a microprocessor, DSP processor, microcontroller, PC, or board tester). During such an externally controlled configuration process, the FPGA acts as a Slave, and the CCLK is an input to it. When using the serial or the SelectMAP interface, 7 series FPGAs can also support the Master mode for configuration from legacy serial PROMs (when applicable) or for custom, CPLD-based configuration state machines driven by the FPGA CCLK [14].
In any case, there are two general configuration datapaths. The first is the serial datapath that is used to minimize the device pin requirements. The second datapath is the 8-bit, 16-bit, or 32-bit datapath used for higher performance or access (or link) to industry-standard interfaces, ideal for external data sources like processors, or x8- or x16-parallel flash memory [14].
To sum up, each one of the external configuration interfaces corresponds to one or more configuration modes and bus width, shown in Table 3.1 below.

Configuration Mode	M[2:0]	Bus Width	CCLK Direction
Master Serial	000	x1	Output
Master SPI	001	x1, x2, x4	Output
Master BPI	010	x8, x16	Output
Master SelectMAP	100	x8, x16	Output
JTAG only	101	x1	Not Applicable (N/A)
Slave SelectMAP	110	x8, x16, x32	Input
Slave Serial	111	x1	Input

Table 3.1: 7 Series FPGA Configuration Modes [14].

In the case of standard full configuration of a 7 series device, any of the external configuration modes listed above (Table 3.1) can be used to load the full configuration bitstream into the

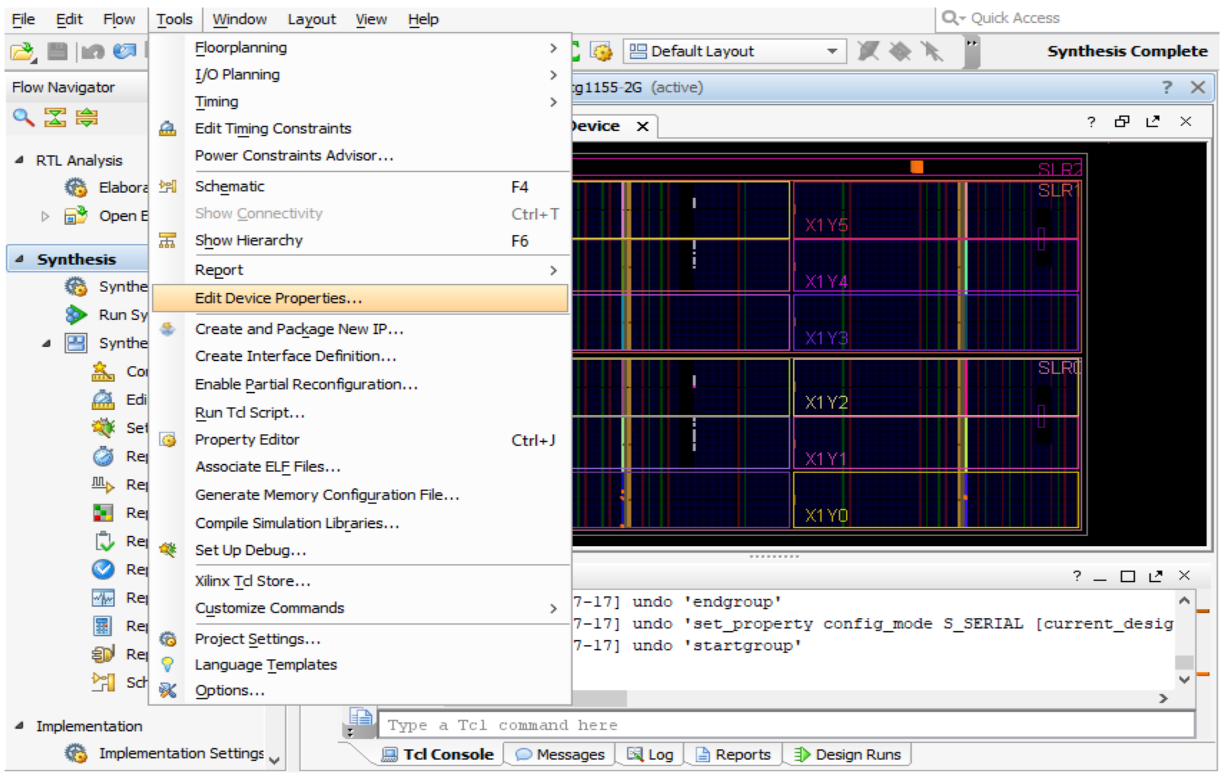
FPGA device. However, only a few of them are able to support the PR. We discuss this in more detail in the subsection 3.3.3.

Setting Configuration Options

The specific configuration mode is selected by setting the appropriate level on the dedicated mode input pins M[2:0] of the FPGA. The M2, M1 and M0 mode pins should be set at a constant DC voltage level, either through pull-up or pull-down resistors (≤ 1 kΩ), or tied directly to ground or VCCO_0 [14].

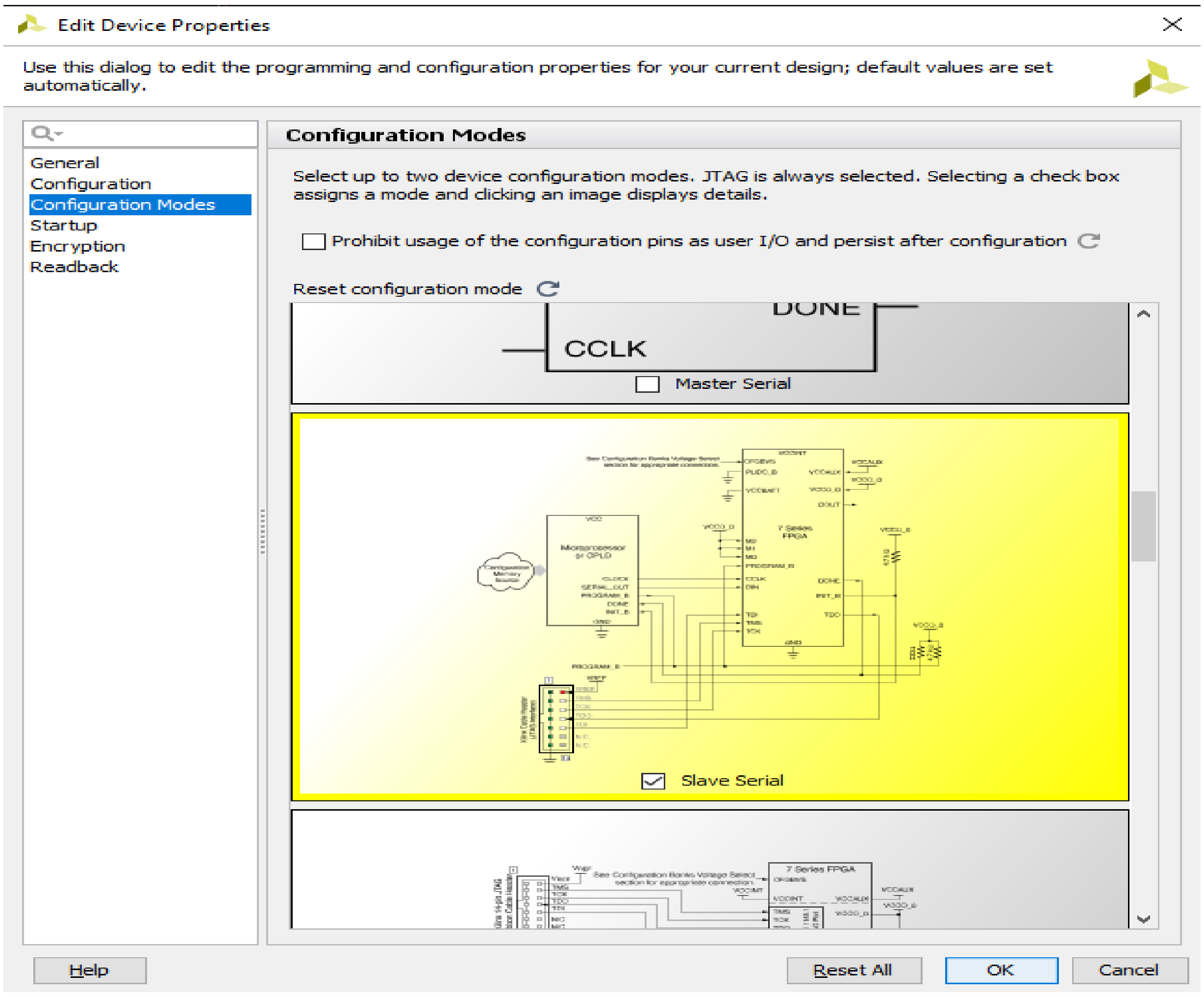
Although, a unique JTAG configuration mode setting is available (101), the JTAG interface can be used at any time regardless of the mode pin settings.

The configuration mode with which the bitstream will be loaded into the device, is an information embedded also in the bitstream and can be defined in the Vivado configuration dialog "Edit Device Properties" or through Tcl (Tool Command Language) commands, during a typical Vivado® PR design flow. More specifically, after our static design has successfully passed the *Synthesis* stage, and before we move on to the next stage of its *Implementation*, apart from creating physical constraints (Pblocks) to define the reconfigurable regions we can also create some additional constraints related to the configuration mode. Figure 3.3 below, shows how we can select the configuration mode using the Vivado-GUI (Graphical User Interface).



```
7-17] undo 'endgroup'
7-17] undo 'set_property config_mode S_SERIAL [current_desig
7-17] undo 'startgroup'
```

(a)


Edit Device Properties

Use this dialog to edit the programming and configuration properties for your current design; default values are set automatically.

- General
- Configuration
- Configuration Modes
- Startup
- Encryption
- Readback

Configuration Modes

Select up to two device configuration modes. JTAG is always selected. Selecting a check box assigns a mode and clicking an image displays details.

☐ Prohibit usage of the configuration pins as user I/O and persist after configuration

Reset configuration mode

☐ Master Serial

☑ Slave Serial

(b)

Figure 3.3: Configuration mode selection in Vivado-GUI.

3.3.2 Internal Configuration Access Port (ICAP)

Apart from the external interfaces presented in the previous subsection, Xilinx® 7 series FPGAs also feature an internal interface for the purpose of configuring the device, named Internal Configuration Access Port (ICAP). In 7 series FPGAs the ICAP interface is available through the use of ICAPE2 primitive (Figure 3.4). The ICAPE2 primitive is a design element gives us post-configuration access to the configuration functions of the FPGA from the FPGA fabric. Using this component, commands and data can be written to and read from the configuration logic.

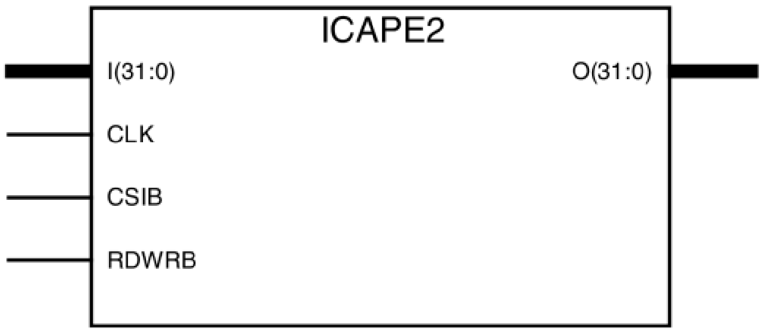


Figure 3.4: ICAPE2 primitive [15].

As shown in Figure 3.4 above, the ICAPE2 primitive has two dedicated and independent, 32-bits width read and write ports. The CSIB input port is the active low ICAP enable, and the RDWRB input port is to select the ICAP operation (read from (1) or write to (0) device's internal configuration memory).

An ICAP controller is always necessary to drive primitive's control signals (i.e. CLK, CSIB and RDWRB) during access to the configuration memory. Depending on the ICAP operation, the controller is also responsible to deliver the configuration data to the write port I [31:0] (if RDWRB=0) or to monitor the output of the read port O [31:0] (in case that RDWRB=1). Xilinx has created two pieces of intellectual property (IP) specifically for the control of the ICAPE2 primitive, the Xilinx® AXI Hardware Internal Configuration Access Port (HWICAP) LogiCORE™ IP core and the Xilinx® Partial Reconfiguration Controller IP core. Alternately, we can create our HDL description of a hardware module that implements a custom state machine for controlling the ICAPE2.

In any case, the only way to make the ICAP interface available for use, is to properly configure our device, implementing a design (into the programmable logic of the FPGA) that instantiates the ICAPE2 primitive and its controller. These two modules (ICAP primitive and the associated controller) should operate without interruption throughout a reconfiguration process, and thus the ICAP interface cannot be used for a full device reconfiguration with a full bitstream. The ICAP internal interface supports only partial reconfiguration (PR) of a 7 series device.

Xilinx's 7 series, monolithic FPGAs have two instances of the ICAP interface known as the top ICAPE2 and the bottom ICAPE2. Since top and bottom ICAP have shared resources in fabric, only one of them can be used at a particular instance of time. The tools automatically use the top ICAPE2 by default. The bottom ICAP site can only be used for a second ICAP component. However, the XC7VH580T FPGA we are targeting for, is not just a simple, monolithic 7 series device. XC7VH580T is a 2.5D IC that is designed using the Stacked Silicon Interconnect (SSI) technology. SSI devices have multiple Super Logic Regions (SLRs), each with its own pair of ICAPE2 primitives (top and bottom). One SLR is defined as the master, while the others are the slaves. Global configuration functions are controlled, by default, from the master SLR, so the ICAPE2 of the master SLR is able to read from and write to all other SLRs. A very important thing to keep in mind when dealing with SSI devices, is that, unlike 7 series monolithic devices, there are circumstances in which ICAP read / write access is restricted as identified in Table 3.1 below [14]:

Device Type	**Configuration Mode**	**ICAP Access to Configuration Memory**	
Monolithic Devices	All modes	Entire Device	
SSI Devices		**Master SLR ICAP**	**Slave SLR ICAP**
	SPIx1, JTAG	Master SLR Only	Slave SLR
	SPIx2, SPIx4		N/A
	All Other Modes	Entire Device	Slave SLR

Table 3.2: ICAP Access to configuration memory [14].

As we can see, if the configuration mode is set to JTAG or to one of the SPI modes (x1, x2 or x4) Master ICAP cannot access configuration memory in slave SLRs. Partial reconfiguration must use the ICAP within the local SLR when one of the SPIx1 or JTAG modes is selected, while ICAP in slave SLRs cannot be used at all when configuring in SPIx2 or SPIx4 modes. As JTAG mode is always available independent of the mode pins, the JTAG mode pin setting is not recommended for SSI devices [14].

Since SSI devices have multiple SLRs and multiple ICAPE2 resources, we should provide the Vivado tool with specific instructions on how the instantiated ICAP(s) will be mapped on to specific primitive(s). More specifically, after our static design has successfully passed the Synthesis stage, and before we move on to the next stage of its Implementation, apart from providing constraints for the Pblocks to define the reconfigurable regions (floorplanning), we should also provide LOC constraints for the ICAP site(s) (putting them all on a separate SLR, in case that more than one ICAPE2 primitives are used).

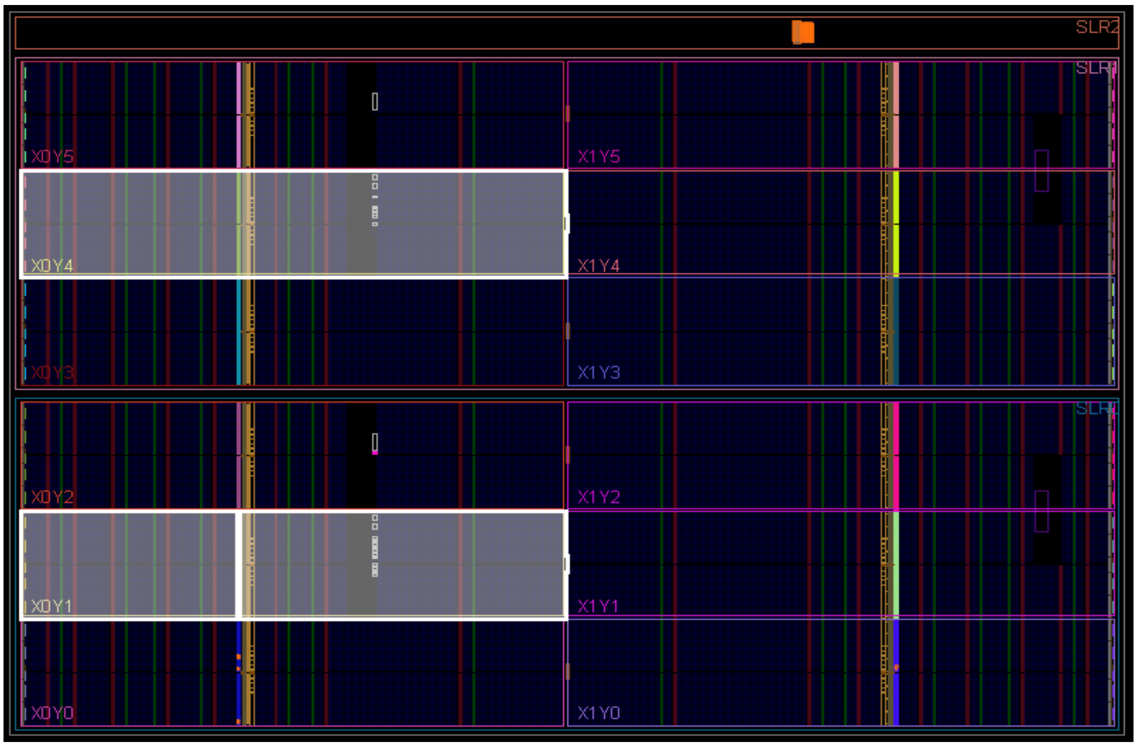


Figure 3.5: Clock regions with ICAPE2 resources in VH580T: X0Y1 (SLR0) & X0Y4 (SLR1).

The XC7VH580T comprises two FPGA die slices (SLR0 and SLR1) and an 8-channel 28Gbps transceiver die (SLR2) in the same package. Since the SLR2 is a die dedicated to transceivers, there are no ICAPE2 resources in it. Each of the two other SLRs (SLR0 and SLR1) has a pair of ICAPE2 primitives (top and bottom), located in a specific site within them (see the highlighted clock regions in Figure 3.5). So, in order, for example, to place an instantiated ICAPE2 primitive, named "master_icap", in the master SLR of our target-device, we should create a Vivado XDC constraints file as follows (lines beginning with # are just comments):

```
# ICAP location constraints
# Force the instantiated ICAPE2 to the top site in the
# SLR0(master-SLR) of the XC7VH580T

set_property LOC ICAP_X0Y1 [get_cells master_icap]
```

3.3.3 Downloading configuration files to the XC7VH580T FPGA

Table 3.3 summarizes the configuration modes supported by the XC7VH580T FPGA, indicating which of them can be used to load a full configuration bitstream into our target-device, and which ones are available in a case of partial reconfiguration of this device.

	Configuration Mode	M[2:0]	Full Configuration	Partial Reconfiguration
External Interfaces	Master Serial	000	Yes	No
	Master SPI	001		No
	Master BPI	010		No
	Master SelectMAP	100		No
	JTAG only	101		Yes
	Slave SelectMAP	110		Yes
	Slave Serial	111		Yes
Internal Interfaces	ICAP	N/A	No	Yes

Table 3.3: XC7VH580T configuration modes.

Full Configuration Bitstreams

All PR designs start with a standard configuration of the full device using a full configuration bitstream [13]. In order to download a full configuration file to the XC7VH580T device, we can use any of its external configuration modes. The only one internal configuration mode of our target-device (i.e. ICAP) is not meant to be used for a full-device configuration.

Partial Bitstreams

As shown in Table 3.3 above, partial reconfiguration of our target-device is supported using the following configuration modes [13]:

- **JTAG:** A good interface for quick and easy testing or debug. Can be driven with the Vivado Logic Analyzer. JTAG interface can be used at any time regardless of the mode pin settings (M[2:0]) for loading both full and partial bitstreams to the XC7VH580T device.
- **Slave SelectMAP or Slave Serial:** A good choice to perform full configuration and Partial Reconfiguration over the same interface. Let’s assume that the configuration pins have been set to one of these two external configuration modes and that we have already delivered a full bitstream through this mode. To keep using the selected external configuration mode for loading a partial bitstream, the configuration pins must be reserved for use after the initial configuration. This is achieved by setting the BITSTREAM.CONFIG.PERSIST property before the Bitstream Generation stage of a standard PR design flow. The Tcl command syntax to set this property is:

```
set_property BITSTREAM.CONFIG.PERSIST Yes [current_design]
```

We should keep in mind, however, that when configuration pins are persisted, the ICAP is disabled; the two features are mutually exclusive [13].

- **ICAP:** A good choice for user configuration solutions. ICAP interface is an internal one. Its use is based on the device's internal programmable resources and primitives, and therefore it's an interface always available for use after a first, proper configuration of the full device. Requires the creation of an ICAP controller as well as logic to drive the ICAP interface. Furthermore, in order to use the ICAP interface after the initial configuration (regardless of the external configuration mode used) we must ensure that the BITSTREAM.CONFIG.PERSIST property is not set. The corresponding Tcl command is:

```
set_property BITSTREAM.CONFIG.PERSIST No [current_design]
```

When it comes to partial reconfiguration, master modes are not directly supported. In fact, SPI and BPI flash memories can be used to store partial bitstreams, but we cannot use the corresponding interfaces to deliver a stored bitstream. The static design would need to be connected to the flash via user IO, and a controller would be used to fetch bitstreams for delivery to the ICAP [13].

Each of the above configuration modes has a maximum theoretical bandwidth depending on the maximum data width of the corresponding datapath and the maximum clock frequency up to which the corresponding configuration interface can, theoretically, be operational. Table 3.4 below, summarizes the maximum bandwidths for these configuration modes in the XC7VH580T-G2HCG1155E which is a -2G speed grade, SLR-based Virtex-7 HT device.

Configuration Mode	Max Clock Rate	Max Data Width	Maximum Bandwidth
ICAP (1)	100 MHz	32 bits	3.2 Gbits/s
SelectMAP (Slave)	100 MHz	32 bits	3.2 Gbits/s
Serial Mode (Slave)	100 MHz	1 bit	100 Mbits/s
JTAG	20 MHz	1 bit	20 Mbits/s

Table 3.4: Maximum Bandwidths for configuration modes in XC7VH580T-G2HCG1155E.

(1) As specified in the Xilinx® documentation [16], the ICAP of an SLR can only be clocked at the maximum frequency of 100 MHz if it is used to access the configuration memory in this particular SLR. A Slave SLR ICAP can only access configuration memory within the local SLR. However, a Master SLR ICAP gives us access to the entire device. If the Master ICAP is used to access the configuration memory in Slave SLRs, it should not be clocked beyond the 70 MHz, and thus, the corresponding maximum bandwidth is decreased to 2.24 Gbits/s.

3.4 Evaluation of various Reconfigurable Architectures

The speed of partial reconfiguration is directly related to the size of the partial bitstream. However, for a partial bitstream of a specific size, the reconfiguration time depends directly on the reconfigurable architecture (RA) used. Depending on the storage medium of the partial configuration file, the configuration interface used, and the way in which we read this file from the storage medium and deliver it to the selected configuration interface, we can build several reconfigurable architectures, each with a different configuration throughput.

In the context of this thesis we dealt with five different reconfigurable architectures. The configuration throughput of each of them is presented in the Table 3.5 below. The first one named as *EXT_JTAG* uses the JTAG interface, while the other four RAs are based on the ICAP. In any case, exactly the same physical constraints were imposed, during the phase of floorplanning the reconfigurable partitions (RPs). Consequently, in all five cases the reconfigurable regions (RRs) were identical in terms of size, location and resources, and thus the amount of data needed to be sent to the FPGA's configuration memory each time was the same. The minor size difference (namely 128 bytes) observed in the following table between the partial bitstream used in the *EXT_JTAG* case and those used in the other four RAs, is due to the fact that the (equivalent) partial configuration files had different formats.
More specifically, in the first case where the JTAG interface was used, we had the possibility to download partial bitstreams to the XC7VH580T device as *.bit* configuration data files. Bitstreams (either full or partial) are generated by default as .bit files from the Vivado design tool, when the *write_bitstream* command is executed. BIT files are binary configuration data files containing a 128-byte header to provide general information about the bitstream name, the FPGA device, the user ID, etc. As these first 128 bytes of a BIT file are not configuration data, they are not meant to be downloaded to the FPGA. So, this file type is mainly used to program devices form the Vivado device programmer with a programming cable. For custom configuration solutions, the header information is useless; we just need a bitstream which will contain pure configuration data. Such a file format, is provided by a *.bin* configuration file, that is generated from Xilinx's Vivado design tool by running the *write_bitstream* command with the *-bin_file* argument. BIN files, that were used in any other case except the EXT_JTAG, are binary configuration data files containing only the device programming data, without the header information found in a standard binary BIT file [17].
Anyway, an arbitrary size of bitstream was selected for our experimentation. It is noted that we expect similar results/differences for smaller or bigger (in size) bitstreams.

Name of RA	Partial Bitstream Format	Partial Bitstream Size	Reconfiguration Time	Throughput
EXT_JTAG	.bit	681,660 Bytes	926.02 msec	5.889071 Mbits/s
EXT_HWICAP	.bin	681,532 Bytes	709 sec	7.690064 Kbits/s
INT_ HWICAP			37.599270 msec	145.009623 Mbits/s
INT_ HWICAP_ASYNC			19.669035 msec	277.199974 Mbits/s
INT_CUSTOM_ICAP			1.703860 msec	3.199943 Gbits/s

Table 3.5: Configuration throughputs of the various Reconfigurable Architectures.

The JTAG-based architecture has been included in Table 3.5 only for comparison purposes, because it is a commonly followed approach for reconfiguration and the easiest one to deploy as well. The other four ones are considered to be our main architectures implemented in the effort to improve the reconfiguration speed. A really important feature, common to all four main architectures, is the implementation of a microprocessor entirely within the device general-purpose memory and logic fabric. The inclusion of a microprocessor in the hardware design facilitates the management of our reconfigurable system from the outside world (e.g. easy and flexible communication with a host-computer) enabling us to remotely control the PR of our device. The result is a hybrid hardware-software system where the functionality that will be optimally implemented in the hardware can be easily defined by the software (the "hardware-enabled, software-defined" concept).

In order to ensure that the global reset events are properly synchronized across all elements in the Reconfigurable Module (RM), and that all Super Long Lines (SLLs) are contained within the static portion of the design (since they are not partially reconfigurable), a Reconfigurable Region (RR) must be fully contained within a single SLR [13]. In the light of the above, the Pblock that was drawn once and used in all cases (remember that the same constraints were always imposed) to define the RR during the floorplanning stage, was entirely placed in the SLR1. In any of the ICAP-based RAs, the instantiated ICAP was also placed in the SLR1, so that it can be clocked at the maximum possible frequency of 100 MHz.

Towards accomplishing the evaluation of the various RAs, two key-components of Xilinx's Vivado® Design Suite 2016.3 were used; the Vivado 2016.3 tool for implementing any hardware design to be loaded on the FPGA fabric, and the Xilinx Software Development Kit (SDK) 2016.3 for developing applications to be executed by an embedded soft processor core, in the cases where the implemented HW design comprised such a module.

The following five subsections provide details about each RA listed in Table 3.5.

3.4.1 *The “EXT_JTAG” architecture*

In this case, partial bitstreams are stored in the memory of an external, smart device; on a common Personal Computer (PC) to be precise. Using a standard A to micro-B USB cable, we are able to deliver partial bitstreams to the JTAG interface through Vivado’s Hardware Manager.

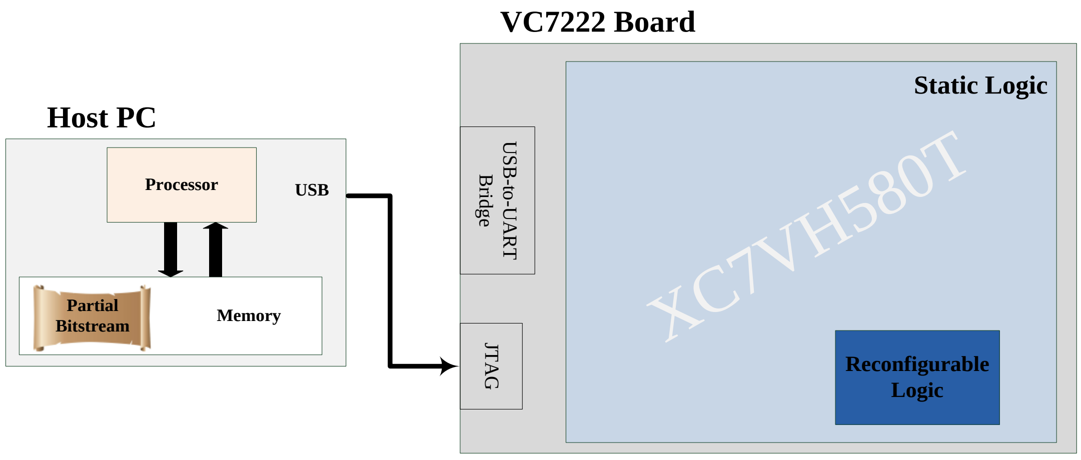


Figure 3.6: Simplified representation of “EXT_JTAG” architecture.

Given the data provided in Table 3.4, and the fact that our main objective is to implement the PR of our device in the shortest possible timeframe, we should have never been interested in this architecture. This is why we never tried to push this architecture to its limits in terms of reconfiguration throughput. However, the maximum bandwidth of the configuration interface used in an RA, is not the only factor that can limit the RA’s overall configuration throughput, and thus, using a higher bandwidth configuration interface does not necessarily mean an RA with better configuration throughput (this will become even more evident when looking at the “EXT_HWICAP” architecture). So, taking into account the fact that this RA offers the most simple and standard way to download bitstreams (full or partial) to any modern FPGA device, it seems quite sensible to know exactly how fast the PR can happen in this case, before making any further effort.

Nevertheless, one thing we must definitely do to achieve the maximum possible reconfiguration throughput, is to base our RA on either the SelectMAP or the ICAP, since only through one of these configuration ports the data can be written to the FPGA’s configuration memory at the highest possible rate of 3.2 Gigabits per second (Table 3.4). Being a good choice for custom user configuration solutions, the ICAP interface was used to access the device’s internal configuration memory in all our attempts that followed.

3.4.2 *The “EXT_ HWICAP” architecture*

Like the “EXT_JTAG”, the “EXT_HWICAP” also relies on an external PC device to store the partial bitstreams. However, in this architecture partial bitstreams are delivered to the ICAP interface of the FPGA device, through the Xilinx’s AXI HWICAP IP core. Using a standard A to mini-B USB cable, the host computer is able to communicate with the FPGA through the USB-to-UART bridge of the VC7222 board. For the purposes of this architecture, a Xilinx’s soft IP core named AXI UART Lite is implemented in the FPGA to provide the controller interface for asynchronous serial data transfer. Both AXI HWICAP and AXI UART Lite are connected to an embedded soft-core processor of Xilinx named MicroBlaze, which is also implemented in the FPGA fabric to control the operation of these two IP cores. More specifically, a software application running on MicroBlaze is responsible for reading bitstream configuration data from the serial input through the AXI UART Lite core and delivering them to the ICAP interface through the AXI HWICAP core. A user of the host computer can send the configuration data to the serial input using a terminal emulation application, such as HyperTerminal, TeraTerm, etc. The software application executed on MicroBlaze, also prints proper messages to this terminal, assisting and guiding the user to go through the PR process.

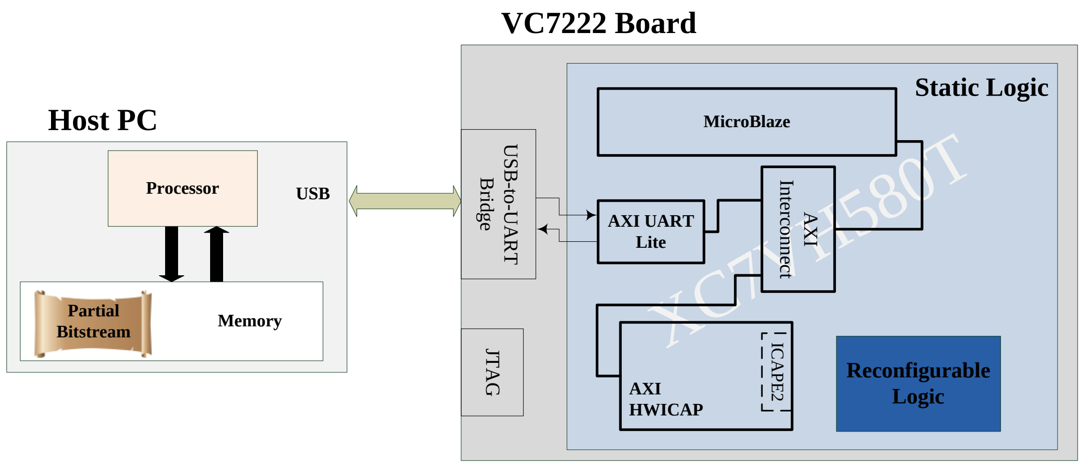


Figure 3.7: Simplified representation of “EXT_HWICAP” architecture.

The entire design was clocked at 100 MHz (the maximum operating frequency recommended by Xilinx for the ICAP). All the IP cores used were customized in the Vivado IP Integrator. The baud rate of the AXI UART Lite core was set to 9600 bits per second and the number of data bits in the serial frame was eight. The read and write FIFOs of the AXI HWICAP core was of Block RAM type, the depth of each of them was set to the maximum allowed value, and the data width of the ICAPE2 primitive, which is automatically instantiated through use of the AXI HWICAP, was set to 32 bits.

Comparing this RA with the previously presented one (i.e. “EXT_JTAG”), we can notice that the time required for PR completion was significantly increased, despite the fact that a higher bandwidth configuration interface was used. This is due to the serial port which is a very slow interface. Furthermore, the reconfiguration time is getting worse due to the MicroBlaze which acts as a proxy between the host computer and the FPGA fabric. In order to benefit from the ICAP’s high throughput capability, we must find a way to store the bitstream configuration data closer to the MicroBlaze embedded soft-core processor. Since our target-board (i.e. the VC7222) does not feature any (external to the FPGA) type of SRAM or DRAM component memory, the embedded memory resources in our target-device (i.e. the XC7VH580T) are the only ones that can be used to store the partial bitstreams closer to the MicroBlaze. But even if there were more than one option, using the FPGA’s embedded memory resources will always be the best one in terms of speed and performance. The primary drawback of this approach, is that we are only allowed to generate very limited capacity memories; is not only that the amount of these memory resources is very small, but also that they should primarily be used for the needs of the design implemented each time in the FPGA device.

3.4.3 *The “INT_ HWICAP” architecture*

Like the “EXT_HWICAP”, the “INT_HWICAP” also uses the AXI HWICAP IP core to access the FPGA’s internal configuration memory. However, in this case partial bitstreams are pre-loaded into the FPGA’s BRAM memory. This time, the software application running on MicroBlaze retrieves configuration data stored into the device’s Block RAM and delivers them to the ICAP interface through the AXI HWICAP core.

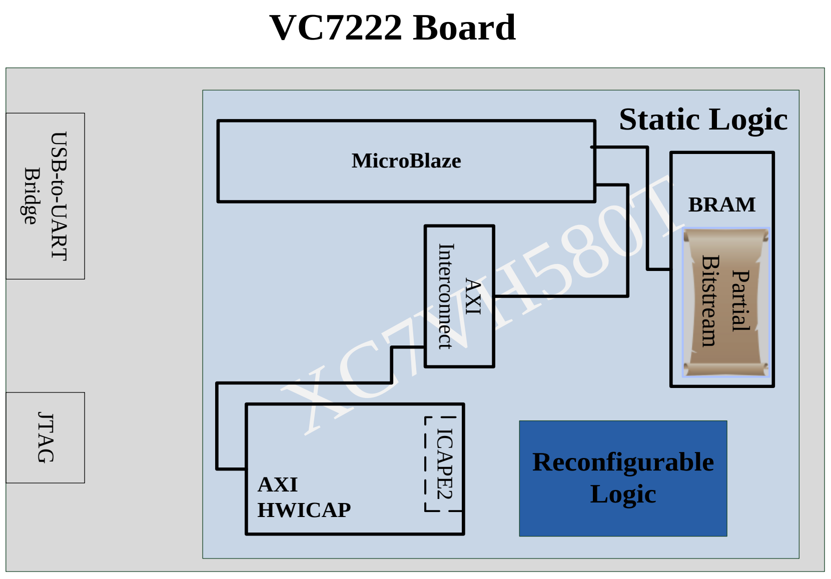


Figure 3.8: Simplified representation of “INT_HWICAP” architecture.

Again, the entire design was clocked at 100 MHz, the read and write FIFOs of the AXI HWICAP core was of Block RAM type, the depth of each of them was set to the maximum allowed value, and the data width of the ICAPE2 primitive was 32 bits. The AXI UART Lite IP core was also used in this case, but is not illustrated in Figure 3.8, as it is not actually a vital module for this RA; it was added just to allow the user of an external computer device to communicate and interact with the MicroBlaze, through the USB-to-UART bridge, and thus manage the PR via the software running on the embedded microprocessor.
Towards storing partial bitstreams into the device's Block RAM, the Xilinx® LogiCORE^{TM} IP Block Memory Generator (BMG) core was used. The BMG IP core is an advanced memory constructor that generates area and performance-optimized memories using embedded Block RAM resources in Xilinx FPGAs [18]. This core was customized in the Vivado IP Integrator in such a way that it can operate like a Stand-Alone, Single Port ROM. The Width and Depth values that were used for read operation in the memory port was set to 32 bits and 170,383 respectively, resulting in a memory with a storage capacity of 170,383*32 bits or 681,532 Bytes (i.e. the size of a partial bitstream). The generated ROM was initialized with the configuration data of a partial bitstream, and the MicroBlaze was able to access them through a fast, local bus that was connecting the MicroBlaze processor data port (i.e. the DLMB interface) to the high-speed, BRAM-based ROM. The LogiCORE™ IP Local Memory Bus (LMB) core, whose frequency and latency are optimized for use with MicroBlaze [19], was instantiated in our design to implement this fast, local bus. To this end, the Local Memory Bus (LMB) Block RAM (BRAM) Interface controller IP core was also added to the design to provide the interface between the BMG core and the Data (D) side LMB core.

The configuration throughput has increased significantly in this case, but judging from the ICAP's maximum theoretical bandwidth, there is still room for improvement. However, in this RA the operating frequency (f) and the data width of the ICAPE2 primitive were 100 MHz and 32 bits respectively, and, additionally, the high-speed, BRAM-based ROM could potentially feed the ICAPE2 with 32 bits of configuration data per clock cycle (cc = 1/f = 10ns). Therefore, connecting the on-chip block RAM to the ICAPE2 primitive through the MicroBlaze and the associated processor buses, is the main bottleneck to the configuration throughput of this RA. The following two architectures will verify the veracity of the above claim.

3.4.4 The "INT_ HWICAP_ASYNC" architecture

The "INT_HWICAP_ASYNC" architecture is just like the "INT_HWICAP" described above. The only difference in this case, is that the whole design, except for the ICAPE2 primitive, was clocked at 200 MHz. In order to keep the ICAP clock at 100 MHz, we had to set the "Enable Async" property of the AXI HWICAP IP core. By doubling the frequency at which the MicroBlaze, the BMG core, the AXI HWICAP core, as well as the associated data buses and interconnections were operated, the reconfiguration time was decreased by almost half. This result strongly confirms the fact that using all these modules (MicroBlaze, BMG, HWICAP, LMB, AXI Interconnects, etc.) to deliver the configuration data from the BRAM to the ICAP by a software application, is the only factor that keeps the configuration throughput of this architecture at a lower level than the ICAP's theoretical maximum bandwidth. This led to the implementation of our last RA described in the following subsection.

3.4.5 *The "INT_CUTOM_ICAP" architecture*

The last architecture named "INT_CUSTOM_ICAP" is similar to the "INT_HWICAP" one. The partial bitstreams are stored in the FPGA's BRAM memory and the device's internal configuration memory is accessed through the ICAP interface. The key difference between these two architectures is that in the case of "INT_CUSTOM_ICAP" architecture, the whole process of retrieving the partial bitstream stored in Block RAM and delivering it to the ICAP interface, is not carried out by a software application, but from a hardware module developed from scratch in RTL to implement a custom state machine in the FPGA fabric. ICAPE2 primitive is automatically instantiated by this custom ICAP controller, which is able to connect directly (without any processor bus) the output port of the BRAM memory to the input port of the ICAP primitive.

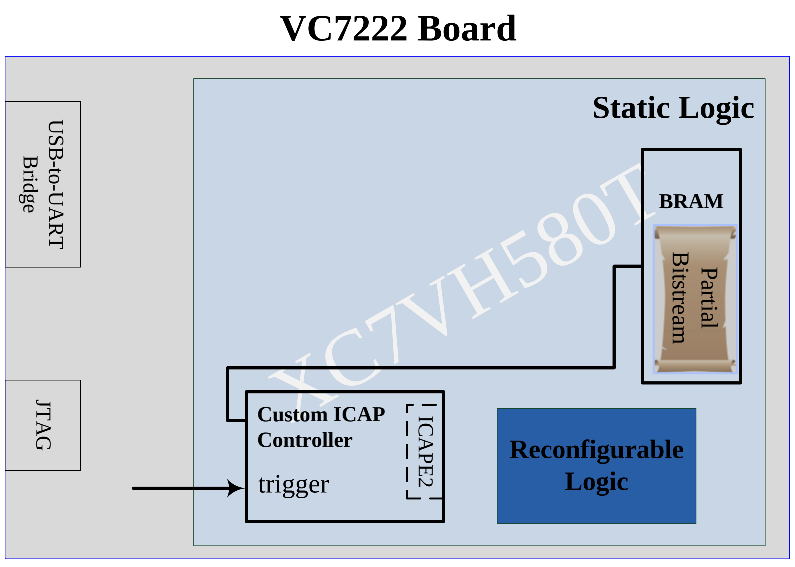


Figure 3.9: Simplified representation of "INT_CUSTOM_ICAP" architecture.

In this case the AXI HWICAP and the MicroBlaze IP cores become useless. However, the MicroBlaze microprocessor is also used by this architecture along with the AXI UART Lite core, just to enable the management of PR via software. More specifically, a software application running on MicroBlaze waits until it receives the appropriate input from the user of an external computer through the USB-to-UART bridge. MicroBlaze then triggers the custom ICAP controller, which in turn writes the configuration data stored in BRAM to device's internal configuration memory through the ICAP primitive. Since, the MicroBlaze and the AXI UART Lite are not actually vital modules of this RA, they are not illustrated in Figure 3.9. The whole design was clocked at 100 MHz. As can be seen from Table 3.5, this RA allows us to fully exploit the ICAP's high throughput capabilities, and therefore this is the architecture on which we will base the implementation of our final hardware, partially reconfigurable, design. The following chapter provides more details on how the Custom ICAP Controller, and more generally this entire RA, are implemented.

4 System Integration

As already mentioned, the ultimate goal of this thesis is to develop a system which will make use of the high-bandwidth GTZ transceivers available on the XC7VH580T, and will be capable of being partially reconfigured within a very short timeframe. The 7 series FPGAs Transceivers Wizard LogiCORE™ IP (Wizard) was used to configure and simplify the use of the eight serial GTZ transceivers in the Virtex®-7 XC7VH580T device. Based on the example design that can be generated by this Wizard, we built the reconfigurable system depicted in Figure 4.2. This system was implemented through the use of Xilinx's Vivado 2016.3 Integrated Design Environment (IDE) tool, part of the Vivado Design Suite 2016.3, by adopting a hierarchical, functional, unit-based design approach in which the whole design was described as a combination of smaller, cooperating hardware modules. Our (hierarchical) design has one top-level module that is the root of the design hierarchy. The top-level module comprises eleven instances from four different basic building modules (Figure 4.1). Two of them, namely the *sys_clk* and the *gtz_raw_data_gt_frame_gen* Verilog modules, are just single elements, while the two others are more complex Verilog modules within which several lower-level (Verilog) modules are instantiated. In other words, each of the *gtz_raw_data_init* and the *pr_*controller is a collection of lower-level module instances.

```
   gtz_raw_data_exdes.v (top module)
|  ______ gtz_raw_data_init.v
|  ______ gtz_raw_data_gt_frame_gen.v (1 per transceiver)
|  ______ sys_clk.v
|  ______ pr_controller.v
```

Figure 4.1: Design hierarchy.

The ***sys_clk*** module generates clock signals for the needs of our design. The ***gtz_raw_data_init*** instantiates eight properly configured GTZ transceivers and offers an easy way to interface the user logic with them. This module is connected to eight frame generators, which are eight different instances of the ***gtz_raw_data_gt_frame_gen*** Verilog module. Each frame generator is dedicated to a particular transceiver channel to provide transmission data to it. The *FRAME_GEN*_0 module, which corresponds to the first transceiver (GT_0) of the GTZ transceiver octal, is defined as a reconfigurable one. The logic within this generator will be able to be modified without disturbing the smooth operation of the rest of the design, through the Partial Reconfiguration (PR) of our device. Finally, based on the conclusions drawn in the Section 3.4, some additional logic was implemented as part of the static logic for the PR needs. This additional logic is described by a suitable block-design (BD) created in the Vivado IP Integrator and instantiated into the top-level module through the ***pr_controller***.

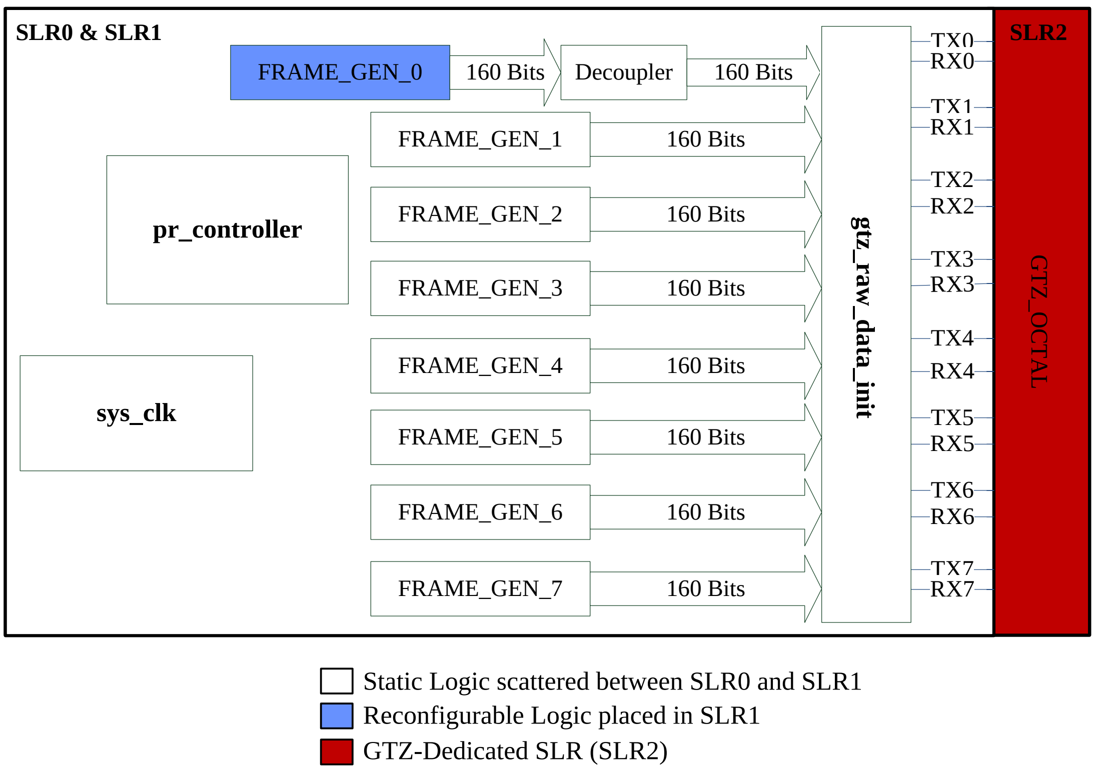


Figure 4.2: Simplified view of the HW design implemented.

The purpose of this chapter is to describe in detail the process followed for the system integration in the context of this thesis. To this end, Section 4.1 first provides some basic information on the 7 Series Transceiver Channel Architecture. After that, Section 4.2 introduces the reader to the 7 series FPGAs Transceivers Wizard LogiCORE™ IP and the example design that can be generated for any instance of this Wizard IP core. This section also deals with the *gtz_raw_data_exdes*, *sys_clk*, *gtz_raw_data_gt_frame_gen* and *gtz_raw_data_init* modules, since they are directly related to the generated example design. Section 4.3 presents the *pr_controller* and the associated HW/SW modules. In Section 4.4 we describe the methodology followed to generate the final (full and partial) bitstreams, and, finally, in Section 4.5 we discuss the VC7222 board experimental setup.

4.1 7 Series Transceiver Architecture

In Chapter 2, the Section 2.2 focused on introducing the broader concept of serial Multi-Gigabit Transceivers (MGTs), a specific category of which are the Xilinx's GTZ transceivers. Following this introduction, and since the use of the GTZ transceivers available on the Virtex-7 XC7VH580T-G2HCG1155E FPGA is one of the main objectives of this thesis, a more detailed presentation on the special features and topology of a GTZ channel primitive would enhance the completeness of the information contained in this document. However, the information that is specific to the GTZ transceivers, and is provided in the *7 Series FPGAs GTZ Transceivers* user guide (UG478), is confidential. In this context, the general architecture of a 7 series transceiver channel is presented below, just to familiarize the reader with some common definitions and terminology used throughout this chapter.

7 series transceivers are based on the following architecture [2]:

- Transmitter (TX) / Receiver (RX) Package
- Physical Medium Attachment Sublayer (PMA). The PMA includes a serial / parallel interface (PISO, SIPO), phase-locked loop (PLL), clock data recovery (CDR), pre-emphasis, and equalization blocks.
- Physical Coding Sublayer (PCS). The PCS contains logic-to-process parallel data and includes FIFO, coding/decoding, and gearbox functionality.
- FPGA logic interface (fabric)

Figure 4.3 illustrates an abstract diagram of a 7 series transceiver channel [2].

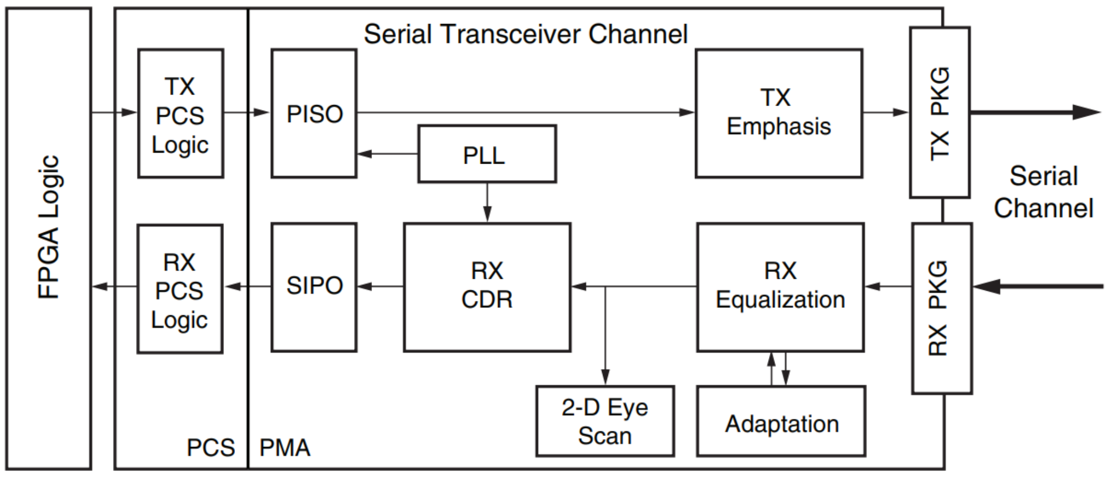


Figure 4.3: 7 Series Transceiver Channel Architecture [2].

Both PCS and PMA are contained within the physical layer of the Open Systems Interconnection (OSI) reference model. The PMA sublayer acts as a SerDes block, the major features of which (i.e. TX pre-emphasis and post-emphasis, TX differential swing, RX equalization, PLL divider settings) are controllable. This layer provides a medium-independent means for the PCS to support various serial physical media [20]. The PCS acts as an interface between the logic implemented in the FPGA (fabric) and the PMA, offering also several configurable features such as data encoding/decoding schemes, scrambling/descrambling, data width conversion, FIFOs, etc. The implementation of these blocks for each type of 7 series transceivers is different based on the connectivity and performance requirements.

4.2 Transceivers Wizard and the associated Example Design

The 7 series FPGAs Transceivers Wizard LogiCORE™ IP (Wizard) automates the task of creating HDL wrappers to configure Xilinx® 7 series FPGA on-chip transceivers. The menu-driven interface allows for configuring one or more high speed serial transceivers using either pre-defined templates supporting popular industry standards, or from scratch to support a wide variety of custom protocols. In addition, the Wizard can produce an example design, test bench, and scripts for simple simulation and hardware usage demonstration. The files generated by the Wizard can be summarized as follows [21]:

- **Transceiver wrapper**, which includes:
  - Specific serial transceiver configuration parameters set using the Wizard.
  - Transceiver primitive selected using the Wizard.
- **Example design** demonstrating the modules required to simulate the wrapper. These include:
  - **FRAME_GEN** module which generates a user-definable data stream for simulation analysis.
  - **FRAME_CHECK** module: tests for correct transmission of data stream for simulation analysis.
- **Test bench**: top-level testbench demonstrating how to stimulate the design.

This Wizard supports and covers all the types of Gigabit Transceivers (GTs) in Xilinx's 7 Series FPGAs (e.g. GTH, GTZ, etc.). The structure and the content of the transceiver wrapper, example design, and testbench files can vary depending on the GT type we are targeting for and the specific configuration of the target-GTs; however, the relationship of these files can generally be considered to be, as illustrated in Figure 4.4 [21].

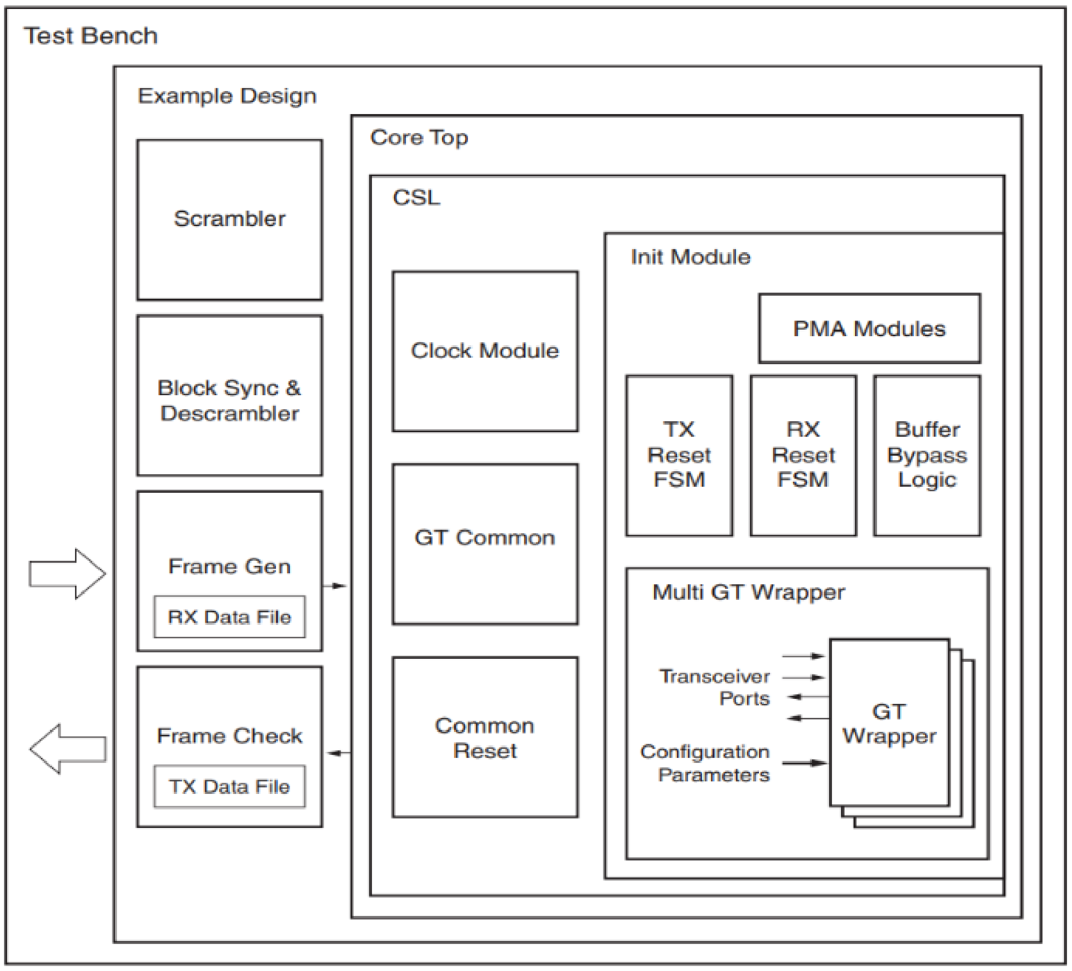


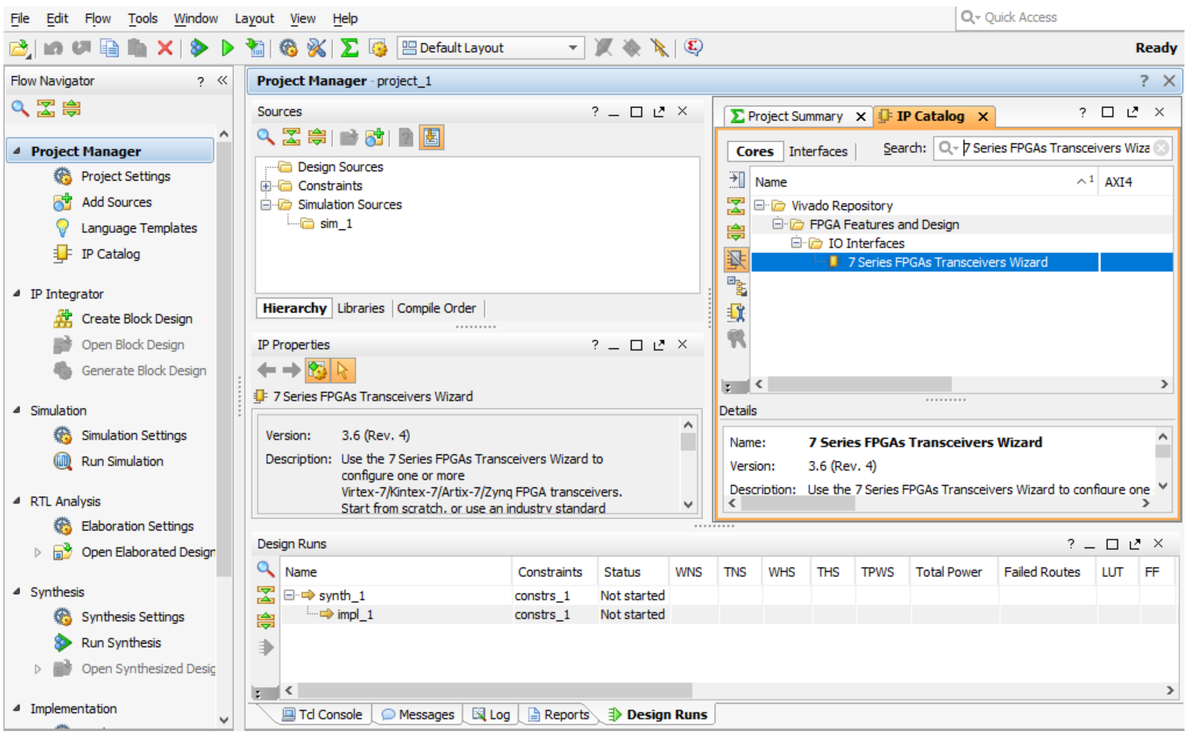
Figure 4.4: Structure of the Transceiver Wrapper, Example Design, and Test Bench [21].

4.2.1 Customizing and Generating the Core

The 7 series FPGAs Transceivers Wizard is not present in the Vivado IP Integrator (IPI). To customize and generate a transceiver core (and its associated example design) through the Vivado IDE we should use the Vivado IP catalog. The exact wizard wrapper steps followed for the purposes of this thesis are listed below:

1. Open Vivado Design Suite. Select "*Create New Project*" and click "*Next*".
2. Select the project name/path and click "*Next*".
3. Select "*RTL Project*" to permit running the example design and click "*Do not specify sources at this time*". Click "*Next*".
4. Select the "*Parts*" option and click the "*xc7vh580thcg1155-2G*". Click "*Next*", and then click "*Finish*".

5. Under the **Project Manager** in the Flow Navigator panel, select "*IP Catalog*" and search for "*7 Series FPGAs Transceivers Wizard*" (see Figure 4.5).



7 Series FPGAs Transceivers Wizard

Version: 3.6 (Rev. 4)
Description: Use the 7 Series FPGAs Transceivers Wizard to configure one or more Virtex-7/Kintex-7/Artix-7/Zynq FPGA transceivers. Start from scratch, or use an industry standard

Name: **7 Series FPGAs Transceivers Wizard**
Version: 3.6 (Rev. 4)
Description: Use the 7 Series FPGAs Transceivers Wizard to configure one

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
synth_1	constrs_1	Not started									
impl_1	constrs_1	Not started									

Figure 4.5: IP Catalog Window.

6. Double-click "*7 Series FPGAs Transceivers Wizard*" to bring up the 7 Series FPGA Transceiver Customize IP dialog box.

7. Page 1 of the Wizard (Figure 4.6) allows us to determine the component name, the GT type, the line rate, the reference clock frequency. Additionally, this page specifies a protocol template.

   a) In the "*Component Name*" field, the string "*gtz_raw_data*" was entered as the name for the Wizard instance.

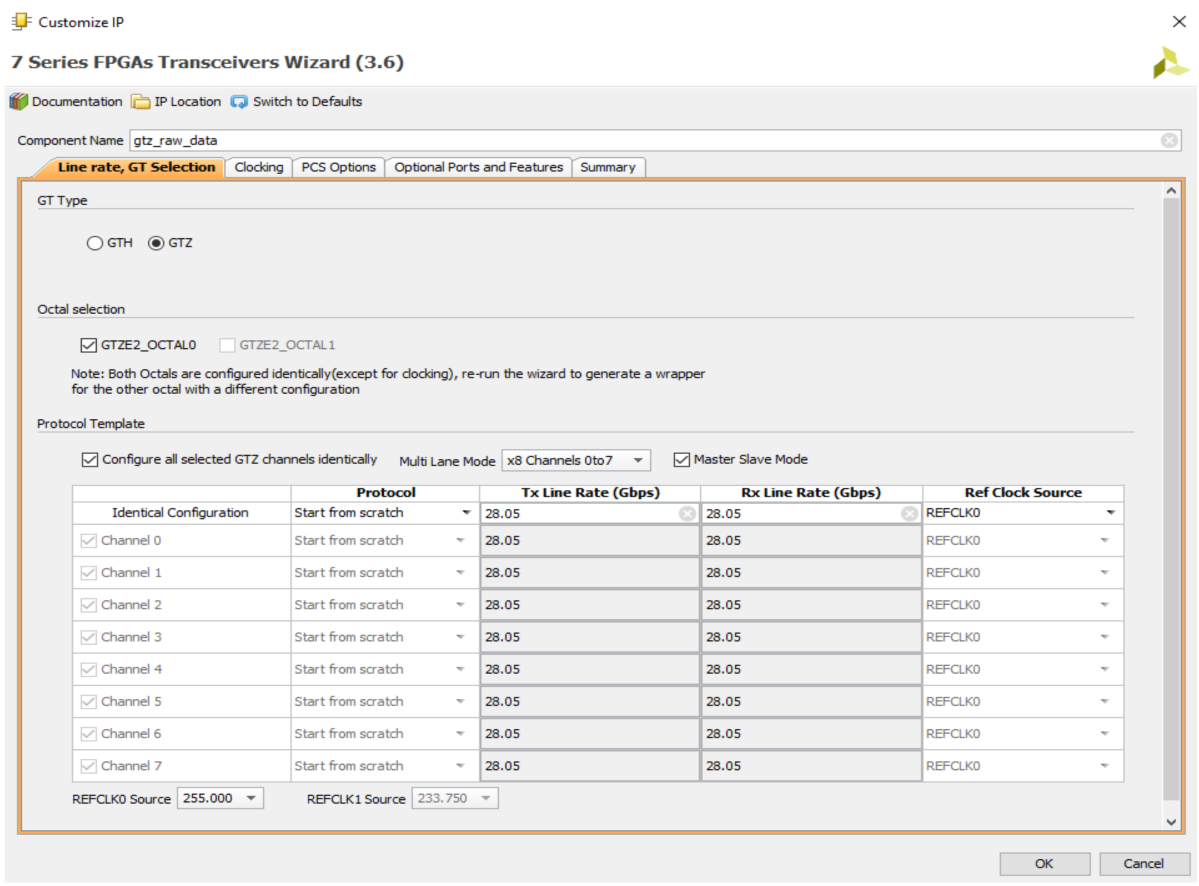
   b) In the "*GT Type*" field the "*GTZ*" was selected (the number of options available in this field depends on the device chosen in Project Options; as already mentioned the xc7vh580thcg1155-2G features 6xGTH-Quad and 1xGTZ-Octal).

   c) In the "*Octal selection*" field the "*GTZE2_OCTAL0*" was selected (the number of available octals depends on the target device and package; in our case there is one GTZ octal, so there is no other option).

   d) In order to be able to manually set all parameters (e.g. Tx /Rx Line Rate, Clocking, PCS options, etc.) the "*Start from scratch*" protocol was selected.

   e) In the "*Multi Lane Mode*" field, the "*x8 Channels 0to7*" was selected. The "*Master Slave Mode*" was used to enable one of the channels as the master and the other as slaves. In addition, the "*Configure all selected GTZ channels identically*" checkbox was selected to configure all the eight channels selected identically.

f) The line rate for both TX and RX was set to the maximum possible value of 28.05 Gb/s ("*Rx Line Rate*" must be the same as "*Tx Line Rate*"). Finally, the "*REFCLK0*" was selected as the Reference Clock Source and it was set to 255MHz.


Customize IP

7 Series FPGAs Transceivers Wizard (3.6)

Documentation | IP Location | Switch to Defaults

Component Name: gtz_raw_data

Line rate, GT Selection | Clocking | PCS Options | Optional Ports and Features | Summary

GT Type: ○ GTH ● GTZ

Octal selection: ☑ GTZE2_OCTAL0 ☐ GTZE2_OCTAL1

Note: Both Octals are configured identically(except for clocking), re-run the wizard to generate a wrapper for the other octal with a different configuration

Protocol Template

☑ Configure all selected GTZ channels identically | Multi Lane Mode: x8 Channels 0to7 | ☑ Master Slave Mode

	Protocol	Tx Line Rate (Gbps)	Rx Line Rate (Gbps)	Ref Clock Source
Identical Configuration	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 0	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 1	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 2	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 3	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 4	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 5	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 6	Start from scratch	28.05	28.05	REFCLK0
☑ Channel 7	Start from scratch	28.05	28.05	REFCLK0

REFCLK0 Source: 255.000 | REFCLK1 Source: 233.750

OK | Cancel

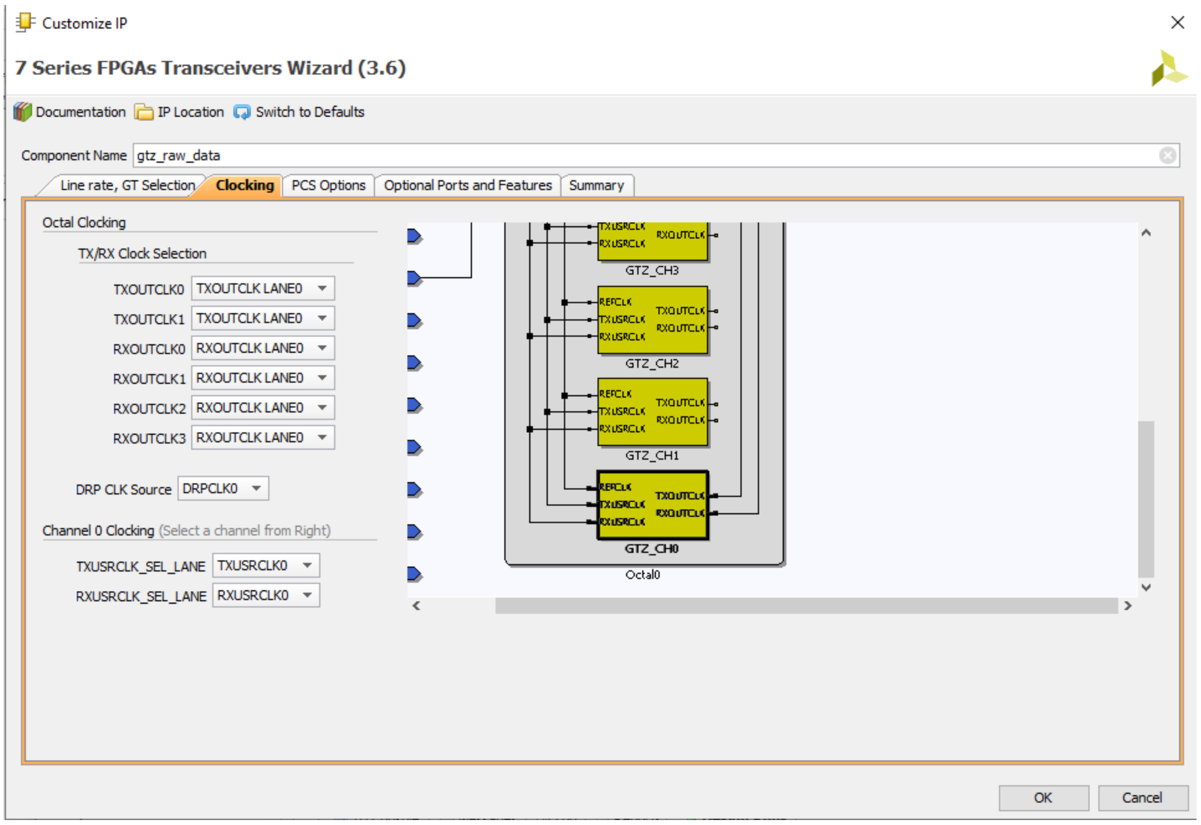
Figure 4.6: Line Rates and Transceiver Selection – GTZ Page 1.

8. The second page of the Wizard (Figure 4.7) allows us to select the clocking for the octal and the enabled channels within the octal (remember that all the eight channels of the GTZ octal were enabled on page 1 of the Wizard).

a) The "*TXOUTCLK LANE0*" and "*RXOUTCLK LANE0*" were selected as the sources for the "*TXOUTCLK<0-1>*" and "*RXOUTCLK<0-3>*" of the octal respectively.

b) The DRP CLK was selected to be sourced from the "*DRPCLK0*".

c) Finally, the "*TXUSRCLK0*" and "*RXUSRCLK0*" were selected for the "*TXUSRCLK_SEL_LANE*" and "*RXUSRCLK_SEL_LANE*" respectively, for each channel. The channel we wished to configure each time, was selected from the image on the right (GTZ_CH<0-7>).


Customize IP

7 Series FPGAs Transceivers Wizard (3.6)

Documentation | IP Location | Switch to Defaults

Component Name: gtz_raw_data

Line rate, GT Selection | **Clocking** | PCS Options | Optional Ports and Features | Summary

Octal Clocking

TX/RX Clock Selection

TXOUTCLK0	TXOUTCLK LANE0
TXOUTCLK1	TXOUTCLK LANE0
RXOUTCLK0	RXOUTCLK LANE0
RXOUTCLK1	RXOUTCLK LANE0
RXOUTCLK2	RXOUTCLK LANE0
RXOUTCLK3	RXOUTCLK LANE0

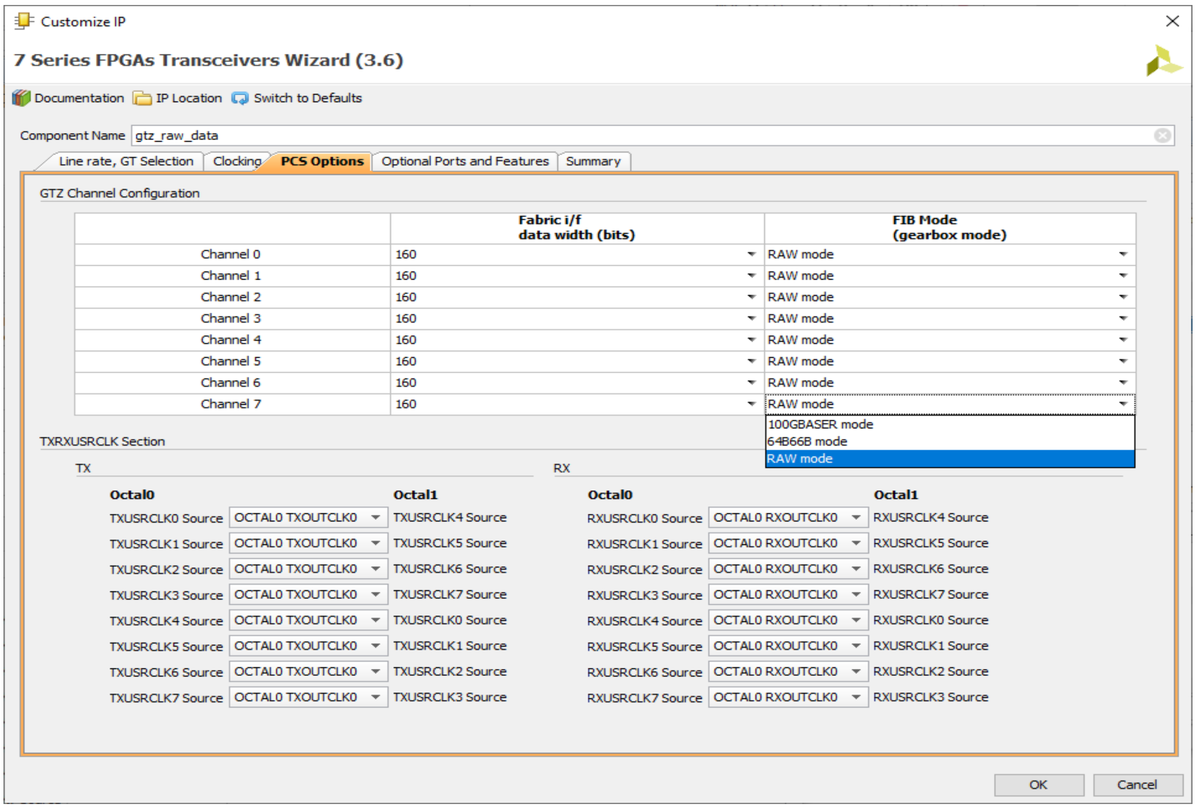
DRP CLK Source: DRPCLK0

Channel 0 Clocking (Select a channel from Right)

TXUSRCLK_SEL_LANE	TXUSRCLK0
RXUSRCLK_SEL_LANE	RXUSRCLK0

Figure 4.7: Octal and Channel Clocking – GTZ Page 2.

9. Page 3 of the Wizard (Figure 4.8) allows us to select the data width and Physical Coding Sublayer (PCS) mode options.
   a) The data width for each of the eight channels enabled on "*Line Rate, GT Selection*" tab (i.e. page 1) was set to 160 bits. The options shown here are dependent on the line rates entered on page 1; for a channel with line rate of 28.05 Gbps, only 160-bit mode is applicable.
   b) Depending on the data width and the protocol selected, there is a number of options for the PCS mode (e.g. 100GBASER, 64B/66B, RAW) which was set to the RAW mode for all eight channels. This means that no data encoding / decoding scheme was used.


Customize IP

7 Series FPGAs Transceivers Wizard (3.6)

Documentation | IP Location | Switch to Defaults

Component Name: gtz_raw_data

Line rate, GT Selection | Clocking | **PCS Options** | Optional Ports and Features | Summary

GTZ Channel Configuration

	Fabric i/f data width (bits)	FIB Mode (gearbox mode)
Channel 0	160	RAW mode
Channel 1	160	RAW mode
Channel 2	160	RAW mode
Channel 3	160	RAW mode
Channel 4	160	RAW mode
Channel 5	160	RAW mode
Channel 6	160	RAW mode
Channel 7	160	RAW mode

FIB Mode options: 100GBASER mode, 64B66B mode, RAW mode

TXRXUSRCLK Section

TX

Octal0		Octal1
TXUSRCLK0 Source	OCTAL0 TXOUTCLK0	TXUSRCLK4 Source
TXUSRCLK1 Source	OCTAL0 TXOUTCLK0	TXUSRCLK5 Source
TXUSRCLK2 Source	OCTAL0 TXOUTCLK0	TXUSRCLK6 Source
TXUSRCLK3 Source	OCTAL0 TXOUTCLK0	TXUSRCLK7 Source
TXUSRCLK4 Source	OCTAL0 TXOUTCLK0	TXUSRCLK0 Source
TXUSRCLK5 Source	OCTAL0 TXOUTCLK0	TXUSRCLK1 Source
TXUSRCLK6 Source	OCTAL0 TXOUTCLK0	TXUSRCLK2 Source
TXUSRCLK7 Source	OCTAL0 TXOUTCLK0	TXUSRCLK3 Source

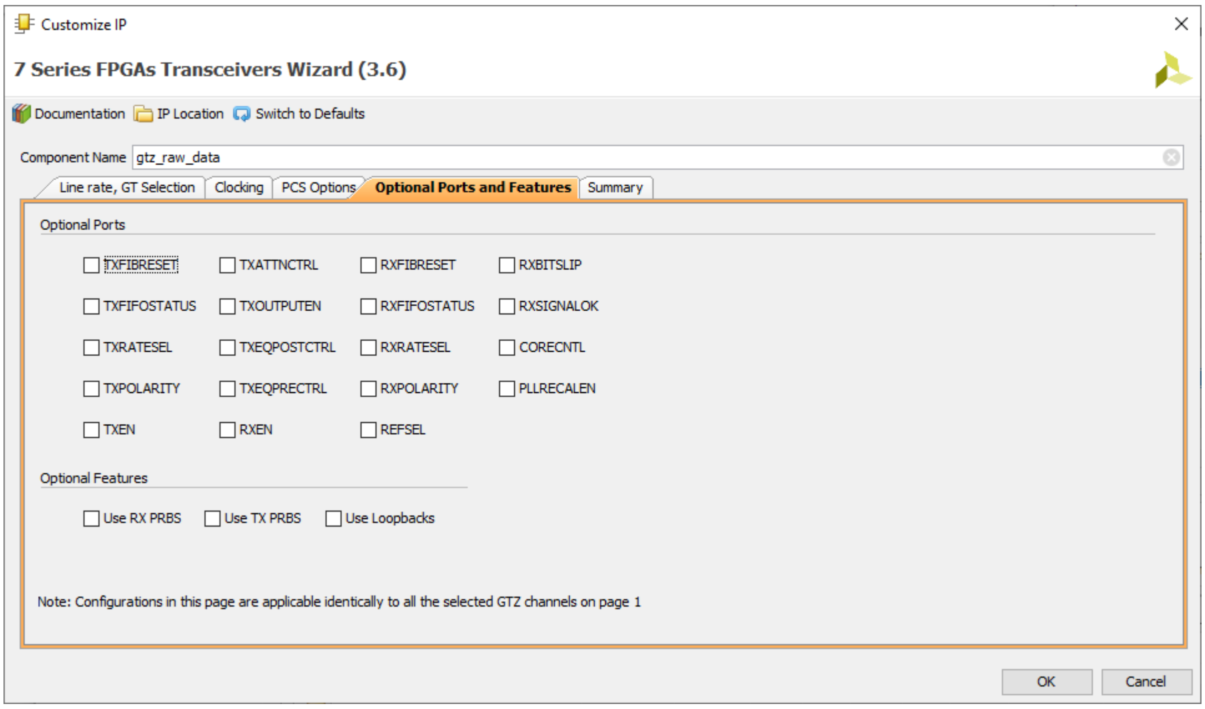
RX

Octal0		Octal1
RXUSRCLK0 Source	OCTAL0 RXOUTCLK0	RXUSRCLK4 Source
RXUSRCLK1 Source	OCTAL0 RXOUTCLK0	RXUSRCLK5 Source
RXUSRCLK2 Source	OCTAL0 RXOUTCLK0	RXUSRCLK6 Source
RXUSRCLK3 Source	OCTAL0 RXOUTCLK0	RXUSRCLK7 Source
RXUSRCLK4 Source	OCTAL0 RXOUTCLK0	RXUSRCLK0 Source
RXUSRCLK5 Source	OCTAL0 RXOUTCLK0	RXUSRCLK1 Source
RXUSRCLK6 Source	OCTAL0 RXOUTCLK0	RXUSRCLK2 Source
RXUSRCLK7 Source	OCTAL0 RXOUTCLK0	RXUSRCLK3 Source

OK | Cancel

Figure 4.8: PCS Options – GTZ Page 3.

10. In the 4th page of the Wizard (Figure 4.9) we are allowed to select the optional ports to bring out to the example top and multi-transceiver wrapper. Nothing was selected here.


Customize IP

7 Series FPGAs Transceivers Wizard (3.6)

Documentation | IP Location | Switch to Defaults

Component Name: gtz_raw_data

Line rate, GT Selection | Clocking | PCS Options | **Optional Ports and Features** | Summary

Optional Ports

☐ TXFIBRESET	☐ TXATTNCTRL	☐ RXFIBRESET	☐ RXBITSLIP
☐ TXFIFOSTATUS	☐ TXOUTPUTEN	☐ RXFIFOSTATUS	☐ RXSIGNALOK
☐ TXRATESEL	☐ TXEQPOSTCTRL	☐ RXRATESEL	☐ CORECNTL
☐ TXPOLARITY	☐ TXEQPRECTRL	☐ RXPOLARITY	☐ PLLRECALEN
☐ TXEN	☐ RXEN	☐ REFSEL	

Optional Features

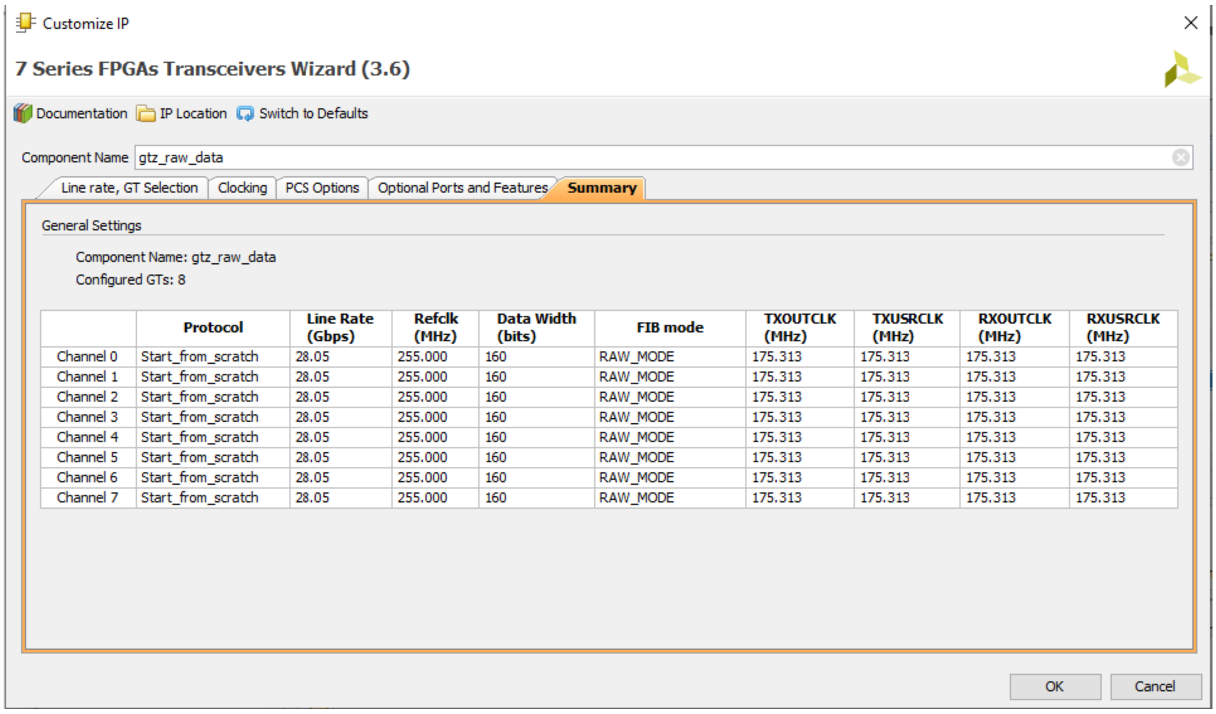
☐ Use RX PRBS ☐ Use TX PRBS ☐ Use Loopbacks

Note: Configurations in this page are applicable identically to all the selected GTZ channels on page 1

OK | Cancel

Figure 4.9: Optional Ports and Features – GTZ Page 4.

11. Finally, the 5th page of the Wizard (Figure 4.10) provides a summary of the selected configuration parameters. After reviewing the settings, we clicked "*Generate*" in the Generate Output Products window that pops up by clicking "*OK*" in the summary page.



Component Name gtz_raw_data

General Settings

Component Name: gtz_raw_data
Configured GTs: 8

	Protocol	Line Rate (Gbps)	Refclk (MHz)	Data Width (bits)	FIB mode	TXOUTCLK (MHz)	TXUSRCLK (MHz)	RXOUTCLK (MHz)	RXUSRCLK (MHz)
Channel 0	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 1	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 2	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 3	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 4	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 5	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 6	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313
Channel 7	Start_from_scratch	28.05	255.000	160	RAW_MODE	175.313	175.313	175.313	175.313

Figure 4.10: Summary – GTZ Page 5.

4.2.2 The core's Example Design

Following the procedure described in the previous subsection we can generate an instance of the 7 series FPGAs Transceivers Wizard IP core with the desired features. In addition to customizing and generating this IP core, the Wizard enables us to produce a really useful example design that provides a starting point for integrating the customized core into our system, including reference clock buffers and example system-level constraints.

For this purpose, after the product generation is complete, in the Project Manager panel, in the Sources window, we have to right-click the core name "*gtz_raw_data*" (gtz_raw_data.xci) and select "*Open IP Example Design*" as shown in Figure 4.11.

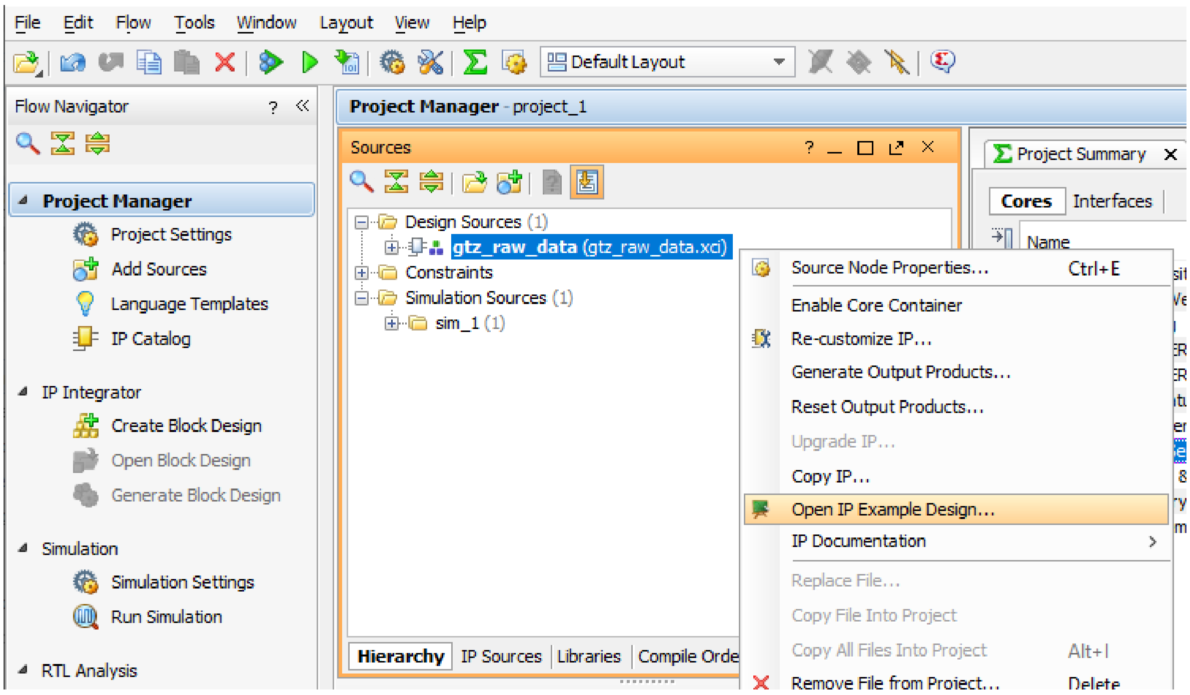
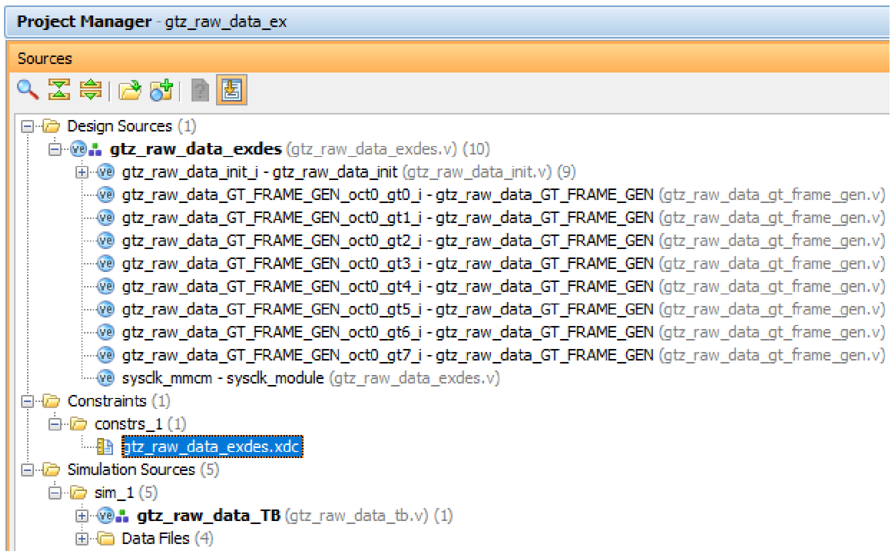


Figure 4.11: Open IP Example Design for the customized core.

A totally new Vivado project is created for the needs of the example design, which opens automatically in a new Vivado window. By default, the names of the new project and its source files (e.g. Verilog, XDC constraint files, etc.) are directly related to the core name; in our case the whole Vivado project was named "*gtz_raw_data*_ex" (this was also the name of the project's root directory). Along with several other Verilog modules, the *gtz_raw_data_exdes*, the *sys_clk*, the *gtz_raw_data_gt_frame_gen*, and the *gtz_raw_data_init* were generated in the context of the example design of our customized core.

In addition, the Wizard created the following two files with all the major timing and physical constraints for the example design:

1) **gtz_raw_data_exdes.xdc:** a Xilinx® Design Constraints (XDC) file including timing constraints (like create_clock, set_false_path, and create_generated_clock) for the example design. An XDC file is essentially a set of Tcl commands that the Vivado Tcl interpreter sequentially reads and parses. This file can be found under the *gtz_raw_data_ex/gtz_raw_data_ex.srcs/constrs_1/imports/example_design* directory, or through the Vivado GUI, in the Project Manager panel, in the Sources window, under the constraint set named "*constrs_1*" (see Figure 4.12).



```
Project Manager - gtz_raw_data_ex
Sources
Design Sources (1)
  gtz_raw_data_exdes (gtz_raw_data_exdes.v) (10)
    gtz_raw_data_init_i - gtz_raw_data_init (gtz_raw_data_init.v) (9)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt0_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt1_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt2_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt3_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt4_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt5_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt6_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    gtz_raw_data_GT_FRAME_GEN_oct0_gt7_i - gtz_raw_data_GT_FRAME_GEN (gtz_raw_data_gt_frame_gen.v)
    sysclk_mmcm - sysclk_module (gtz_raw_data_exdes.v)
Constraints (1)
  constrs_1 (1)
    gtz_raw_data_exdes.xdc
Simulation Sources (5)
  sim_1 (5)
    gtz_raw_data_TB (gtz_raw_data_tb.v) (1)
    Data Files (4)
```

Figure 4.12: Hierarchy in the "gtz_raw_data_ex" Vivado project.

2) **v7ht.tcl:** a Vivado Tcl script containing physical constraints for the example design. This file is actually a collection of Tcl commands that imposes three different types of physical constraints:
   - **Netlist constraints**, such as DON'T_TOUCH and LOCK_PINS, that are set on netlist objects (i.e. ports, pins, nets or cells) to require the compilation tool to handle them in special way.
   - **Placement constraints**, like PACKAGE_PIN, PROHIBIT, BEL, and LOC, that are applied to cells to control their locations within the device.
   - **Routing constraints**, like FIXED_ROUTE, that are applied to net objects to control their routing resources.

   All these constraints concern exclusively the GTZ transceivers' design elements, and without them, it would be impossible to properly use the GTZ transceivers available on the XC7VH580T-G2HCG1155E FPGA. The v7ht.tcl file was generated under the *gtz_raw_data_ex/gtz_raw_data_ex.srcs/sources_1/ip/gtz_raw_data/tcl* directory (remember that the "*gtz_raw_data_ex*" is the project's root directory).

The rest of this subsection will provide details on how the, generated by the Wizard, modules and constraints were modified and used for the needs of our final design.

The "gtz_raw_data_exdes" module

As already mentioned, the *gtz_raw_data_exdes* is intended to be the top-level module of our design. In other words, the *gtz_raw_data_exdes* will be the wrapper of the entire design to be implemented in the FPGA. To this end, this module was modified to contain one instance of the *sys_clk* module, one instance of the *gtz_raw_data_init* module, one instance of the *pr_controller* module (we will discuss this module further in the next section), eight instances of the *gtz_raw_data_gt_frame_gen* module (one per GTZ channel), and all the data objects (e.g. wires, regs, etc.) needed to enable interconnection and communication between these modules.

Furthermore, the *gtz_raw_data_exdes* creates some Input / Output (I/O) ports to be assigned to physical pins on the FPGA device, thus enabling the HW design implemented in it (i.e. in the FPGA) to make use of other modules that are installed, along with the FPGA, on the VC7222 board (e.g. clock sources, general purpose DIP switches, LEDs, pushbuttons, USB-to-UART Bridge, connector pads for the transceivers, etc.).

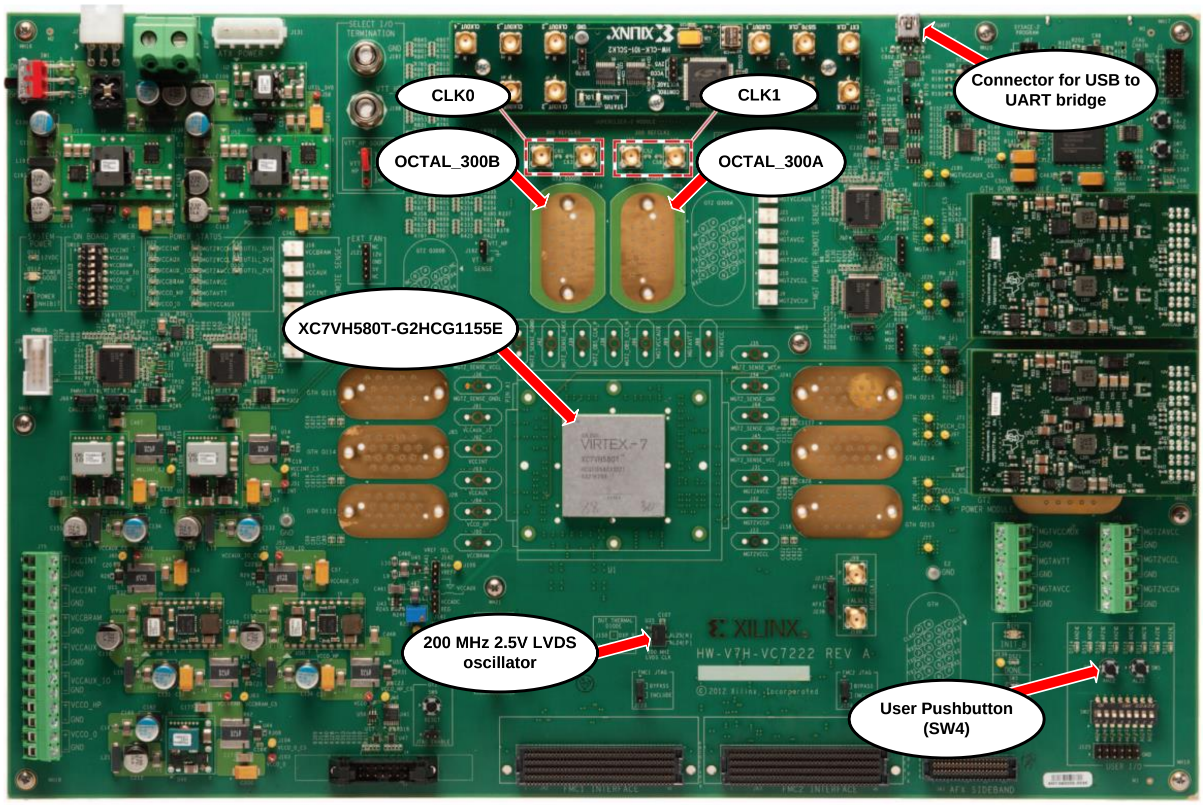


Figure 4.13: Modules on VC7222 board used by the final design.

As Figure 4.13 illustrates, our final design needs to be boosted by the following hardwired modules available on the VC7222 board:

- **200 MHz 2.5V LVDS oscillator:** this oscillator (connected to multi-region clock capable (MRCC) inputs on the FPGA) is the source of a 200 MHZ, free running, differential System/board clock used to drive the FPGA logic in our design.
- **User Pushbutton SW4 (active high):** force the *pr_controller* module and all the eight frame generators (instances of the *gtz_raw_data_gt_frame_gen* module) to be reset by pushing this button.

- **Connector for USB to UART bridge:** the AXI UART Lite soft IP core of Xilinx, instantiated into the *pr_controller* module, is brought out to a mini-B USB connector on our target-board.
- **GTZ SMA Connectors:** the GTZ reference clocks (CLK0 and CLK1) are brought out to two pairs of differential SMA (SubMiniature version A) transceiver clock inputs.
- **GTZ Connector Pads (300A & 300B):** the GTZ octal is brought out to two connector pads, which interface with Samtec BullsEye connectors used with the Samtec HDR-155805-01-BEYE cable assembly. As shown in Figure 4.14, the GTZ channels 0 – 3 are brought out to the GTZ 300A connector, while the GTZ channels 4 – 7 are brought out to the GTZ 300B connector. The TXi and RXi in the figure below ($0 \leq i \leq 7$), illustrate the transmitter and receiver of the ith GTZ channel respectively, and each of them is a differential pair of signals.

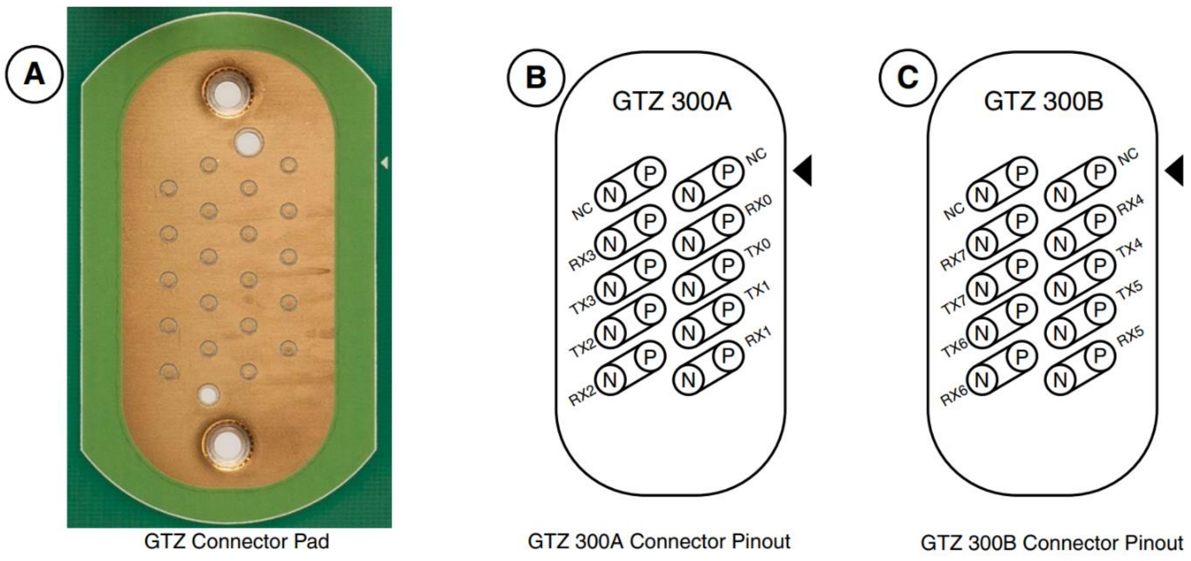


Figure 4.14: A – GTZ Connector Pad. B – GTZ 300A Connector Pinout. C – GTZ 300B Connector Pinout [12].

Towards interconnecting the final design implemented in the XC7VH580T-G2HCG1155E with the aforementioned modules on the VC7222 board, the *gtz_raw_data_exdes* was modified to have the following ports (lines beginning with // are just comments in Verilog):

```
//
// I/O ports of the top-level module
//

module gtz_raw_data_exdes
  (
   // GTZ octal signals
   output wire OCT0_GT0_TXP_OUT,
   output wire OCT0_GT0_TXN_OUT,
   input wire OCT0_GT0_RXP_IN,
   input wire OCT0_GT0_RXN_IN,

    output wire OCT0_GT1_TXP_OUT,
    output wire OCT0_GT1_TXN_OUT,
    input wire OCT0_GT1_RXP_IN,
    input wire OCT0_GT1_RXN_IN,

    output wire OCT0_GT2_TXP_OUT,
    output wire OCT0_GT2_TXN_OUT,
    input wire OCT0_GT2_RXP_IN,
    input wire OCT0_GT2_RXN_IN,

    output wire OCT0_GT3_TXP_OUT,
    output wire OCT0_GT3_TXN_OUT,
    input wire OCT0_GT3_RXP_IN,
    input wire OCT0_GT3_RXN_IN,

    output wire OCT0_GT4_TXP_OUT,
    output wire OCT0_GT4_TXN_OUT,
    input wire OCT0_GT4_RXP_IN,
    input wire OCT0_GT4_RXN_IN,

    output wire OCT0_GT5_TXP_OUT,
    output wire OCT0_GT5_TXN_OUT,
    input wire OCT0_GT5_RXP_IN,
    input wire OCT0_GT5_RXN_IN,

    output wire OCT0_GT6_TXP_OUT,
    output wire OCT0_GT6_TXN_OUT,
    input wire OCT0_GT6_RXP_IN,
    input wire OCT0_GT6_RXN_IN,

    output wire OCT0_GT7_TXP_OUT,
    output wire OCT0_GT7_TXN_OUT,
    input wire OCT0_GT7_RXP_IN,
    input wire OCT0_GT7_RXN_IN,

    // GTZ reference clocks
    (* CLOCK_BUFFER_TYPE = "NONE" *) input wire OCT0_REFCLK0P_IN,
    (* CLOCK_BUFFER_TYPE = "NONE" *) input wire OCT0_REFCLK0N_IN,
    (* CLOCK_BUFFER_TYPE = "NONE" *) input wire OCT0_REFCLK1P_IN,
    (* CLOCK_BUFFER_TYPE = "NONE" *) input wire OCT0_REFCLK1N_IN,

    // System Clock
    input wire SYSCLK_P_IN,
    input wire SYSCLK_N_IN,

    // Universal Asynchronous Receiver-Transmitter (UART)
    input wire UART_RXD,
    output wire UART_TXD,

    // Reset Pushbutton - SW4
    input wire HARD_RESET

    );
```

The interconnection is eventually achieved by mapping each of the top-level module's ports (described in Verilog as shown above) to a specific physical pin on the FPGA. For this purpose, the following constraints were added in the *gtz_raw_data_exdes.xdc* file:

```
# Map I/O ports of the top-level module to specific
# pins on the XC7VH580T-G2HCG1155E FPGA, based on the
# VC7222 board master XDC file
# REFCLK
set_property PACKAGE_PIN E17 [get_ports OCT0_REFCLK0P_IN]
set_property PACKAGE_PIN E16 [get_ports OCT0_REFCLK0N_IN]
set_property PACKAGE_PIN E21 [get_ports OCT0_REFCLK1P_IN]
set_property PACKAGE_PIN E20 [get_ports OCT0_REFCLK1N_IN]

# System Clock - 200MHz diff clock
set_property PACKAGE_PIN AL24 [get_ports SYSCLK_P_IN]
set_property IOSTANDARD LVDS [get_ports SYSCLK_P_IN]
set_property PACKAGE_PIN AL25 [get_ports SYSCLK_N_IN]
set_property IOSTANDARD LVDS [get_ports SYSCLK_N_IN]

# UART
set_property PACKAGE_PIN AK10 [get_ports UART_TXD]
set_property IOSTANDARD LVCMOS18 [get_ports UART_TXD]
set_property PACKAGE_PIN AL10 [get_ports UART_RXD]
set_property IOSTANDARD LVCMOS18 [get_ports UART_RXD]

# Reset PB – PushButton2(SW4)
set_property PACKAGE_PIN AM22 [get_ports HARD_RESET]
set_property IOSTANDARD LVCMOS18 [get_ports HARD_RESET]

# TX - RX Pins
set_property PACKAGE_PIN C29 [get_ports OCT0_GT0_TXP_OUT]
set_property PACKAGE_PIN C28 [get_ports OCT0_GT0_TXN_OUT]
set_property PACKAGE_PIN A28 [get_ports OCT0_GT1_TXP_OUT]
set_property PACKAGE_PIN A27 [get_ports OCT0_GT1_TXN_OUT]
set_property PACKAGE_PIN C26 [get_ports OCT0_GT2_TXP_OUT]
set_property PACKAGE_PIN C25 [get_ports OCT0_GT2_TXN_OUT]
set_property PACKAGE_PIN A25 [get_ports OCT0_GT3_TXP_OUT]
set_property PACKAGE_PIN A24 [get_ports OCT0_GT3_TXN_OUT]
set_property PACKAGE_PIN C17 [get_ports OCT0_GT4_TXP_OUT]
set_property PACKAGE_PIN C16 [get_ports OCT0_GT4_TXN_OUT]
set_property PACKAGE_PIN A16 [get_ports OCT0_GT5_TXP_OUT]
set_property PACKAGE_PIN A15 [get_ports OCT0_GT5_TXN_OUT]
set_property PACKAGE_PIN C14 [get_ports OCT0_GT6_TXP_OUT]
set_property PACKAGE_PIN C13 [get_ports OCT0_GT6_TXN_OUT]
set_property PACKAGE_PIN A13 [get_ports OCT0_GT7_TXP_OUT]
set_property PACKAGE_PIN A12 [get_ports OCT0_GT7_TXN_OUT]
set_property PACKAGE_PIN C23 [get_ports OCT0_GT0_RXP_IN]
set_property PACKAGE_PIN C22 [get_ports OCT0_GT0_RXN_IN]
set_property PACKAGE_PIN A22 [get_ports OCT0_GT1_RXP_IN]
set_property PACKAGE_PIN A21 [get_ports OCT0_GT1_RXN_IN]
set_property PACKAGE_PIN C20 [get_ports OCT0_GT2_RXP_IN]
set_property PACKAGE_PIN C19 [get_ports OCT0_GT2_RXN_IN]
set_property PACKAGE_PIN A19 [get_ports OCT0_GT3_RXP_IN]
set_property PACKAGE_PIN A18 [get_ports OCT0_GT3_RXN_IN]
set_property PACKAGE_PIN C11 [get_ports OCT0_GT4_RXP_IN]
set_property PACKAGE_PIN C10 [get_ports OCT0_GT4_RXN_IN]
set_property PACKAGE_PIN A10 [get_ports OCT0_GT5_RXP_IN]
set_property PACKAGE_PIN A9 [get_ports OCT0_GT5_RXN_IN]
set_property PACKAGE_PIN C8 [get_ports OCT0_GT6_RXP_IN]
set_property PACKAGE_PIN C7 [get_ports OCT0_GT6_RXN_IN]
set_property PACKAGE_PIN A7 [get_ports OCT0_GT7_RXP_IN]
set_property PACKAGE_PIN A6 [get_ports OCT0_GT7_RXN_IN]
```

The "sys_clk" module

This module takes the differential clock sourced by the 200 MHz 2.5V LVDS oscillator as input, and generates a single-ended clock running at 100 MHz. The differential system clock is divided down internally using a 7 series FPGAs Mixed-Mode Clock Manager (MMCM) to satisfy timing constraints. To this end, the MMCME2_ADV primitive is instantiated (and used) into the *sys_clk* module.

The "gtz_raw_data_init" module

This is the main transceiver wrapper, used to:

- Instantiate individual transceiver wrappers, which in turn instantiate the selected transceivers with settings for the selected protocol.
- Instantiate modules to perform the necessary reset sequence for each GTZ channel in the octal.
- Interface the user logic with the GTZ transceivers.

The "gtz_raw_data_gt_frame_gen" module

This module has two input ports for receiving clock and reset signals, and one 160-bit width output port that can be connected, through the *gtz_raw_data_init* module, to a GTZ channel to provide data for transmission. At the positive edge of the clock signal, a word of 160 bits is made available by the frame generator to the corresponding GTZ transceiver in order for it to be transmitted as a stream of serial bits. In case that the reset signal is asserted, all 160 bits of this module's output signal are set to zero. Otherwise, the *gtz_raw_data_gt_frame_gen* obtains the data to be transmitted, from a 512-word ROM (a word consists of 160 bits) implemented in the FPGA's internal memory resources. Starting from the ROM's 1st address, the module reads one word per clock cycle from the memory, until it reads all 512 words and starts all over again from the beginning. The ROM is created and initialized by the *gtz_raw_data_gt_frame_gen* module itself.

```
module gtz_raw_data_gt_frame_gen
   (
    output reg [159:0] TX_DATA_OUT,
    input wire         USER_CLK,
    input wire         SYSTEM_RESET
    );
```

In order for each channel of the GTZ-octal to have its own, dedicated frame generator, this module is instantiated eight times in the top-level module of our design. The ith instance of the *gtz_raw_data_gt_frame_gen*, FRAM_GEN_i ($0 \leq i \leq 7$), is connected to the ith transceiver, GT_i ($0 \leq i \leq 7$), of the GTZ-octal through the *gtz_raw_data_init* module. The reset signal of each FRAM_GEN_i (where $0 \leq i \leq 7$), is driven by the *HARD_RESET* signal of the *gtz_raw_data_exdes* module, so that all eight frame generators can be reset simultaneously by simply pressing the Pushbutton SW4 available on the VC7222 board.

Being the only one reconfigurable frame generator, FRAME_GEN_0 differs from all the others in the following respects:

- The FRAME_GEN_0 comes in three flavors. The first one called *FRAME_GEN_0_dummy,* is used just to synthesize the static logic before the Partial Reconfiguration (PR) workflow. For this reason, the *FRAME_GEN_0_dummy* only contains a top file with definitions of the signals seen outside the generator and no additional logic. The two others, called *FRAME_GEN_0_v1* and *FRAME_GEN_0_v2*, are the Reconfigurable Modules to be loaded in the Reconfigurable Region (RR) of our device. *FRAME_GEN_0_v1* and *FRAME_GEN_0_*v2 are interchangeable and each of them contains all the logic needed to implement a regular frame generator (e.g. create and initialize a ROM, provide output data, etc.). What makes the *FRAME_GEN_0_v1* different from the *FRAME_GEN_0_v2*, is the data that each of them provides to the corresponding transceiver (GT_0).
- Because the logic inside the Reconfigurable Partition is modified while the device is operating, the static logic connected to outputs of Reconfigurable Modules (RMs) must ignore all incoming data during the PR procedure. The reconfigurable logic is in an unknown state during reconfiguration and RMs do not output valid data until PR is complete and the RMs are reset. In the light of the above, the necessary logic was added in the top-level module (i.e. the *gtz_raw_data_exdes*) in order to implement the following:
    - The output of the FRAME_GEN_0 is not directly connected to the GT_0; it is one of the two (160-bit width) input signals of a 2-to-1 multiplexer (mux). The second input signal of this multiplexer is a word of 160 zero bits. The mux has one 160-bit width output port connected to the GT_0 through the *gtz_raw_data_init* module. If the mux's *selector* signal is asserted, the second input signal (i.e. the one containing 160 zero bits) is sent to the multiplexer's output and, consequently, to the GT_0 transceiver. Otherwise, the mux allows the output of the FRAME_GEN_0 to pass through it unaffected. The mux's *selector* is driven by the *busy* signal of the *pr_controller* module which is asserted as long as the PR takes place (we will discuss this module further in the next section).
    - Apart from the *HARD_RESET* signal of the *gtz_raw_data_exdes* module, the *busy* signal generated by the *pr_controller* module also drives the reset signal of the FRAME_GEN_0. The *busy* signal, which is asserted during the PR procedure, forces the logic within the FRAME_GEN_0 to be automatically reset after each PR.

4.3 The "*pr_controller*"

The last modification made to the example-design presented in the previous section, was to implement some additional logic (as part of the static logic) to enable the PR of our device in the manner described in subsection 3.4.5 above. This additional logic is provided by a suitable block-design (BD) created in the Vivado IP Integrator, within the "*gtz_raw_data*_ex" Vivado project. The *pr_controller* is essentially the wrapper around the block design; it is used to instantiate the BD into our top-level module (i.e. the *gtz_raw_data_exdes*) and is defined as follows (Table 4.1):

```
module pr_controller
   (
    input wire Clk,
    output wire busy,
    input wire ena,
    output wire ready,
    input wire reset_rtl,
    input wire uart_rtl_rxd,
    output wire uart_rtl_txd,
   );
```

Signal	**Direction**	**Width (bits)**	**Description**
Clk	input	1	All IP cores within the block design are synchronous to this 100 MHz clock signal. It is provided by the *sys_clk* module presented in the previous section.
busy	output	1	This signal is asserted as long as the Partial Reconfiguration (PR) takes place.
ena	input	1	It controls the operation of the read-only memory generated using the Block Memory Generator IP core. When deasserted, no Read or reset operations are performed on the port A of this memory. This signal was always asserted.
ready	output	1	It is asserted just for one clock cycle to indicate that PR has been successfully completed.
reset_rtl	input	1	Active high reset signal for the entire BD. This input is driven by the *HARD_RESET* signal of the *gtz_raw_data_exdes* module.
uart_rtl_rxd	input	1	Input signal of the UART Lite IP core to receive data. It is connected to the *UART_RXD* port of the *gtz_raw_data_exdes*.
uart_rtl_txd	output	1	Output signal of the UART Lite IP core to transmit data. It is connected to the *UART_TXD* port of the *gtz_raw_data_exdes*.

Table 4.1: I/O signals of the "pr_controller" wrapper.

Figure 4.15 shows the said block-design, the IP cores of which are presented in the following subsections.

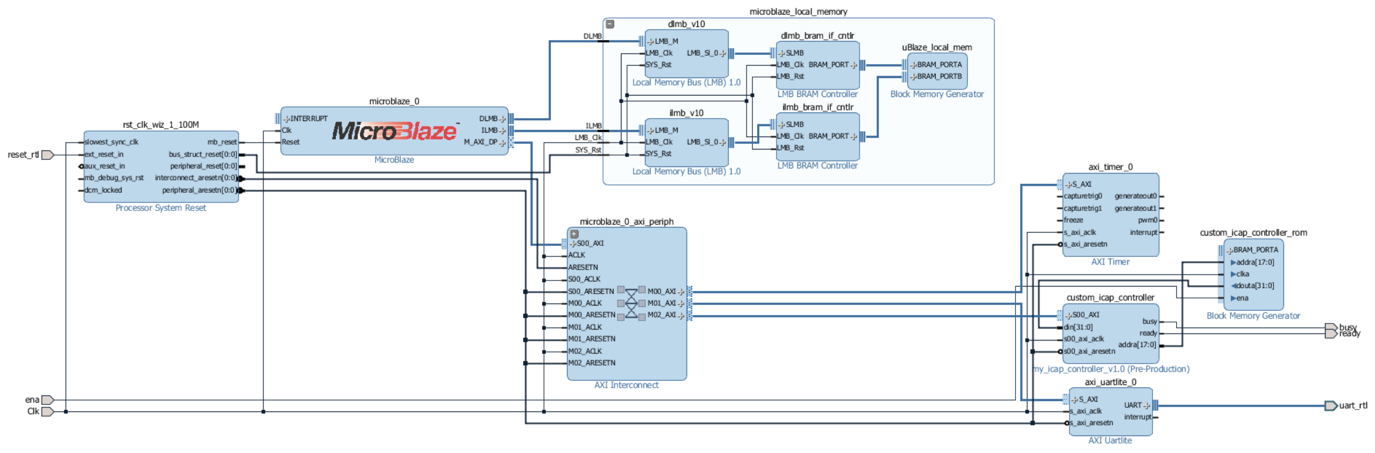


Figure 4.15: The “pr_controller”–related Block Design.

4.3.1 MicroBlaze$^{™}$ Processor

The MicroBlaze$^{™}$ embedded processor is a reduced instruction set computer (RISC) core, optimized for implementation in Xilinx® FPGAs. The following figure shows a functional block design of the MicroBlaze core [22].

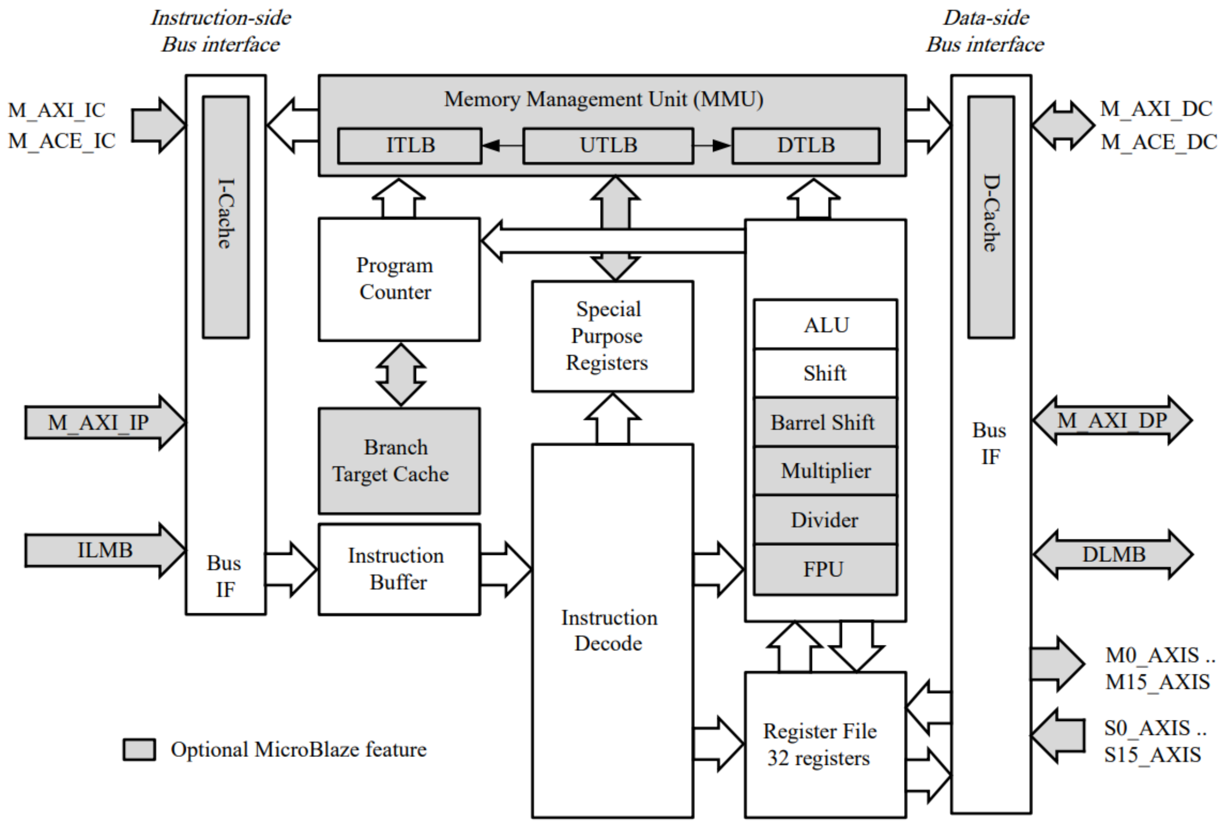


Figure 4.16: Block Design of MicroBlaze Core [22].

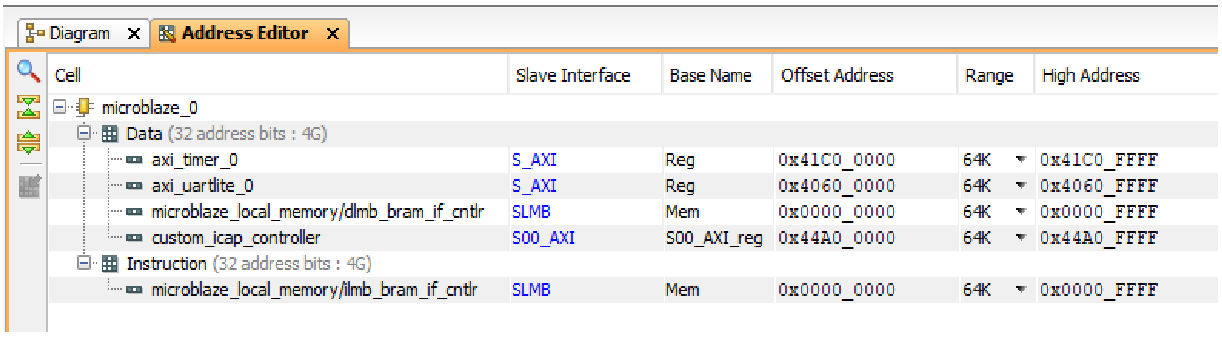
The MicroBlaze is a highly configurable processor which, as a soft-core, is implemented entirely in the FPGA resources (LUTs, BRAMs, DSPs, etc.). It is organized as a Harvard architecture with separate bus interface units for data and instruction accesses. MicroBlaze does not separate data accesses to I/O and memory; it uses memory-mapped I/O. The processor has up to three interfaces for memory accesses [22]:

- Local Memory Bus (LMB)
- ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface 4 (AXI4) for peripheral access.
- ARM AMBA® AXI4 or AXI Coherency Extension (ACE) for cache access.

In the context of this thesis the MicroBlaze was used as a 32-bit processor configured to have the following interfaces:

1) **Local memory Bus Instruction Interface (ILMB):** provides access to fast local memory for instructions.
2) **Local memory Bus Data Interface (DLMB):** provides access to fast local memory for data and vectors.
3) **Peripheral AXI Data Interface (M_AXI_DP):** this interface connects to peripheral I/O using AXI4-Lite.
4) **Core Interface:** miscellaneous signals for: clock, reset, interrupt.

The local memory of the MicroBlaze processor, as well as the *axi_uartlite_0*, *axi_timer_0*, and *custom_icap_controller* peripherals that were connected to it, were memory-mapped in the "Address Editor” window of the IP Integrator as shown in the following figure:



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
axi_timer_0	S_AXI	Reg	0x41C0_0000	64K	0x41C0_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF
custom_icap_controller	S00_AXI	S00_AXI_reg	0x44A0_0000	64K	0x44A0_FFFF
Instruction (32 address bits : 4G)					
microblaze_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF

Figure 4.17: Memory-mapped slaves of the MicroBlaze system.

As we can see, the instruction and data memory ranges were made to overlap by mapping them both to the same physical memory.

4.3.2 MicroBlaze Local Memory

The MicroBlaze processor needs to use a memory to store instructions and data of SW applications running on it. To this end, the Xilinx® LogiCORE™ IP Block Memory Generator (BMG) core was used to create a fast, local memory out of block RAM resources available on the FPGA device. The BMG core (named *uBlaze_local_mem*) was configured as a True Dual-Port RAM with 32-bits address interfaces, and 64 KBytes storage capacity. The True Dual-Port RAM provides two ports, A and B, as illustrated in Figure 4.15. Read and Write accesses to the memory are allowed on either port. The BMG core was used in "BRAM Controller" mode, so that each of its ports (i.e. A and B) can be connected to the corresponding MicroBlaze LMB interface (i.e. DLMB and ILMB) through a dedicated set of IP cores composed by a Local Memory Bus (LMB) and an LMB BRAM Interface Controller (see Figure 4.15).

4.3.3 Processor System Reset

The Xilinx LogiCORE™ IP Processor System Reset Module core provides customized resets for the entire MicroBlaze system, including the processor, the interconnect and peripherals [23]. The *ext_reset_in* port of this IP core was exposed as *reset_rtl* to the *pr_controller* wrapper in order to be connected to the *HARD_RESET* signal of the top-level module (i.e. *gtz_raw_data_exdes*). As a result, the entire MicroBlaze system could be reset by pressing the SW4 Pushbutton available on the VC7222 board.

4.3.4 AXI Interconnect

AXI Interconnect IP connects one or more AXI memory-mapped Master devices to one or more AXI memory-mapped Slave devices. The devices can vary in terms of data width, clock domain and AXI sub-protocol (AXI4, AXI3, or AXI4-Lite) [24]. In our case this IP core was used in a 1-to-3 configuration in order for the MicroBlaze (Master device) to access the following three memory-mapped Slave peripherals (see Figure 4.15): the *axi_uartlite_0*, the *axi_timer_0*, and the *custom_icap_controller*.

4.3.5 AXI Timer

The LogiCORE™ IP AXI Timer is a 32/64-bit timer module that interfaces to the AXI4-Lite interface [25]. This IP core was used in 64-bit mode to measure the reconfiguration time.

4.3.6 AXI Uartlite

The LogiCORE™ IP AXI Universal Asynchronous Receiver Transmitter (UART) Lite core provides between UART signals and the AMBA® AXI interface and also provides a controller interface for asynchronous serial data transfer. This soft LogiCORE™ IP core is designed to interface with the AXI4-Lite protocol [26]. The core was customized in the Vivado IP Integrator as follows:

- Baud Rate = 9600 bits per second (bps)
- Number of data bits in the serial frame = 8
- Parity was not used

The *rx* (receive data) and *tx* (transmit data) signals of this core, bundled in the *UART* interface, are exposed as *uart_rtl_rxd* and *uart_rtl_txd*, respectively, to the *pr_controller* wrapper in order to be matched with the *UART_RXD* and *UART_TXD* signals of the *gtz_raw_data_exdes* module (respectively).

4.3.7 custom_icap_controller_rom

The *custom_icap_controller_rom* is the second instance of the Xilinx® LogiCORE™ IP Block Memory Generator (BMG) core in our Block Design (the first one was the *uBlaze_local_mem* used as a True Dual-Port RAM for the needs of processor's local memory). As already mentioned, the BMG core uses embedded Block Memory primitives in Xilinx® FPGAs to extend the functionality and capability of a single primitive to memories of arbitrary widths and depths. Sophisticated algorithms within the BMG core produce optimized solutions to provide convenient access to memories for a wide range of configurations [18].

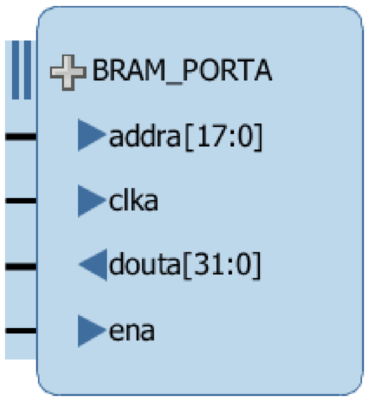


Figure 4.18: The "custom_icap_controller_rom".

In the case of *custom_icap_controller_rom*, our device's BRAM resources are used to generate a read-only memory storing the partial bitstream that will be loaded in the Reconfigurable Region of the device when PR is performed on it. This core was configured as a Single Port ROM in the Vivado IP Integrator. The Width and Depth values that were used for read operation in the memory port named Port A, was set to 32 bits and 170,383 respectively, resulting in a memory with a storage capacity of 170,383*32 bits or 681,532 Bytes (i.e. the size of a partial bitstream). The *custom_icap_controller* (which will be presented in the following subsection) was directly connected to the *custom_icap_controller_rom* without the need for using any memory controller (e.g. LMB or AXI), buses, or interconnects that implement a standard and predefined communication protocol. So, unlike the *uBlaze_local_mem*, in this case the BMG core was used in "Stand Alone" mode, and the I/O signals that are associated with the Port A and are bundled in the *BRAM_PORTA* interface, were handled individually as described in the following table:

Signal	Direction	Width (bits)	Description
addra	input	18	Input provided by the *custom_icap_controller*. It addresses the memory space for the Port A Read operation. The width of this port is determined by the memory depth ($2^{17} < 170{,}383 < 2^{18}$).
clka	input	1	Port A clock connected to the *Clk* signal of the *pr_controller* wrapper. Port A Read operations are synchronous to this clock.
douta	output	32	Data output from Read operations through port A. As already mentioned, the width was set to 32 bits. This output signal was monitored by the *custom_icap_controller*.
ena	input	1	If asserted, it enables all operations through Port A. It was exposed as *ena* to the *pr_controller* wrapper and its value was always set to 1.

Table 4.2: I/O signals of the "custom_icap_controller_rom".

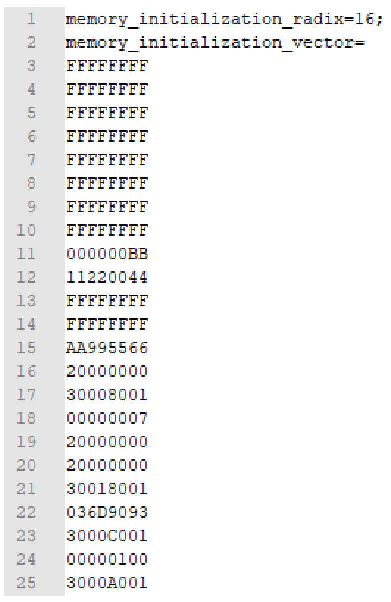
Finally, the *custom_icap_controller_rom* was configured to be initialized with the contents of a partial bitstream file, by selecting the "*Load Init File*" option in the "*Other Options*" tab of the BMG configuration window. Unfortunately, the bitstream cannot be loaded directly into the BRAM using the Xilinx's BMG core. This IP core only supports .coe files for memory initialization. A file of COE format is a text file which specifies two parameters [18]:

- **memory_initialization_radix:** The radix of the values in the memory_initialization_vector. Valid choices are 2, 10, or 16.
- **memory_initialization_vector:** Defines the contents of each memory element. Each value is LSB-justified and assumed to be in the radix defined by memory_initialization_radix. The values (or coefficients) can be separated by a space, a comma, or by placing one value in each line with a carriage return.

In the light of the above, we developed a C application that automates the conversion of a BIN file, generated by the Vivado tool to contain bitstream configuration data, into the equivalent COE file with the radix be set to 16 (i.e. hexadecimal system is used). This simple app takes a .bin file as input and does the followings:

1) It creates a .coe output file setting the memory_initialization_radix to 16.
2) Starting from the beginning, it reads 32-bit words of binary data from the .bin file and writes them in hexadecimal format to the .coe file, until reading the entire .bin file.
3) The 32-bit words (i.e. coefficients) written to the .coe file, are separated by placing one value in each line with a carriage return. The last coefficient is followed by the semicolon (;) special character indicating the end of the memory_initialization_vector.

Figure 4.19 shows the first twenty-five lines of the .coe file used to initialize the *custom_icap_controller_rom* (according to Table 3.5 the size of the .bin file containing the partial bitstream is 681,532 Bytes or 170,383 32-bit words, and therefore the equivalent .coe file, a small part of which is shown in the figure below, consists of 170,385 (170,383 + 2) lines).



```
memory_initialization_radix=16;
memory_initialization_vector=
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
FFFFFFFF
000000BB
11220044
FFFFFFFF
FFFFFFFF
AA995566
20000000
30008001
00000007
20000000
20000000
30018001
036D9093
3000C001
00000100
3000A001
```

Figure 4.19: Sectional view of the COE converted partial bitstream file.

4.3.8 *custom_icap_controller*

The custom_icap_controller instantiates the ICAPE2 primitive (see Subsection 3.3.2) in our design and implements a custom state machine through which the partial bitstream stored in the *custom_icap_controller_rom* is delivered to the ICAPE2 write port to replace functionality in a pre-defined device region during normal device operation. This module was implemented writing Verilog HDL code from scratch. To make it easy for this module to be distributed to third parties, as well as to be integrated into a microprocessor-based system (e.g. MicroBlaze, Zynq) so that it can be controlled by a SW application running on the microprocessor, the RTL was packaged as an AXI IP core (Figure 4.20) with the help of the Vivado® IDE **Create and Package New IP wizard**.

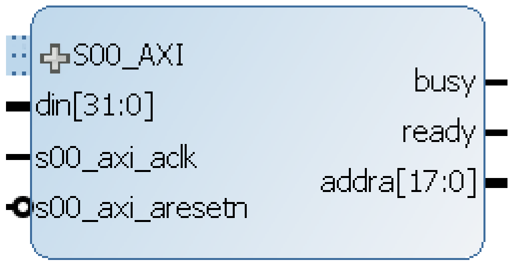


Figure 4.20: The "custom_icap_controller".

Table 4.3 describes how custom_icap_controller communicates and interacts with the rest of the HW design.

Signal	**Direction**	**Width (bits)**	**Description**
S00_AXI	input/output	N/A	Group of I/O signals implementing a Slave AXI4-Lite interface of 32-bit data width. This interface enables MicroBlaze to access the custom_icap_controller through register reads and writes over an AXI bus.
s00_axi_aclk	input	1	AXI clock connected to the *Clk* signal of the *pr_controller* wrapper. All operations are synchronous to this clock.
s00_axi_aresetn	input	1	Active-Low, AXI reset signal provided by the *Processor System Reset* module.
addra	output	18	Output port connected to the homonym port of the *custom_icap_controller_rom*. It addresses the memory space for the Read operations through port A of the *custom_icap_controller_rom* module.
din	input	32	Data input from Read operations through port A of the *custom_icap_controller_rom* module. It is connected to the *dout* port of the *custom_icap_controller_rom*.
busy	output	1	This signal is asserted as long as the Partial Reconfiguration (PR) takes place. It is exposed as *busy* to the *pr_controller* wrapper in order to be connected to the *SYSTEM_RESET* port of the *FRAME_GEN_0*.
ready	output	1	It is asserted just for one clock cycle to indicate that PR has been successfully completed.

Table 4.3: I/O signals of the "custom_icap_controller".

The custom state machine controlling the partial reconfiguration process directly drives the pins of the ICAP primitive as shown in Figure 4.21.

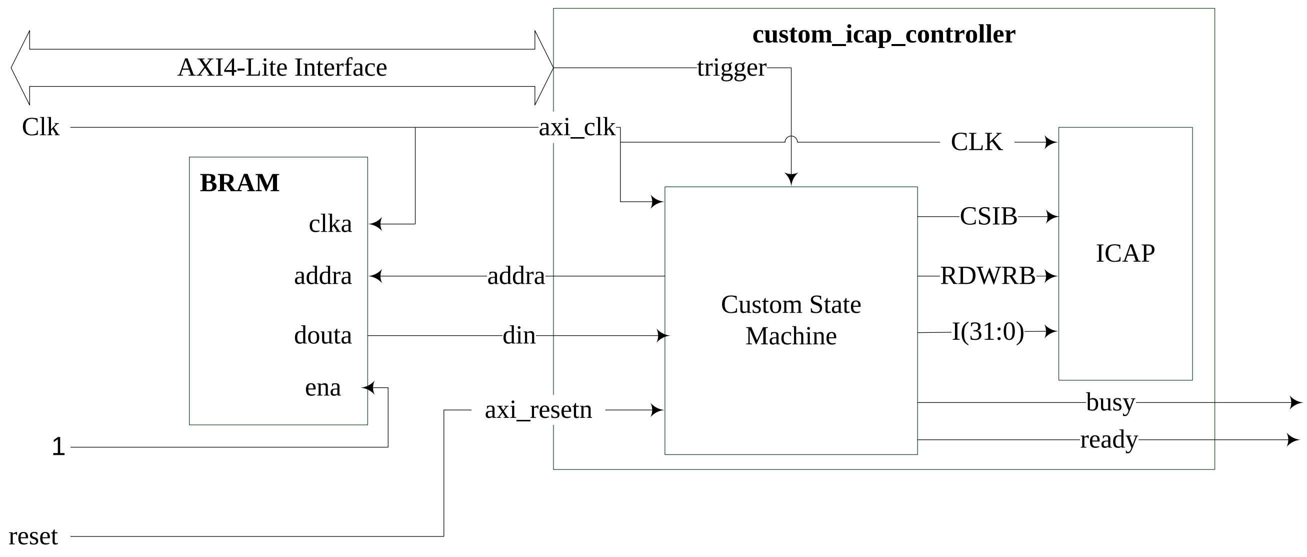


Figure 4.21: Custom state machine interface to the BMG core and ICAPE2 primitive.

While in rest mode, the custom state machine disables the ICAP by pulling the *CSIB* pin high, and sets the *busy*, the *ready* and all bits of the *addra* signal to zero. The PR process is triggered by means of an external trigger supplied by the user to the MicroBlaze through the serial port, and the MicroBlaze in turn propagates the trigger signal to the custom state machine through its AXI4-Lite interface (the next subsection provides more details on how the MicroBlaze is involved in this process). The moment the PR process is triggered the *busy* pin is pulled high and *addra* is increased by one. The output data provided by the BMG core (i.e. the *custom_icap_controller_rom*) are not valid during this clock cycle. For each of the next two clock cycles the custom state machine just increases the *addra* by one without sending data to the ICAP, since the output data from BRAM are still invalid. After that, the custom state machine reads a 32-bit configuration word from the *douta* port of the BMG core per clock cycle, and delivers it to the input port, *I*, of the ICAP by pulling the *CSIB* and *RDWRB* pins low to enable the ICAP and write operations to it. When the last 32-bit configuration word of the partial bitstream stored in BRAM is sent to the ICAP, the custom state machine also asserts the *ready* signal for one clock cycle, and then goes into rest mode. This whole process is illustrated by the flow diagram of the state machine in Figure 4.22.

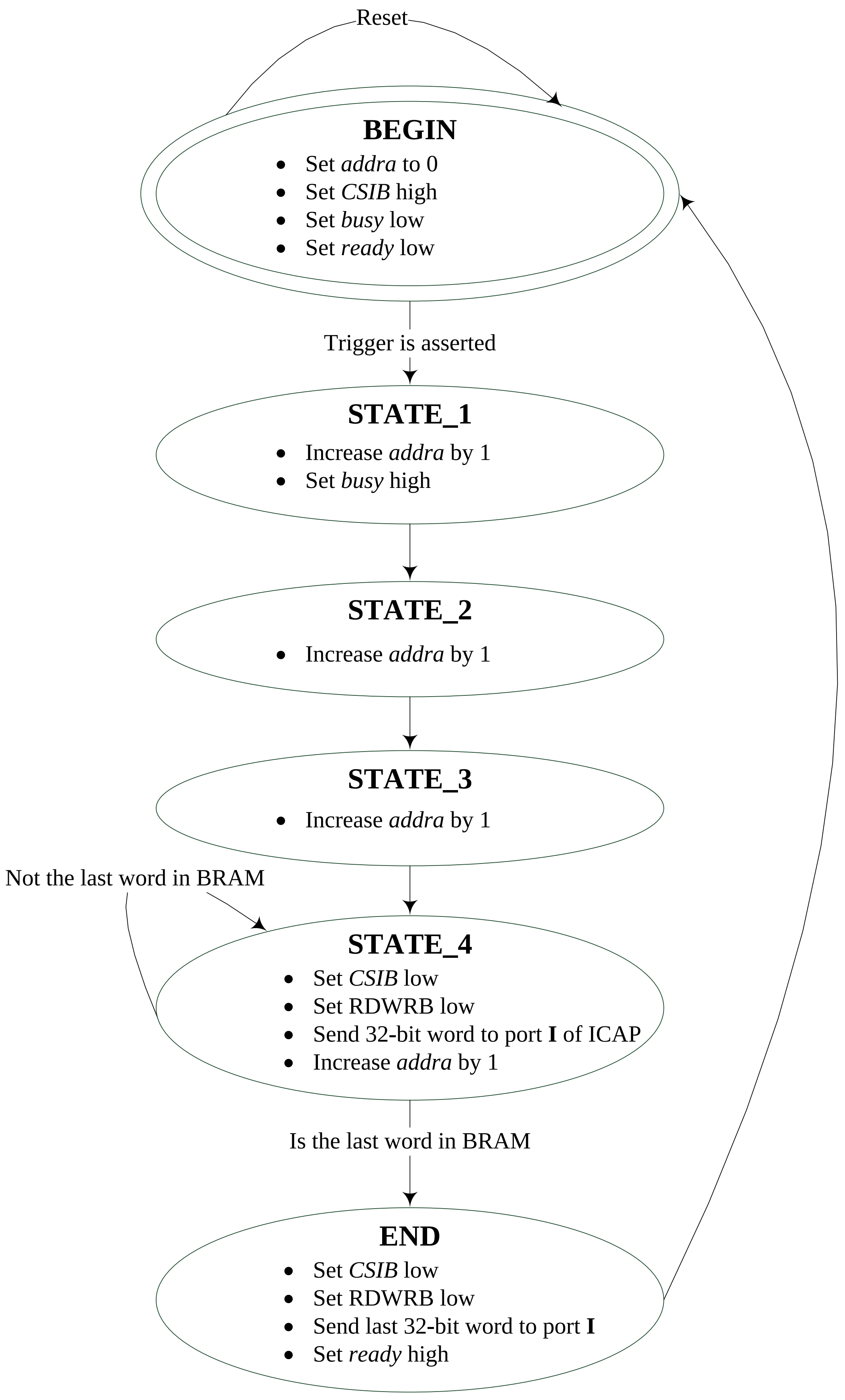


Figure 4.22: Partial reconfiguration state machine flow diagram.

It is essential to bear in mind that the bit-swap rule applies to the 7 series FPGA ICPAE2 interface. As shown in Figure 4.23, bit swapping is the swapping of the bits within a byte [14].

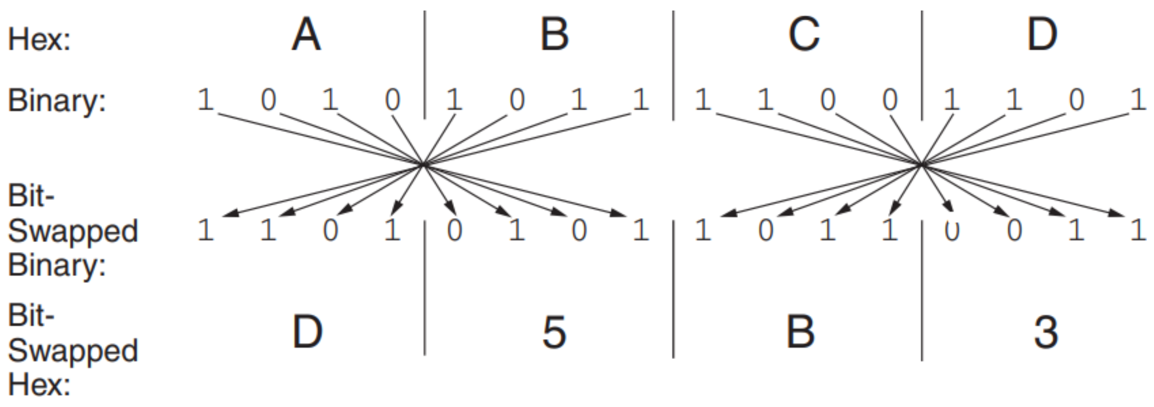


Figure 4.23: Bit swapping two bytes of data (0xABCD).

Since the BIN files generated by the Vivado tool are never bit swapped, the custom state machine should also undertake the task of swapping the bits within the bytes of each 32-bit configuration word in the partial bitstream. The following table shows an example of how the 0xAA995566 configuration word of the partial bitstream is bit swapped within each byte by the custom state machine, before it is sent to the port I of the ICAP.

	[31:24]	**[23:16]**	**[15:8]**	**[7:0]**
Bitstream Format	0xAA	0x99	0x55	0x66
Bit Swapped	0x55	0x99	0xAA	0x66

Table 4.4: Bit swap example of the 0xAA995566 configuration word.

4.3.9 Bare-Metal Application Development

The way in which the MicroBlaze soft processor controls the operation of its peripherals (i.e. the *axi_uartlite_0*, the *axi_timer_0*, and the *custom_icap_controller*) is determined by a bare-metal application developed and compiled for a MicroBlaze system using the Xilinx SDK 2016.3 tool. This SW application enables the user of a host PC (personal computer) to trigger the PR process using a serial terminal program (e.g. PuTTY). More specifically, using a standard A to mini-B USB cable, the host PC is able to communicate with the soft processor through the *axi_uartlite_0* IP core. Assuming that the host PC user has opened a serial connection at the relevant port and baud rate (i.e. 9600 bps) by using a serial terminal program, the SW app will print the following message to the terminal: “Would you like to change device’s configuration? [Y/n]”. If the user’s answer is ‘yes’, the application sends a trigger signal to the *custom_icap_controller* and the PR process begins. The SW app also sends a start signal to the *axi_timer_0* to measure the reconfiguration time. Finally, when the PR of our device ends, a new message is displayed on the terminal console informing the user that the device was successfully reconfigured and how long this process took. Since this application runs directly on hardware level without the support of any operating system (bare-metal), the processor's peripherals are accessed through register reads/writes.

4.4 Bitstream Generation

In order to generate the full and partial bitstreams for our design, the Xilinx PR workflow described in Section 3.2 was followed. However, since the partial bitstream corresponding to the reconfigurable logic must be stored in the *custom_icap_controller_rom* (see subsection 4.3.7) and thus be part of the static logic, we had to perform three iterations of this workflow:

1st Iteration

- **Step 1 –** Synthesize the static logic and the two RMs (i.e. *FRAME_GEN_0_v1* and *FRAME_GEN_0_v2*), separately:
  The respective design checkpoints were saved as *static_synth.dcp, RM1_synth.dcp* and *RM2_syth.dcp*. As regards the static logic, the first time we perform this step, there is no partial bitstream that we can store in the *custom_icap_controller_rom*. In fact, we cannot even know the required memory capacity for the (BRAM-based) ROM (the partial-bitstream size is directly related to the size of the RR defined in Step 3). In this context, the BMG core was initially configured as a stand-alone, non-initialized Single Port ROM, setting its capacity to the minimum possible, to reduce the synthesis time of the static logic.
- **Step 2 –** Open the *static_synth.dcp* in a Vivado project and read the *RM1_synth.dcp* in order to add the *FRAME_GEN_0_v1* RM to the static design. In addition, set the HD.RECONFIGURABLE property on the Reconfigurable Partition of our design, and impose all constraints contained in the *gtz_raw_data_exdes.xdc* and *v7ht.tcl* files.
- **Step 3 –** Impose physical constraints to define the Reconfigurable Region (Floorplanning):
  The first time we perform this step, we create these constraints by drawing a Pblock in the device window of Vivado project. In order to be able to impose exactly the same constraints for the next two times we will perform this step, we saved them as *fplan.xdc*.

- **Step 4 –** Implement the complete design and save a design checkpoint for the full routed design as *RM1_impl.dcp*.
- **Step 5 –** Remove the *FRAME_GEN_0_v1* RM from the full routed design, lock the static placement and routing, and save a static-only design checkpoint as *static_only_impl.dcp*.
- **Step 6 –** Add the *FRAME_GEN_0_v2* RM to the static-only design by reading the *RM2_syth.dcp* and implement this new configuration, saving a checkpoint for the full routed design as *RM2_impl.dcp*.
- **Step 7 –** Run the verification utility (pr_verify) on the two configurations (i.e. *RM1_impl.dcp* and *RM2_impl.dcp*) and create (full and partial) bitstreams for each of them.

2nd Iteration

- **Step 1 –** Synthesize the static logic and the two RMs (i.e. *FRAME_GEN_0_v1* and *FRAME_GEN_0_v2*), separately:
  Knowing the size of a partial bitstream and provided that the physical constraints associated with the RR definition never change (see Step 3), the second time we performed this step, the *custom_icap_controller_rom* was reconfigured in order to set its capacity to the bitstream's size. This change in the ROM's capacity, causes changes to the static logic that also affect the reconfigurable one. As a result, the partial bitstreams generated from the first iteration do not correspond to the new design, and therefore they should not be used to initialize the *custom_icap_controller_rom*.
- **Step 2 –** Open the *static_synth.dcp* in a Vivado project and read the *RM1_synth.dcp* in order to add the *FRAME_GEN_0_v1* RM to the static design. In addition, set the HD.RECONFIGURABLE property on the Reconfigurable Partition of our design, and impose all constraints contained in the *gtz_raw_data_exdes.xdc* and *v7ht.tcl* files.
- **Step 3 –** Impose physical constraints to define the Reconfigurable Region (Floorplanning):
  Impose the physical constraints contained in the *fplan.xdc* file (created the first time this step was performed) to ensure that the partial bitstreams that will be generated in the seventh step of this iteration, will be of exactly the same size as the ones generated the first time we performed the Step 7. Otherwise, we may need to start again from the beginning by redefining the *custom_icap_controller_rom* capacity.
- **Step 4 –** Implement the complete design and save a design checkpoint for the full routed design as *RM1_impl.dcp*.

- **Step 5 –** Remove the *FRAME_GEN_0_v1* RM from the full routed design, lock the static placement and routing, and save a static-only design checkpoint as *static_only_impl.dcp*.
- **Step 6 –** Add the *FRAME_GEN_0_v2* RM to the static-only design by reading the *RM2_syth.dcp* and implement this new configuration, saving a checkpoint for the full routed design as *RM2_impl.dcp*.
- **Step 7 –** Run the verification utility (pr_verify) on the two configurations (i.e. *RM1_impl.dcp* and *RM2_impl.dcp*) and create (full and partial) bitstreams for each of them:

  More specifically, one full and a partial bitstream were stored in both BIT and BIN file formats for each implementation, as shown in the table below.

	RM1_impl.dcp	**RM2_impl.dcp**
Full Bitstreams	RM1.bit	RM2.bit
	RM1.bin	RM2.bin
Partial Bitstreams	RM1_partial.bit	RM2_partial.bit
	RM1_partial.bin	RM2_partial.bin

Table 4.5: Full and partial bitstreams for each implementation.

3rd Iteration

- **Step 1 –** Synthesize the static logic and the two RMs (i.e. *FRAME_GEN_0_v1* and *FRAME_GEN_0_v2*), separately:

  The third (and last) time this step is performed, the *custom_icap_controller_rom* is reconfigured in order to be initialized with the contents of the previously created file, *RM2_partial.bin*. Given that the physical constraints associated with the RR definition never change, this slight change in the ROM's content does not affect the reconfigurable logic. As a result, the partial bitstreams that will be generated in the seventh step of this iteration, will be identical with the ones generated the second time we performed the Step 7.
- **Step 2 –** Open the *static_synth.dcp* in a Vivado project and read the *RM1_synth.dcp* in order to add the *FRAME_GEN_0_v1* RM to the static design. In addition, set the HD.RECONFIGURABLE property on the Reconfigurable Partition of our design, and impose all constraints contained in the *gtz_raw_data_exdes.xdc* and *v7ht.tcl* files.
- **Step 3 –** Impose physical constraints to define the Reconfigurable Region (Floorplanning):

Impose the physical constraints contained in the *fplan.xdc* file created the first time this step was performed.

- **Step 4 –** Implement the complete design and save a design checkpoint for the full routed design as *RM1_impl.dcp*.
- **Step 5 –** Remove the *FRAME_GEN_0_v1* RM from the full routed design, lock the static placement and routing, and save a static-only design checkpoint as *static_only_impl.dcp*.
- **Step 6 –** Add the *FRAME_GEN_0_v2* RM to the static-only design by reading the *RM2_syth.dcp* and implement this new configuration, saving a checkpoint for the full routed design as *RM2_impl.dcp*.
- **Step 7 –** Run the verification utility (pr_verify) on the two configurations (i.e. *RM1_impl.dcp* and *RM2_impl.dcp*) and create (full and partial) bitstreams for each of them.

4.5 Experimental Setup

This section provides a procedure for setting up the Virtex®-7 FPGA VC7222 GTH and GTZ Transceiver Characterization Board to test and demonstrate our design on real hardware.

1. Make sure the following modules are installed on the VC7222 board:
    - The *7 series GTZ transceiver power module* that supplies MGTZAVCC, MGTZVCCL and MGTZVCCH voltages to the FPGA GTZ transceivers.
    - The *SuperClock-2 module* (Figure 4.24) that provides programmable, LVDS (low-voltage differential signaling) clock outputs for the GTZ transceivers reference clock (in our case the frequency is preset to 255.00 MHz).

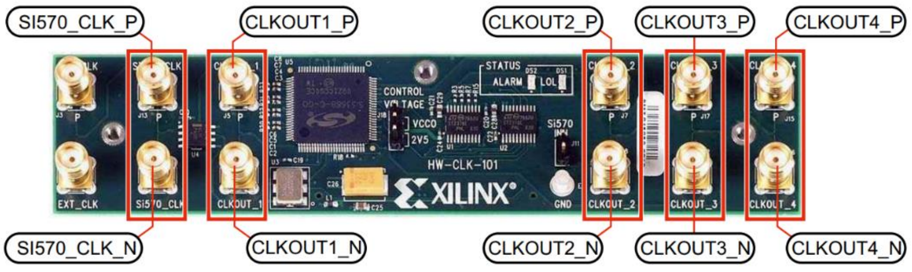


Figure 4.24: SuperClock-2 Module Output Clock SMA Locations [27].

2. Connect the GTZ reference clock CLK0 (see Figure 4.13) to the SuperClock-2 module as follows (note that the GTZ reference clock CLK1 can be left disconnected):
    - REFCLK0_P → SMA connector → CLKOUT1_P (on SuperClock-2 module)

- REFCLK0_N → SMA connector → CLKOUT1_N (on SuperClock-2 module)

3. Attach a Samtec BullsEye cable to each of the GTZ Quads, i.e. the Q300A and the Q300B (see Figure 4.13) and connect the transmit and receive cables as follows (see Figure 4.14 to identify the location of the P and N pins of the GTZ transmitters and receivers lanes):
    - Screw down a 50Ω SMA terminator onto each of the sixteen receive cables (RXi_P and RXi_N, $0 \leq i \leq 7$), since we do not intend to use the GTZ receivers.
    - Screw down a 50Ω SMA terminator onto one of the two transmit cables (either the P or the N) of each transmitter lane. The eight transmit cables left (one per transmitter), will be connected to a powerful and high-performance scope to monitor the GTZ transmitters output data.
4. Connect a host computer to the VC7222 board using a standard-A plug to micro-B plug USB cable. The standard-A plug connects to a USB port on the host computer and the micro-B plug connects to the Digilent USB JTAG configuration port on the VC7222 board.
5. With the board powered ON, start Vivado Design Suite on the host computer and use the Vivado Hardware Manager to detect and open our hardware target (i.e. the XC7VH580T device) in a new Hardware Manager window.
6. Configure the SuperClock-2 module so that it will provide a 255 MHz frequency clock source. For this purpose, the *setup_scm2_225_00_GTZ.tcl* script provided by Xilinx in the context of VC7222 IBERT Getting Started Guide (UG971) [27] can be used. More specifically, click **Tools > Run Tcl Script** (in the Vivado window), navigate to the *setup_scm2_225_00_GTZ.tcl* (in the Run Script window that pops up) and click **OK**.
7. Configure the FPGA by downloading the *RM1.bit* to the XC7VH580T device through the (always available) JTAG interface, with the help of Vivado Hardware Manager. Note that the *RM1.bit* is the full configuration file generated the 3rd time we performed the 7th step of the bitstream generation process described in the previous section (Section 4.4). At this point we will be able to monitor the data generated by the frame generators on scope's display.
8. Enable communication between the host computer and the MicroBlaze soft processor through the serial port:
    - Connect the host computer to the VC7222 board using a standard-A plug to mini-B plug USB cable. The standard-A plug connects to a USB port on the host computer and the mini-B plug connects to the board's connector providing the USB-to-UART bridge.

- Open a serial terminal application (e.g. PuTTY) and set the terminal configuration to the COM port where the Silicon Laboratories USB-UART bridge is connected. Set the baud rate to 9600, data to 8 bits, no parity or flow control, and 1 stop bit with no delays.

9. Launch the bare-metal application described in subsection 4.3.9 on the MicroBlaze. A message will be displayed on the terminal console asking for user input.
10. Trigger the PR process and notice that the GT_0 output signal displayed on the scope changes.

5 *Conclusions and future directions*

5.1 Summary

In the context of this thesis, we had the opportunity to explore the Partial Reconfiguration technology on FPGAs and apply the knowledge acquired to implement a high-bandwidth telecom system on a Virtex-7 H580T device. This device is not just an ordinary, monolithic FPGA; it is actually the result of combining two FPGA dices and an 8-channel 28Gbps transceiver die (GTZ transceivers) into a single package using Xilinx's SSI technology.

Our high-bandwidth, reconfigurable system makes use of all eight GTZ transceiver channels available on the Virtex®-7 H580T device. Each GTZ channel was configured to transmit raw data (that is, non-encoded data) at the maximum possible line rate of 28.05 Gbps. The transmission data were generated and provided to each channel by a (dedicated to the channel) hardware module called frame generator, and they were monitored (in a real-world environment) using a special, high-performance scope.

The logic implemented by the first frame generator (that is, the one corresponding to the first transceiver of the GTZ transceiver octal) was defined as reconfigurable, so that it can be dynamically modified through the Partial Reconfiguration of our device. To this end, two hardware modules were constructed as different flavors of the first frame generator, and also a third, "dummy" hardware module containing only definitions of the signals seen outside the two aforementioned generators and no additional logic. The "dummy" module was necessary to synthesize the static design and was used as a black-box in which the different flavors of the first frame generator were loaded prior the implementation of the various configurations. In addition, we implemented all this logic needed to isolate outcoming signals from the Reconfigurable Partitions to the rest of the system during Partial Reconfiguration and reset the Reconfigurable Modules to a known initial state after reconfiguring was done.

In the direction of improving our system's reconfiguration throughput, five different reconfigurable architectures were tested and compared in terms of reconfiguration time. One of the things we primarily had to do towards reducing the reconfiguration time, was to make use of the high-speed embedded memory resources in our target-device (i.e. the XC7VH580T) by storing the partial bitstream configuration data in the (on-chip) Block RAM. With our efforts being focused on BRAM-based architectures, we realized that the reconfiguration speed was mainly limited by the buses used to connect the configuration controller (i.e. one of the ready-to-use ICAP controllers provided by Xilinx) to the (BRAM-based) memory where the partial bitstream was stored. The only way to overcome this limitation, is to create a custom ICAP controller which could directly connect the (BRAM-based) memory output port to the ICAP input port. In the light of the above, we developed from scratch in RTL a hardware module to implement a custom state machine through which the partial bitstream stored in the Block RAM is delivered to the ICAP write port. By comparing the throughput of this architecture to the maximum theoretical throughput of the ICAP it was established that it allows us to fully exploit the ICAP's high throughput capabilities. However, being BRAM-based, the main disadvantage of this architecture is the limited amount of BRAM available to store configuration data.

Finally, to further enhance our reconfigurable system, a microprocessor was implemented entirely within the device general-purpose memory and logic fabric, by using Xilinx's soft IP core, MicroBlaze. The inclusion of a microprocessor in the hardware design facilitates the management of our system from the outside world (e.g. easy and flexible communication with a host-computer) enabling us to remotely control the Partial Reconfiguration of our device. The result is a hybrid hardware-software system where the functionality that will be optimally implemented in the hardware can be easily defined by the software (the "hardware-enabled, software-defined" concept).

5.2 Final Thoughts and Future Work

During this work, we encountered many difficulties in our efforts to:

- use the high performance GTZ transceivers and put them under our full control in a real-world environment.
- improve our system's reconfiguration throughput by exploring the unique and not so known nature of a heterogeneous, SSI device and constructing a custom configuration controller.
- implement a MicroBlaze based hardware design, creating, essentially, a hybrid device which is both software and hardware programmable.

Having overcome all these challenges, we, eventually, built a reconfigurable system whose sole intention was to demonstrate the "proof of concept". However, during the development of this thesis, we had in mind the greater picture of the technologies used. Based on the acquired knowledge, we could implement a more sophisticated system for it to be able to operate in a real-world scenario. We could extend, for instance, the functionality of our high-bandwidth telecom system in order to implement a hardware accelerated Software-Defined Radio (SDR) system. SDR is a communication system where components that are traditionally implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, encoders/decoders etc.) are instead implemented by means of software. This gives the ability to a system to change transmission protocols and a single device can be used in a wide variety of communication schemes. However, performing computationally-intensive signal processing algorithms in software is inefficient or even practically impossible in the case of a telecom system that needs to operate reliably at extremely high rates (just like the one proposed here). On the contrary, our reconfigurable system could function as an ideal SDR that implements efficiently all these tasks in hardware.

Another interesting endeavor, would be to further increase the reconfiguration throughput of our system by trying to overclock the ICAP. It has been shown in the literature ([28], [29]) that by adding custom hardware to control the ICAP (just like in our case), it is possible to clock the ICAP above the Xilinx-recommended 100 MHz. Although, none of these projects deals with a Virtex-7, heterogeneous SSI device, it is quite possible that we will be able to clock the ICAP above 200 MHz, since the reconfigurable architecture proposed in this thesis is based on a custom ICAP controller and BRAM (the maximum clock frequency of which, according to the Xilinx documentation [18], is 450 MHz). So, to investigate the maximum clock frequency of the ICAP in the proposed architecture, we should gradually increase the frequency at which our RA’s components are clocked, until the reconfiguration process fails. If our predictions hold true, then the (already short) reconfiguration time could be reduced by at least half.

References

[1] E. Mohsen, “Designing Nx100G Applications with Heterogeneous 3D FPGAs,” p. 9, 2013.

[2] H. Fu, “Leveraging 7 Series FPGA Transceivers for High-Speed Serial I/O Connectivity,” p. 16, 2013.

[3] S. Ahmadi, “Xilinx Solutions and Enablers for Next-Generation Wireless Systems,” p. 32, 2016.

[4] J. Rajewski, *Learning FPGAs: Digital Design for Beginners with Mojo and Lucid HDL*. O’Reilly Media, Inc., 2017.

[5] P. Sundararajan, “High Performance Computing Using FPGAs,” p. 15, 2010.

[6] “FPGA-based microelectronics in Space Avionics,” *Blog GMV*, Jun. 04, 2019. https://www.gmv.com/blog_gmv/language/en/the-silicon-throne-fpga-based-microelectronics-in-space-avionics/ (accessed Jul. 29, 2019).

[7] “Urban Dictionary: RTL,” *Urban Dictionary*. https://www.urbandictionary.com/define.php?term=RTL.

[8] “Vivado Design Suite User Guide: Implementation,” p. 192, 2019.

[9] C. Maxfield, *The design warrior’s guide to FPGAs: devices, tools, and flows*. Boston: Newnes/Elsevier, 2004.

[10] “SDAccel Environment User Guide,” p. 158, 2019.

[11] “Xilinx Virtex-7 FPGA VC7222 Characterization Kit.” https://www.xilinx.com/products/boards-and-kits/ck-v7-vc7222-g.html#hardware (accessed Aug. 24, 2019).

[12] “Virtex-7 FPGA VC7222 GTH and GTZ Transceiver Characterization Board User Guide (UG965),” p. 64, 2015.

[13] “Vivado Design Suite User Guide: Partial Reconfiguration (UG909),” p. 151, 2019.

[14] “7 Series FPGAs Configuration User Guide (UG470),” p. 180, 2018.

[15] “Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide,” p. 604, 2019.

[16] “Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics,” p. 78, 2019.

[17] “Vivado Design Suite Tcl Command Reference Guide,” p. 1872, 2019.

[18] “Block Memory Generator v8.4 LogiCORE IP Product Guide,” p. 129, 2019.

[19] “Local Memory Bus (LMB) v3.0 LogiCORE IP Product Guide (PG113),” p. 20, 2016.

[20] S. V. Kartalopoulos, *Free Space Optical Networks for Ultra-Broad Band Services*. John Wiley & Sons, 2011.

[21] "7 Series FPGAs Transceivers Wizard v3.6 LogiCORE IP Product Guide," p. 147, 2016.

[22] "MicroBlaze Processor Reference Guide," p. 390, 2019.

[23] "Processor System Reset Module v5.0 LogiCORE IP Product Guide (PG164)," p. 33, 2015.

[24] "AXI Interconnect v2.1 LogiCORE IP Product Guide (PG059)," p. 171, 2017.

[25] "AXI Timer v2.0 LogiCORE IP Product Guide (PG079)," p. 37, 2016.

[26] "AXI UART Lite v2.0 LogiCORE IP Product Guide (PG142)," p. 26, 2017.

[27] "Virtex-7 FPGA VC7222 Characterization Kit IBERT Getting Started Guide (UG971)," p. 73, 2015.

[28] S. G. Hansen, D. Koch and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," *2011 IEEE Int. Symp. Parallel Distrib. Process. Workshop Phd Forum*, 2011, doi: 10.1109/IPDPS.2011.139.

[29] J. C. Hoffman and M. S. Pattichis, "A High-Speed Dynamic Partial Reconfiguration Controller Using Direct Memory Access Through a Multiport Memory Controller and Overclocking with Active Feedback," *Int. J. Reconfigurable Comput.*, vol. 2011, p. 439072, Aug. 2011, doi: 10.1155/2011/439072.