ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Σύνδεση Εικονικής και Πραγματικής Πλατφόρμας Υλοποίησης Αρχιτεκτονικών Συστήματος σε Ψηφίδα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Χ. Μπρισκόλας

Επιβλέπων : Γεώργιος Οικονομάκος

Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Σύνδεση Εικονικής και Πραγματικής Πλατφόρμας Υλοποίησης Αρχιτεκτονικών Συστήματος σε Ψηφίδα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Χ. Μπρισκόλας

Επιβλέπων : Γιώργος Οικονομάκος
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Ιουλίου 2013.

.....	.....	.....
Γιώργος Οικονομάκος Επ. Καθηγητής Ε.Μ.Π.	Κιαμάλ Πεκμεστζή Καθηγητής Ε.Μ.Π.	Δημήτρης Σούντρης Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

.....

Γεώργιος Χ. Μπρισκόλας

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

Περίληψη

Σκοπός της διπλωματικής εργασίας καθίσταται η σύνδεση μιας εικονικής και μιας πραγματικής πλατφόρμας υλοποίησης αρχιτεκτονικών συστήματος σε ψηφίδα. Αντικείμενο της σύνδεσης αυτής είναι ο σχεδιασμός περιφερειακών μονάδων τόσο στην εικονική όσο και στην πραγματική πλατφόρμα μέσω εργαλείων Σύνθεσης Υψηλού Επιπέδου για επιτάχυνση της διαδικασίας. Αρχικά, η σχεδίαση της εικονικής πλατφόρμας πραγματοποιήθηκε στο περιβάλλον εικονικής προσομοίωσης Open Virtual Platforms (OVP). Αφού επιλέχθηκε η εφαρμογή που θέλουμε να έχει το ενσωματωμένο σύστημα δημιουργήσαμε ένα περιφερειακό, στο οποίο τοποθετήθηκαν οι πιο πολύπλοκοι υπολογισμοί αυτής της εφαρμογής ώστε να υλοποιηθούν σε hardware για μεγαλύτερη ταχύτητα εκτέλεσης. Στην συνέχεια, χρησιμοποιήσαμε αυτό το περιφερειακό και μετά από Σύνθεση υψηλού επιπέδου (High Level Synthesis), με τη βοήθεια του εργαλείου της Xilinx VivadoHLS, μετατρέψαμε άμεσα και με βέλτιστη υλοποίηση τον κώδικα του από C σε Γλώσσα Περιγραφής Υλικού (HDL). Μετά από αυτό το βήμα, σχεδιάσαμε την πραγματική πλατφόρμα χρησιμοποιώντας την ομάδα εργαλείων του Embedded Development Kit (EDK) και του PlanAHead της Xilinx και να προσθέσουμε το περιφερειακό που παράχθηκε από το VivadoHLS στο σύστημα. Με αυτόν τον τρόπο το περιφερειακό που σχεδιάσαμε στο εικονικό περιβάλλον, το χρησιμοποιούμε και στο πραγματικό με τη σύνδεση εικονικής και πραγματικής πλατφόρμας να πραγματοποιείται επιτυχώς.

Η διπλωματική εργασία χωρίζεται σε έξη κεφάλαια. Αρχικά, γίνεται μία εισαγωγή στα ενσωματωμένα συστήματα και μετά μία σύνδεσή αυτών με την διπλωματική εργασία. Στο πρώτο κεφάλαιο γίνεται μία περιγραφή των FPGAs και εν συνεχεία αναλύονται το Virtex-7, που είναι το FPGA που χρησιμοποιούμε, και ο επεξεργαστής MicroBlaze, ο οποίος αποτελεί τον επεξεργαστή που θα έχουν η εικονική αλλά και η πραγματική πλατφόρμα. Στο δεύτερο κεφάλαιο περιγράφεται αναλυτικά το περιβάλλον εικονικής προσομοίωσης Open Virtual Platforms και παρουσιάζεται λεπτομερώς η διαδικασία σχεδίασης της εικονικής πλατφόρμας . Στο κεφάλαιο τρία βρίσκεται η περιγραφή του εργαλείου VivadoHLS της Xilinx και τα βήματα που ακολουθήθηκαν για τη μετατροπή του κώδικα του περιφερειακού σε Γλώσσα Περιγραφής Υλικού. Επίσης, η διαδικασία σχεδίασης της πραγματικής πλατφόρμας ξεκίνησε στο κεφάλαιο τέσσερα με το εργαλείο Planahead και συνεχίστηκε στο πέμπτο κεφάλαιο με τα εργαλεία σχεδίασης του EDK. Τέλος, στο έκτο κεφάλαιο συνοψίζονται και σχολιάζονται τα αποτελέσματα που προέκυψαν, καθώς επίσης παρατίθενται κάποιες προτάσεις για μελλοντική εργασία.

Λέξεις Κλειδιά

Ενσωματωμένα συστήματα, MicroBlaze, OvpWorld, Vivado HLS, EDK, SDK, XPS, PlanAhead, Virtex-7, FPGA, Εικονική Πλατφόρμα, Πραγματική Πλατφόρμα.

Abstract

The purpose of the present diploma thesis is to integrate Implementation and Virtual Prototyping SoC Architecture Platforms. The way of achieving this integration is to design peripherals for both virtual and real platform by using High Level Synthesis tools for speeding up the process. Initially, the design of virtual platform held in a virtual simulation environment called Open Virtual Platforms (OVP). Having chosen the application we wanted to integrate in the embedded system, we created a peripheral, which performed the most complicated and time consuming functions of this application. Integrating those function as hardware we achieved more speed in application execution. After using the Xilinx tool, VivadoHLS, for High Level Synthesis we converted peripheral 's C code into hardware description language (HDL) extremely quick and efficienty. Moreover, we designed a real platform with that peripheral by using the appropriate set of Xilinx tools like Embedded Development Kit (EDK) and PlanAHead. As a result the peripheral we designed in the virtual environment has been used in the real platform which means that both virtual and real platform have been connected successfully.

The diploma thesis is divided into six chapters. To begin with, the introduction refers to the embedded systems and the linking to my diploma thesis. The first chapter contains a description of FPGAs in generall, Virtex-7, which is the FPGA we use in the real platform, and MicroBlaze processor, which is the processor of both the virtual and the real platform. The second chapter describes in detail the virtual environment simulation Open Virtual Platforms and the design process of a virtual platform. Chapter three is a description of the tool VivadoHLS of Xilinx and the steps followed to convert peripheral's C code into hardware description language. The detailed design process of the real platform starts by using the Xilinx Planahead tool in chapter four and continued by the use of EDK tools in the fifth chapter. Finally, the sixth chapter not only summarizes and comments the results obtained but also provides some suggestions for future work.

KeyWords

Embedded systems, MicroBlaze, OvpWorld, Vivado HLS, EDK, SDK, XPS, PlanAhead, Virtex-7, FPGA, virtual platform, real platform

Ευχαριστίες

Για την εκπόνηση της παρούσας διπλωματικής εργασίας θα ήθελα να ευχαριστήσω κατά κύριο λόγο τον επίκουρο καθηγητή κ. Γεώργιο Οικονομάκο, για τις συμβουλές και τις ιδέες τους σε όλη την διάρκεια της εργασίας. Επίσης, θα ήθελα να ευχαριστήσω όλα τα μέλη του Εργαστηρίου Μικροϋπολογιστών και Ψηφιακών Συστημάτων, και ιδιαίτερα τον Ευστάθιο Σωτηρίου- Ξανθόπουλο, του οποίου η συμβολή υπήρξε καθοριστική για την ολοκλήρωση της εργασίας. Τέλος θεωρώ χρέος μου να ευχαριστήσω την οικογένειά μου και το φιλικό μου περιβάλλον, καθώς στάθηκαν δίπλα μου, με στήριξαν, και συνέβαλαν τα μέγιστα τόσο στην ολοκλήρωση της εργασίας όσο και καθ' όλη τη διάρκεια των σπουδών μου.

Πίνακας Περιεχομένων

Κεφάλαιο 2: Open Virtual Platforms

Κεφάλαιο 3: Vivado HLS

Κεφάλαιο 4: PlanAhead

Κεφάλαιο 5: Embedded Development Kit

Κεφάλαιο 6: Σύνοψη, Συμπεράσματα και Μελλοντική Εργασία

Ευρετήριο Σχημάτων

Εισαγωγή

Κεφάλαιο 1

Κεφάλαιο 2

Κεφάλαιο 3

Κεφάλαιο 4

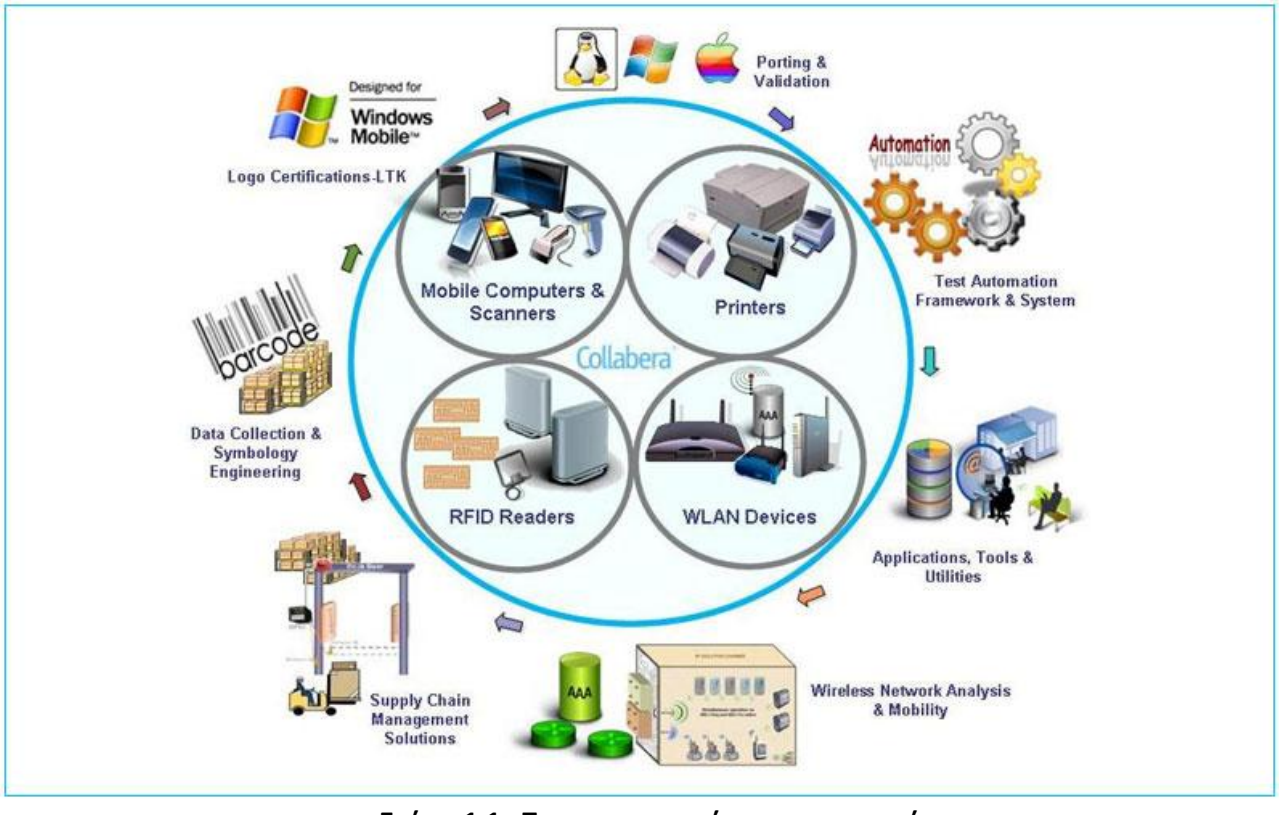
Κεφάλαιο 5

Κεφάλαιο 6

Εισαγωγή

Σύγχρονα Ενσωματωμένα Συστήματα και FPGAs

Από τα smart phones μέχρι τον ιατρικό εξοπλισμό και από τους φούρνους μικροκυμάτων μέχρι τους "εγκεφάλους" των σημερινών αυτοκινήτων, τα σύγχρονα project ενσωματωμένων συστημάτων αναπτύσσουν υπολογιστικές μηχανές που αποτελούν αναπόσπαστο κομμάτι της κοινωνίας μας. Για την ανάπτυξη αυτών των προϊόντων, ηλεκτρολόγοι μηχανικοί επιστρατεύουν μια γκάμα εργαλείων και τεχνολογιών για την υλοποίηση ενσωματωμένων συστημάτων από hardware και software στοιχεία. Ένα τέτοιο στοιχείο, ο επί τόπου προγραμματιζόμενος πίνακας πυλών (Field-Programmable Gate Array, FPGA) χρησιμοποιείται σήμερα όλο και περισσότερο. Άτυπα, ένα FPGA θα μπορούσε να θεωρηθεί ως μια "κενή κατάσταση" (blank state), στην οποία μπορεί να υλοποιηθεί κάθε ψηφιακό κύκλωμα. Εν συντομία, το FPGA παρέχει στον σχεδιαστή ενσωματωμένων συστημάτων, ένα "προγραμματιζόμενο υλικό".



Σχήμα 1.1 : Τα ενσωματωμένα στην κοινωνία

Ο ρόλος των συσκευών FPGA έχει εξελιχθεί με την πάροδο των χρόνων. Αρχικά, χρησιμοποιήθηκαν με σκοπό να αντικαταστήσουν κάποιες συσκευές ολοκληρωμένων κυκλωμάτων μικρής και μεσαίας κλίμακας, με μία μόνο συσκευή FPGA. Με τις γνωστές βελτιώσεις και εξελίξεις στην τεχνολογία των ημιαγωγών, ο αριθμός των τρανζίστορ ανά IC-chip αυξήθηκε εκθετικά. Αυτό ήταν μια θετική εξέλιξη για τις συσκευές FPGA, οι οποίες αύξησαν δραματικά τόσο την χωρητικότητα προγραμματιζόμενης λογικής, όσο και τη λειτουργική τους ικανότητα. Με τον όρο χωρητικότητα, αναφερόμαστε στον αριθμό των διαθέσιμων λογικών πυλών, ενώ με τον όρο ικανότητα εννοούμε μια ποικιλία σταθερών, ειδικού σκοπού blocks.

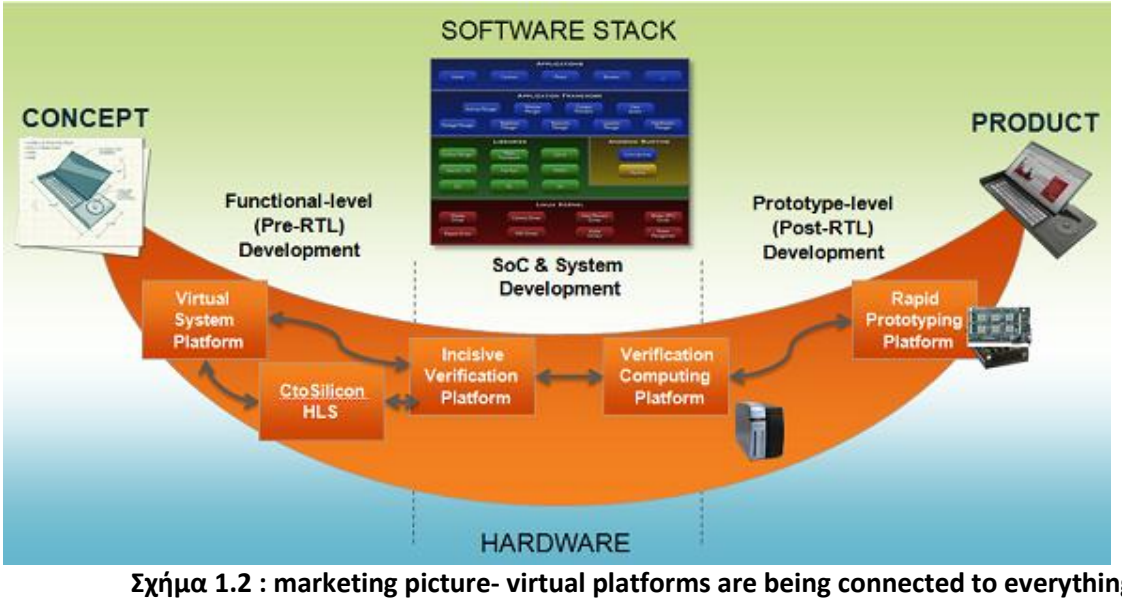
Ως αποτέλεσμα αυτής της ανάπτυξης, τα σύγχρονα FPGA είναι σε θέση να υποστηρίξουν επεξεργαστές, διαύλους, ελεγκτές μνήμης και διεπαφές δικτύου, καθώς και έναν όλο και αυξανόμενο αριθμό κοινών περιφερειακών, όλα σε μία μόνο συσκευή. Με την προσθήκη ενός σύγχρονου λειτουργικού συστήματος, όπως τα Linux, τα FPGA αρχίζουν να εμφανίζονται όλο και περισσότερο σαν ένας σταθερός προσωπικός υπολογιστής όσον αφορά τη χρηστικότητα και την λειτουργικότητα.

Γενικά, τα FPGA προσφέρουν τεράστιες δυνατότητες και προοπτικές στους σχεδιαστές ενσωματωμένων συστημάτων. Πέρα από το ότι απλά μειώνουν των αριθμό των τσιπ που χρησιμοποιούνται σε μια υλοποίηση, προσφέρουν στον σχεδιαστή τεράστια ευελιξία. Όπως αποδεικνύεται, η ευελιξία είναι εξαιρετικά πολύτιμη κατά των σχεδιασμό συστημάτων ώστε να ανταποκριθούν στις σύνθετες και απαιτητικές ανάγκες των ενσωματωμένων υπολογιστικών συστημάτων. Συχνά, αυτή η ευελιξία είναι που κάνει τα FPGA τόσο ελκυστικά σε σχέση με τις λύσεις με παραδοσιακούς μικροεπεξεργαστές, ASIC και άλλες System-on-a-Chip (SoC) λύσεις.

Όμως μαζί με τα πλεονεκτήματα της τεχνολογίας των FPGA εμφανίζεται και μια σειρά από νέες προκλήσεις. Παλαιότερα, ένα από τα κυριότερα διλήμματα για τον σχεδιαστή ενσωματωμένων συστημάτων, ήταν η επιλογή του κατάλληλου μικροεπεξεργαστή. Η επιλογή αυτή ήταν που καθόριζε σε μεγάλο βαθμό την υπόλοιπη αρχιτεκτονική (ή τουλάχιστον περιόριζε το φάσμα των επιλογών σχεδιασμού). Με τις πλατφόρμες FPGA, οι σχεδιαστές συστημάτων είναι αναγκαίο να μπορούν να συνδυάσουν γνώσεις αρχιτεκτονικής υπολογιστών, ψηφιακών κυκλωμάτων και προγραμματισμού ώστε να κατορθώσουν τελικά να αξιοποιήσουν πλήρως τις δυνατότητες των συσκευών αυτών.

Σύνδεση με την εργασία

Όπως μπορεί κανείς εύκολα να καταλάβει τα ενσωματωμένα συστήματα παίζουν πολύ σημαντικό ρόλο στην ζωή του ανθρώπου. Συνεπώς, η ανάγκες για δημιουργία τέτοιων συστημάτων αυξάνονται με ραγδαίους ρυθμούς και ως αποτέλεσμα η σχεδίασή τους πρέπει να γίνεται ταχύτατα και ολοένα και περισσότερο αυτοματοποιημένα. Αυτή η αυτοματοποίηση λοιπόν, είναι ο βασικός στόχος της εργασίας. Ο συνηθισμένος τρόπος σχεδίασης ενσωματωμένων συστημάτων συμπεριλαμβάνει δύο ξεχωριστά μεταξύ τους στάδια: Την εικονική σχεδίαση και προσομοίωση του συστήματος, και την πραγματική του σχεδίαση. Αυτά τα δύο στάδια προσπαθήσαμε να συνδέσουμε και το πετύχαμε αυτό μέσω High Level Synthesis.



Σχήμα 1.2 : marketing picture- virtual platforms are being connected to everything

Έτσι λοιπόν, σκοπός της διπλωματικής εργασίας καθίσταται η σύνδεση μιας εικονικής και μιας πραγματικής πλατφόρμας υλοποίησης αρχιτεκτονικών συστήματος σε ψηφίδα. Αντικείμενο της σύνδεσης αυτής είναι ο σχεδιασμός περιφερειακών μονάδων τόσο στην εικονική όσο και στην πραγματική πλατφόρμα μέσω εργαλείων Σύνθεσης Υψηλού Επιπέδου για επιτάχυνση της διαδικασίας.

Αρχικά, η σχεδίαση της εικονικής πλατφόρμας πραγματοποιήθηκε στο περιβάλλον εικονικής προσομοίωσης Open Virtual Platforms (OVP). Αφού επιλέχθηκε η εφαρμογή που θέλουμε να έχει το ενσωματωμένο σύστημα δημιουργήσαμε ένα περιφερειακό, στο οποίο εκτελούνται οι υπολογισμοί αυτής της εφαρμογής και έτσι την υλοποιήσαμε σε hardware. Στην συνέχεια, χρησιμοποιήσαμε αυτό το περιφερειακό και μετά από Σύνθεση υψηλού επιπέδου (High Level Synthesis), με τη βοήθεια του εργαλείου της Xilinx VivadoHLS, μετατρέψαμε άμεσα και με βέλτιστη υλοποίηση τον κώδικα του από C σε Γλώσσα Περιγραφής Υλικού (HDL). Μετά από αυτό το βήμα, σχεδιάσαμε την πραγματική πλατφόρμα χρησιμοποιώντας την ομάδα εργαλείων του Embedded Development Kit (EDK) και του PlanAHead της Xilinx και να προσθέσουμε το περιφερειακό που παράχθηκε από το VivadoHLS στο σύστημα. Με αυτόν τον τρόπο το περιφερειακό που σχεδιάσαμε στο εικονικό περιβάλλον, το χρησιμοποιούμε και στο πραγματικό με τη σύνδεση εικονικής και πραγματικής πλατφόρμας να πραγματοποιείται επιτυχώς.

Όλα τα παραπάνω αναλύονται λεπτομερέστατα στα κεφάλαια που ακολουθούν.

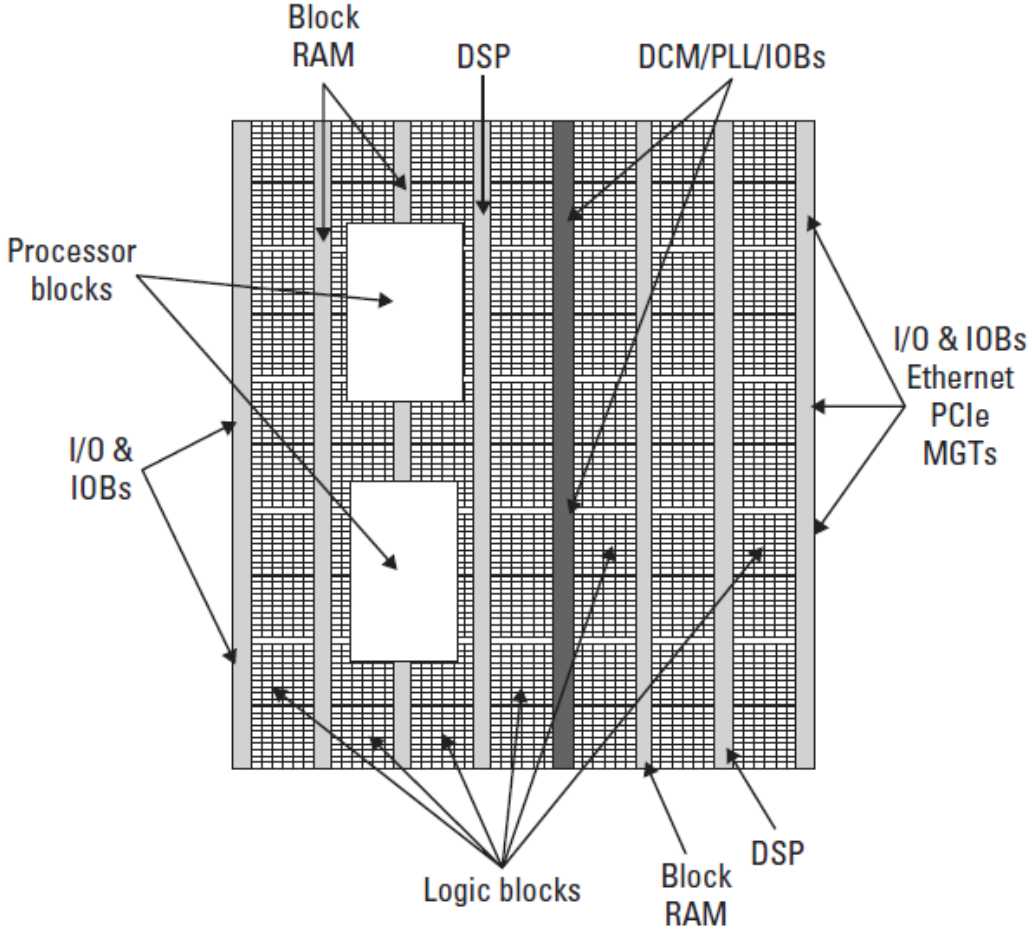
- Στο πρώτο κεφάλαιο γίνεται μία περιγραφή των FPGAs και εν συνεχεία αναλύονται το Virtex-7, που είναι το FPGA που χρησιμοποιούμε, και ο επεξεργαστής MicroBlaze, ο οποίος αποτελεί τον επεξεργαστή που θα έχουν η εικονική αλλά και η πραγματική πλατφόρμα.
- Στο δεύτερο κεφάλαιο περιγράφεται αναλυτικά το περιβάλλον εικονικής προσομοίωσης Open Virtual Platforms και παρουσιάζεται λεπτομερώς η διαδικασία σχεδίασης της εικονικής πλατφόρμας.
- Στο τρίτο κεφάλαιο βρίσκεται η περιγραφή του εργαλείου VivadoHLS της Xilinx και τα βήματα που ακολουθήθηκαν για τη μετατροπή του κώδικα του περιφερειακού σε Γλώσσα Περιγραφής Υλικού.
- Στο τέταρτο κεφάλαιο ξεκινάει η διαδικασία σχεδίασης της πραγματικής πλατφόρμας με χρήση του εργαλείου Planahead.
- Στο πέμπτο κεφάλαιο συνεχίζεται και ολοκληρώνεται η σχεδίαση της πραγματικής πλατφόρμας, αλλά με τα εργαλεία σχεδίασης του Embedded Development Kit (EDK).
- Στο έκτο κεφάλαιο συνοψίζονται και σχολιάζονται τα αποτελέσματα που προέκυψαν, καθώς επίσης παρατίθενται κάποιες προτάσεις για μελλοντική εργασία.

Κεφάλαιο 1

FPGA, Xilinx Virtex-7, MicroBlaze

1.1 Επί-Τόπου Προγραμματιζόμενος Πίνακας Πυλών (FPGA)

Ένα σύγχρονο FPGA (Σχήμα 1.3), αποτελείται από έναν δισδιάστατο πίνακα από προγραμματιζόμενα λογικά blocks (CLBs) , από blocks σταθερής λειτουργίας και πηγές δρομολόγησης υλοποιημένες στη τεχνολογία CMOS. Κατά μήκος της περιμέτρου του FPGA υπάρχουν ειδικά λογικά blocks συνδεδεμένα με εξωτερικές συσκευασίες ακροδεκτών I/O. Τα λογικά blocks αποτελούνται από πολλαπλά λογικά κελιά, ενώ τα λογικά κελιά περιέχουν γεννήτριες συναρτήσεων και αποθηκευτικά στοιχεία.



Σχήμα 1.3 - FPGA

1.1.1 Γεννήτριες Συναρτήσεων

Οι συσκευές FPGA χρησιμοποιούν γεννήτριες συναρτήσεων για την υλοποίηση της Boolean λογικής και όχι φυσικές πύλες σε αντίθεση με τις υπόλοιπες συσκευές προγραμματιζόμενης λογικής (PLAs, PALs, CPLDs). Για τις εισόδους μιας Boolean συνάρτησης δημιουργείται αρχικά ένας πίνακας αλήθειας. Για κάθε είσοδο ο πίνακας αλήθειας περιγράφει την τιμή της εξόδου της συνάρτησης. Κάθε bit της εξόδου της συνάρτησης αποθηκεύεται σε ξεχωριστό κελί μιας στατικής μνήμης. Τα κελιά αυτά συνδέονται ως είσοδοι σε έναν πολυπλέκτη όπου ανάλογα με την είσοδο επιλέγεται η κατάλληλη έξοδος. Το αποτέλεσμα είναι γνωστό ως **Look-Up Table** (LUT). Αντίθετα με τα ψηφιακά κυκλώματα που υλοποιούνται με λογικές πύλες, η καθυστέρηση διάδοσης στα LUT είναι σταθερή. Αυτό σημαίνει ότι ανεξάρτητα από την πολυπλοκότητα του Boolean κυκλώματος, εφόσον "χωράει" σε ένα LUT, η καθυστέρηση διάδοσης είναι η ίδια. Αυτό επίσης ισχύει και για κυκλώματα που εκτείνονται σε περισσότερα LUT, αλλά εδώ η καθυστέρηση διάδοσης εξαρτάται και από των αριθμό των LUT που χρησιμοποιούνται.

1.1.2 Στοιχεία Αποθήκευσης

Ενώ οι γεννήτριες συναρτήσεων αποτελούν το βασικό block για την υλοποίηση
συνδυαστικών κυκλωμάτων, τα FPGA διαθέτουν επιπρόσθετα στοιχεία παρέχοντας έτσι ένα πλούτο λειτουργιών. Όπως και στα block των PLA, D Flip-Flop ενσωματώνονται επίσης και στα FPGA. Τα Flip-Flops μπορούν να χρησιμοποιηθούν με διάφορους τρόπους, με πιο απλό, την αποθήκευση δεδομένων. Συνήθως, μια έξοδος του LUT συνδέεται με την είσοδο ενός Flip-Flop. Επίσης, το Flip-Flop μπορεί να χρησιμοποιηθεί και ως μανδαλωτής που λειτουργεί είτε σε θετική είτε σε αρνητική λογική.

1.1.3 Λογικά Κελιά

Συνδυάζοντας ένα LUT και ένα D Flip-Flop προκύπτει αυτό που αναφέρεται ως λογικό κελί. Τα λογικά κελιά αποτελούν το χαμηλότερου επιπέδου δομικό block σε ένα FPGA. Τόσο η συνδυαστική όσο και η ακολουθιακή λογική μπορούν να υλοποιηθούν από ένα λογικό κελί ή μια συλλογή από αυτά.

1.1.4 Λογικά Blocks

Ενώ τα λογικά κελιά μπορούν να θεωρηθούν ως τα βασικά δομικά blocks των εφαρμογών FPGA, στην πραγματικότητα είναι σύνηθες να ομαδοποιούμε μερικά λογικά κελιά σε blocks και να προσθέτουμε κυκλώματα ειδικού σκοπού με σκοπό να δημιουργήσουμε ένα λογικό block. Αυτό επιτρέπει σε μια ομάδα από λογικά κελιά, τα οποία βρίσκονται κοντά μεταξύ τους, να έχουν γρήγορα μονοπάτια επικοινωνίας, μειώνοντας έτσι την καθυστέρηση διάδοσης και βελτιώνοντας την υλοποίηση του σχεδίου. Για παράδειγμα, η οικογένεια Virtex-7 της Xilinx ομαδοποιεί τέσσερα λογικά κελιά σε ένα slice. Δυο slices και ένα carry-logic σχηματίζουν ένα προγραμματιζόμενο λογικό block (Configurable Logic Block, CLB). Τα λογικά κελιά, συνδέονται με ένα δίκτυο διασυνδέσεων ώστε να παρέχουν υποστήριξη για πιο πολύπλοκες υλοποιήσεις κυκλωμάτων. Αυτό το δίκτυο διασυνδέσεων αποτελείται από switch boxes. Ένα switch box χρησιμοποιείται για τη δρομολόγηση μεταξύ των εισόδων/εξόδων ενός λογικού κελιού με το γενικό δίκτυο δρομολόγησης του τσιπ. Είναι επίσης υπεύθυνο για τη διέλευση σημάτων από ένα τμήμα καλωδίωσης σε ένα άλλο. Τα τμήματα καλωδίωσης μπορεί να είναι είτε μικρά, συνδέοντας μόνο δυο λογικά κελιά είτε μεγάλα, διατρέχοντας όλη την επιφάνεια του τσιπ.

1.1.5 Blocks Εισόδου/Εξόδου

Τα blocks εισόδου/εξόδου (Input/Output Blocks – IOBs) , που βρίσκονται περιμετρικά του τσιπ, συνδέουν τον πινάκα των λογικών blocks και τις πηγές δρομολογήσεων με τους εξωτερικούς ακροδέκτες της συσκευής.

1.1.6 Blocks Ειδικού Σκοπού

Πολλά σχέδια απαιτούν την χρήση κάποιου ποσού της On-Chip μνήμης. Χρησιμοποιώντας λογικά κελιά είναι δυνατή η κατασκευή στοιχείων μνήμης διάφορων μεγεθών. Ωστόσο, όσο αυξάνονται οι απαιτήσεις για μνήμη τόσο μειώνονται οι διαθέσιμοι πόροι. Η λύση βρίσκεται στην ύπαρξη ενός σταθερού ποσού ενσωματωμένης on-chip μνήμης μέσα στο σώμα του FPGA που ονομάζεται **Block Ram** (BRAM). Τοπικές on-chip αποθηκευτικές μονάδες όπως μνήμες RAM, ROM ή Buffers μπορούν να κατασκευαστούν από BRAMs. Οι τελευταίες μπορούν να συνδυαστούν μεταξύ τους ώστε να σχηματίσουν μεγαλύτερες BRAMs. Είναι επίσης μνήμες dual-ported, επιτρέποντας την ανεξάρτητη ανάγνωση και εγγραφή από κάθε port συμπεριλαμβανομένων και των ανεξάρτητων ρολογιών. Πολλές συσκευές FPGA, θέλοντας να επιτρέπουν πιο σύνθετα σχέδια, συμπεριλαμβανομένης της ψηφιακής επεξεργασίας σήματος ή μιας γκάμας πολλαπλασιασμών, συμπεριλαμβάνουν στο σώμα τους Blocks Ψηφιακής Επεξεργασίας Σήματος (Digital Signal Processing Block, DSP). Όπως και με τις BRAMs, είναι δυνατό να υλοποιήσουμε αυτά τα στοιχεία χρησιμοποιώντας την προγραμματιζόμενη λογική, αλλά είναι πιο αποδοτικό από άποψη απόδοσης και κατανάλωσης ισχύος να ενσωματώσουμε πολλά από αυτά τα στοιχεία κατευθείαν στο σώμα του FPGA. Είναι δυνατό να συνδυάσουμε μεταξύ τους DSP Blocks ώστε να εκτελέσουμε μεγαλύτερες πράξεις όπως πρόσθεση, αφαίρεση, πολλαπλασιασμό, διαίρεση και τετραγωνική ρίζα floating point αριθμών απλής και διπλής ακρίβειας. Αναμφισβήτητα, μία από τις πιο σημαντικές προσθήκες στο σώμα των FPGA είναι η προσθήκη ενσωματωμένων επεξεργαστών. Υπάρχει ένας μεγάλος αριθμός διεπαφών για τη διασύνδεση των επεξεργαστών με την προγραμματιζόμενη λογική των FPGA ώστε να είναι δυνατή η επικοινωνία με τους hardware πυρήνες. Τα περισσότερα συστήματα διαθέτουν ένα μόνο εξωτερικό ρολόι που παράγει μια σταθερή συχνότητα ρολογιού. Παρόλα αυτά, υπάρχουν διάφοροι λόγοι για τους οποίους ένας σχεδιαστής μπορεί να επιθυμεί οι hardware πυρήνες του να λειτουργούν σε διαφορετικές συχνότητες. Ο Manager Ψηφιακού Ρολογιού (Digital Clock Manager, DCM), επιτρέπει την παραγωγή διαφορετικών συχνοτήτων ρολογιού από ένα μόνο ρολόι αναφοράς. Είναι δυνατό να διαιρέσουμε το υπάρχων σήμα ρολογιού ώστε να πετύχουμε παλμούς χαμηλότερης συχνότητας. Το πλεονέκτημα της χρήσης των DCMs είναι ότι οι παραγόμενοι παλμοί θα έχουν μικρή απόκλιση από την επιθυμητή τιμή (Jitter) και μια προκαθορισμένη σχέση φάσης.

1.2 Πλεονεκτήματα και μειονεκτήματα των FPGA

Τα FPGA παρέχουν στις σχεδιάσεις και εφαρμογές μας πληθώρα πλεονεκτημάτων αλλά παρουσιάζουν και σημαντικά μειονεκτήματα συγκρινόμενα με τους επεξεργαστές ψηφιακού σήματος. Τα κυριότερα από τα πλεονεκτήματα και τα μειονεκτήματα αυτά παρουσιάζονται παρακάτω.

1.2.1 Πλεονεκτήματα των FPGA

Τα FPGA (Field Programmable Gate Arrays) είναι μία ειδική κατηγορία ολοκληρωμένων κυκλωμάτων. Τα κύρια πλεονεκτήματα που προσφέρουν τα FPGA είναι:

1) **Απόδοση:** Λόγω του hardware παραλληλισμού, τα FPGA υπερβαίνουν την υπολογιστική δύναμη των DSP, σπάζοντας το πρότυπο της ακολουθιακής εκτέλεσης και επιτυγχάνοντας περισσότερους κύκλους ανά παλμό του ρολογιού. Επίσης ελέγχοντας τις εισόδους και τις εξόδους στο hardware επίπεδο πετυχαίνουμε γρηγορότερους χρόνους απόκρισης και εξειδικευμένη λειτουργία για να ταιριάζει σχεδόν ακριβώς στις απαιτήσεις του συστήματος.

2) **Time to Market:** Η FPGA τεχνολογία προσφέρει ευελιξία και ταχεία προτυποποίηση για να ανταπεξέλθει στις προσδοκίες της αγοράς. Μπορούμε να τεστάρουμε μία ιδέα και να την επαληθεύσουμε στο hardware χωρίς να μπούμε στην μακρά διαδικασία της ASIC σχεδίασης. Μπορούμε μετά να υλοποιήσουμε σταδιακές αλλαγές και να επανελέγξουμε την FPGA σχεδίαση μέσα σε ώρες αντί για εβδομάδες. Είναι επίσης διαθέσιμο Commersial Off The Self (COTS) hardware με διαφορετικού τύπου εισόδους – εξόδους οι οποίες είναι ήδη ενωμένες στο FPGA chip. Τέλος η αυξανόμενη διαθεσιμότητα software εργαλείων υψηλού επιπέδου μειώνει την απαραίτητη προγραμματιστική γνώση με αρκετά επίπεδα αφαίρεσης και συχνά προσφέρει IP Cores (προκατασκευασμένες συναρτήσεις) για προηγμένο έλεγχο και ανάλυση σήματος.

3) Κόστος: Το NRE (Non Recurring Engineering) κόστος της ASIC σχεδίασης ξεπερνά κατά πολύ αυτό των βασισμένων σε FPGA hardware λύσεων. Το μεγάλο αρχικό κόστος επένδυσης στα ASICs αποσβένεται για εφαρμογές που πουλούν χιλιάδες chip το χρόνο, αλλά κάποιοι χρήστες χρειάζονται συγκεκριμένη hardware λειτουργία για δεκάδες ή εκατοντάδες συστήματα. Επίσης η φύση αυτή του προγραμματιζόμενου πυριτίου του FPGA σημαίνει μηδενικό κόστος κατασκευής όπως και συναρμολόγησης. Επειδή οι απαιτήσεις του συστήματος συχνά αλλάζουν με το χρόνο, το κόστος για μικρές αλλαγές του hardware στο FPGA είναι αμελητέο συγκρινόμενο με αυτό της ανακατασκευής του ASIC.

4) **Αξιοπιστία:** Τα Processor – based συστήματα συχνά περιλαμβάνουν αρκετά στρώματα αφαίρεσης για να βοηθήσουν την χρονοδρομολόγηση των εργασιών και τον διαμοιρασμό των πόρων ανάμεσα στις πολλαπλές διαδικασίες. Για κάθε δοσμένο πυρήνα επεξεργαστή, μία μόνο εντολή μπορεί να εκτελεστεί ανά φορά. Στα Processor – based συστήματα είναι συνεχώς σε κίνδυνο οι time – critical εργασίες να επηρεάζουν η μία την άλλη. Τα FPGA από την άλλη ελαχιστοποιούν τους κινδύνους αξιοπιστίας με την πραγματική παράλληλη εκτέλεση και το ντετερμινιστικό hardware κομμάτι το οποίο είναι αποκλειστικά αφιερωμένο για την κάθε εργασία.

5) Μακροπρόθεσμη Διατήρηση: Τα FPGA chip είναι Field – Upgradable και δεν χρειάζονται τον χρόνο και τα έξοδα που σχετίζονται με την ASIC επανασχεδίαση. Τα ψηφιακά πρωτόκολλα επικοινωνίας, για παράδειγμα, θέτουν στα συστήματα διάφορες προδιαγραφές οι οποίες μπορεί να αλλάζουν με τον καιρό. Άρα οι αναπροσαρμοστικότητα των FPGA τους επιτρέπει να ακολουθούν τις μελλοντικές, απαραίτητες αλλαγές. Τέλος καθώς το προϊόν ή το σύστημα ωριμάζει, μπορούμε να κάνουμε λειτουργικές βελτιώσεις χωρίς να σπαταλούμε χρόνο για την επανασχεδίαση του hardware ή την τροποποίηση του ήδη υπάρχοντος όπως θα απαιτούνταν στα ASIC.

1.2.2 Μειονεκτήματα των FPGA

1) Μεγαλύτερη κατανάλωση ισχύος.
2) Λιγότερο αποδοτική χρήση του πυριτίου.
3) Είναι τάξεις μεγέθους πιο δύσκολο να προγραμματιστούν.
4) Το 70% του κόστους ανάπτυξης λογισμικού προέρχεται από την ανάπτυξη της Εισόδου/Εξόδου διεπαφής (αντί από αλγόριθμους).
5) Δεν είναι η καταλληλότερη λύση για εφαρμογές που εμπεριέχουν υπολογισμούς αριθμών κινητής υποδιαστολής. Ακόμα και οι υπολογισμοί με σταθερής υποδιαστολής αριθμούς υψηλής ακρίβειας χρειάζονται μεγάλη ποσότητα λογικών κυττάρων για να υλοποιηθούν.

1.3 Γλώσσες Περιγραφής Υλικού

Χρησιμοποιούμε τις Γλώσσες Περιγραφής Υλικού (Hardware Description Language, HDL), ως γλώσσες υψηλού επιπέδου για την περιγραφή ενός κυκλώματος που πρόκειται να υλοποιηθεί σε FPGA. Οι απαρχές των γλωσσών περιγραφής υλικού, έχουν τις ρίζες τους στην ανάγκη να τεκμηριωθεί η συμπεριφορά του υλικού. Με την πάροδο του χρόνου, αναγνωρίστηκε πως οι περιγραφές θα μπορούσαν να χρησιμοποιηθούν για την προσομοίωση hardware κυκλωμάτων σε έναν επεξεργαστή γενικής χρήσης. Αυτή η διαδικασία μετάφρασης ενός κώδικα HDL σε μια μορφή κατάλληλη για έναν τέτοιο επεξεργαστή ώστε να μιμηθεί το hardware που περιγράφεται, ονομάζεται προσομοίωση (simulation). Η προσομοίωση έχει αποδειχθεί πως είναι ένα εξαιρετικά χρήσιμο εργαλείο για την ανάπτυξη του υλικού και την επαλήθευση της λειτουργικότητας πριν τη φυσική κατασκευή του υλικού. Δυστυχώς, ενώ η προσομοίωση παρέχει ένα μεγάλο αριθμό δομών ώστε να βοηθήσει τον σχεδιαστή στον έλεγχο και την ανάλυση του σχεδίου, πολλές από αυτές εκτείνονται πέρα από τη φυσική υλοποίηση του υλικού ή συνθέτουν αναποτελεσματικά τους πόρους του FPGA. Ως εκ τούτου, μόνο ένα υποσύνολο μιας HDL μπορεί να χρησιμοποιηθεί στην σύνθεση των σχεδίων σε υλικό.

1.4 Xilinx Virtex 7

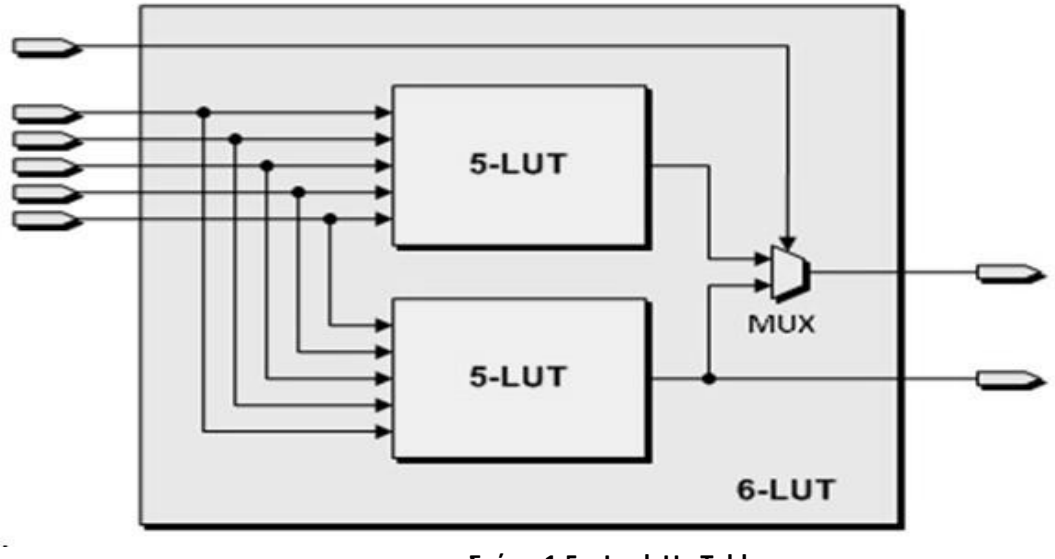
Στο παρόν κεφάλαιο, ακολουθεί μια περιγραφή των χαρακτηριστικών του Xilinx Virtex-7 XC7VX485T καθώς και του μικροεπεξεργαστή Microblaze που χρησιμοποιήθηκε στο σύστημά μας.

Σχήμα 1.4 – Virtex7

Η οικογένεια Virtex 7 της Xilinx, αποτελείται από αρκετές όμοιες μεν, αλλά μεταξύ τους διαφορετικές συσκευές. Οι διαφορετικές αυτές συσκευές δηλώνονται με τα σύμβολα LX, SX, TX και FX. Τα σύμβολα αυτά, αναφέρονται σε διαφορετικούς συνδυασμούς των προγραμματιζόμενων λογικών blocks και των hardware πυρήνων της συσκευής. Το LX (και κάθε παραλλαγή του, όπως LXT) αναφέρεται σε συσκευές που περιέχουν μεγάλο αριθμό προγραμματιζόμενων λογικών blocks σε σχέση με τους hardware πυρήνες της συσκευής. Οι πόροι στο FPGA είναι κυρίως προσανατολισμένοι στην προγραμματιζόμενη λογική, σε αντίθεση με έναν ενσωματωμένο επεξεργαστή στο σώμα του FPGA. Το SX αναφέρεται στις συσκευές της σειράς που σχετίζεται με την επεξεργασία σήματος, διαθέτοντας τους περισσότερους πόρους για εφαρμογές ψηφιακής επεξεργασίας σήματος. Το σύμβολο TX, αναφέρεται σε εκείνες τις συσκευές οι οποίες διαθέτουν επιπλέον σειριακούς, υψηλής ταχύτητας πομποδέκτες για συνδέσεις με χωρητικότητα υψηλού bandwidth. Τέλος, το FX αναφέρεται στα FPGA που περιέχουν έναν ή περισσότερους επεξεργαστές ενσωματωμένους στο σώμα του FPGA.

1.4.1 Look-Up Table

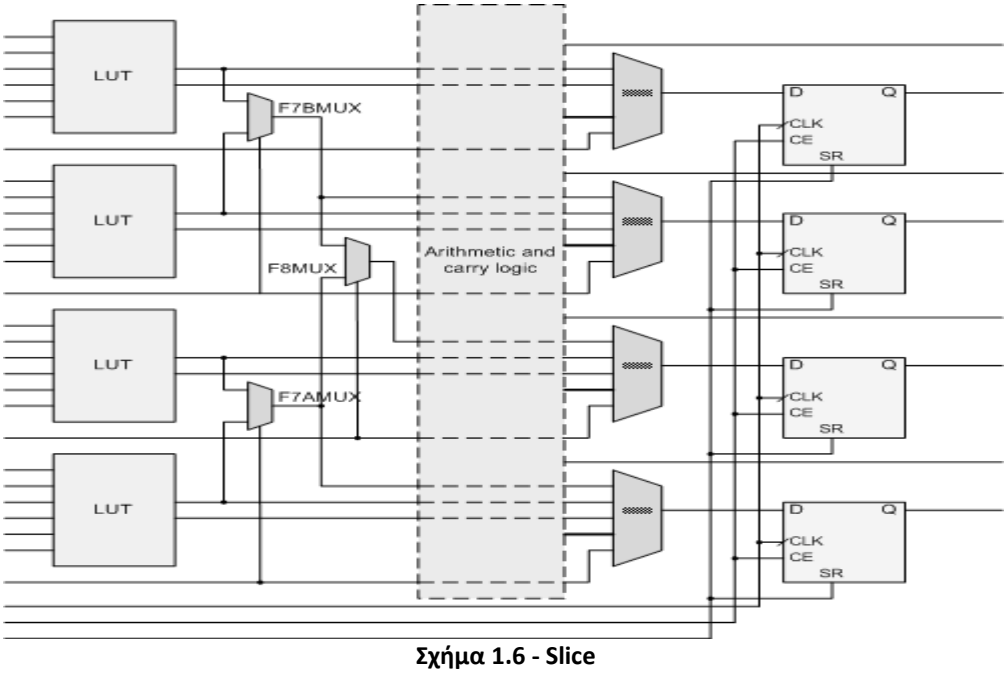
Η Xilinx αναφέρει τις γεννήτριες συναρτήσεων στο σώμα του FPGA ως Look-Up Tables (LUT). Τα FPGA της σειράς Virtex 7 δομούνται από LUTs 6 εισόδων. Αυτά τα LUTs των 6 εισόδων μπορούν να χρησιμοποιηθούν είτε ως άπλα 6-LUTs, είτε ως δυο 5-LUTs εφόσον τα τελευταία μοιράζονται τις ίδιες εισόδους (Σχήμα 1.5).



Σχήμα 1.5 – Look Up Table

1.4.2 Slice

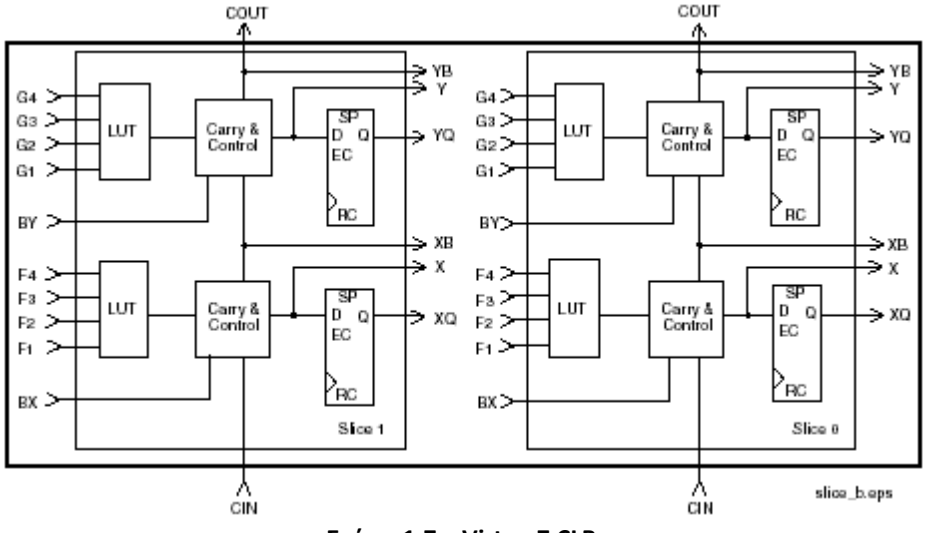
Στο προηγούμενο κεφάλαιο αναφέρθηκε ο όρος λογικό κελί για να περιγράψει τη δομή που αποτελείται από ένα LUT και ένα Flip-Flop. Τα Virtex 7 συνδυάζουν 4 τέτοια λογικά κελιά, μαζί με τα flip-flops στις εξόδους των κελιών, για να δημιουργήσουν ένα slice. Με τέσσερα 6-LUTs και τέσσερα D Flip-Flops να βρίσκονται σε κοντινή απόσταση, είναι δυνατή η κατασκευή πιο σύνθετων σχεδίων (Σχήμα 1.6). Εκτός από τη Boolean λογική, ένα slice μπορεί να χρησιμοποιηθεί για αριθμητικές πράξεις καθώς και για την κατασκευή μνημών RAM και ROM. Μερικά slices συνδέονται με τέτοιο τρόπο ώστε μπορούν να χρησιμοποιηθούν για αποθήκευση δεδομένων, όπως διανεμημένες RAM και καταχωρητές ολίσθησης. Αυτό είναι δυνατό συνδυάζοντας πολλά LUTs σε ένα slice. Σε όλες αυτές τις δυνατές χρήσεις, τα D Flip-Flops μπορούν να χρησιμοποιηθούν για διαδικασία της σύγχρονης ανάγνωσης. Το Virtex-7 XC7VX485T διαθέτει συνολικά 75.900 slices.



Σχήμα 1.6 - Slice

1.4.3 Προγραμματιζόμενα Λογικά Blocks (CLBs)

Στα Virtex 7, ένα CLB περιέχει δύο slices και ένα carry-logic για τη σύνδεση των γειτονικών slices. Ένα CLB θεωρείται ως το υψηλότερο επίπεδο αφαίρεσης για το προγραμματιζόμενο σώμα του FPGA. Τα δύο slices του CLB δεν είναι άμεσα συνδεδεμένα, αλλά βρίσκονται σε διαφορετικές στήλες slices. Κάθε CLB συνδέεται σε έναν switch matrix, παρέχοντας με αυτόν τον τρόπο τη δυνατότητα στα προγραμματιζόμενα σχέδια να εκτείνονται σε πολλά CLBs. Ο switch matrix αποτελείται από μακριές και κοντές καλωδιώσεις (long and short wires) ώστε να παρέχει άμεσες, point-to-point συνδέσεις μεταξύ των CLBs. Ένα Virtex 7 CLB παρουσιάζεται στο Σχήμα 1.7



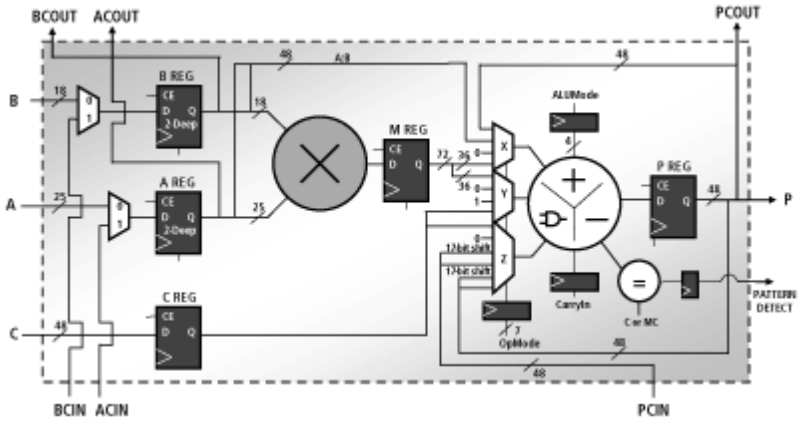
Σχήμα 1.7 – Virtex-7 CLB

1.4.4 Block RAM (BRAM)

Ένα Block Ram αναφέρεται στις μνήμες τυχαίας προσπέλασης που ομαδοποιούνται σε blocks των 36Kbit στα FPGA της οικογένειας Virtex 7. Η BRAM παρέχει μνήμη εντός του τσιπ, που μπορεί να χρησιμοποιηθεί για τον σχηματισμό μεγάλων LUTs, buffer δεδομένων (FIFOs) και στην τοπική αποθήκευση. Κάθε τέτοιο block έχει δύο ανεξάρτητες θύρες πρόσβασης οι οποίες ανταλλάσσουν δεδομένα μόνο με τα κελιά της μνήμης. Τα ρολόγια σχετίζονται με κάθε θύρα ξεχωριστά. Κάθε 36Kbit BRAM, μπορεί να ρυθμιστεί ως 32K×1 (32.768 δεδομένα του ενός Bit), 16K×2, 8K×4, 4K×9, 2K×18, ή 1K×36. Οι BRAM μπορούν να συνδυαστούν μεταξύ τους ώστε να δημιουργήσουν μεγαλύτερες (τόσο στο βάθος-depth όσο και στο πλάτος-width) BRAMs. Ομοίως, μια BRAM μπορεί να διαιρεθεί σε δύο ανεξάρτητες 16Kb BRAMs. Αυτό δίνει τη δυνατότητα στα FPGA Virtex 7 να αξιοποιούν τους πόρους τους περισσότερο αποτελεσματικά. Το Virtex-7 XC7VX485T διαθέτει 1030× 36Kb BRAMs ή 2060 x 18Kb BRAMs.

1.4.5 DSP Slices

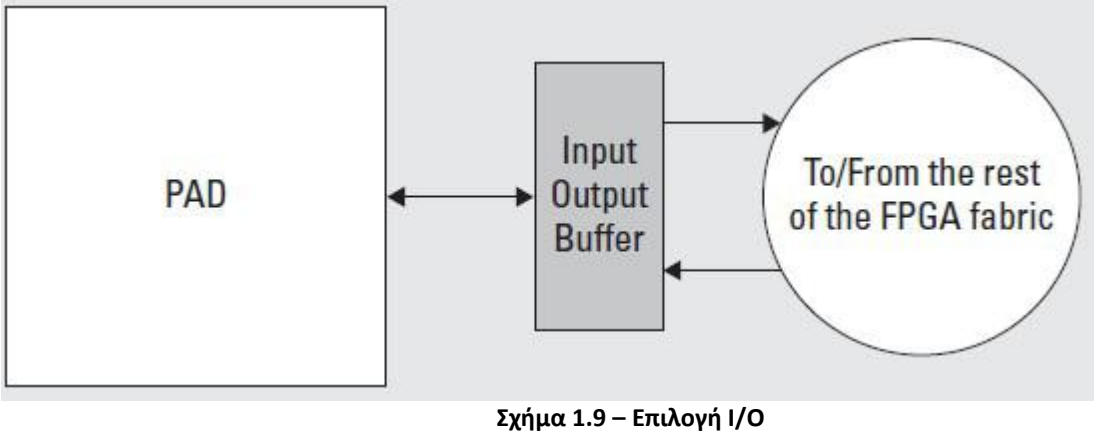
Μια κύρια χρήση των FPGA είναι η ψηφιακή επεξεργασία σήματος. Για την βελτίωση της απόδοσης, ορισμένα FPGA διαθέτουν ενσωματωμένα ειδικά DSP Blocks. Στα Virtex 7, τα DSP slices περιλαμβάνουν έναν 25×18 πολλαπλασιαστή συμπληρώματος του δύο, έναν συσσωρευτή, έναν αθροιστή/αφαιρέτη για πράξεις συνεχούς διοχέτευσης (pipelined) και λογικές πράξεις σε επίπεδο bit. Ενσωματώνοντας αυτή τη λειτουργία μέσα στα slices, μπορούμε να πετύχουμε σημαντική εξοικονόμηση των πόρων του FPGA, αφού η υλοποίηση του σχεδίου μόνο σε LUTs είναι αρκετά “ακριβή”. Το Virtex-7 XC7VX485T διαθέτει 2800 DSP Slices .



Σχήμα 1.8 – Dsp Slices

1.4.6 Επιλογή I/O

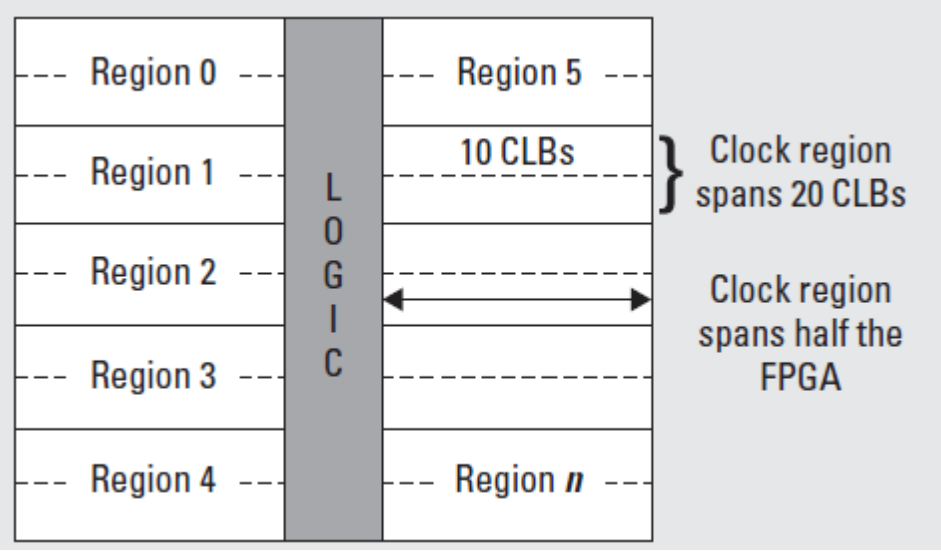
Η Xilinx χρησιμοποιεί τον όρο Select I/O για να αναφερθεί στις προγραμματιζόμενες
εισόδους και εξόδους, οι οποίες υποστηρίζουν μια ποικιλία από τυποποιημένες διεπαφές (HT, SSTL, LVDS, κτλ). Όπως φαίνεται και στο Σχήμα 1.8 κάθε I/O συνδέεται σε ένα pad το οποίο με τη σειρά του συνδέεται σε έναν ακροδέκτη της συσκευής. Το Select I/O επωφελείται επίσης από το Digitally Controlled Impedance (DCI) για την εξάλειψη των πρόσθετων αντιστάσεων στους ακροδέκτες της συσκευής, πράγμα αναγκαίο για την αποφυγή της υποβάθμισης του σήματος. Το DCI μπορεί να ρυθμίσει τις αντιστάσεις εισόδου και εξόδου ώστε να ταιριάζουν με το ίχνος της αντίστασης τόσο στην οδήγηση, όσο και στη λήψη. Το Virtex-7 XC7VX485T διαθέτει 700 user I/O .



Σχήμα 1.9 – Επιλογή I/O

1.4.7 Ρολόγια

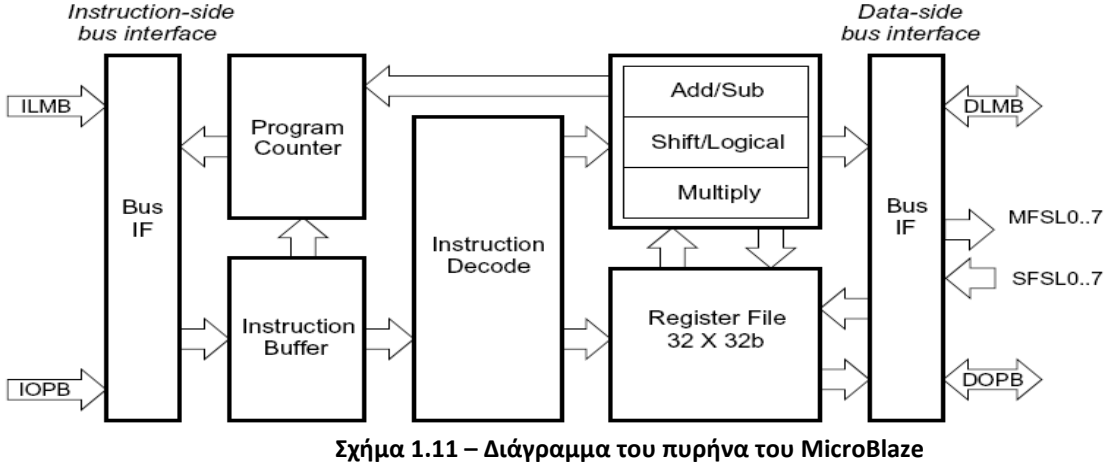
Είναι δυνατόν, διάφοροι hardware πυρήνες να λειτουργούν με διαφορετικές μεταξύ τους συχνότητες. Στον παραδοσιακό σχεδιασμό, κάθε ρολόι παράγεται εκτός του τσιπ και συνδέεται ως είσοδος στο σύστημα. Στις υλοποιήσεις με FPGA είναι δυνατό να δημιουργήσουμε διαφορετικούς ρυθμούς ρολογιών από ένα ή περισσότερα ρολόγια. Ο αριθμός των clock regions στα FPGA Virtex7 κυμαίνεται από 4 σε 24 (Σχήμα 1.10). Για να βοηθήσει στον σχεδιασμό, τη χρήση και τη διαχείριση αυτών των ρολογιών η Xilinx χρησιμοποιεί τους Μάνατζερ Ψηφιακού Ρολογιού (Digital Clock Managers, DCMs). Γενικά, ένας DCM παίρνει ένα ρολόι εισόδου και δημιουργεί ένα ρυθμιζόμενο ρολόι εξόδου.



Σχήμα 1.10 – Clock Regions

1.5 MicroBlaze

Το Virtex-7 XC7VX485T διαθέτει έναν ενσωματωμένο μικροεπεξεργαστή MicroBlaze. Ο επεξεργαστής MicroBlaze είναι ένας 32-bit RISC επεξεργαστής, που προορίζεται ειδικά για υλοποιήσεις σε FPGAs. Το διάγραμμα του πυρήνα του παρουσιάζεται παρακάτω:



Σχήμα 1.11 – Διάγραμμα του πυρήνα του MicroBlaze

Τα κυριότερα από τα χαρακτηριστικά του MicroBlaze παρουσιάζονται παρακάτω:

- Διαθέτει 32 καταχωρητές γενικού σκοπού και δύο καταχωρητές ειδικού σκοπού, όλοι μήκους 32-bit
- Το μήκος της εντολής του MicroBlaze είναι 32 bit και χρησιμοποιεί μέχρι τρεις τελεστές και δύο μεθόδους διευθυνσιοδότησης
- Διαθέτει ξεχωριστούς διαδρόμους δεδομένων, των 32 bit, για τις εντολές και για τα δεδομένα, που ακολουθούν τις προδιαγραφές του διαδρόμου δεδομένων OPB (On-chip Peripheral Bus) της IBM
- Διαθέτει ξεχωριστούς διαδρόμους δεδομένων, των 32 bit, για τις εντολές και για τα δεδομένα, με άμεση σύνδεση με block RAM, μέσω του διαδρόμου δεδομένων LMB (Local Memory Bus)

- Το μήκος των διευθύνσεων που χρησιμοποιεί είναι 32 bit
- Εφαρμόζει την παράλληλη, μέσω pipeline, εκτέλεση εντολών, με το pipeline να χωρίζεται σε τρία στάδια
- Διαθέτει γρήγορη μνήμη τόσο για τις εντολές όσο και για τα δεδομένα
- Υποστηρίζει την γρήγορη, σημείο – με – σημείο σύνδεση μέσω της διεπαφής FSL (Fast Simplex Link)
- Διαθέτει hardware πολλαπλασιαστή

1.5.1 Αναλυτική περιγραφή

Καταχωρητές

Ο MicroBlaze διαθέτει 32 καταχωρητές γενικού σκοπού και δύο καταχωρητές ειδικού σκοπού. Οι καταχωρητές γενικού σκοπού έχουν τις ονομασίες R0 έως R31, με τον R0 να είναι πάντοτε μηδενικός αγνοώντας οτιδήποτε γράφεται σε αυτόν. Οι δύο καταχωρητές ειδικού σκοπού είναι ο Program Counter (PC) και ο Machine Status Register (MSR). Όπως μαρτυρά η ονομασία του, ο PC δείχνει σε κάθε στιγμή την διεύθυνση της επόμενης εντολής που πρόκειται να φορτωθεί για εκτέλεση. Ο MSR περιλαμβάνει το flag για το κρατούμενο υπερχείλισης και τα bits ενεργοποίησης για τις διακοπές, για την γρήγορη μνήμη (cache), για την διακοπή λειτουργίας του OPB καθώς και άλλες πληροφορίες κατάστασης και ελέγχου. Μια πλήρης περιγραφή της χρήσης των bits του MSR γίνεται στο σχήμα 1.12

Bits	Name	Description	Reset Value
0	CC	Arithmetic Carry Copy Copy of the Arithmetic Carry (bit 29). Read only.	0
1:23	Reserved		
24	DCE	Data Cache Enable 0 Data Cache is Disabled 1 Data Cache is Enabled	0
25	DZ	Dvision by Zero 0 No divison by zero has occured 1 Division by zero has occured	0
26	ICE	Instruction Cache Enable 0 Instruction Cache is Disabled 1 Instruction Cache is Enabled	0
27	FSL	FSL Error 0 FSL get/put had no error 1 FSL get/put had mismatch in instruction type and value type	0
28	BIP	Break in Progress 0 No Break in Progress 1 Break in Progress Source of break can be software break instruction or hardware break from Ext_Brk or Ext_NM_Brk pin.	0

Σχήμα 1.12 – Ο καταχωρητής MSR

1.5.2 Εντολές

Όλες οι εντολές του MicroBlaze έχουν μήκος 32 bit και χωρίζονται σε δύο κατηγορίες, τις εντολές τύπου Α και τις εντολές τύπου Β. Οι εντολές τύπου Α χρησιμοποιούν δύο καταχωρητές προέλευσης και έναν καταχωρητή προορισμού, ενώ οι εντολές τύπου Β έναν καταχωρητήπροέλευσης μαζί με ένα άμεσο όρισμα των 16 bit και έναν καταχωρητή προορισμού. Όλες οι εντολές με την σημασιολογία (semantics) τους αναφέρονται στο σχήμα 1.13.

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	00000000000	Rd := Rb + Ra
RSUB Rd,Ra,Rb	000001	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDC Rd,Ra,Rb	000010	Rd	Ra	Rb	00000000000	Rd := Rb + Ra + C
RSUBC Rd,Ra,Rb	000011	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + C
ADDK Rd,Ra,Rb	000100	Rd	Ra	Rb	00000000000	Rd := Rb + Ra
RSUBK Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + 1
ADDKC Rd,Ra,Rb	000110	Rd	Ra	Rb	00000000000	Rd := Rb + Ra + C
RSUBKC Rd,Ra,Rb	000111	Rd	Ra	Rb	00000000000	Rd := Rb + $\overline{Ra}$ + C
CMP Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000001	Rd := Rb cmp Ra (signed)
CMPU Rd,Ra,Rb	000101	Rd	Ra	Rb	00000000011	Rd := Rb cmp Ra (unsigned)
ADDI Rd,Ra,Imm	001000	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBI Rd,Ra,Imm	001001	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIC Rd,Ra,Imm	001010	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIC Rd,Ra,Imm	001011	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
ADDIK Rd,Ra,Imm	001100	Rd	Ra	Imm		Rd := s(Imm) + Ra
RSUBIK Rd,Ra,Imm	001101	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + 1
ADDIKC Rd,Ra,Imm	001110	Rd	Ra	Imm		Rd := s(Imm) + Ra + C
RSUBIKC Rd,Ra,Imm	001111	Rd	Ra	Imm		Rd := s(Imm) + $\overline{Ra}$ + C
MUL Rd,Ra,Rb	010000	Rd	Ra	Rb	00000000000	Rd := Ra * Rb
BSRL Rd,Ra,Rb	010001	Rd	Ra	Rb	00000000000	Rd : = Ra >> Rb
BSRA Rd,Ra,Rb	010001	Rd	Ra	Rb	01000000000	Rd := Ra[0], (Ra >> Rb)
BSLL Rd,Ra,Rb	010001	Rd	Ra	Rb	10000000000	Rd := Ra << Rb
MULI Rd,Ra,Imm	011000	Rd	Ra	Imm		Rd := Ra * s(Imm)
BSRLI Rd,Ra,Imm	011001	Rd	Ra	0000	0000.. Imm	Rd : = Ra >> Imm
BSRAI Rd,Ra,Imm	011001	Rd	Ra	0000	0100.. Imm	Rd := Ra[0], (Ra >> Imm)
BSLLI Rd,Ra,Imm	011001	Rd	Ra	0000	1000.. Imm	Rd := Ra << Imm
IDIV Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000000	Rd := Rb/Ra, signed
IDIVU Rd,Ra,Rb	010010	Rd	Ra	Rb	00000000001	Rd := Rb/Ra, unsigned
GET Rd,FSLx	011011	Rd	00000	0000	FSLx	Rd := FSLx (blocking data read)
PUT Ra,FSLx	011011	00000	Ra	1000	FSLx	FSLx := Ra (blocking data write)
nGET Rd,FSLx	011011	Rd	00000	0100	FSLx	Rd := FSLx (non-blocking data read)
nPUT Ra,FSLx	011011	00000	Ra	1100	FSLx	FSLx := Ra (non-blocking data write)

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
cGET Rd,FSLx	011011	Rd	00000	0010	FSLx	Rd := FSLx (blocking control read)
cPUT Ra,FSLx	011011	00000	Ra	1010	FSLx	FSLx := Ra (blocking control write)
ncGET Rd,FSLx	011011	Rd	00000	0110	FSLx	Rd := FSLx (non-blocking control read)
ncPUT Ra,FSLx	011011	00000	Ra	1110	FSLx	FSLx := Ra (non-blocking control write)
OR Rd,Ra,Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
AND Rd,Ra,Rb	100001	Rd	Ra	Rb	00000000000	Rd := Ra and Rb
XOR Rd,Ra,Rb	100010	Rd	Ra	Rb	00000000000	Rd := Ra xor Rb
ANDN Rd,Ra,Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{Rb}$
SRA Rd,Ra	100100	Rd	Ra	0000000000000001		Rd := Ra[0], (Ra >> 1); C := Ra[31]
SRC Rd,Ra	100100	Rd	Ra	0000000000100001		Rd := C, (Ra >> 1); C := Ra[31]
SRL Rd,Ra	100100	Rd	Ra	0000000001000001		Rd := 0, (Ra >> 1); C := Ra[31]
SEXT8 Rd,Ra	100100	Rd	Ra	0000000001100000		Rd[0:23] := Ra[24]; Rd[24:31] := Ra[24:31]
SEXT16 Rd,Ra	100100	Rd	Ra	0000000001100001		Rd[0:15] := Ra[16]; Rd[16:31] := Ra[16:31]
WIC Rd,Ra	100100	Ra	Ra	Rb	01101000	ICache_Tag := Ra, ICache_Data := Rb
WDC Rd,Ra	100100	Ra	Ra	Rb	01100100	DCache_Tag := Ra, DCache_Data := Rb
MTS Sd,Ra	100101	00000	Ra	110000000000000d		Sd := Ra , where S1 is MSR
MFS Rd,Sa	100101	Rd	00000	100000000000000a		Rd := Sa , where S0 is PC and S1 is MSR
BR Rb	100110	00000	00000	Rb	00000000000	PC := PC + Rb
BRD Rb	100110	00000	10000	Rb	00000000000	PC := PC + Rb
BRLD Rd,Rb	100110	Rd	10100	Rb	00000000000	PC := PC + Rb; Rd := PC
BRA Rb	100110	00000	01000	Rb	00000000000	PC := Rb
BRAD Rb	100110	00000	11000	Rb	00000000000	PC := Rb
BRALD Rd,Rb	100110	Rd	11100	Rb	00000000000	PC := Rb; Rd := PC
BRK Rd,Rb	100110	Rd	01100	Rb	00000000000	PC := Rb; Rd := PC; MSR[BIP] := 1
BEQ Ra,Rb	100111	00000	Ra	Rb	00000000000	if Ra = 0: PC := PC + Rb
BNE Ra,Rb	100111	00001	Ra	Rb	00000000000	if Ra /= 0: PC := PC + Rb
BLT Ra,Rb	100111	00010	Ra	Rb	00000000000	if Ra < 0: PC := PC + Rb
BLE Ra,Rb	100111	00011	Ra	Rb	00000000000	if Ra <= 0: PC := PC + Rb
BGT Ra,Rb	100111	00100	Ra	Rb	00000000000	if Ra > 0: PC := PC + Rb
BGE Ra,Rb	100111	00101	Ra	Rb	00000000000	if Ra >= 0: PC := PC + Rb
BEQD Ra,Rb	100111	10000	Ra	Rb	00000000000	if Ra = 0: PC := PC + Rb

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
BNED Ra,Rb	100111	10001	Ra	Rb	00000000000	if Ra /= 0: PC := PC + Rb
BLTD Ra,Rb	100111	10010	Ra	Rb	00000000000	if Ra < 0: PC := PC + Rb
BLED Ra,Rb	100111	10011	Ra	Rb	00000000000	if Ra <= 0: PC := PC + Rb
BGTD Ra,Rb	100111	10100	Ra	Rb	00000000000	if Ra > 0: PC := PC + Rb
BGED Ra,Rb	100111	10101	Ra	Rb	00000000000	if Ra >= 0: PC := PC + Rb
ORI Rd,Ra,Imm	101000	Rd	Ra	Imm		Rd := Ra or s(Imm)
ANDI Rd,Ra,Imm	101001	Rd	Ra	Imm		Rd := Ra and s(Imm)
XORI Rd,Ra,Imm	101010	Rd	Ra	Imm		Rd := Ra xor s(Imm)
ANDNI Rd,Ra,Imm	101011	Rd	Ra	Imm		Rd := Ra and $\overline{\text{s(Imm)}}$
IMM Imm	101100	00000	00000	Imm		Imm[0:15] := Imm
RTSD Ra,Imm	101101	10000	Ra	Imm		PC := Ra + s(Imm)
RTID Ra,Imm	101101	10001	Ra	Imm		PC := Ra + s(Imm); MSR[IE] := 1
RTBD Ra,Imm	101101	10010	Ra	Imm		PC := Ra + s(Imm); MSR[BIP] := 0
BRID Imm	101110	00000	10000	Imm		PC := PC + s(Imm)
BRLID Rd,Imm	101110	Rd	10100	Imm		PC := PC + s(Imm); Rd := PC
BRAI Imm	101110	00000	01000	Imm		PC := s(Imm)
BRAID Imm	101110	00000	11000	Imm		PC := s(Imm)
BRALID Rd,Imm	101110	Rd	11100	Imm		PC := s(Imm); Rd := PC
BRKI Rd,Imm	101110	Rd	01100	Imm		PC := s(Imm); Rd := PC; MSR[BIP] := 1
BEQI Ra,Imm	101111	00000	Ra	Imm		if Ra = 0: PC := PC + s(Imm)
BNEI Ra,Imm	101111	00001	Ra	Imm		if Ra /= 0: PC := PC + s(Imm)
BLTI Ra,Imm	101111	00010	Ra	Imm		if Ra < 0: PC := PC + s(Imm)
BLEI Ra,Imm	101111	00011	Ra	Imm		if Ra <= 0: PC := PC + s(Imm)
BGTI Ra,Imm	101111	00100	Ra	Imm		if Ra > 0: PC := PC + s(Imm)
BGEI Ra,Imm	101111	00101	Ra	Imm		if Ra >= 0: PC := PC + s(Imm)
BEQID Ra,Imm	101111	10000	Ra	Imm		if Ra = 0: PC := PC + s(Imm)
BNEID Ra,Imm	101111	10001	Ra	Imm		if Ra /= 0: PC := PC + s(Imm)
BLTID Ra,Imm	101111	10010	Ra	Imm		if Ra < 0: PC := PC + s(Imm)
BLEID Ra,Imm	101111	10011	Ra	Imm		if Ra <= 0: PC := PC + s(Imm)
BGTID Ra,Imm	101111	10100	Ra	Imm		if Ra > 0: PC := PC + s(Imm)
BGEID Ra,Imm	101111	10101	Ra	Imm		if Ra >= 0: PC := PC + s(Imm)
LBU Rd,Ra,Rb	110000	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:23] := 0, Rd[24:31] := *Addr

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
Type B	0-5	6-10	11-15	16-31		
LHU Rd,Ra,Rb	110001	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd[0:15] := 0, Rd[16:31] := *Addr
LW Rd,Ra,Rb	110010	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; Rd := *Addr
SB Rd,Ra,Rb	110100	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[24:31]
SH Rd,Ra,Rb	110101	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd[16:31]
SW Rd,Ra,Rb	110110	Rd	Ra	Rb	00000000000	Addr := Ra + Rb; *Addr := Rd
LBUI Rd,Ra,Imm	111000	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd[0:23] := 0, Rd[24:31] := *Addr
LHUI Rd,Ra,Imm	111001	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd[0:15] := 0, Rd[16:31] := *Addr
LWI Rd,Ra,Imm	111010	Rd	Ra	Imm		Addr := Ra + s(Imm); Rd := *Addr
SBI Rd,Ra,Imm	111100	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[24:31]
SHI Rd,Ra,Imm	111101	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd[16:31]
SWI Rd,Ra,Imm	111110	Rd	Ra	Imm		Addr := Ra + s(Imm); *Addr := Rd

Σχήμα 1.13 – Σύνοψη των εντολών του MicroBlaze

Όταν ενεργοποιηθεί το σήμα Reset ή το Debug_Rst, ο MicroBlaze ξεκινά να εκτελεί την εντολή που βρίσκεται στην διεύθυνση 0, ακολουθούμενη από τις εντολές που καθορίζει το πρόγραμμα που είναι φορτωμένο στην μνήμη. Εκτός όμως από τις εντολές που εκτελούνται μέσω του software, ιδιαίτερη μνεία αξίζει να γίνει στις εντολές που εκτελούνται όταν συμβαίνουν διακοπές (interrupts), εξαιρέσεις (exceptions) και σταμάτημα λειτουργίας (break), που οφείλονται στο υλικό (hardware).

Όταν συμβεί μία διακοπή, ο MicroBlaze αναβάλλει την εκτέλεση της επόμενης εντολής προκειμένου να την εξυπηρετήσει. Η διεύθυνση της επόμενης εντολής που επρόκειτο να εκτελεστεί αποθηκεύεται στον καταχωρητή γενικού σκοπού R14 και ο MicroBlaze διακλαδίζεται στην διεύθυνση 0x00000010. Παράλληλα απενεργοποιεί τυχόν μελλοντικές διακοπές θέτοντας το κατάλληλο flag, bit 30, ‘0’ στον καταχωρητή MSR. Όταν το bit αυτό είναι 0 οι διακοπές αγνοούνται, όπως συμβαίνει και στην περίπτωση που το bit 28 του MSR είναι ‘1’, καθώς τα breaks έχουν μεγαλύτερη προτεραιότητα από τις διακοπές.

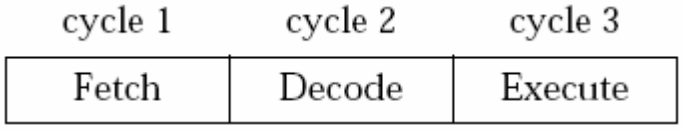
Ομοίως με τις διακοπές, όταν συμβεί μία εξαίρεση ο MicroBlaze αναβάλλει την εκτέλεση της επόμενης εντολής προκειμένου να την χειριστεί. Η διεύθυνση της επόμενης εντολής που επρόκειτο να εκτελεστεί αποθηκεύεται στον καταχωρητή γενικού σκοπού R17 και ο MicroBlaze διακλαδίζεται στην διεύθυνση 0x00000008.

Τέλος έχουμε τα break υλικού που οδηγούνται μέσω των ακροδεκτών Ext_Brk και Ext_NM_Brk. Και στις δύο περιπτώσεις ο MicroBlaze αναβάλλει την εκτέλεση της επόμενης εντολής προκειμένου να χειριστεί το break. Η διεύθυνση της επόμενης εντολής που επρόκειτονα εκτελεστεί αποθηκεύεται στον καταχωρητή γενικού σκοπού R16 και ο MicroBlaze διακλαδίζεται στην διεύθυνση 0x00000018. Παράλληλα ο MicroBlaze αποκλείει τυχόν μελλοντικά break θέτοντας το bit 28 του MSR ‘1’ ώστε να δείχνει ότι υπάρχει ήδη ένα break σε εξέλιξη. Μονάχα όταν το bit αυτό έχει την τιμή ‘0’, είναι δυνατόν να χειριστεί ένα break οδηγούμενο από το σήμα Ext_Brk, στην περίπτωση όμως που το break προκαλείται από το σήμα Ext_NM_Brk, τότε ο MicroBlaze το χειρίζεται άμεσα.

1.5.3 Pipeline

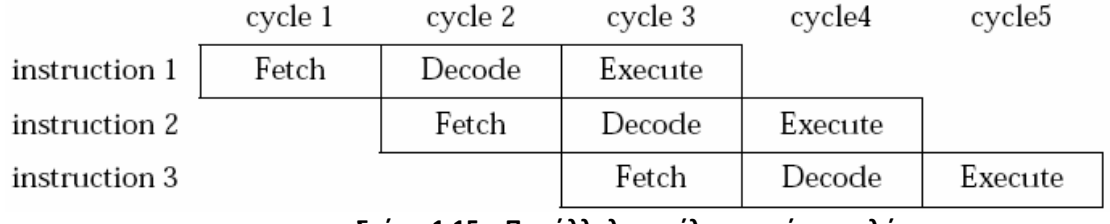
Το pipeline του MicroBlaze χωρίζεται σε τρία στάδια:
• Ανάκληση της εντολής
• Αποκωδικοποίηση της εντολής
• Εκτέλεση της εντολής

Γενικά κάθε στάδιο χρειάζεται έναν κύκλο ρολογιού για να ολοκληρωθεί, επομένως χρειάζονται τρεις κύκλοι ρολογιού, χωρίς να υπολογίζονται οι καθυστερήσεις ή οι αναβολές, για να εκτελεστεί μία εντολή.



Σχήμα 1.14 – Κάθε εντολή εκτελείται σε τρεις κύκλους

Στο παράλληλο pipeline του MicroBlaze κάθε στάδιο είναι ενεργό σε κάθε κύκλο ρολογιού. Τρεις εντολές είναι δυνατόν να εκτελούνται ταυτόχρονα, μία σε κάθε ένα από τα τρία στάδια του pipeline. Παρά το ότι χρειάζονται τρεις κύκλοι για την εκτέλεση μιας εντολής, κάθε στάδιο του pipeline μπορεί να είναι απασχολημένο με διαφορετική εντολή. Σε ένα κύκλο ρολογιού γίνεται ανάκληση μιας νέας εντολής, μία άλλη αποκωδικοποιείται και μία τρίτη ολοκληρώνει την εκτέλεσή της. Με τον τρόπο αυτό επιτυγχάνεται η εκτέλεση μιας εντολής ανά κύκλο ρολογιού.



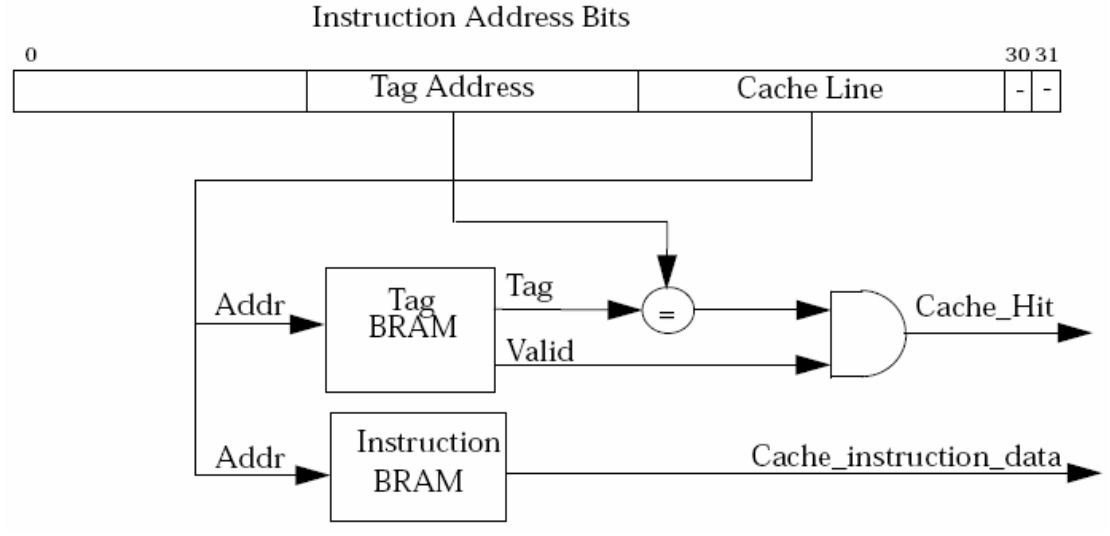
Σχήμα 1.15 – Παράλληλη εκτέλεση τριών εντολών

Όπως συμβαίνει με όλα τα pipeline των επεξεργαστών υπάρχουν εντολές που αλλάζουν την ροή του προγράμματος, εντολές διακλαδώσεων, και κατ' επέκταση τον ρυθμό εκτέλεσης των εντολών. Όταν μια εντολή διακλάδωσης βρεθεί στο στάδιο της εκτέλεσης, τότε η δουλειά που έχει γίνει στα προηγούμενα δύο στάδια δεν έχει καμία χρησιμότητα. Η δουλειά αυτή απορρίπτεται και το pipeline αδειάζει. Προκειμένου το pipeline να ξαναγεμίσει με τις σωστές εντολές, και δεδομένου ότι η εκτέλεση της εντολής διακλάδωσης χρειάζεται τρεις κύκλους ρολογιού, υπάρχει μία καθυστέρηση δύο κύκλων ρολογιού. Κάποιες εντολές διακλάδωσης καθιστούν δυνατό να μειωθεί ο χρόνος καθυστέρησης από δύο κύκλους σε έναν. Αυτό γίνεται με το να απορρίπτεται μόνο η εργασία που γίνεται στο πρώτο στάδιο του pipeline, ανάκλησηεντολής. Με τον τρόπο αυτό η εντολή που βρίσκεται μετά την εντολή διακλάδωσης, εκτελείται κανονικά και χάνεται μονάχα ένας κύκλος ρολογιού.

1.5.4 Γρήγορη Μνήμη

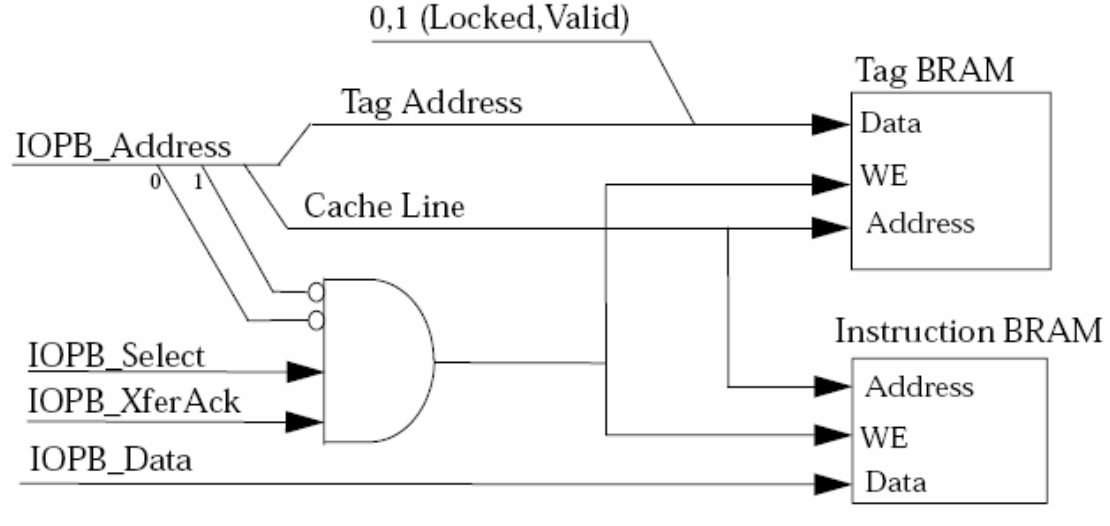
Ο MicroBlaze χρησιμοποιεί γρήγορη μνήμη, τόσο για τις εντολές, instruction cache, όσο και για τα δεδομένα, data cache. Η γρήγορη μνήμη για τις εντολές χρησιμοποιείται για καλύτερη επίδοση όταν υπάρχει εκτελέσιμος κώδικας που ανήκει σε μνήμες πέραν του εύρους διευθύνσεων του LMB. Στην περίπτωση αυτή ο χώρος της μνήμης χωρίζεται σε δύο μέρη, το ένα δύναται να εγγράφεται στην γρήγορη μνήμη , ενώ το άλλο, στο οποίο πρέπει να ανήκουν και οι διευθύνσεις του LMB, όχι. Το πρώτο

τμήμα καθορίζεται από δύο παραμέτρους, C_ICACHE_BASEADDR και C_ICACHE_HIGHADDR. Όλες οι διευθύνσεις μέσα σε αυτό το εύρος χωρίζονται σε δύο τμήματα, στο τμήμα cache line και στο τμήμα tag address. Τα μεγέθη των δύο τμημάτων καθορίζονται από τον χρήστη, ενώ το τμήμα μεταξύ του bit 0 και του tag address αγνοείται και τα bit 30 και 31 είναι δεσμευμένα. Το μέγεθος του cache line είναι μεταξύ 10 και 14 bit, που μεταφράζεται σε μέγεθος γρήγορης μνήμης που κυμαίνεται μεταξύ 4 Kbytes και 64 Kbytes. Δεν υπάρχει περιορισμός στο μέγεθος του tag address.



Σχήμα 1.16 – Η οργάνωση της instruction cache

Στο στάδιο της ανάκλησης της εντολής, ο MicroBlaze γράφει την διεύθυνση της επόμενης εντολής στον αντίστοιχο διάδρομο δεδομένων και περιμένει το σήμα επιβεβαίωσης. Για οικονομία χρόνου το αίτημα της ανάκλησης γίνεται ταυτόχρονα και στον LMB και στον OPB. Αν το σήμα επιβεβαίωσης έρθει από τον LMB στον επόμενο κύκλο, τότε ακυρώνεται η πρόσβαση στον OPB. Κάθε φορά η γρήγορη μνήμη των εντολών ανιχνεύει εάν η διεύθυνση της εντολής ανήκει στο εύρος των εντολών που δύνανται να εγγράφονται στην γρήγορη μνήμη. Αν κάτι τέτοιο δεν συμβαίνει, τότε η γρήγορη μνήμη αφήνει την διεκπεραίωση της αίτησης στους LMB ή OPB. Αν ισχύει το αντίθετο τότε η γρήγορη μνήμη ελέγχει αν έχει την εντολή που ζητείται. Αυτό συμβαίνει όταν το tag address της ζητούμενης εντολής είναι το ίδιο με αυτό που έχει η γρήγορη μνήμη, αντιστοιχίζοντας το cache line, και εάν το bit εγκυρότητας είναι ενεργοποιημένο. Εάν η εντολή βρίσκεται στην γρήγορη μνήμη, τότε θα στείλει το σήμα επιβεβαίωσης στον MicroBlaze και μαζί την εντολή. Εάν όμως δεν την έχει θα περιμένει έως ότου η αίτηση διεκπεραιωθεί από τον OPB, και έπειτα θα ενημερώσει το περιεχόμενό της με τις νέες πληροφορίες.



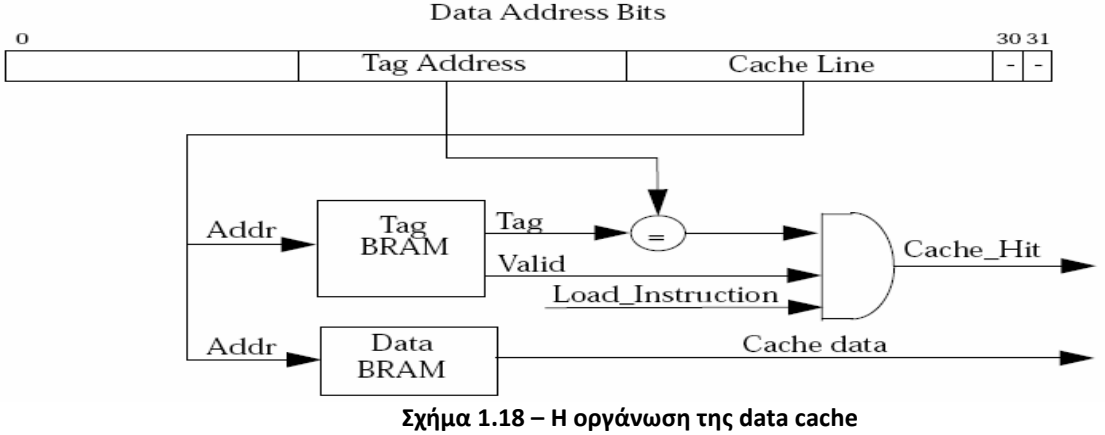
Σχήμα 1.17 – Διαδικασία εγγραφής στην γρήγορη μνήμη

Η ενεργοποίηση ή μη της γρήγορης μνήμης, για τις εντολές, ελέγχεται από το bit 26 του MSR. Τα δεδομένα της γρήγορης μνήμης διατηρούνται ανέπαφα μετά την απενεργοποίησή της, παρ' όλ' αυτά υπάρχει η δυνατότητα μέσω της εντολής

WIC Ra, Rb

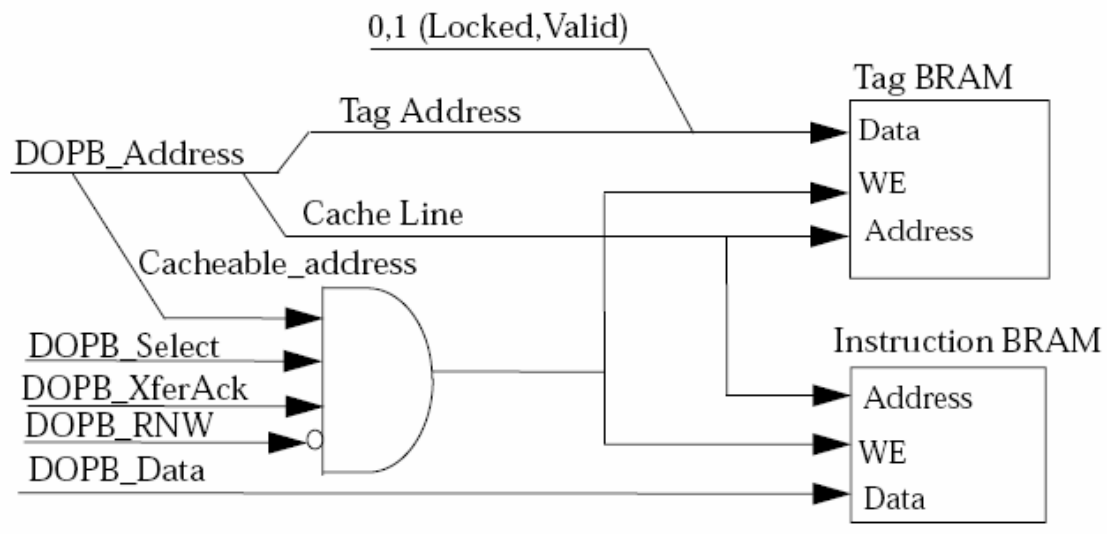
να αλλάξει το περιεχόμενο της γρήγορης μνήμης, στην διάρκεια μόνο που αυτή είναι απενεργοποιημένη. Ο καταχωρητής Rb περιέχει την εντολή. Ο καταχωρητής Ra περιέχει τα τμήματα cache line και tag address, καθώς και το bit εγκυρότητας, Ra(31), με το bit κλειδώματος, RA(30). Το bit κλειδώματος, επιτρέπει την μόνιμη παραμονή μιας εντολής στην γρήγορη μνήμη, με αποτέλεσμα τον εγγυημένο χρόνο εκτέλεσής της. Η συγκεκριμένη τεχνική όμως έχει ως αποτέλεσμα την μείωση του μεγέθους της γρήγορης μνήμης και κατ' επέκταση τον αριθμό των επιτυχημένων αναζητήσεων στην γρήγορη μνήμη (hits). Για το λόγο αυτό είναι προτιμότερο οι εντολές αυτού του τύπου να βρίσκονται στην μνήμη του LMB.

Η οργάνωση και η λειτουργία της γρήγορης μνήμης για τα δεδομένα είναι σχεδόν η ίδια με αυτήν της γρήγορης μνήμης για τις εντολές. Και σε αυτήν την περίπτωση η μνήμη χωρίζεται σε δύο τμήματα, εκ των οποίων το ένα, που οριοθετείται από τις παραμέτρους C_DCACHE_BASEADDR και C_DCACHE_HIGHADDR, περιλαμβάνει διευθύνσεις δεδομένων που είναι δυνατό να βρίσκονται στην γρήγορη μνήμη. Στο τμήμα αυτό δεν πρέπει να περιλαμβάνονται οι διευθύνσεις που αντιστοιχούν στην μνήμη του LMB. Οι διευθύνσεις χωρίζονται σε δύο τμήματα, tag address και cache line, και το μέγεθος της γρήγορης μνήμης κυμαίνεται από 4 Kbytes έως 64 Kbytes.



Σχήμα 1.18 – Η οργάνωση της data cache

Όταν εκτελείται μία εντολή αποθήκευσης δεδομένων, η διαδικασία ακολουθείται κανονικά με την διαφορά ότι εάν η διεύθυνση αποθήκευσης ανήκει στο τμήμα που αναφέρθηκε παραπάνω, τότε γίνεται και ενημέρωση των δεδομένων στην γρήγορη μνήμη. Στην περίπτωση που ο MicroBlaze εκτελεί μια εντολή φόρτωσης δεδομένων, πρώτα ελέγχεται εάν η διεύθυνση ανήκει στην ομάδα των διευθύνσεων που είναι δυνατό να βρίσκονται στην κύρια μνήμη, και μετά ελέγχεται εάν το ζητούμενο δεδομένο βρίσκεται στην γρήγορη μνήμη. Εάν η γρήγορη μνήμη διαθέτει το δεδομένο τότε στέλνει ένα σήμα επιβεβαίωσης στον MicroBlaze και μαζί τα δεδομένα. Αντιθέτως εάν η γρήγορη μνήμη δεν έχει το δεδομένο, αφήνει την διεκπεραίωση της φόρτωσης στον OPB.



Σχήμα 1.19 – Διαδικασία εγγραφής στην γρήγορη μνήμη

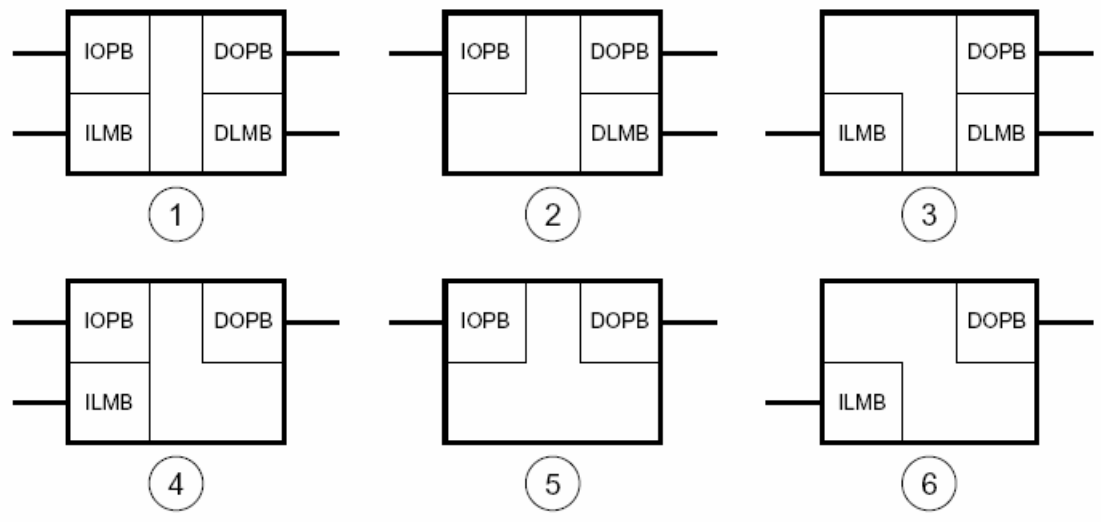
Η ενεργοποίηση και η απενεργοποίηση της γρήγορης μνήμης δεδομένων ελέγχονται από το bit 24 του MSR. Τα δεδομένα διατηρούνται και μετά την απενεργοποίηση της γρήγορης μνήμης, ενώ τον ρόλο της εντολής WIC, που χρησιμοποιείται για την γρήγορη μνήμη εντολών, παίζει η
WDC Ra, Rb
τα ορίσματα της οποίας περιέχουν ότι και τα αντίστοιχα της WIC με την διαφορά ότι ο καταχωρητής Rb περιέχει δεδομένα και όχι εντολές.

1.5.5 Διεπαφή FSL (Fast Simplex Link)

Ο MicroBlaze διαθέτει 16 διεπαφές FSL, οκτώ εισόδου και οκτώ εξόδου. Οι διεπαφές αυτές είναι αποκλειστικές συνδέσεις σημείου προς σημείο μεταξύ του MicroBlaze και των περιφερειακών. Το μήκος του διαδρόμου δεδομένων του FSL είναι 32 bit, τα οποία μπορεί να είναι είτε εντολές ελέγχου, είτε δεδομένα. Ένα ξεχωριστό bit καταδεικνύει πότε μία λέξη που εκπέμπεται ή λαμβάνεται, είναι δεδομένο ή εντολή ελέγχου.
Ο MicroBlaze διαβάζει από μια είσοδο FSL με τέσσερις εντολές, δύο για τα δεδομένα και δύο για τις εντολές ελέγχου. Η εντολή get παγώνει το pipeline του MicroBlaze μέχρις ότου δεδομένα να γίνουν διαθέσιμα στην είσοδο της FSL σύνδεσης. Η εντολή nget επίσης περιμένει δεδομένα στην είσοδο της FSL σύνδεσης αλλά, δεν μπλοκάρει το pipeline. Αντί για αυτό θέτει την τιμή ‘0’ στο bit 29 του MSR όταν τα δεδομένα είναι διαθέσιμα ώστε να διαβαστούν. Και στις δύο εντολές αν επιχειρηθεί να περαστούν εντολές ελέγχου και όχι δεδομένα ενεργοποιείται το bit λάθους του FSL, που αντιστοιχεί στο bit 27 του MSR. Αντίστοιχες ιδιότητες και λειτουργίες με τις εντολές get και nget, έχουν οι εντολές cget και ncget, με την μόνη διαφορά ότι αφορούν αποκλειστικά εντολές ελέγχου και όχι δεδομένα. Οι τέσσερις εντολές ανάγνωσης από την διεπαφή FSL ολοκληρώνονται σε δύο κύκλους ρολογιού από την στιγμή που τα δεδομένα είναι διαθέσιμα. Ανάλογη είναι και η λογική των εντολών που χρησιμοποιούνται για την εγγραφή μέσω μιας σύνδεσης FSL. Υπάρχουν δύο εντολές για την εγγραφή δεδομένων, η put που γράφει μπλοκάροντας το pipeline του MicroBlaze μέχρις ότου τα δεδομένα να μπορέσουν να προωθηθούν, και η nput που δεν μπλοκάρει το pipeline αλλά θέτει την τιμή ‘0’ στο bit 29 του MSR όταν η εγγραφή ολοκληρωθεί επιτυχώς. Και οι δύο εντολές χειρίζονται μονάχα δεδομένα. Για τις εντολές ελέγχου χρησιμοποιούνται οι εντολές cput, που μπλοκάρει το pipeline, και η ncput που δεν το μπλοκάρει και χρησιμοποιεί και αυτή το bit 29 του καταχωρητή MSR. Ομοίως οι δύο παραπάνω εντολές χειρίζονται αποκλειστικά εντολές ελέγχου, ενώ όλες οι εντολές εγγραφής χρειάζονται δύο κύκλους ρολογιού για να ολοκληρωθούν από την στιγμή που τα δεδομένα για εγγραφή είναι διαθέσιμα.

1.5.6 Διάδρομοι Δεδομένων του MicroBlaze

Οι δύο κύριοι διάδρομοι δεδομένων του MicroBlaze, είναι ο Local Memory Bus (LMB), ο οποίος παρέχει πρόσβαση σε μία dual-port μνήμη BRAM που είναι προσπελάσιμη σε ένα κύκλο ρολογιού, και ο On-chip Peripheral Bus (OPB), που παρέχει σύνδεση με εσωτερικά και εξωτερικά περιφερειακά και μνήμη. Οι LMB και OPB χωρίζονται περαιτέρω στους DLMB, Data interface LMB, DOPB, Data interface OPB, που χειρίζονται μόνο δεδομένα και στους ILMB, Instruction interface LMB, IOPB, Instruction interface OPB, που χειρίζονται μόνο εντολές. Εδώ πρέπει να τονιστεί ότι όλα τα περιφερειακά συνδέονται στον DOPB. Εκτός από τους LMB και OPB ο MicroBlaze διαθέτει και οκτώ κύριες (Master) και οκτώ εξαρτημένες (Slave) διεπαφές, για συνδέσεις FSL.
Η δομή του MicroBlaze όσον αφορά τους διαδρόμους δεδομένων είναι δυνατόν να διαμορφωθεί με έξι διαφορετικούς τρόπους, όπως φαίνεται στο σχήμα 1.20 και στο σχήμα 1.21. Σε αυτούς τους έξι τρόπους δεν συνυπολογίζεται η χρησιμοποίηση ή μη των συνδέσεων FSL.



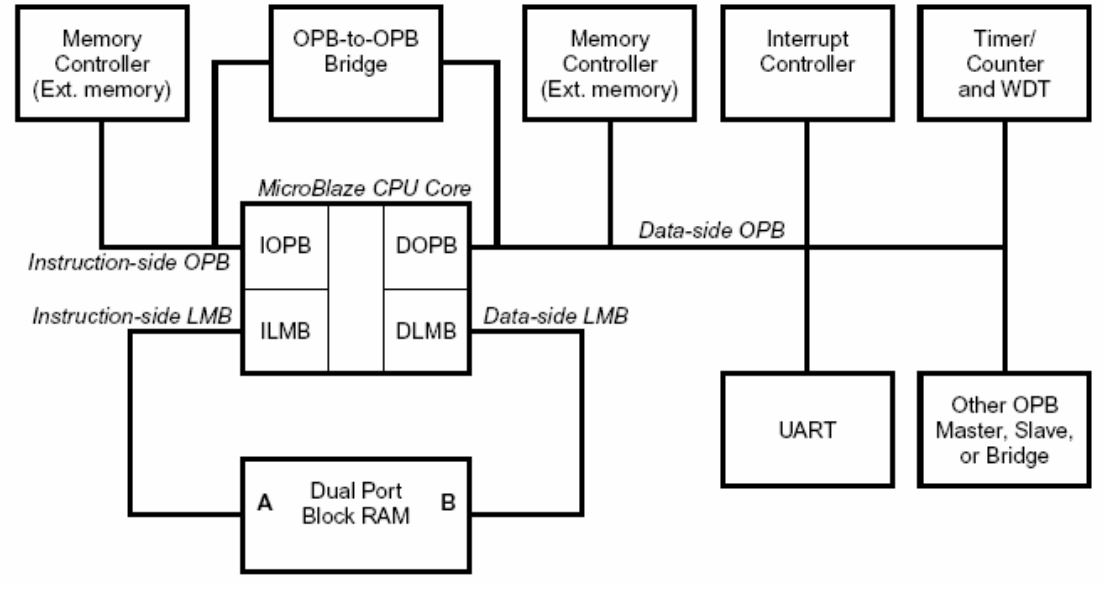
Σχήμα 1.20 – Οι έξι τρόποι διαμόρφωσης των διαδρόμων δεδομένων του MicroBlaze

Η επιλογή κάθε ενός από τους έξι τρόπους διαμόρφωσης εξαρτάται από το μέγεθος του εκτελέσιμου κώδικα, την ανάγκη σε χώρο για αποθήκευση δεδομένων και στην απαίτηση για γρήγορη πρόσβαση στην εσωτερική μνήμη BRAM.

	Διαμόρφωση	Μέγιστη Συχνότητα	Υποστηριζόμενο Μοντέλο Μνήμης	Τυπικές Εφαρμογές
1	IOPB+ILMB+DOPB+DLMB	110	Μεγάλη εξωτερική μνήμη Γρήγορη εσωτερική μνήμη	Για εφαρμογές που χρειάζονται μνήμη μεγαλύτερη από αυτή που προσφέρει η BRAM. MPEG αποκωδικοποιητές, ελεγκτές επικοινωνιών
2	IOPB+DOPB+DLMB	125	Μεγάλη εξωτερική μνήμη Γρήγορη εσωτερική μνήμη για δεδομένα (BRAM)	Παρόμοιες με την περίπτωση 1 Επιτυγχάνεται μεγαλύτερη συχνότητα αλλά η λειτουργία είναι πιο αργή λόγω απουσίας ILMB
3	ILMB+DOPB+DLMB	125	Μεγάλη εξωτερική μνήμη για δεδομένα Γρήγορη εσωτερική μνήμη (BRAM)	Για εφαρμογές που αρκούνται στην μνήμη BRAM για τον εκτελέσιμο κώδικα Μικρές ή μεσαίες μηχανές καταστάσεων
4	IOPB+ILMB+DOPB	110	Μεγάλη εξωτερική μνήμη Γρήγορη εσωτερική μνήμη για εντολές (BRAM)	Παρόμοιες με την περίπτωση 1
5	IOPB+DOPB	125	Μεγάλη εξωτερική μνήμη	Παρόμοιες με την περίπτωση 2
6	ILMB+DOPB	125	Μεγάλη εξωτερική μνήμη για δεδομένα Γρήγορη εσωτερική μνήμη για εντολές (BRAM)	Εφαρμογές όπου η BRAM είναι αρκετή για τον εκτελέσιμο κώδικα, αλλά όχι για τα δεδομένα Μικροί ελεγκτές, μικρές ή μεσαίες μηχανές καταστάσεων

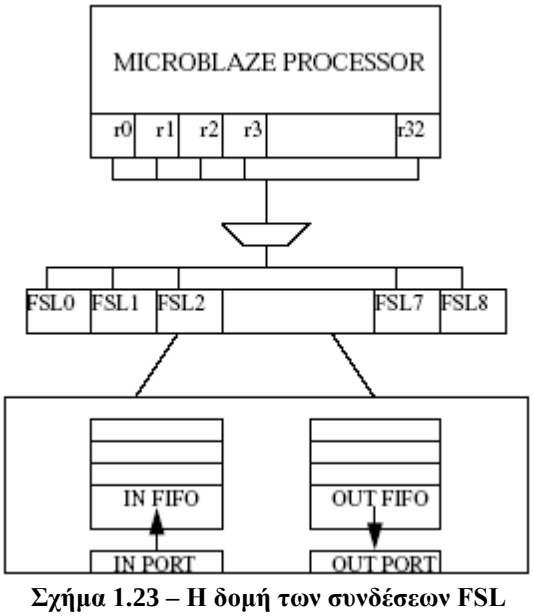
Σχήμα 1.21 – Οι έξι διαμορφώσεις των διαδρόμων δεδομένων του MicroBlaze

Το σχήμα 1.22 απεικονίζει την πρώτη περίπτωση διαμόρφωσης, από την οποία βέβαια απορρέουν και οι υπόλοιπες.



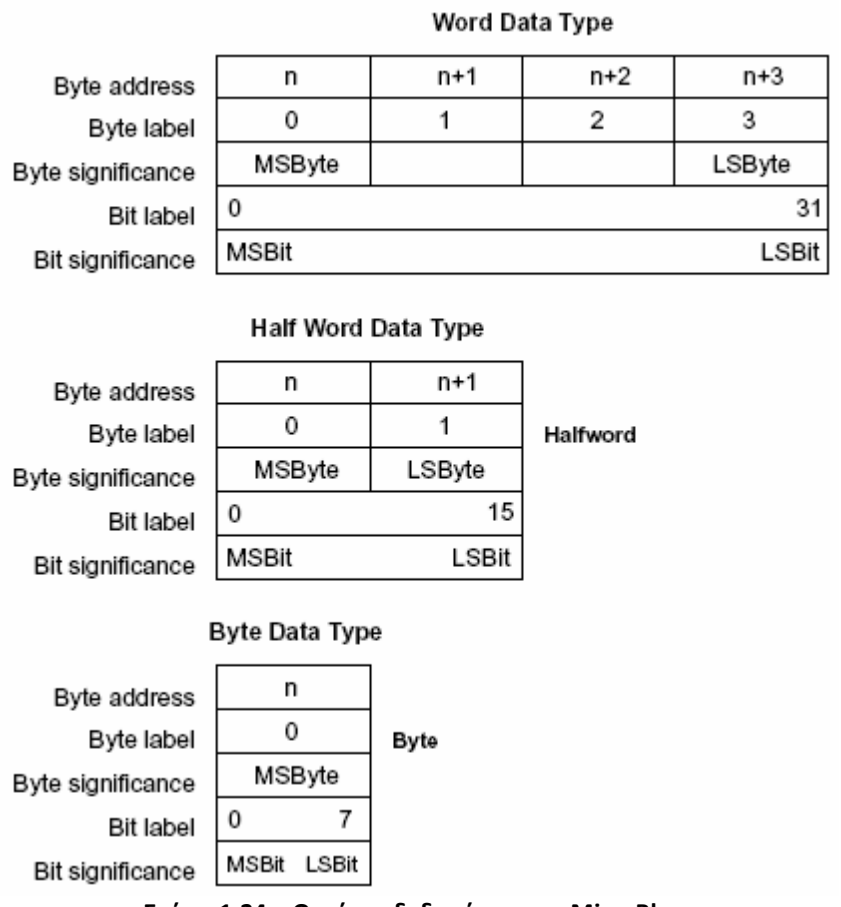
Σχήμα 1.22 – Η πρώτη περίπτωση διαμόρφωσης

Όπως αναφέρθηκε παραπάνω είναι δυνατόν σε κάθε έναν από τους έξι τρόπους διαμόρφωσης να υπάρχουν και συνδέσεις FSL μεταξύ του MicroBlaze και των περιφερειακών, που καθιστούν την επικοινωνία ταχύτερη και είναι ιδανικές για εφαρμογές επεξεργασίας σημάτων και ελέγχου δικτύων.



Σχήμα 1.23 – Η δομή των συνδέσεων FSL

Οι διάδρομοι δεδομένων του MicroBlaze μεταφέρουν δεδομένα τριών τύπων (σε παρένθεση οι αντιστοιχίσεις αυτών με assembly και C), word (data8 και char), half word (data16 και short, pointer) και byte (data32 και int, long int, enum, pointer). Οι τύποι δεδομένων και η συμβάσεις ονοματολογίας που χρησιμοποιούνται από τους διαδρόμους δεδομένων του MicroBlaze φαίνονται στο σχήμα 1.24.



Σχήμα 1.24 – Οι τύποι δεδομένων του MicroBlaze

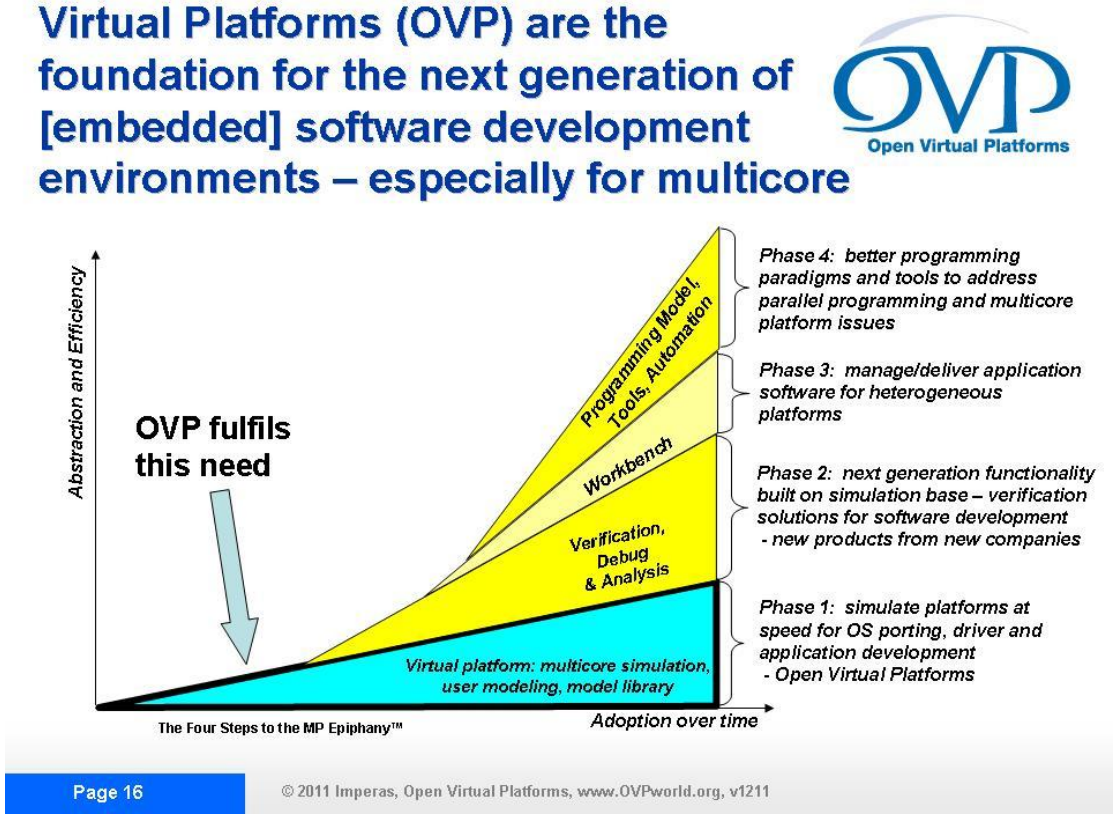
Κεφάλαιο 2

Open Virtual Platforms

2.1.Εισαγωγή- Γενικά για το OVP

Το σκεπτικό που βρίσκεται στο επίκεντρο του OVP είναι να επιταχυνθεί η υιοθέτηση ενός νέου τρόπου ανάπτυξης ενσωματωμένων συστημάτων - ειδικά για System-On-Chip (SoC) και MultiProcessor System-On-Chip (MPSoC) πλατφόρμες. Πολλές φορές, ωστόσο, η ανάγκη για πολλαπλούς επεξεργαστές ή πυρήνες στην σχεδίαση κρίνεται επιτακτική. Συνεπώς, αυτό που χρειάζεται είναι το μοντέλο της πλατφόρμας που χρησιμοποιούμε για το σύστημα μας να περιλαμβάνει όχι μόνο τα μοντέλα όλων των επεξεργαστών ή πυρήνων που χρειαζόμαστε, αλλά και την περιγραφή των περιφερειακών και της συμπεριφοράς τους, ώστε να πραγματοποιείται η επικοινωνία τους με το software. Αυτή είναι η έννοια της εικονικής πλατφόρμας, ή πιο απλά ένα μοντέλο προσομοίωσης της σχεδίασής μας.

Το OVP, λοιπόν, παρέχει βιβλιοθήκες του επεξεργαστή με πρότυπα συμπεριφοράς και Διεπαφή Προγραμματισμού Εφαρμογών (Application Programming Interface-APIs), επιτρέποντας τον σχεδιασμό σε software του δικού μας επεξεργαστή, με συγκεκριμένα περιφερειακά στην κατάλληλη πλατφόρμα. Όλα αυτά είναι ακριβώς ό, τι χρειάζεται κανείς για να χρησιμοποιήσει τα υπάρχοντα μοντέλα ή να δημιουργήσει το σύστημα της επιλογής του. Το OVP είναι αρκετά εύκολο στη χρήση, ανοικτό , ευέλικτο, και το σημαντικότερο, δωρεάν για μη εμπορική χρήση.



Σχήμα 2.1- OVP

Τα βασικότερα χαρακτηριστικά του OVP συγκεντρώνονται παρακάτω:

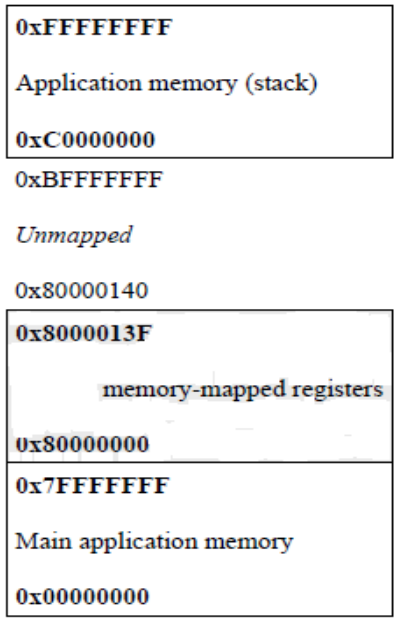
- Εύκολη δημιουργία εικονικής πλατφόρμας πολλών περιφερειακών και πολλών επεξεργαστών.
- Εύκολη δημιουργία των δικών μας επεξεργαστών, περιφερειακών και πλατφόρμας.
- Βιβλιοθήκη με μοντέλα επεξεργαστών και περιφερειακών.
- Μοντελοποίηση μόνο όσων χρειάζονται στα περιφερειακά.
- Οι προσομοιώσεις είναι Instruction Accurate, υψηλής ταχύτητας.
- Αποτελεσματικό, ολοκληρωμένο περιβάλλον για embedded software development.

2.2. Περιγραφή δημιουργίας ενσωματωμένου συστήματος

Σύμφωνα με τις ανάγκες της εργασίας, χρησιμοποιήσαμε το OVP, ως το καταλληλότερο εργαλείο ανάπτυξης του δικού μας συστήματος. Οι βιβλιοθήκες, τα μοντέλα επεξεργαστών, περιφερειακών και τα παραδείγματα που μας παρέθεσε, βοήθησαν καθοριστικά στη δημιουργία μίας εικονικής πλατφόρμας με επεξεργαστή τον Microblaze της Xilinx και ένα περιφερειακό με συγκεκριμένη λειτουργία. Η δουλειά που έγινε ήταν αφού επιλεγεί η εφαρμογή του συστήματος μας, να εντοπίσουμε μετά από profiling τις πιο βαριές συναρτήσεις της και να τις υλοποιήσουμε σε hardware βάζοντάς τες στο περιφερειακό. Τα βήματα που ακολουθήσαμε για να χτίσουμε το δικό μας σύστημα περιγράφονται αναλυτικότερα στη συνέχεια.

2.2.1 Σχεδιασμός εικονικής πλατφόρμας

Αρχικά, έγινε η επιλογή του επεξεργαστή που θα χρησιμοποιήσουμε στην εικονική πλατφόρμα. Επειδή η πραγματική πλακέτα που θα χρησιμοποιήσουμε θα έχει τον Microblaze της Xilinx για επεξεργαστή, για να γίνει επιτυχής η σύνδεση της με την εικονικής, διαλέξαμε να βάλουμε τον ίδιο επεξεργαστή και στις δύο πλατφόρμες. Εν συνεχεία, χωρίσαμε την μνήμη σε διαστήματα ώστε να υπάρχει χώρος για τους καταχωρητές, το περιφερειακό, αλλά και την εφαρμογή.



Σχήμα 2.2- Memory Map

Όσων αφορά στον κώδικα, δημιουργήσαμε τον δίαυλο δεδομένων πάνω στον οποίο θα συνδεθούν τα διάφορα μέρη για να επικοινωνούν μεταξύ τους.

```
// create the processor bus
  icmBusP bus = icmNewBus("busMain", 32);
```

Στην συνέχεια, ακολούθησε η δημιουργία των συσκευών μνήμης και η σύνδεσή τους με τον δίαυλο. Οι διευθύνσεις βάσης των μνημών καθορίζονται όταν συνδέονται με το δίαυλο, ενώ το μέγεθος τους ορίζεται, όταν αυτές δημιουργηθούν.

```
// create two memory regions mapping all memory except the DMAC registers
    icmMemoryP mem1 = icmNewMemory("mem1", ICM_PRIV_RWX, 0x3fffffff);
    icmMemoryP mem2 = icmNewMemory("mem2", ICM_PRIV_RWX, 0x7fffffff);


    // connect memories to the bus
    icmConnectMemoryToBus(bus, "sp", mem1, 0xc0000000);
    icmConnectMemoryToBus(bus, "sp", mem2, 0x00000000);
```

Δημιουργούμε τον επεξεργαστή που θα χρησιμοποιήσουμε, δηλαδή τον Microblaze:

```
// create a processor instance
    icmProcessorP cpu1_c = icmNewProcessor(
        "cpu0",             // CPU name
        "microblaze",       // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                  // address bits
        microblazeModel,    // model file
        "modelAttrs",       // morpher attributes
        SIM_ATTRS,          // attributes
        cpu1_attr,          // user-defined attributes
        microblazeSemihost, // semi-hosting file
        "modelAttrs"        // semi-hosting attributes
  );
```

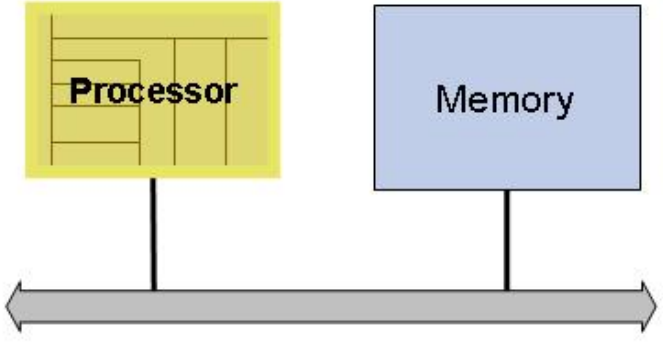
Συνδέουμε τον Microblaze με τον δίαυλο:

```
// connect the processor instruction and data busses to the bus
   icmConnectProcessorBusses(cpu1_c, bus, bus);
```

Μετα φορτώνουμε την εφαρμογή στη μνήμη :

```
// load the application executable file into processor memory space
   if(!icmLoadProcessorMemory(cpu1_c, appName, False, False, True)) {
      return False;
   }
```

Ουσιαστικά, μέχρι αυτό το σημείο έχει δημιουργηθεί ο επεξεργαστής, η μνήμη και ο δίαυλος που τα συνδέει:



Σχήμα 2.3- Connecting Bus

Αυτό που λείπει είναι να δηλώσουμε το περιφερειακό στην πλατφόρμα και να το συνδέουμε στο bus.

```
// instantiate the peripheral
   icmPseP dmac = icmNewPSE("dmac", "pse.pse", NULL, NULL, NULL);

 // connect the peripheral slave port on the bus and define the address range it occupies
   icmConnectPSEBus(dmac, bus, "DMACSP", False, 0x80000000, 0x8000013f);
```

2.2.2 Σχεδιασμός περιφερειακού

Η δημιουργία του περιφερειακού ήταν αρκετά πιο πολύπλοκη από αυτήν της πλατφόρμας με τον επεξεργαστή. Αρχικά, έπρεπε να καθοριστεί η δομή και οι πόροι που θα χρησιμοποιεί το περιφερειακό, όμως για να το κάνουμε αυτό έπρεπε πρώτα να καθορίσουμε την κύρια λειτουργία που θα εκτελεί. Επιλέξαμε, λοιπόν, για αρχή μία εύκολη εφαρμογή, η οποία υπολογίζει και εμφανίζει έναν αριθμό Fibonacci. Στο σημείο αυτό αξίζει να προσθέσουμε ότι το περιφερειακό που θα δημιουργήσουμε πρέπει να περιλαμβάνει μονάχα ένα thread, καθώς για να μπορέσουμε να περάσουμε τον κώδικα από High Level Synthesis η περιφερειακή μονάδα πρέπει να είναι single-threaded.

Το περιφερειακό, γενικά, μπορεί να είναι μία συσκευή slave, η οποία είναι συνδεδεμένη με τον δίαυλο και εκθέτει τους καταχωρητές της ή τη μνήμης της ώστε να μπορούν να προσπελαστούν από τον δίαυλο αυτόν. Οι προσπελάσεις σε αυτή την περιοχή που εκτίθεται, ελέγχουν την συμπεριφορά του περιφερειακού και, συγχρόνως, επιστρέφουν πληροφορίες για την κατάσταση του. Τυπικά, αυτή η επικοινωνία μπορεί να επιτευχθεί μέσω καταχωρητών αποθηκευμένων στην μνήμη, οι οποίοι διοχετεύουν και επιστρέφουν πληροφορίες σχετικά με το περιφερειακό.

i.Παράθυρο Μνήμης

Για να δημιουργήσουμε την διεπαφή τύπου slave του περιφερειακού με τον δίαυλο πρέπει να δημιουργήσουμε μία περιοχή στη μνήμη όπου θα αποθηκευτούν οι καταχωρητές που θα χρησιμοποιεί το περιφερειακό. Συμπερασματικά η θύρα του περιφερειακού slave θα καταλαμβάνει ένα παράθυρο μνήμης στο χώρο διευθύνσεων του συστήματος. Το μέγεθος του παραθύρου μνήμης ορίζεται ως τμήμα του περιφερειακού και η θέση ορίζεται ως μέρος του σχεδιασμού στον οποίο το περιφερειακό χρησιμοποιείται.

Το παράθυρο προϋποθέτει ότι υπάρχει μια περιοχή αποθήκευσης που ορίζεται εντός του περιφερειακού. Αυτή η περιοχή είναι η πηγή και ο προορισμός όλων των προσβάσεων τύπου σκλάβου στο περιφερειακό και στον κώδικα δηλώνεται με αυτόν τον τρόπο:

```
static unsigned char DMACSP_Window [0x140];
```

Το παράθυρο μνήμης στο περιφερικό εκτίθεται στο υπόλοιπο σύστημα κατά το άνοιγμα ενός διαύλου θύρας τύπου slave χρησιμοποιώντας την PPM ppmOpenSlaveBusPort εντολή ως εξής:

```
static void installSlavePorts(void) {
   handles.DMACSP = ppmCreateSlaveBusPort("DMACSP", 320);
   if (!handles.DMACSP) {
      bhmMessage("E", "PPM_SPNC", "Could not connect port 'DMACSP'");
   }
```

Η θύρα αυτή επιτυγχάνει την σύνδεση μεταξύ της πλατφόρμας και του software του περιφερειακού και επιτρέπει την πρόσβαση στην μνήμη αυτή.

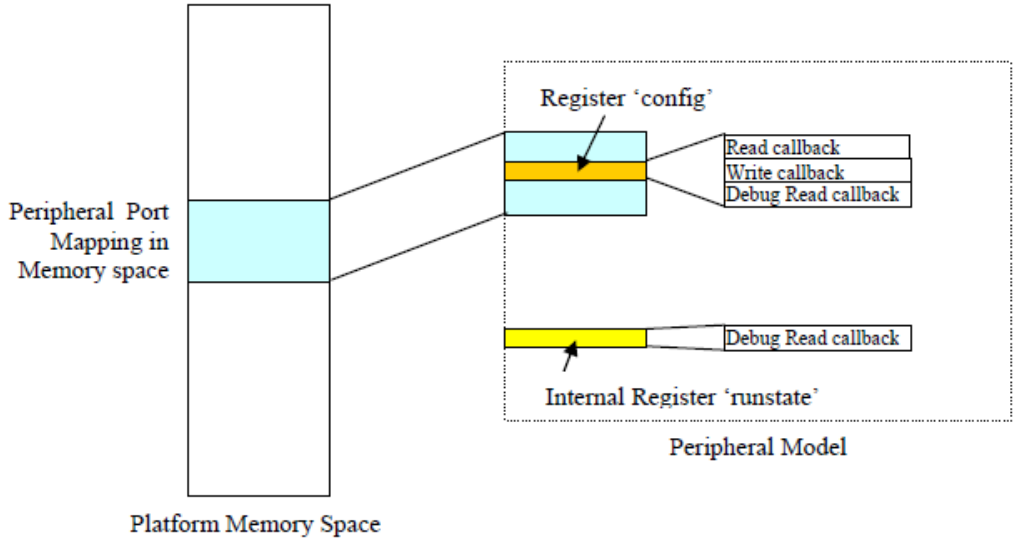
ii. Καταχωρητές

- Μια εγγραφή στο εκτεθειμένο παράθυρο του περιφερικού από έναν δίαυλο θα έχει ως αποτέλεσμα την αποθήκευση της τιμής που έχουν στον πίνακα μνήμης.
- Μια ανάγνωση από το παράθυρο θα επιστρέψει την τιμή από την εκτιθέμενη μνήμη που αντιστοιχεί στη διεύθυνση πρόσβασης.

Όταν μια ανάγνωση ή εγγραφή στο παράθυρο μνήμης συγκεκριμένης διεύθυνσης, απαιτεί εκτός των άλλων λειτουργικότητα, για παράδειγμα, μια εγγραφή σε έναν καταχωρητή ελέγχου, τότε προσθέτουμε συναρτήσεις επανάκλησης (callback functions), οι οποίες πραγματοποιούν την επιθυμητή λειτουργία.

Κατά τη δημιουργία ενός καταχωρητή σε ένα περιφερειακό μοντέλο, η συνιστώμενη προσέγγιση είναι να χρησιμοποιήσετε τις συναρτήσεις API ppmCreateRegister ή ppmCreateInternalRegister. Και οι δύο συναρτήσεις δημιουργούν καταχωρητές στο περιφερειακό μοντέλο, οι οποίοι μάλιστα, όταν χρησιμοποιηθούν για read ή write, προκαλούν ένα "γεγονός", το οποίο είναι ικανό να "ξυπνήσει" το περιφερειακό που περιμένει.

Η παρακάτω εικόνα δείχνει δύο είδη καταχωρητών εκ των οποίων το πρώτο είναι προσβάσιμο μέσω του παραθύρου μνήμης του περιφερειακού, ενώ το άλλο είναι προσβάσιμο μονάχα από το ίδιο το περιφερειακό.



Σχήμα 2.4- Προσβάσιμοι και μη Προσβάσιμοι Καταχωρητές στο Περιφερειακό

Σύμφωνα με τα παραπάνω δημιουργήσαμε τους αναγκαίους καταχωρητές ως εξής:

```
ppmCreateRegister("srcAddr",
    "channel 0 source address",
    handles.DMACSP,
    256,
```

```
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch0_data.srcAddr.value),
    True
  );
```

Όπου ο καταχωρητής που δημιουργούμε βρίσκεται στην θέση της βάσης της μνήμης που φτιάξαμε πριν συν 256 θέσεις και καταλαμβάνει 4 bytes. Επίσης, οι λειτουργίες που θα πραγματοποιηθούν όταν γράψουμε (regWr32) ή διαβάσουμε τους καταχωρητές (regRd32) καθορίζονται καθαρά και μόνο από το περιεχόμενο των callback συναρτήσεων.

Για παράδειγμα, η callback function του regWr32 είναι:

```
PPM_REG_WRITE_CB(regWr32) {
        printf("im in the callback data/n");
bhmTriggerEvent(DMAState.ch[channel].start);
 *(Uns32*)user =  byteSwap(data);
}
```

Και οι λειτουργίες που κάνει, ουσιαστικά ενεργοποιούν το περιφερειακό, το οποίο αρχίζει να εκτελείται μέχρι να ξανασυναντήσει κάποια εντολή που θα το "παγώσει", και γράφουν στην μνήμη του περιφερειακού την τιμή που περιέχει.

iii.Νήματα-Threads

Ένα αρχικό νήμα δημιουργείται από την κύρια ρουτίνα main () εντός του περιφερειακού. Τυχόν επιπλέον νήματα που απαιτούνται εντός του περιφερειακού δημιουργούνται χρησιμοποιώντας την συνάρτηση bhmCreateThread.

Μία περιοχή που θα χρησιμοποιείται ως στοίβα για κάθε επιπλέον νήμα πρέπει να οριστεί. Στο περιφερειακό επίσης, ορίζεται και μία δομή που περιλαμβάνει έναν πίνακα χαρακτήρων για κάθε νήμα, και χειρίζεται το νήμα όταν δημιουργείται και όταν γίνεται κάποιο συμβάν.

```
#define THREAD_STACK (8*1024)
typedef struct {
bhmThreadHandle thread;
bhmEventHandle start;
Bool busy;
char stack[THREAD_STACK];
} channelState;
```

Κάθε ένα από τα κανάλια δημιουργούνται με τον ίδιο τρόπο. Ένα συμβάν ορίζεται στη δομή που θα επιτρέψει στο κανάλι να ξεκινήσει και το νήμα να δημιουργηθεί.

```
// Create threads for the channels
for (i=0; i<NUM_CHANNELS; i++) {
// Event to start the thread
DMAState.ch[i].start = bhmCreateEvent();
DMAState.ch[i].busy = False;
sprintf(threadName, "ch%u", i);
DMAState.ch[i].thread = bhmCreateThread(
channelThread,
(void*) i,
threadName,
&DMAState.ch[i].stack[THREAD_STACK] // top of downward
growing stack
);
}
```

Τα γεγονότα που ενεργοποιούν ένα νήμα αποσκοπούν στον καλύτερο συγχρονισμό και έλεγχο των threads, καθώς όταν είναι πολλά πρέπει με κάποιο τρόπο η σειρά εκτέλεσής τους να είναι ελεγχόμενη.

Αρκεί λοιπόν για την δημιουργία ενός event να γράψουμε

```
// Event to start the thread
DMAState.ch[i].start = bhmCreateEvent();
```

Και για την αναμονή ενός event αρκεί:

```
static void channelThread(void *user)
{
Uns32 ch = (Uns32) user;
for (;;) {
… code …
bhmWaitEvent(DMAState.ch[ch].start);
… code …
}
}
```

Όπως προαναφέρθηκε, η κυρίως λειτουργία του περιφερειακού είναι ο υπολογισμός ενός αριθμού Fibonacci. Συνεπώς, το νήμα μόλις ξυπνήσει καλεί μία συνάρτηση, την dmaburst(); όπου και γίνεται ο ζητούμενος υπολογισμός, όπως φαίνεται από τον παρακάτω κώδικα:

```
static Uns32 dmaBurst(Uns32 ITERATIONS){
   int prev = -1;
   int result = 1;
   int sum;
            int i;
            for (i=0; i<=ITERATIONS; ++i) {
                        sum = result + prev;
      prev = result;
      result = sum;
            }
return result;}
```

2.2.3 Σχεδιασμός εφαρμογής

Η εφαρμογή που δημιουργήσαμε για να τρέξει στην πλατφόρμα μας το μόνο που κάνει είναι να ενεργοποιεί το περιφερειακό ώστε να πραγματοποιηθεί ο υπολογισμός του ζητούμενου αριθμού Fibonacci. Ο τρόπος που ενεργοποιεί το περιφερειακό είναι γράφοντας σε έναν καταχωρητή που ανήκει στο παράθυρο μνήμης που ορίσαμε πριν για το περιφερειακό.

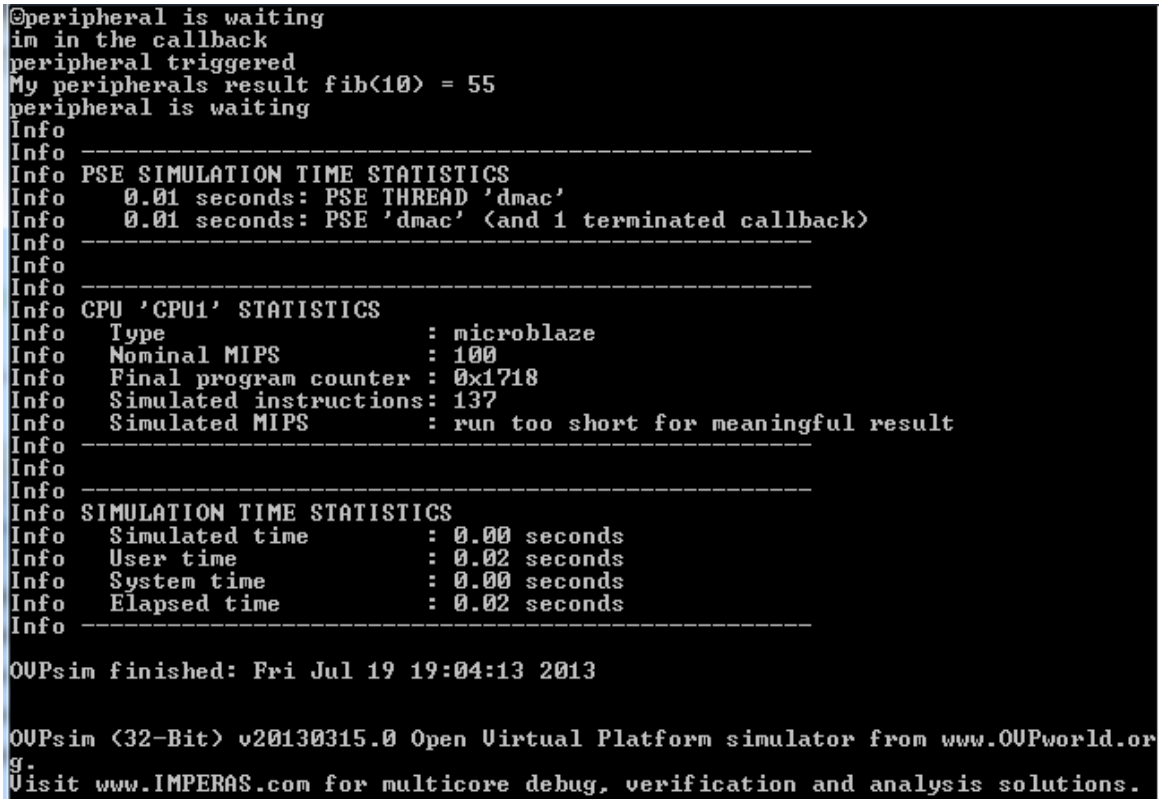
```
writeReg32(DMA_BASE, DMA_C0_CONFIGURATION, 10);
```

Έτσι καλείται μια callback function η οποία παράγει ένα event ξυπνώντας έτσι το περιφερειακό. Ο κώδικας της εφαρμογής θα παρουσιαστεί μαζί με τα υπόλοιπα προγράμματα της προσομοίωσης στην ενότητα Παράρτημα.

2.3. Προσομοίωση- Αποτελέσματα- Συγκρίσεις

Για να γίνει η προσομοίωση σε λειτουργικό Windows 7 χρειάστηκε να κατεβάσουμε το MinGW environment. Αυτό το περιβάλλον χρησιμοποιεί τα Imperas tools και το OVPsim ώστε να επιτευχθεί η δημιουργία του συστήματος που σχεδιάσαμε.

Έτσι, λοιπόν, αφού κάναμε compile στα προγράμματα, δοκιμάσαμε να τα τρέξουμε και πατώντας στο τερματικό ./platform.Windows32.exe dmaTest.elf, πήραμε τα εξής αποτελέσματα:



Σχήμα 2.5- Integration in Hardware

Η πρώτη προσομοίωση που παρουσιάζεται στο προηγούμενο σχήμα αντιπροσωπεύει την υλοποίηση της εφαρμογής μας σε hardware, δηλαδή στο περιφερειακό.

Όπως φαίνεται από την προσομοίωση παρατηρούμε ότι το περιφερειακό, μόλις δεχθεί σήμα από την εφαρμογή για να ξεκινήσει, υπολογίζει και εμφανίζει τον δέκατο αριθμό Fibonacci που είναι ο 55.

Εν συνεχεία, παρουσιάζει κάποιες χρήσιμες πληροφορίες όπως:

- Final program counter - Simulated instructions: Όπως εξηγήσαμε στα παραπάνω το OVP Simulation μας παρέχει instruction accurate πληροφορίες και μας δίνει τον program counter

- Simulated time - System time - Elapsed time: Παρέχει επίσης και πληροφορίες για τους χρόνους εκτέλεσης του συστήματος.

Από αυτές τις πληροφορίες, παρατηρούμε το αποτέλεσμα των εκτελούμενων εντολών είναι Hardware Simulated Instructions = 137.

Κρατάμε αυτή την τιμή, καθώς τώρα θα επιχειρήσουμε να σχεδιάσουμε την εφαρμογή εξολοκλήρου σε software και θα πρέπει να συγκρίνουμε την αντίστοιχη τιμή της νέας προσομοίωσης. Αξίζει να σημειώσουμε σε αυτό το σημείο ότι αν χρησιμοποιούσαμε μία πιο βαριά εφαρμογή θα χρησιμοποιούσαμε και άλλες παραμέτρους που μας παρέχει η προσομοίωση για συγκρίσεις.

2.4 Σύγκριση με υλοποίηση της εφαρμογής ως software - profiling

Όπως επισημάνθηκε και στην εισαγωγή, το περιφερειακό εκτελεί τις πιο βαριές συναρτήσεις της εφαρμογής έτσι ώστε να προσθέσει ταχύτητα στο σύστημα. Αυτή η διαδικασία είναι από τις σημαντικότερες κατά την σχεδίαση ενσωματωμένων συστημάτων και ονομάζεται profiling. Οι σύγχρονοι σχεδιαστές, λοιπόν, πρωτού περάσουν στην πραγματική υλοποίηση του συστήματος που θέλουν να σχεδιάσουν, δημιουργούν μία εικονική πλατφόρμα και δοκιμάζουν διάφορους τρόπους σχεδίασης επιλέγοντας πάντα τον πιο αποδοτικό. Η επικρατούσα ιδέα καλής σχεδίασης είναι η εξής: στην εικονική προσομοίωση του συστήματος εντοπίζονται οι πιο πολύπλοκες και χρονοβόρες συναρτήσεις της εφαρμογής. Αυτές απομονώνονται από τις υπόλοιπες και εν συνεχεία υλοποιούνται ως hardware μέσα σε ένα περιφερειακό της πλατφόρμας. Με αυτόν τον τρόπο εξασφαλίζεται η βέλτιστη υλοποίηση με τον καλύτερο δυνατό χρόνο εκτέλεσης.

Στην προκειμένη περίπτωση, δοκιμάσαμε την εφαρμογή υπολογισμού του δέκατου αριθμού Fibonacci να υλοποιηθεί εξολοκλήρου ως software, καθώς η εφαρμογή δεν αποτελείται από άλλες συναρτήσεις. Ουσιαστικά το περιφερειακό δεν κάνει κάποιο υπολογισμό.

Η προσομοίωση έδωσε τα παρακάτω αποτελέσματα:

```
©peripheral is waiting
My software result fib(10) = 55
im in the callback
peripheral triggered
peripheral is waiting
Info
Info ------------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info     0.00 seconds: PSE THREAD 'dmac'
Info     0.01 seconds: PSE 'dmac' (and 1 terminated callback)
Info ------------------------------------------------------
Info
Info ------------------------------------------------------
Info CPU 'CPU1' STATISTICS
Info   Type                  : microblaze
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1848
Info   Simulated instructions: 4,736
Info   Simulated MIPS        : run too short for meaningful result
Info ------------------------------------------------------
Info
Info ------------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.03 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.03 seconds
Info ------------------------------------------------------

OVPsim finished: Fri Jul 19 19:00:46 2013


OVPsim (32-Bit) v20130315.0 Open Virtual Platform simulator from www.OVPworld.or
g.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.
```

Σχήμα 2.6- Integration in Software

Παρατηρούμε, λοιπόν σε αυτό το σημείο ότι οι εντολές που εκτελέστηκαν είναι :

Software Simulated Instructions = 4.736

Όπως φαίνεται και από την προσομοίωση στο OVP η υλοποίηση αυτή δεν είναι συμφέρουσα και για αυτό τον λόγο απορρίφθηκε. Αυτή είναι και η διαδικασία που ακολουθείται στην αγορά, συγκρίνοντας βέβαια πιο πολλές παραμέτρους καθώς οι εφαρμογές είναι μεγαλύτερες και πιο απαιτητικές.

2.5 Η Εφαρμογή JPEG2000

Μια πιο μεγάλη εφαρμογή με την οποία ασχοληθήκαμε είναι ο JPEG2000, ένα πρότυπο συμπίεσης εικόνας και σύστημα κωδικοποίησης που δημιουργήθηκε με την πρόθεση να αντικατασταθεί το βασισμένο στον μετασχηματισμό συνημιτόνου (discrete cosine transform) πρότυπο JPEG με μία νέα μέθοδο σχεδίασης κυμάτων (Discrete Wavelet Transform).

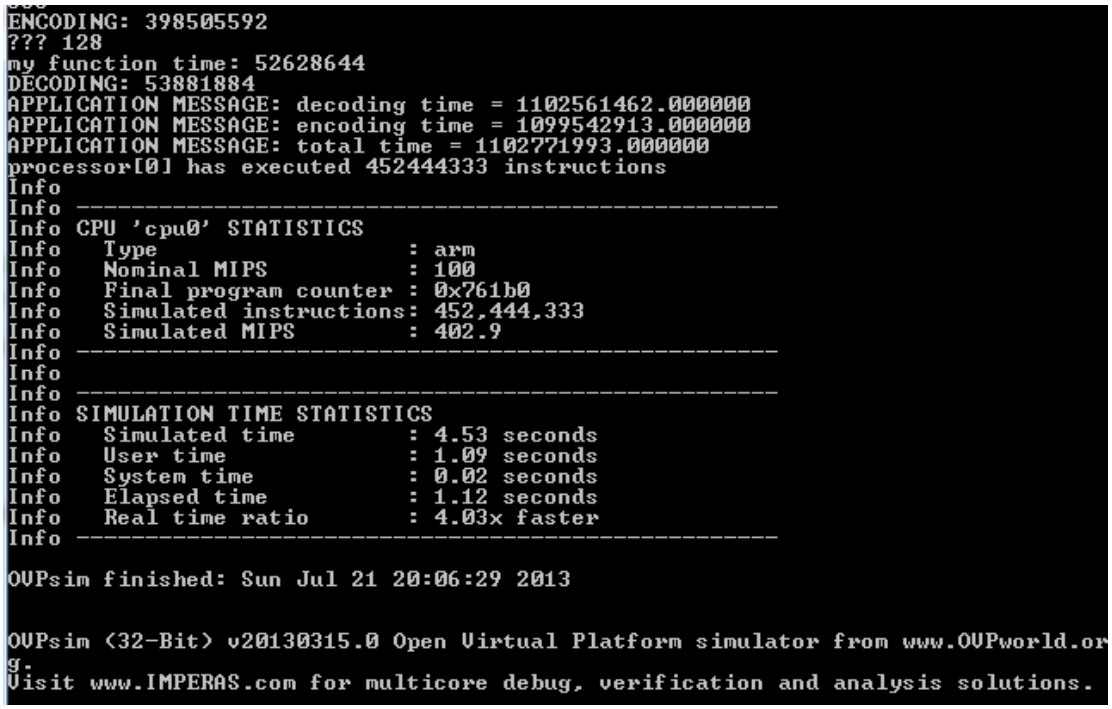
Έτσι λοιπόν, εφαρμόσαμε μια στρατηγική διαχωρισμού του software και του hardware σε μία αρκετά μεγαλύτερη εφαρμογή από την Fibonacci, ώστε να αποδώσουμε καλύτερα την μεθοδολογία που ακολουθείται και δεν μπορέσαμε να αποτυπώσουμε πλήρως στο προηγούμενο παράδειγμα.

Η μεθοδολογία που οδήγησε σε μια συγκεκριμένη στρατηγική διαχωρισμού υλικού/ λογισμικού παρουσιάζεται, περιλαμβάνοντας την ανάλυση προδιαγραφών και χαρακτηριστικών του Τζάσπερ, ενός open-source software-based για την εφαρμογή του κωδικοποιητή JPEG2000. Ένα λεπτομερές σύνολο των χαρακτηριστικών χρονισμού παρουσιάζεται για τον κώδικα του Jasper. Η ανάλυση αυτών των χαρακτηριστικών οδήγησε στην απόφαση για την επιλογή του Inverse Discrete Wavelet Transform για την εφαρμογή του υλικού.

2.5.1 Διαχωρισμός Hardware/Software

Όπως γνωρίζουμε, οι βασικοί υπολογισμοί της εφαρμογής αυτής είναι η κωδικοποίηση και η αποκωδικοποίηση. Αρχικά, προσπαθήσαμε να βρούμε ποια από τις δύο διαδικασίες είναι πιο χρονοβόρα και γιαυτό προσθέσαμε στον κώδικα του jasper.c κάποιες εντολές, με τη βοήθεια των οποίων μετρήσαμε το πλήθος εντολών που εκτελούνται σε αυτές.

Μετά την προσομοίωση στο OVP τα αποτελέσματα που πήραμε παρουσιάζονται στην παρακάτω εικόνα:



Σχήμα 2.7 - JPEG2000 Simulation

Από την προσομοίωση βλέπουμε ότι

- Encoding: 398505592
- Decoding: 53881884

Ένα πρώτο σημαντικό συμπέρασμα που παρατηρήθηκε είναι ότι η διεργασία συμπίεσης (κωδικοποιητή) είναι πιο χρονοβόρα σε σχέση με της αποσυμπίεσης (αποκωδικοποιητής). Πιο συγκεκριμένα, ο χρόνος που δαπανάται κατά την αποκωδικοποίηση της εικόνας περίπου 70% του συνολικού χρόνου που καταναλώνεται στην κωδικοποίσηση της εικόνας.

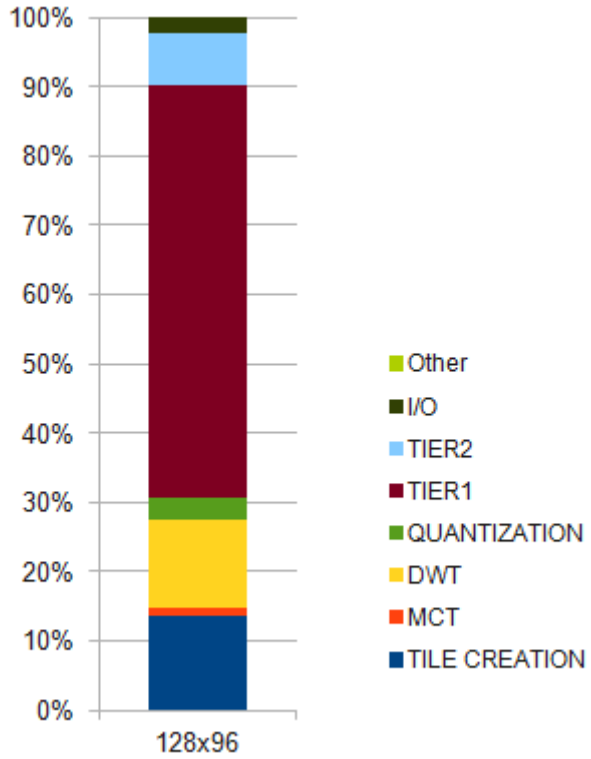
Εν συνεχεία, αυτό που κάναμε ήταν να χωρίσουμε την κώδικοποίηση σε κομμάτια και κάνουμε την ίδια διαδικασία για να βρούμε ποιες συναρτήσεις είναι οι πιο απαιτητικές από άποψη χρόνου. Και καταλήξαμε στα εξής:

	128x96
PART1 [TILE CREATION]	6093862
PART2	1258745
PART3	19
PART4[MCT]	654618
PART5 [DWT]	6912706
PART6 [QUANTIZATION]	1671667
PART7	3945
PART8	99
PART9	1348

PART10 [TIER 1 (EBCOT+AEC)]	32438658
PART11	535
PART12a [TIER-2 I]	2642649
PART12b [TIER-2 II]	1374235
PART12c [STREAM OUT]	1242394
| |/	
	128x96
TILE CREATION	7352607
MCT	654618
DWT	6912706
QUANTIZATION	1671667
TIER1	32403461
TIER1 [AEC]	35197
TIER2	4016884
I/O	1242394
Other	5946

Σχήμα 2.8 – Αριθμός Εντολών των κομματιών του JPEG2000

Αυτά τα αποτελέσματα φαίνονται πιο καλά στο διάγραμμα:



Σχήμα 2.9 – bar chart

Και οι βασικές συναρτήσεις που χωρίζουν τα μέρη του encoder και του decoder μπορούν να περιοριστούν στις παρακάτω συναρτήσεις:

Category	Encoder	Decoder
Utilities	bitstoint() inttobits()	bitstoint() inttobits()
	pgxwordtoint()	pgxwordtoint()
	jasmalloc()	jasmalloc()
Image Transfer	jas_image_writecmpt()	jas_image_writecmpt()
	jas_image_readcmpt()	jas_image_readcmpt()
EBCOT	jpc_encsigpass()	jpc_decsigpass()
	jpc_encrefpass()	jpc_decrefpass()
	jpc_encclnpass()	jpc_decclnpass()
DWT	jpc_ft_analyze()	jpc_ft_synthesize()
AEC	jpc_mqenc_codelps()	jpc_mqdec_codelps()
	jpc_mqenc_codemps2()	jpc_mqdec_codemps2()

Σχήμα 2.10 – most time-consuming functions

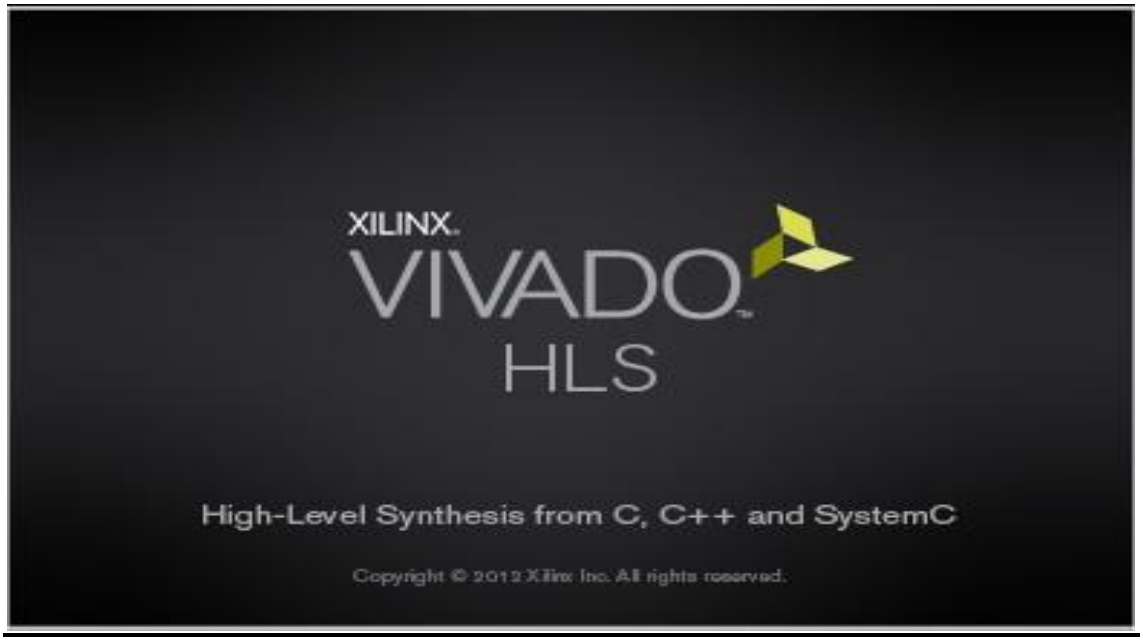
Σύμφωνα, λοιπόν και με τα προηγούμενα αποτελέσματα τα δύο πιο time-consuming μέρη του αλγορίθμου είναι το EBCOT (Coefficient Bit Modelling Routines) και το DWT (Wavelet Transform Routines). Συμπεραίνουμε λοιπόν, ότι οι βασικές συναρτήσεις που υλοποιούν το EBCOT και το DWT είναι αυτές που θα πρέπει να βάλουμε στο περιφερειακό μας ώστε η κωδικοποίηση JPEG2000 να επιταχυνθεί σημαντικά.

Η διαδικασία του hardware/software partitioning έφτασε στο τέλος της. Τώρα πια αυτό που απομένει είναι να υλοποίησουμε τις πιο βαριές συναρτήσεις σε hardware. Μόλις τις βάλουμε στο περιφερειακό και λειτουργήσει η προσομοίωση τότε θα έχουμε ολοκληρώσει την εικονική πλατφόρμα. Εν συνεχεία, θα περάσουμε τις συναρτήσεις αυτές από High Level Synthesis και αφού δημιουργήσουμε το περιφερειακό γραμμένο σε HDL Language μπορούμε πλέον να πραγματοποιήσουμε το implementation του Fpga με τη βοήθεια των εργαλείων τις Xilinx.

Τώρα πλέον η διαδικασία του partitioning περιγράφηκε αναλυτικά και αποτυπώνει πλήρως τον τρόπο με τον οποίο μπορούμε να εργαζόμαστε σε κάθε εφαρμογή. Ωστόσο ο στόχος της διπλωματικής είναι η σύνδεση της εικονικής με την πραγματική πλατφόρμα και όχι η εφαρμογή. Για το λόγο αυτό θα προχωρήσουμε με την εφαρμογή Fibonacci, η οποία είναι ένα πολύ πιο απλό και αντιπροσωπευτικό παράδειγμα.

Κεφάλαιο 3

Vivado HLS



Σχήμα 3.1- VivadoHLS

3.1 Εισαγωγή στο VivadoHLS

Τα εργαλεία σύνθεσης υψηλού επιπέδου (High Level Synthesis - HLS) είναι εξαιρετικά χρήσιμα καθώς μας βοηθούν να μετατρέψουμε ένα software design σε hardware design (Register Transfer Level) πολύ γρήγορα. Γενικά, με την χρήση ενός HLS εργαλείου, οι περιορισμοί του χρήστη μπορούν να οριστούν αυτόματα και ο χρόνος προς την αγορά να μειωθεί. Συγκεκριμένα, το Vivado High-Level-Synthesis είναι ένα HLS εργαλείο για FPGAs και ASICs το οποίο επιταχύνει την σύνθεση του design. Δέχεται έναν κώδικα γραμμένο σε C/C++ ή SystemC και τον μετατρέπει αμέσως σε RTL level ώστε να επιτευχθεί η σύνθεσή του σε ένα FPGA.

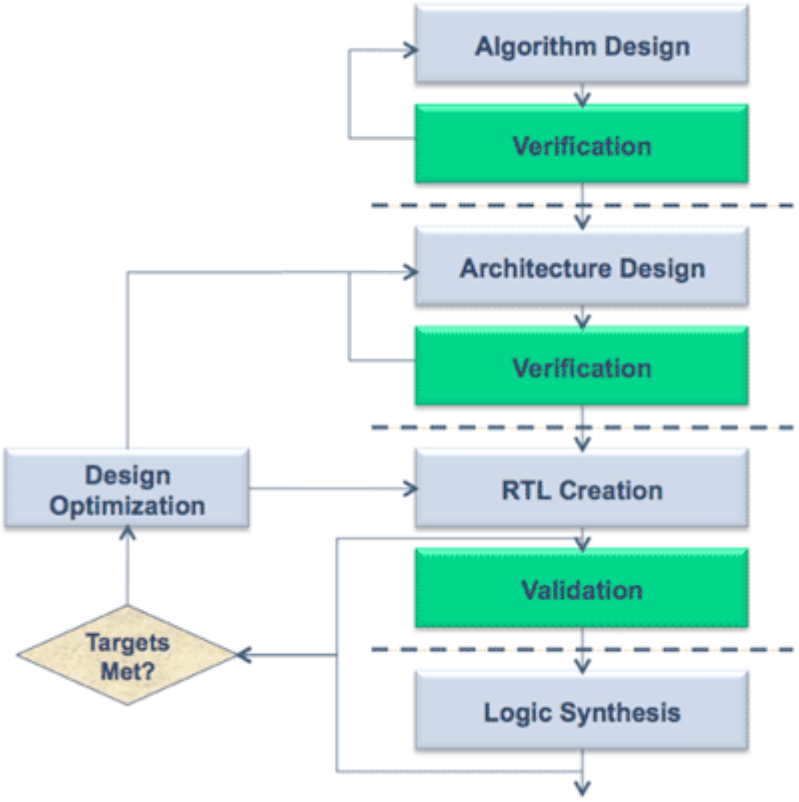
3.2 High Level Synthesis

Το High-Level Synthesis(HLS) ή διαφορετικά electronic system level synthesis(ESL) είναι μια αυτοματοποιημένη διαδικασία σχεδίασης κατά την οποία μια αλγοριθμική περιγραφή περνάει από το στάδιο της μετάφρασης προκειμένου να δημιουργηθεί hardware που υλοποιεί τον ίδιο αλγόριθμο. Στα σύγχρονα εργαλεία hls η σύνθεση γίνεται σε ANSI C/C++/SystemC.

Κατά τη σύνθεση υψηλού επιπέδου ως είσοδοι στο σύστημα δίνονται:

- ο behavioral προσδιορισμός
- σχεδιαστικοί περιορισμοί (κόστος, απόδοση, κατανάλωση ισχύος)
- μια βιβλιοθήκη που δείχνει το υλικό που έχουμε διαθέσιμο

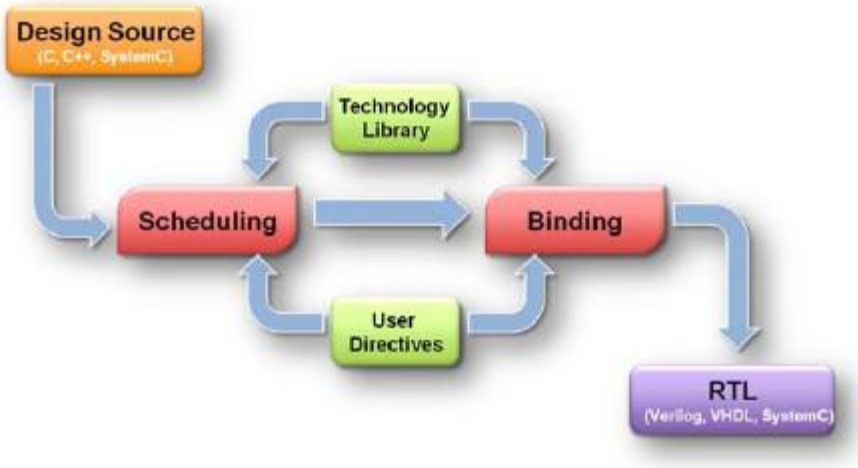
Ως έξοδο παίρνουμε την RTL υλοποίηση καθώς και κάποιες πληροφορίες που αφορούν την γεωμετρία της σχεδίασης. Το βασικό μοντέλο που ακολουθήθηκε κατά την Σύνθεση Υψηλού Επιπέδου φαίνεται παρακάτω:



Σχήμα 3.2- Το βασικό μοντέλο High Level Synthesis

Βασικά βήματα:
Για την διαδικασία της σύνθεσης υψηλού επιπέδου απαιτούνται κάποια βήματα. Αυτά είναι:

- Λεξική επεξεργασία
- Αλγοριθμική βελτιστοποίηση
- Dataflow ανάλυση
- Κατανομή των πόρων
- Scheduling
- Binding
- Επεξεργασία εξόδου



Σχήμα 3.3- Απαραίτητα βήματα HLS

Τα διάφορα hls εργαλεία κάνουν τα παραπάνω βήματα με διαφορετική σειρά το καθένα, ανάλογα με τον αλγόριθμο που χρησιμοποιούν. Πολλές φορές μπορεί να γίνει συνδυασμός των παραπάνω βημάτων ή και επανάληψη τους προκειμένου τελικά να συγκλίνουμε στην επιθυμητή υλοποίηση.

Μερικά βασικά θέματα τα οποία αποτελούν και θεμελιώδης έννοιες της hls:

1)Χρονοδρομολόγηση
Κατά την χρονοδρομολόγηση καθορίζεται ποιές διεργασίες θα τρέξουν σε κάθε κύκλο.

2)Κατανομή πόρων
Επιλογή των τύπων των στοιχείων υλικού και του αριθμού των στοιχείων για κάθε τύπο που πρέπει να περιλαμβάνονται στην τελική εφαρμογή.

3)Binding
η διαδικασία κατά την οποία καθορίζεται ποιός πόρος υλικού ή ποιός πυρήνας θα χρησιμοποιηθεί για κάθε χρονοδρομολογημένη διεργασία

4)Σύνθεση ελεγκτή
Σχεδιασμός του τρόπου ελέγχου και χρονισμού.

5)Compilation της γλώσσας προδιαγραφής εισόδου στην εσωτερική αναπαράσταση.

6)Δυνατότητα παραλληλισμού

Λαμβάνοντας υπόψιν και τα παραπάνω θέματα, στόχος του High-Level Synthesis είναι η δημιουργία μιας RTL σχεδίασης, η οποία θα έχει την επιθυμητή συμπεριφορά, ενώ παράλληλα θα ικανοποιεί τους σχεδιαστικούς περιορισμούς που έχουν τεθεί και θα βελτιστοποιεί την δοσμένη συνάρτηση κόστους

3.3 RTL Synthesis

Στην ψηφιακή σχεδίαση κυκλωμάτων, Register Transfer Level (RTL) σχεδίαση είναι η σχεδίαση που μοντελοποιεί ένα σύγχρονο ψηφιακό κύκλωμα από την άποψη της ροής των ψηφιακών σημάτων (δεδομένα) μεταξύ των καταχωρητών του υλικού, καθώς και τις λογικές πράξεις που εκτελούνται στα εν λόγω σήματα.

Η σχεδίαση RTL χρησιμοποιείται σε γλώσσες περιγραφής υλικού (HDLs), όπως είναι η Verilog και η VHDL για υψηλού επιπέδου αναπαραστάσεις ενός κυκλώματος, από το οποίο μπορούν να προκύψουν χαμηλότερου επιπέδου αναπαραστάσεις και τελικά καλωδίωση.

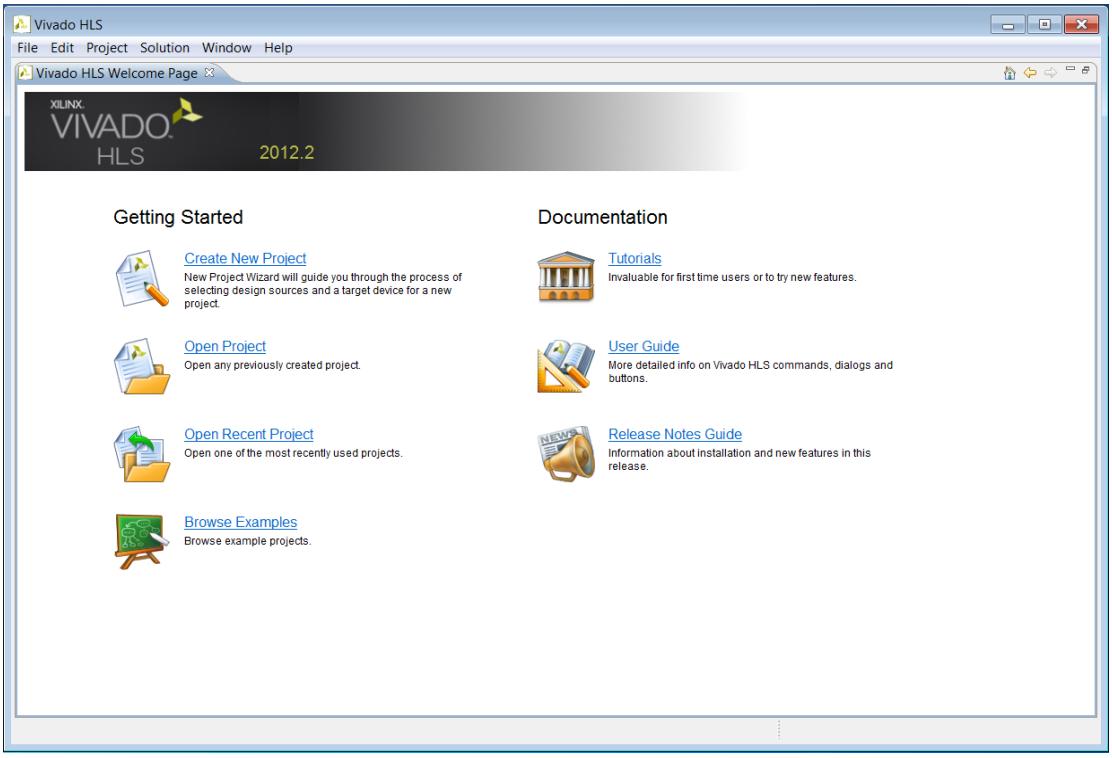
3.4 Δημιουργία Περιφερειακού - Χρήση του εργαλείου

Ο κύριος στόχος μας είναι η δημιουργία του ενσωματωμένου συστήματος, που σχεδιάσαμε εικονικά στο OVP, σε πραγματική πλατφόρμα και συγκεκριμένα στην Xilinx Virtex-7 FPGA VC707 Evaluation Kit. Συνεπώς, θα χρειαστούμε τον κώδικα C που υπολογίζει τον αριθμό Fibonacci. Αυτό το πρόγραμμα είναι το ίδιο με αυτό της συνάρτησης dmaburst() που γράψαμε για το περιφερειακό του OVP και περιγράφουμε σε προηγούμενο κεφάλαιο. Μέσω του Vivado HLS θα το μετατρέψουμε σε Vhdl ώστε να μπορούμε να προγραμματίσουμε ως περιφερειακό στο FPGA.

Η διαδικασία που ακολουθήθηκε περιγράφεται αναλυτικά στα επόμενα.

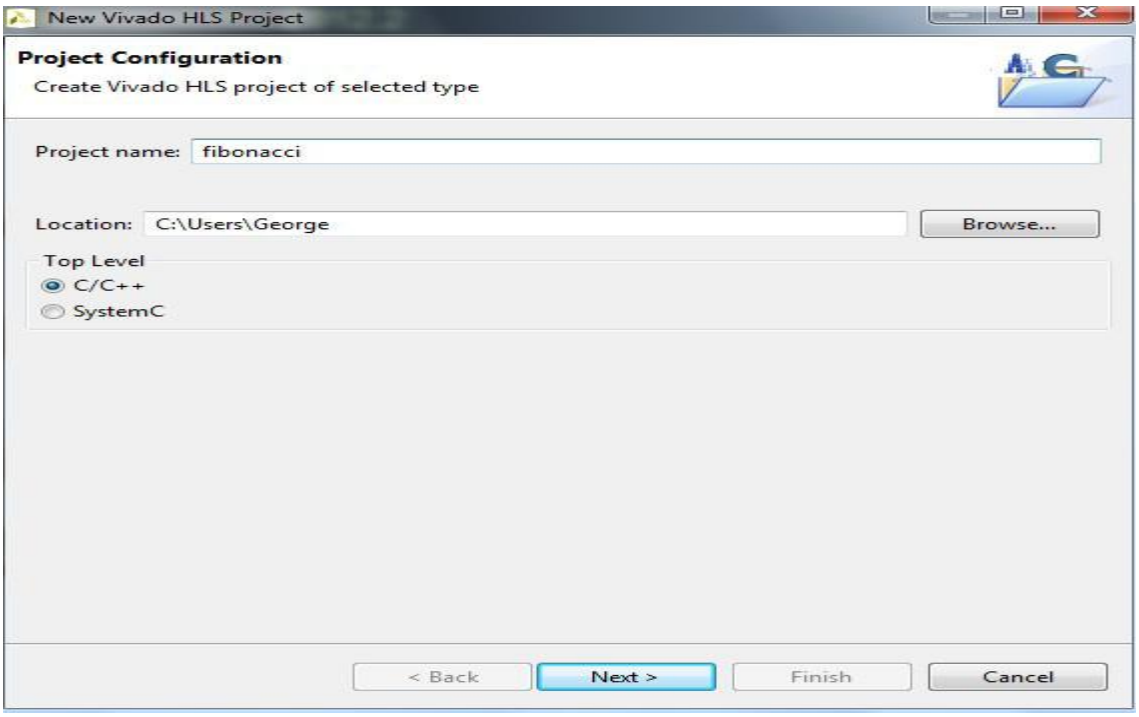
3.4.1 Δημιουργία νέου project

Αρχικά, αφού ανοίξουμε το Vivado HLS, επιλέγουμε create new project για να αρχίσει η δημιουργία του δικού μας project.



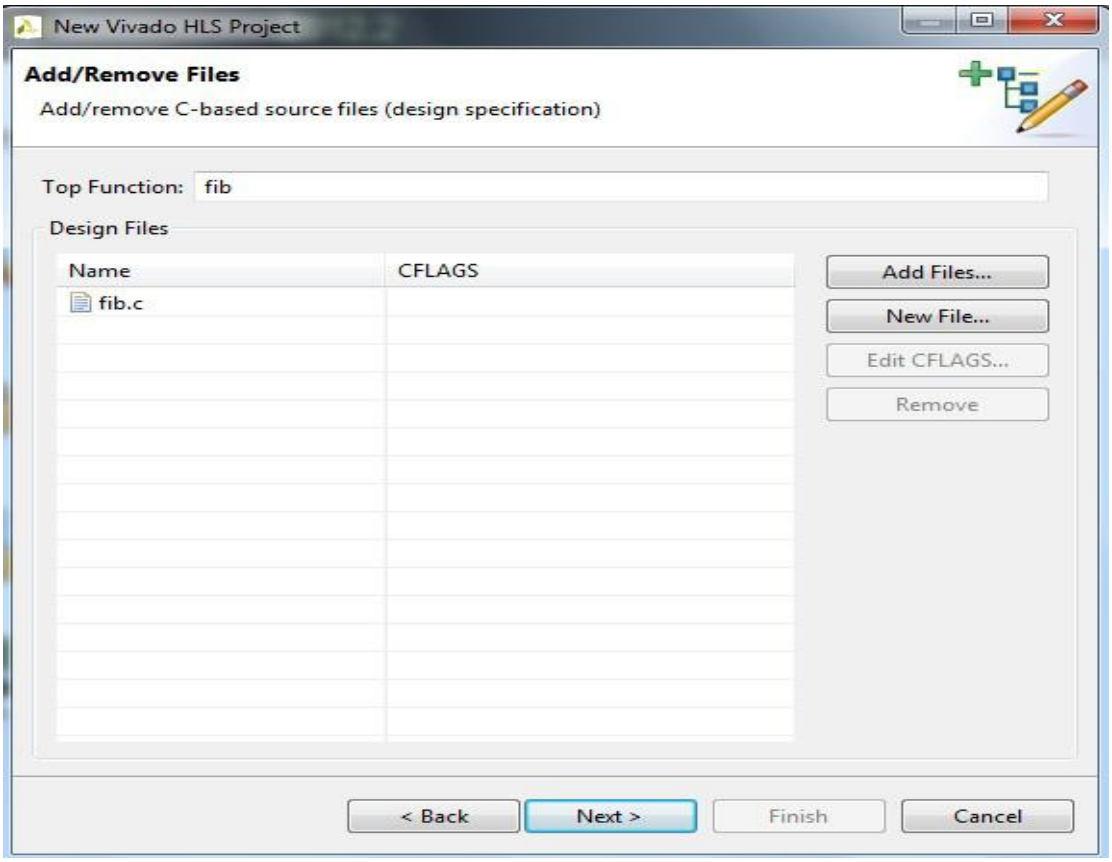
Σχήμα 3.4- Open New Project

Στην συνέχεια, αφού ονομάσουμε το project μας πατάμε Browse και διαλέγουμε την τοποθεσία όπου θα αποθηκεύονται τα αρχεία του. Επίσης, επιλέγουμε C/C++ αφού ο κώδικας είναι γραμμένος σε C.



Σχήμα 3.5- Project Configuration

Πατώντας Next φτάνουμε στο σημείο που πρέπει να καθορίσουμε τα αρχεία που θα χρησιμοποιήσουμε για τη σύνθεση.

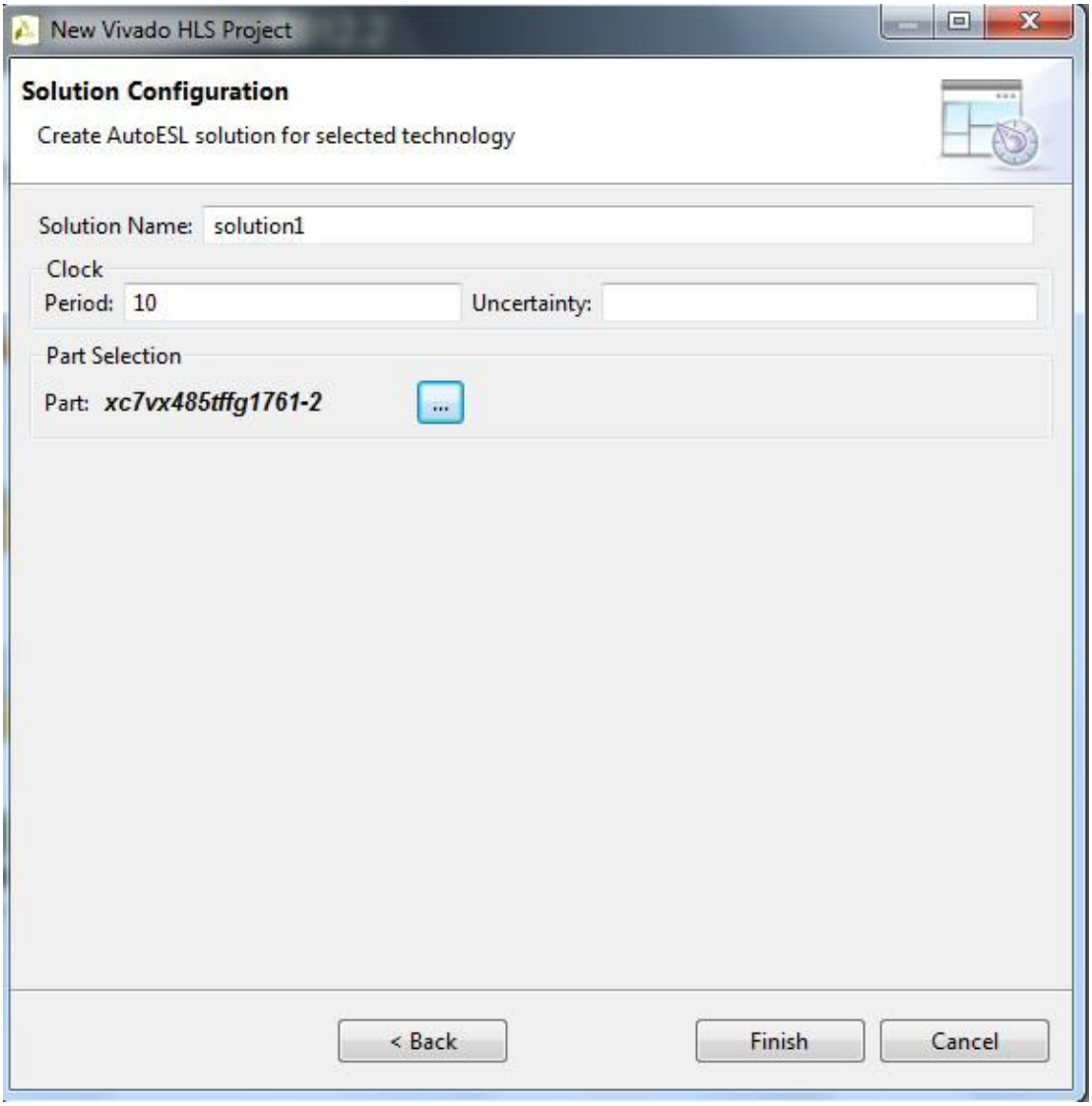


Σχήμα 3.6- Add/Remove Files

Η συνάρτηση fib.c παρατίθεται παρακάτω:

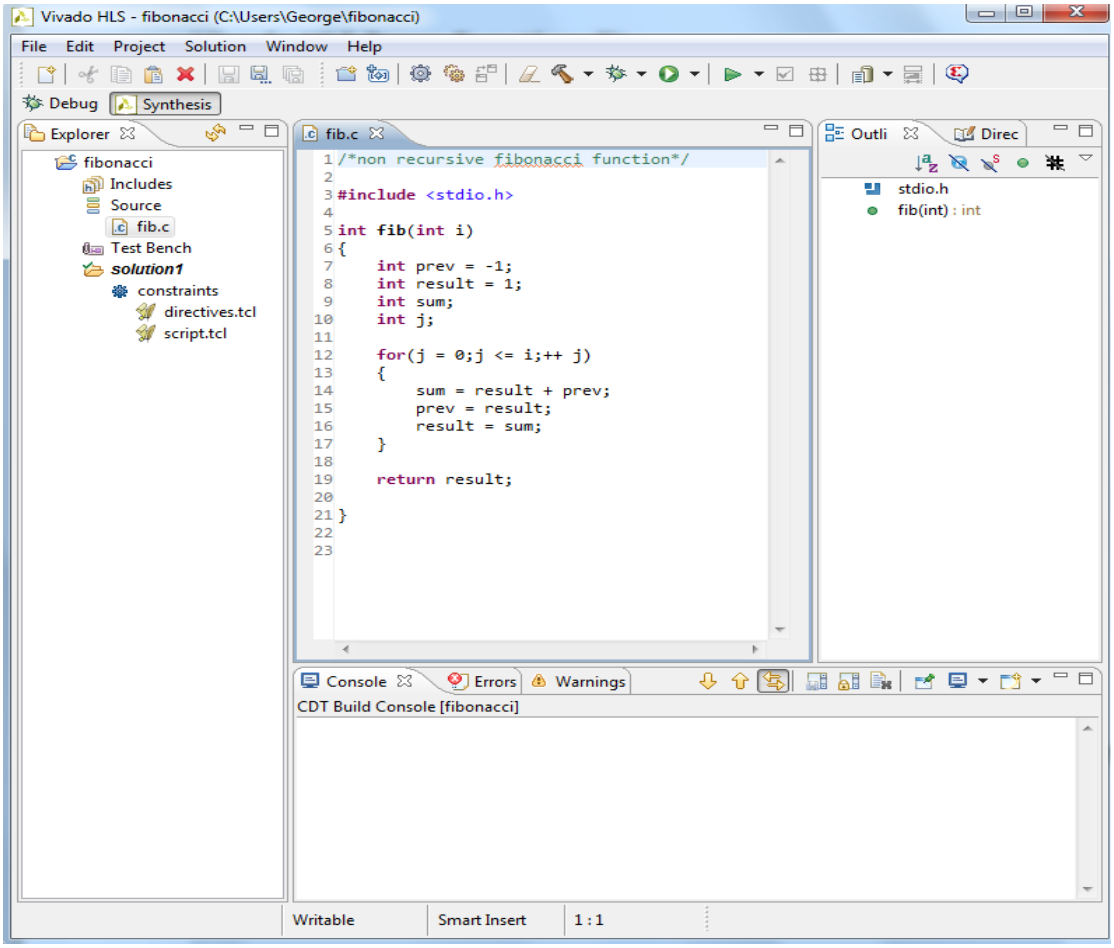
```
/*non recursive fibonacci function*/

int fib(int i)
{
   int prev = -1;
   int result = 1;
   int sum;
   int j;

   for(j = 0;j <= i;++ j)
   {
      sum = result + prev;
      prev = result;
      result = sum;
   }

   return result;

}
```

Αφού καθορίσουμε την top function που θα συντεθεί μετά, αν υπάρχουν, μπορούμε να προσθέσουμε αρχεία TestBench τα οποία ουσιαστικά ελέγχουν εάν ο κώδικας που κάναμε σύνθεση δουλεύει σωστά. Εμείς δεν χρησιμοποιούμε testbench, συνεπώς το συγκεκριμένο πεδίο παραμένει άδειο.



Σχήμα 3.7- Solution Configuration

Τέλος, ονομάζουμε την λύση που θα προκύψει με συγκεκριμένο όνομα, καθώς στη συνέχεια μπορεί να προκύψουν και άλλες λύσεις του ίδιου project με διαφορετικό FPGA, περιορισμούς, ταχύτητα ή άλλα τεχνικά χαρακτηριστικά. Επίσης επιλέγουμε το FPGA που θα χρησιμοποιήσουμε, στην προκειμένη περίπτωση το Virtex7. Πατώντας Finish, η δημιουργία του project ολοκληρώνεται και ανοίγει το περιβάλλον του Vivado HLS μαζί με όλες τις πληροφορίες και τα αρχεία που ορίσαμε κατά τη δημιουργία του.



Σχήμα 3.8- Hls Environment

3.4.2 Σύνθεση

Προς το παρών όμως, θα επικεντρωθούμε στα βήματα που θα ακολουθήσουμε ώστε να δημιουργήσουμε ένα Pcore με AXI bus από το Vivado HLS.

Όπως γνωρίζουμε το Pcore θα υπολογίζει την τιμή ενός αριθμού Fibonacci, ωστόσο θα πρέπει να επικεντρωθούμε ιδιαιτέρως στη δημιουργία του AXI bus που θα ενώνει τον επεξεργαστή με το Pcore για να επικοινωνούν. Για αυτό το λόγο προσθέσαμε στον κώδικα Fibonacci τα παρακάτω:

// Define the RTL interfaces

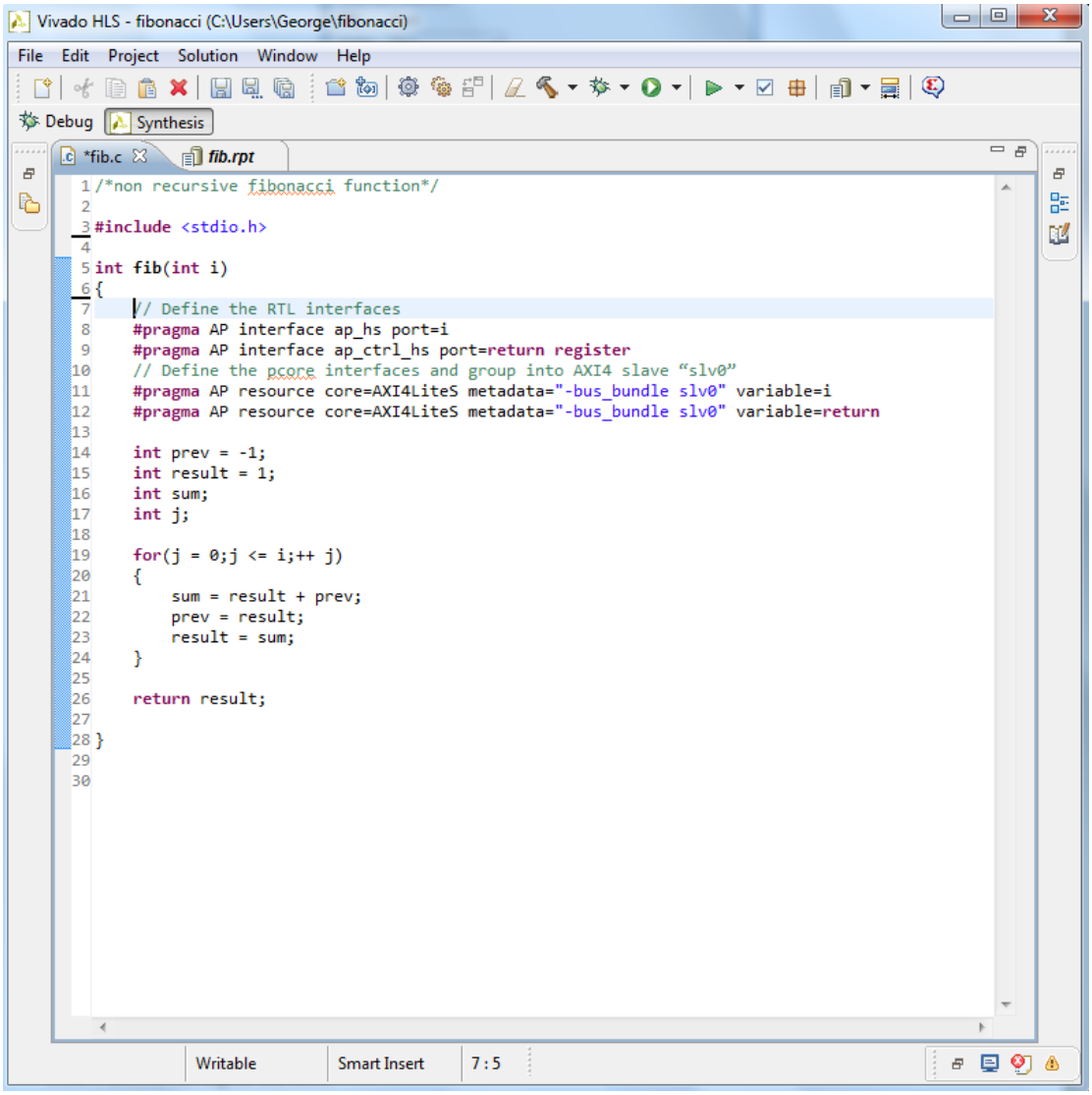
#pragma AP interface ap_hs port=i

#pragma AP interface ap_ctrl_hs port=return register

//Define the pcore interfaces and group into AXI4 slave “slv0”

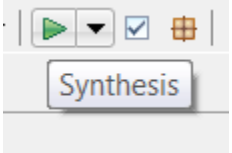
#pragma APresource core=AXILiteS metadata=”-bus_bundle slv0” variable=i

#pragma APresource core=AXILiteS metadata=”-bus_bundle slv0” variable=return



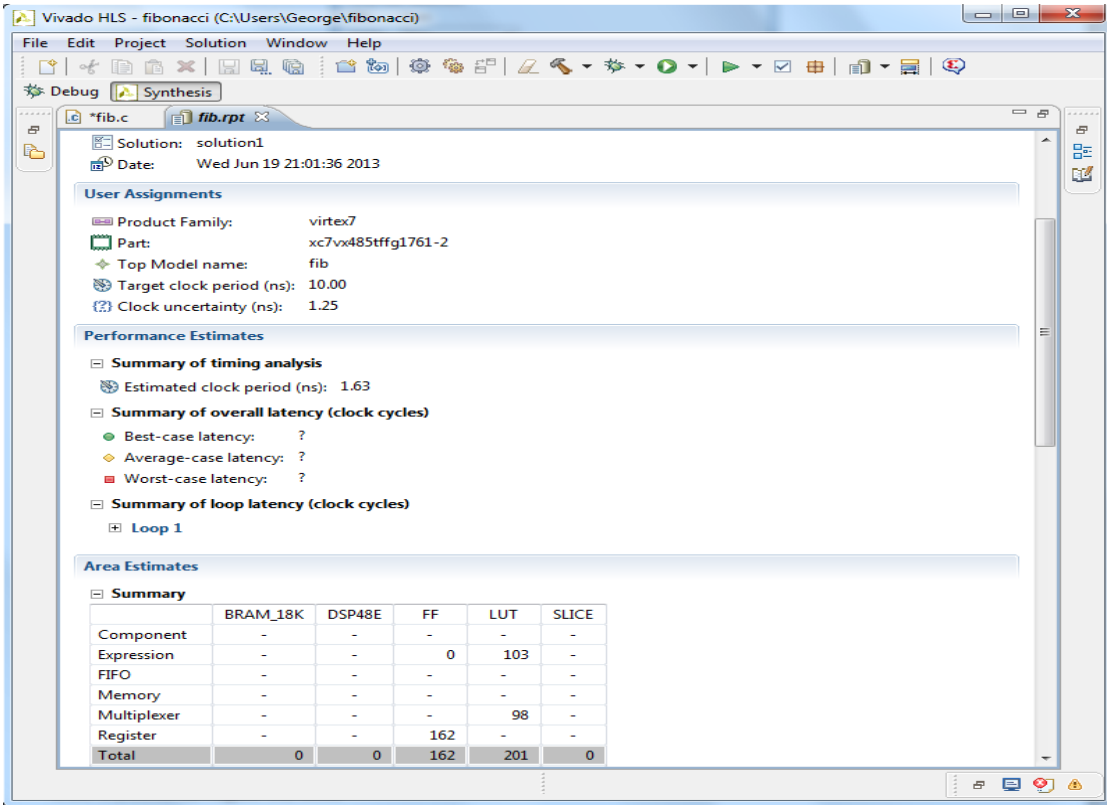
Σχήμα 3.9- Source C Code

Πατώντας λοιπόν την επιλογή synthesis από τη γραμμή εργαλείων,

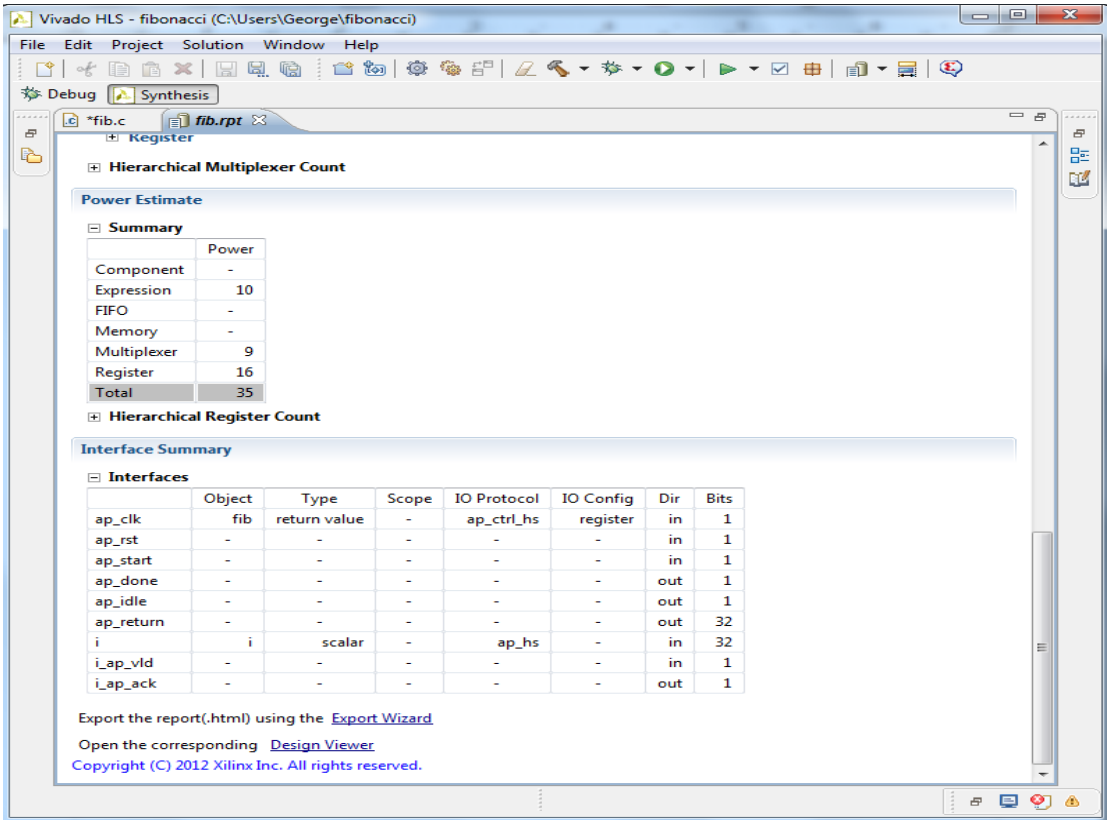


Σχήμα 3.10- Synthesis Task Bar Button

η σύνθεση ξεκινάει και μόλις ολοκληρωθεί επιτυχώς εμφανίζονται τα αποτελέσματά της :



Σχήμα 3.11- HLS Results (1)

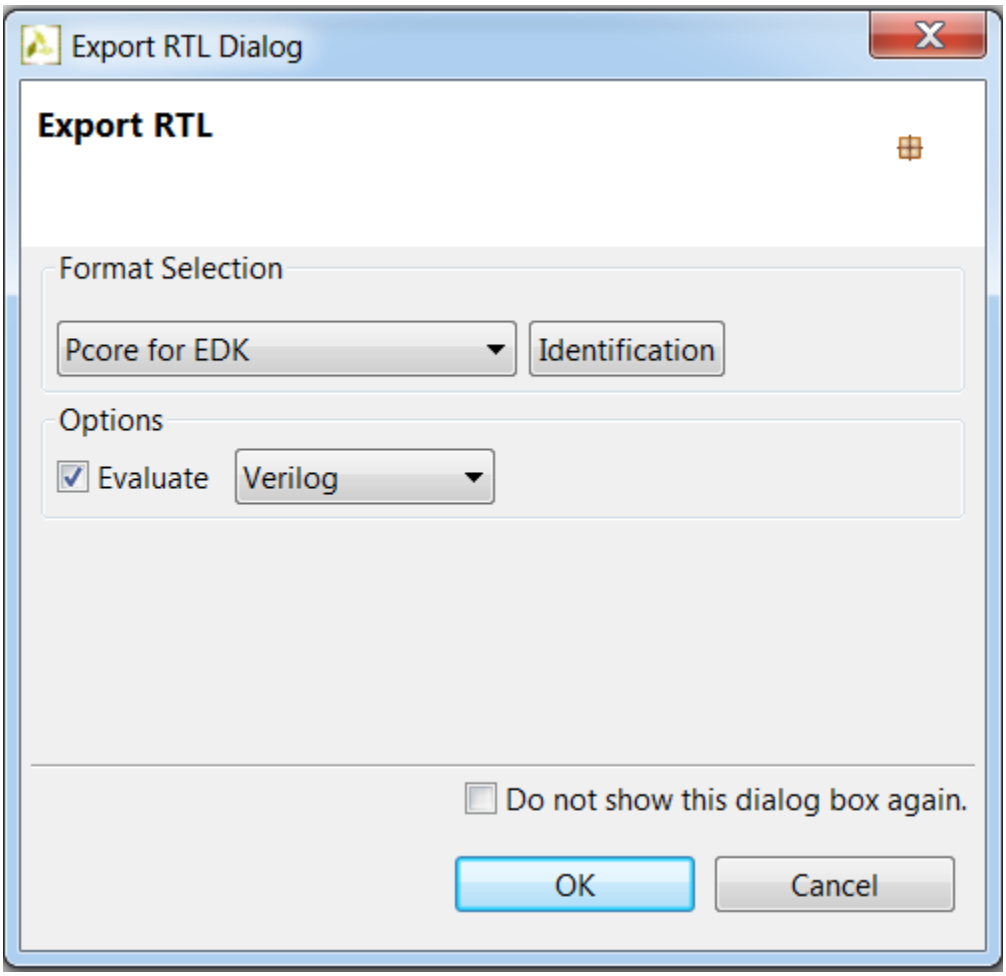


Σχήμα 3.12- HLS Results (2)

πλέον ο χρήστης μπορεί να ενημερωθεί για τα αποτελέσματα της σύνθεσης και γενικότερα να ελέγξει όποια παράμετρο επιθυμεί.

3.4.3 Export RTL

Έχοντας κάνει σύνθεση, πατάμε το πλήκτρο Export RTL και εμφανίζεται το παράθυρο:



Σχήμα 3.13- Export RTL

Επιλέγουμε Pcore for EDK, και για γλώσσα ανάλογα με το τι θέλουμε verilog ή vhdl και πατώντας ok δημιουργείται το Pcore στον impl φάκελο της πρώτης λύσης (solution_1).

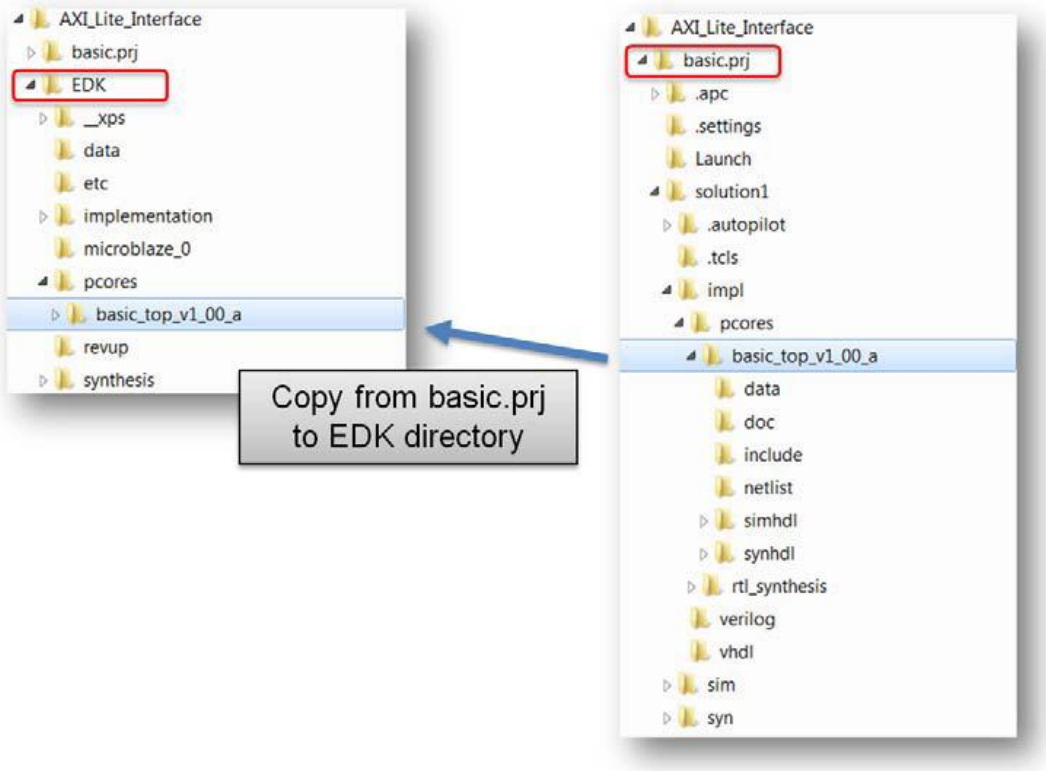
Όταν ολοκληρωθεί η διαδικασία του export RTL, αμέσως θα δημιουργηθεί ένα Pcore, το οποίο στη συνέχεια και θα συνδέσουμε στον Microblaze μέσω ενός άλλου εργαλείου της Xilinx, του Platform Studio (XPS), όπως θα αναλύσουμε σε επόμενο κεφάλαιο.

Με την δημιουργία του Pcore , λοιπόν, μπορούμε να κλείσουμε το VivadoHLS, έχοντας πλέον δημιουργήσει ένα περιφερειακό σε γλώσσα περιγραφής υλικού, το οποίο μπορούμε να συνδέσουμε κατευθείαν με το υπόλοιπο ενσωματωμένο σύστημα. Οι κώδικες που δημιούργησε το VivadoHLS παρατίθενται στο παράρτημα.

3.5 Ενσωμάτωση του Pcore

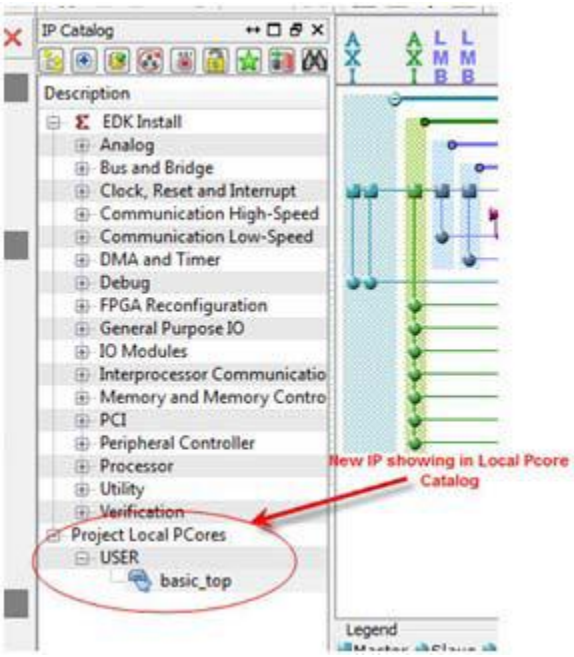
Για να ενσωματώσουμε το Pcore που δημιουργήσαμε στον Microblaze επεξεργαστή χρησιμοποιώντας το εργαλείο XPS ακολουθούμε την διαδικασία:

- Αντιγράφουμε τον φάκελο με το Pcore που φτιάξαμε από τον φάκελο του Vivado HLS στον φάκελο του XPS, όπως φαίνεται στο σχήμα.



Σχήμα 3.14- Pcore Copy

Στο XPS, προσθέτουμε το Pcore από τον IP catalog επιλέγοντας το. Το νέο Pcore εμφανίζεται κάτω από το Project Local Pcores και ουσιαστικά περιλαμβάνει το περιφερειακό που σχεδιάσαμε στο OVP σε Γλώσσα Περιγραφής Υλικού και συγκεκριμένα σε Verilog. Ουσιαστικά λοιπόν, σε αυτό ακριβώς το σημείο επιτυγχάνεται η ένωση της εικονικής με την πραγματική πλατφόρμα.



Σχήμα 3.15- My New IP

Η παραπάνω εικόνα παρουσιάζει την πραγματική πλατφόρμα την οποία θα αρχίσουμε να δημιουργούμε στο επόμενο κεφάλαιο με τη βοήθεια του εργαλείου της Xilinx Planahead.

Κεφάλαιο 4

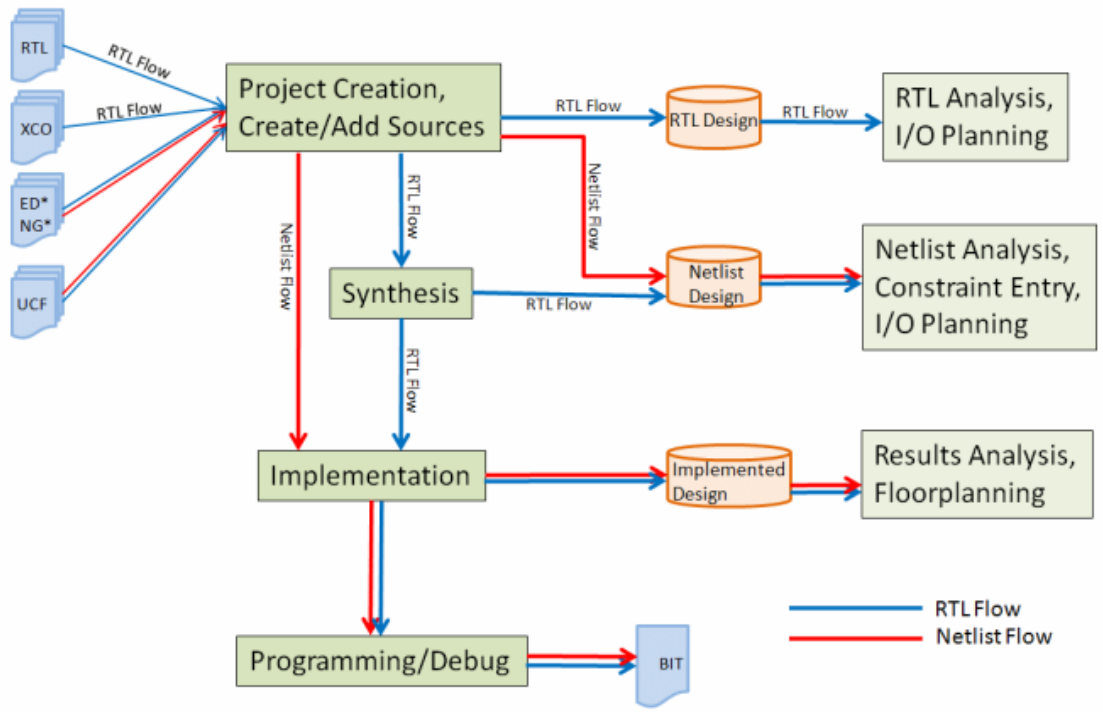
Planahead

Σχήμα 4.1- PlanAhead

4.1 Εισαγωγή

Μία από τις βασικότερες λειτουργίες κατά το σχεδιασμό ψηφιακών συστημάτων είναι η σύνθεση. Η σύνθεση μετασχηματίζει με μοναδικό τρόπο μια αφαιρετική γλωσσική περιγραφή σε κυκλωματική περιγραφή, αρχικά ανεξάρτητα από την τεχνολογία υλοποίησης (σύνθεση επιπέδου μεταφορών καταχωρητών – RTL), και στη συνέχεια βασισμένη στην τεχνολογία υλοποίησης (τεχνολογική σύνθεση). Η πρώτη, χρησιμοποιεί ως δομικές μονάδες συνηθισμένα ψηφιακά κυκλώματα, συνδυαστικά και ακολουθιακά (πύλες, πολυπλέκτες, κωδικοποιητές, καταχωρητές, μετρητές, κ.α.). Η δεύτερη χρησιμοποιεί ψηφιακά κυκλώματα που υπάρχουν στην τεχνολογική βιβλιοθήκη που παρέχεται από τον τελικό κατασκευαστή, όπως γεννήτριες συναρτήσεων LUT και flip-flop για τεχνολογίες FPGA ή τυπικά κελιά για τεχνολογίες ASIC.

Από τα πλέον διαδεδομένα περιβάλλοντα σύνθεσης στη βιομηχανία σήμερα είναι το περιβάλλον ISE Design Suite και περιλαμβάνει μια σειρά εργαλείων για υλοποίηση σε τεχνολογία Xilinx FPGA μέσα στα οποία βρίσκεται και το εργαλείο PlanAhead. Το PlanAhead είναι ένα προϊόν σχεδίασης και ανάλυσης λογισμικού που χρησιμοποιείται για τη σχεδίαση μεγάλων διατάξεων FPGA. Η τεχνολογία πυρήνα περιλαμβάνει ένα ιεραρχικό εργαλείο floorplanning που μπορεί να διασπάσει την φυσική σχεδίαση σε μικρότερα και πιο εύκολα διαχειρήσιμα κομμάτια, μειώνοντας έτσι τον χρόνο για την κατανόηση , την σχεδίαση, την επαλήθευση και την υλοποίηση του FPGA. Παρουσιάζει δηλαδή, με εποπτικό τρόπο τόσο της τεχνολογικής όσο και της μη τεχνολογικής σύνθεσης, και μπορεί να αποτελέσει χρήσιμη βοήθεια στην εκμάθηση των γλωσσών περιγραφής υλικού.



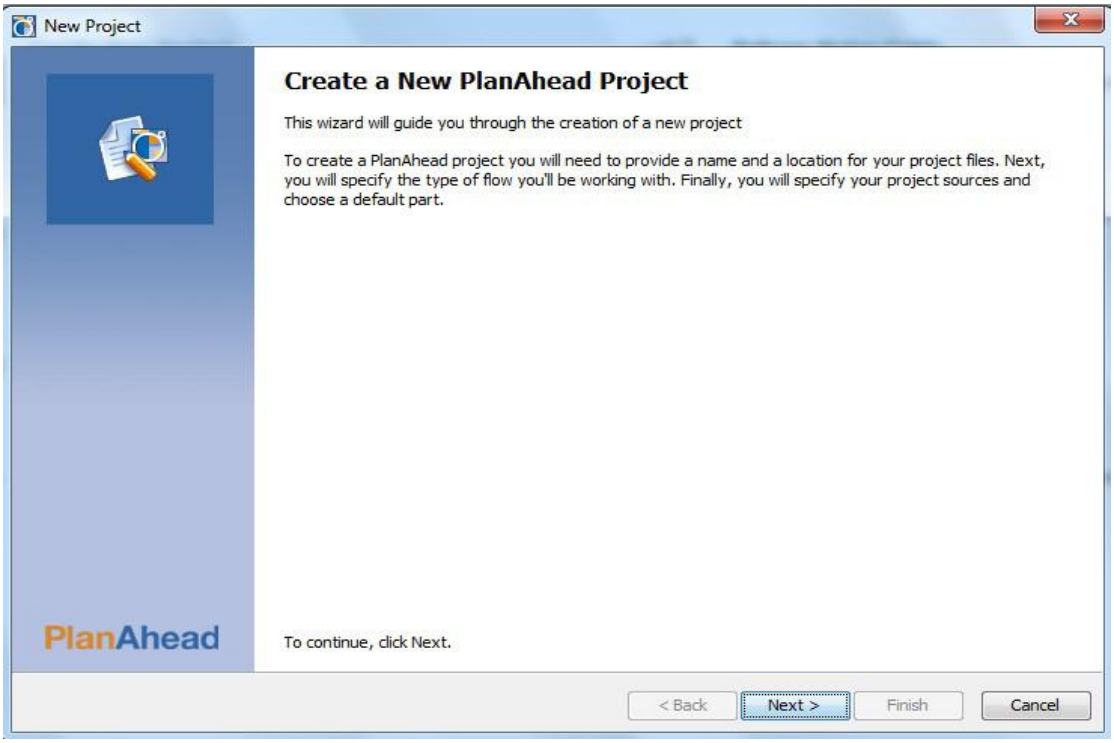
Σχήμα 4.2- Το βασικό μοντέλο του PlanAhead

4.2 Δημιουργία νέου project

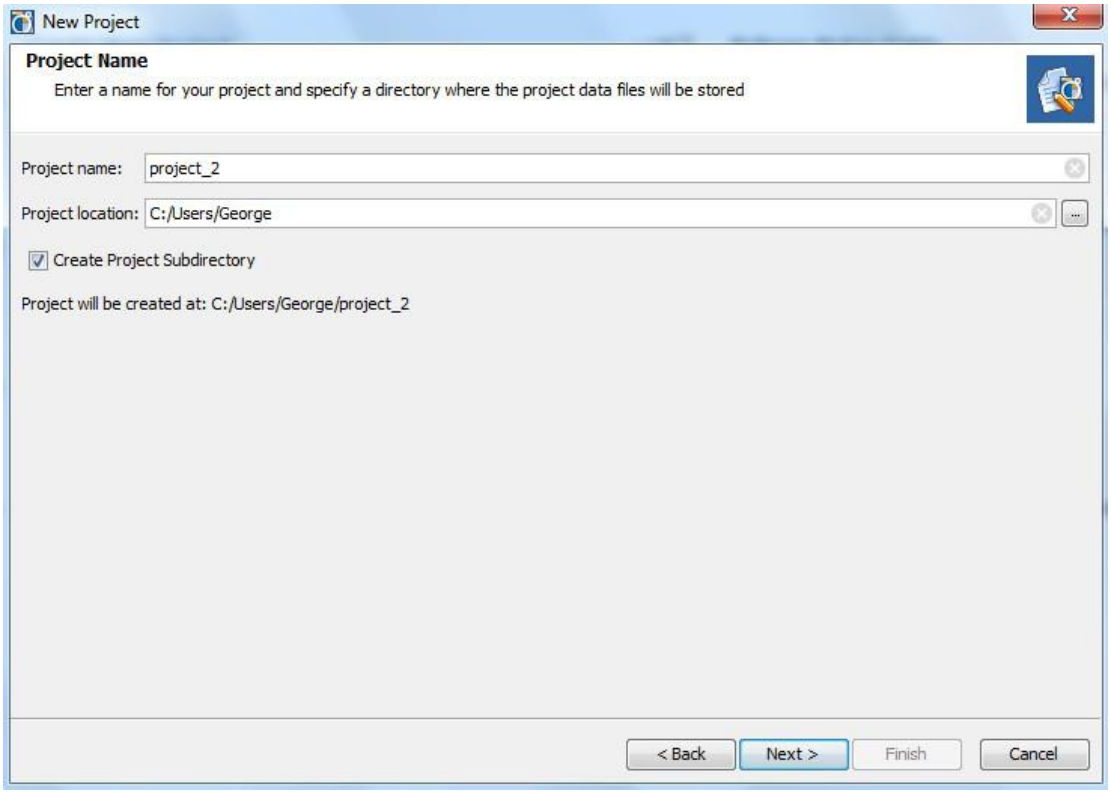
Το εργαλείο PlanAhead υποστηρίζει τη δημιουργία διαφορετικού τύπου έργων, ανάλογα με τα διαθέσιμα αρχεία κώδικα. Η αρχική εκτέλεσή του (καλώντας planAhead σε περιβάλλον Linux ή πατώντας το αντίστοιχο εικονίδιο σε περιβάλλον Windows), έχει ως αποτέλεσμα την εμφάνιση της αρχικής οθόνης του σχήματος 4.3, από την οποία ξεκινάει η διαδικασία δημιουργίας νέου έργου με την εντολή Create New Project.

Σχήμα 4.3- New Project

Το αποτέλεσμα της εντολής είναι η εμφάνιση του εισαγωγικού παραθύρου του σχήματος 4.4, και στη συνέχεια του παραθύρου του σχήματος 4.5, στο οποίο καθορίζονται το όνομα και η θέση (υποκατάλογος) του νέου έργου. Ο υποκατάλογος αυτός δεν είναι απαραίτητο να περιέχει αρχεία κώδικα, αυτά αντιγράφονται αυτόματα από το εργαλείο στην κατάλληλη θέση.

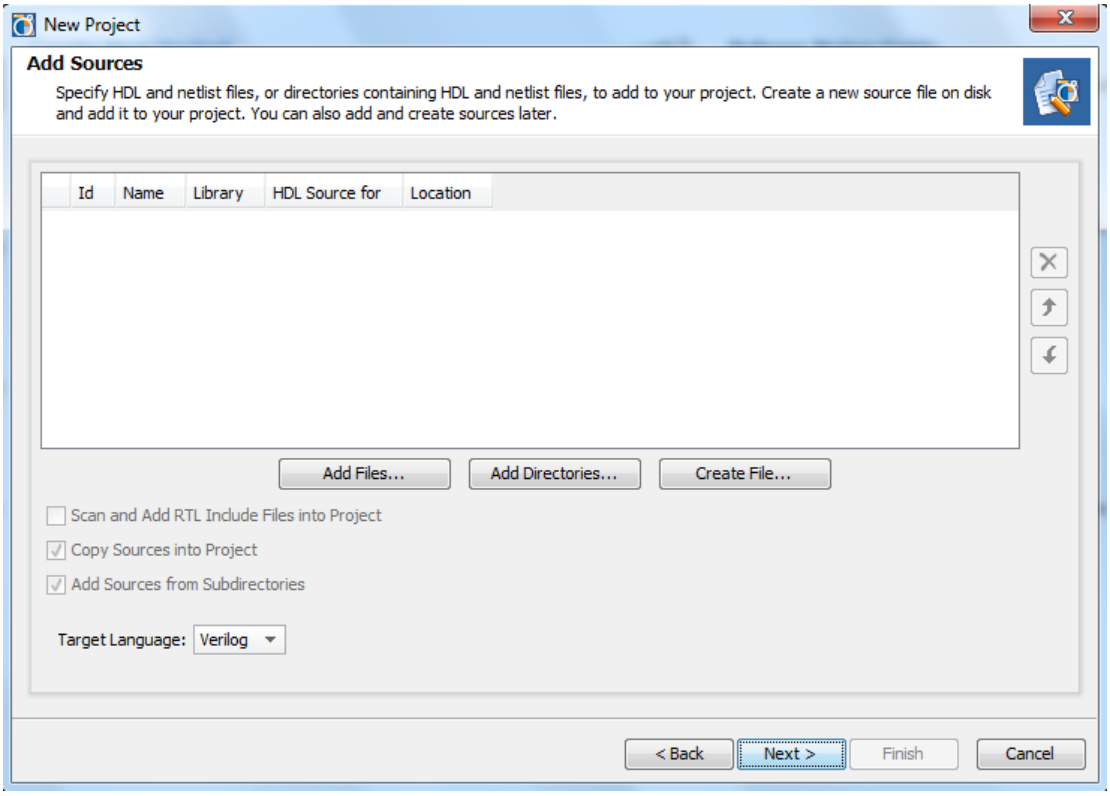


Σχήμα 4.4- Create New PlanAhead Project



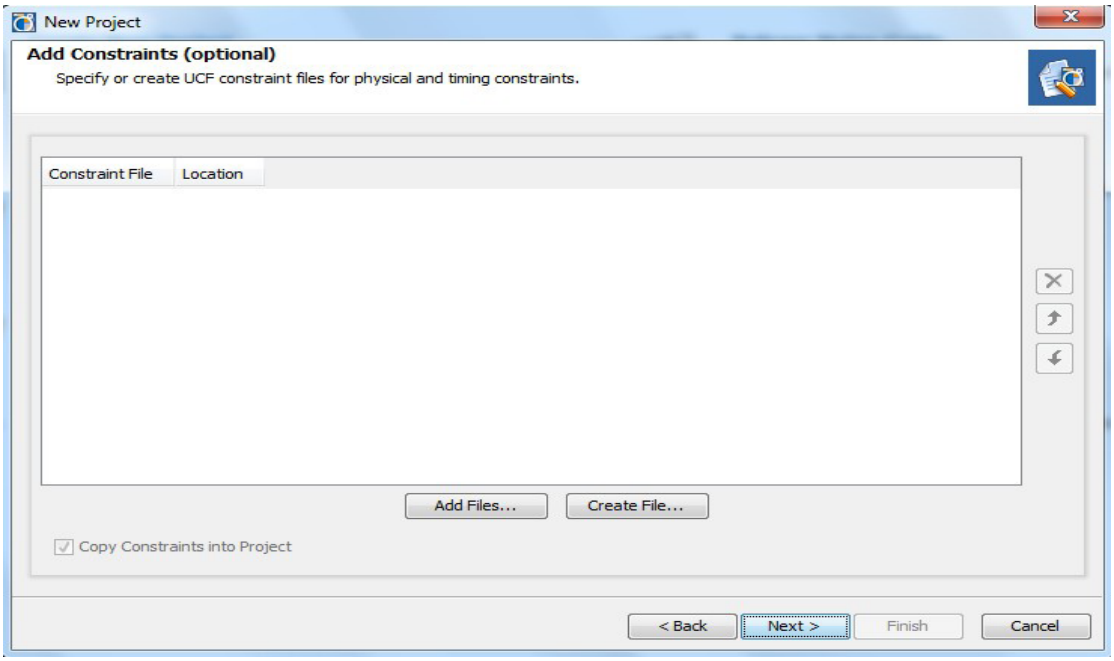
Σχήμα 4.5- Project Name

Στη συνέχεια εμφανίζεται το παράθυρο του σχήματος 4.6, όπου επιλέγονται τα αρχεία που επιθυμούμε να συμπεριλάβουμε στο project σε γλώσσα Verilog ή Vhdl. Τα αρχεία αυτά θα περάσουν από σύνθεση και μέσω του Planahead θα διαμορφώσουν το FPGA.



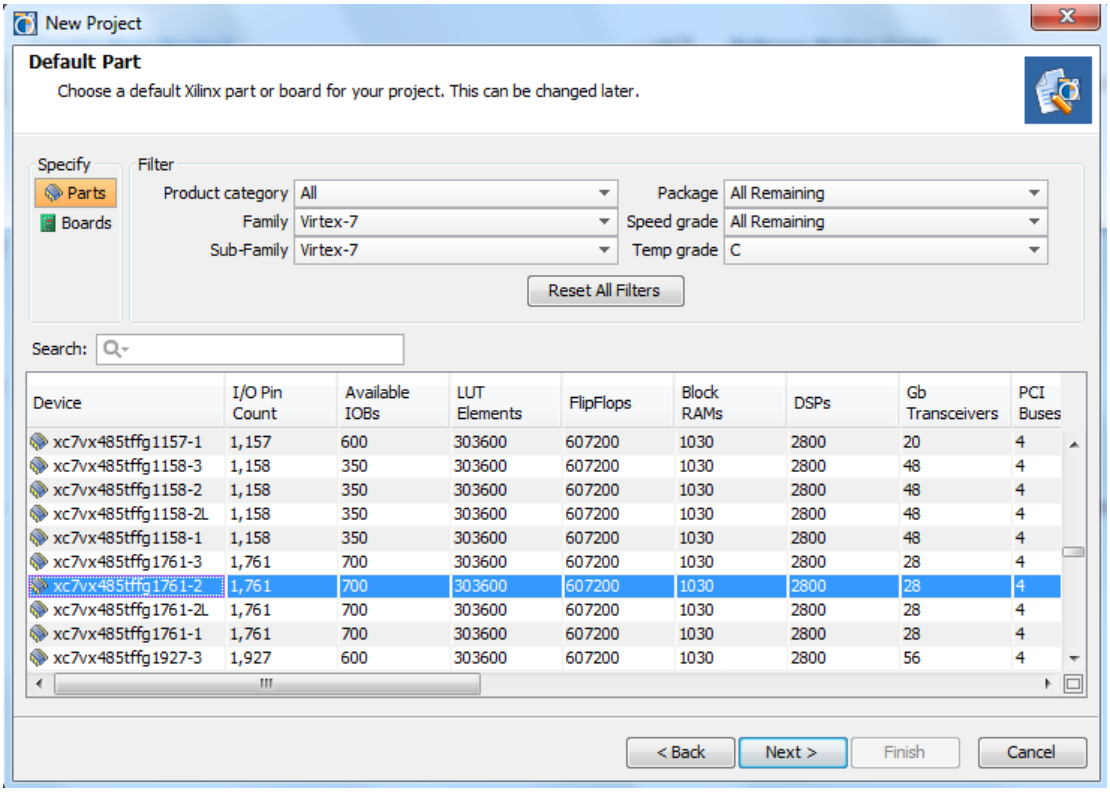
Σχήμα 4.6- Add Sources

Πατώντας την επιλογή Next εμφανίζεται το παράθυρο προαιρετικής εισαγωγής αρχείου περιορισμών (σχήμα 4.7), που έχει να κάνει με την τελική υλοποίηση και αρχικά μπορεί να μείνει κενό.



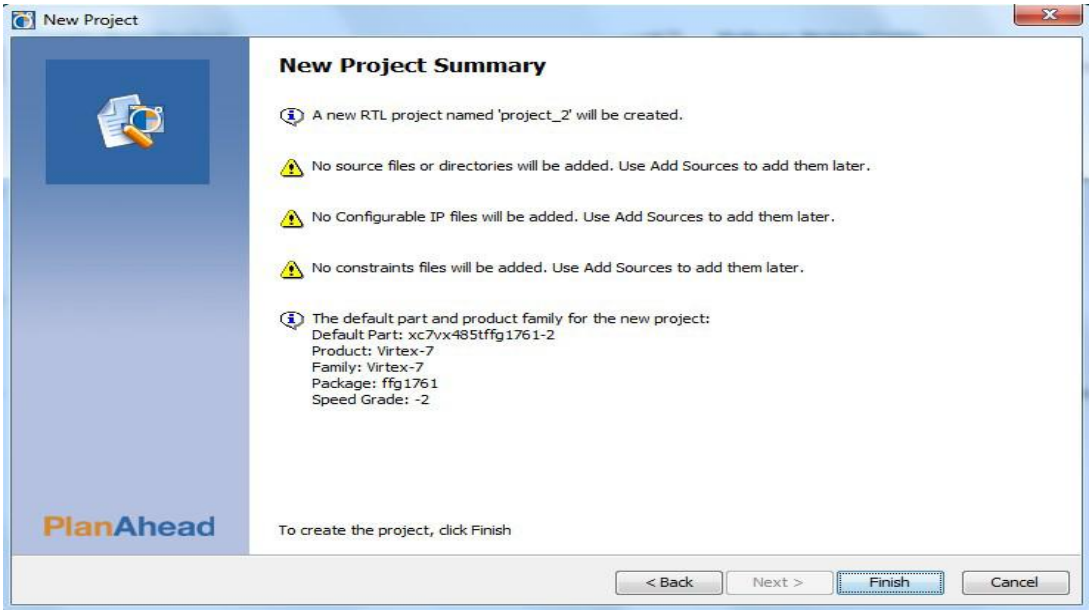
Σχήμα 4.7- Add Constrains

Ακολουθεί το παράθυρο επιλογής μονάδας υλοποίησης (σχήμα 4.8), όπου επιλέγεται η μονάδα FPGA στην οποία θα γίνει υλοποίηση. Επιλέγοντας στοιχεία όπως οικογένεια μονάδων (Family) ή συσκευασία (Package), η λίστα μονάδων περιορίζεται σημαντικά και μπορούμε ευκολότερα να εντοπίσουμε αυτή που μας ενδιαφέρει.



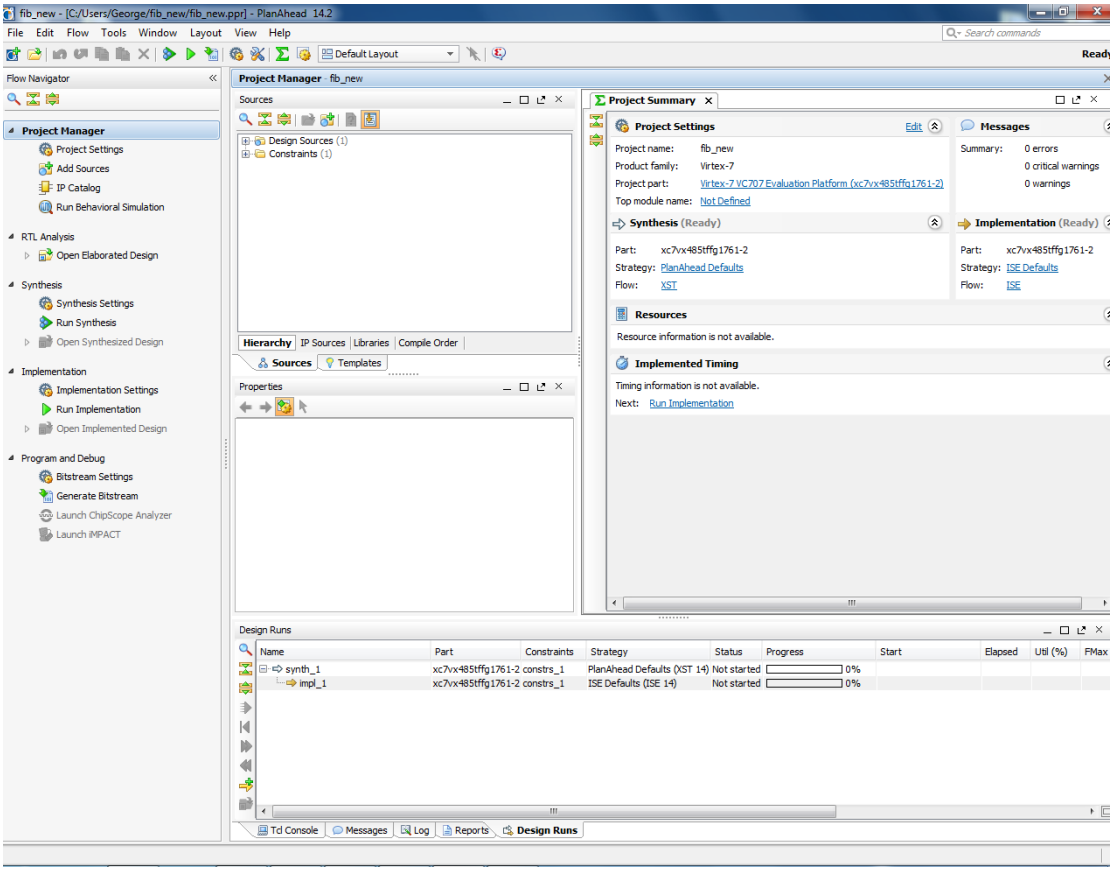
Σχήμα 4.8- FPGA Choice

Τέλος, εμφανίζεται το παράθυρο περίληψης επιλογών (σχήμα 4.9), μετά το οποίο και πατώντας Finish δημιουργείται το νέο έργο και εμφανίζεται η αρχική οθόνη του, όπως εικονίζεται στο σχήμα 4.10.



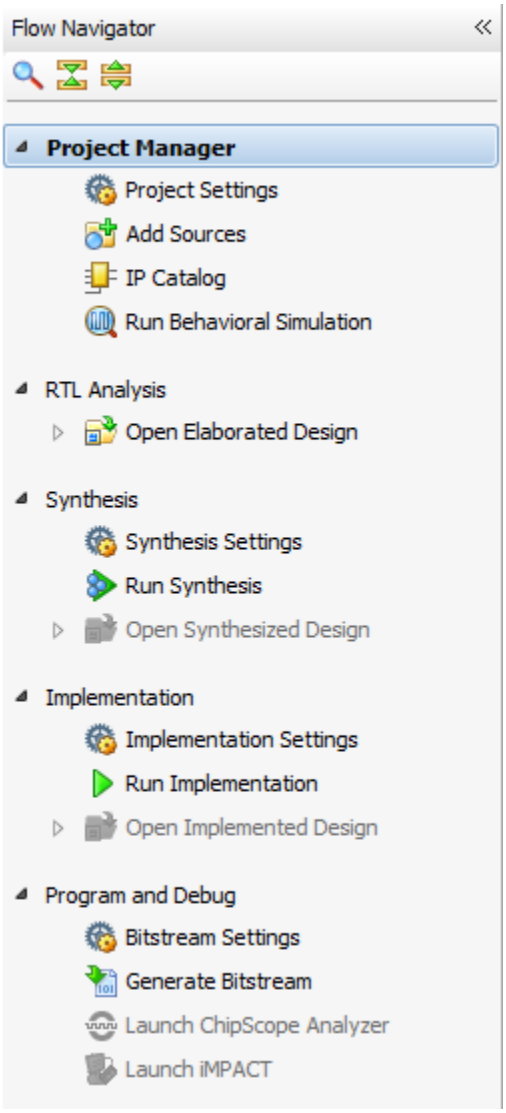
Σχήμα 4.9- New Project Summary

Η αρχική οθόνη του νέου έργου είναι πολύ πλούσια σε εργαλεία και πληροφορίες.



Σχήμα 4.10- Κεντρικό Παράθυρο Σχεδίασης του PlanAhead

Συνοπτικά, στο κέντρο εμφανίζεται το παράθυρο κώδικα, το οποίο περιέχει όλα τα αρχεία κώδικα και περιορισμών του έργου. Πατώντας δεξί κουμπί πάνω σε αυτά, εμφανίζεται ένα μενού πλούσιο σε εντολές. Για παράδειγμα, η εντολή Find in Files. μπορεί να μας βοηθήσει να βρούμε φράσεις (π.χ. ονόματα σημάτων ή θυρών Ε/Ε) κοινές σε όλα τα αρχεία του έργου (πολύ χρήσιμο για μεγάλα σχέδια). Αριστερά, όπως εικονίζεται στο σχήμα 4.11 είναι το παράθυρο διαχείρισης έργου, από το οποίο δίνονται εντολές για τα βήματα που επιθυμεί να κάνει ο σχεδιαστής και χωρίζεται σε τρία πεδία που αντιστοιχούν σε αντίστοιχες φάσης σχεδιασμού: Synthesis (μη τεχνολογική σύνθεση), Implementation (τεχνολογική σύνθεση) και Programm and Debug (δημιουργία αρχείου Bitstream - υλοποίηση). Τέλος, δεξιά στην αρχική οθόνη νέου έργου, βρίσκεται το παράθυρο πολλαπλών πληροφοριών, στο οποίο παρουσιάζονται όλα τα αποτελέσματα.

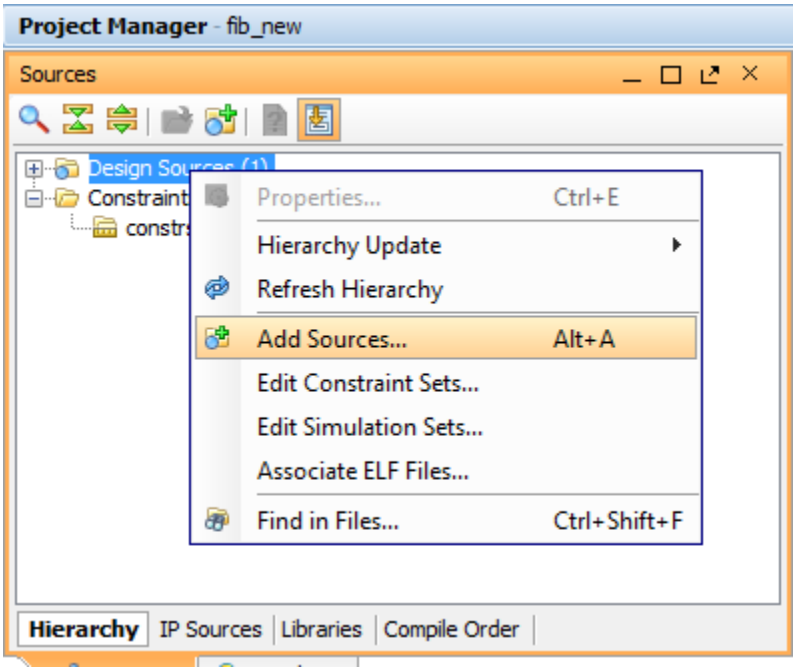


Σχήμα 4.11- Flow Navigator

4.3 Εφαρμογή

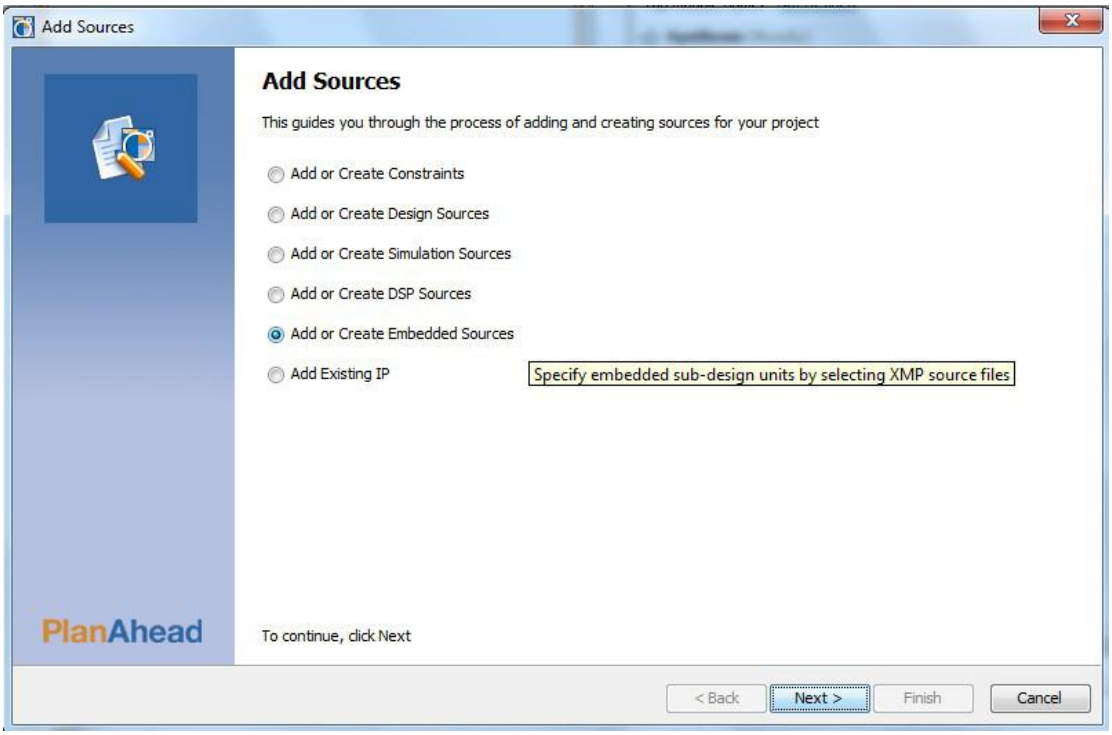
Στην δική μας εφαρμογή, αφού φτιάξαμε το νέο project με τη βοήθεια του Planahead, το επόμενο βήμα είναι να δημιουργήσουμε την πλατφόρμα με τον επεξεργαστή Microblaze, κάνοντας χρήση ενός άλλου εργαλείου της Xilinx, το Xilinx Platform Studio, που είναι ειδικό για αυτή τη δουλειά όπως αναλύεται σε επόμενο κεφάλαιο.

Για να το πετύχουμε αυτό κάνουμε δεξί κλικ στο Design Source, που βρίσκεται στο παράθυρο με τα Sources και επιλέγουμε από τις επιλογές που παρουσιάζονται Add Sources.



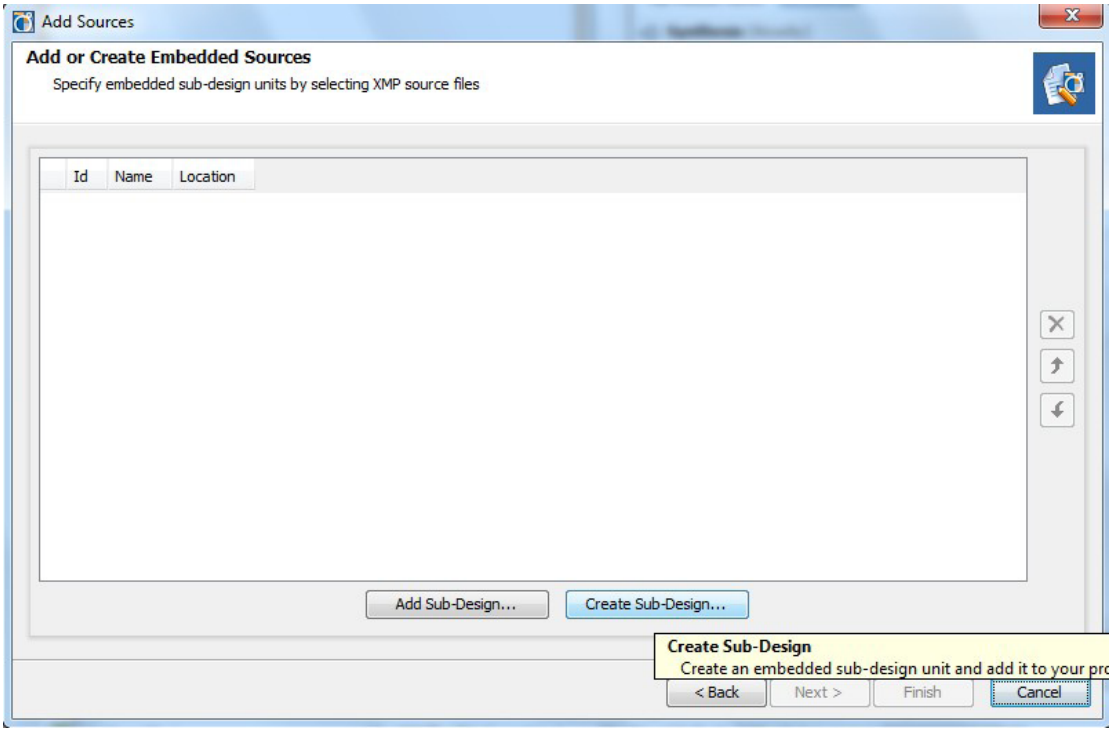
Σχήμα 4.12- How To Add Sources

Όπως αναφέραμε και στην αρχή θέλουμε να δημιουργήσουμε ένα ενσωματωμένο σύστημα. Ως συνέπεια, επιλέγουμε Add or Create Embedded Sources. Αυτή η επιλογή θα επιτρέψει την προσθήκη του Microblaze στο σύστημά μας.



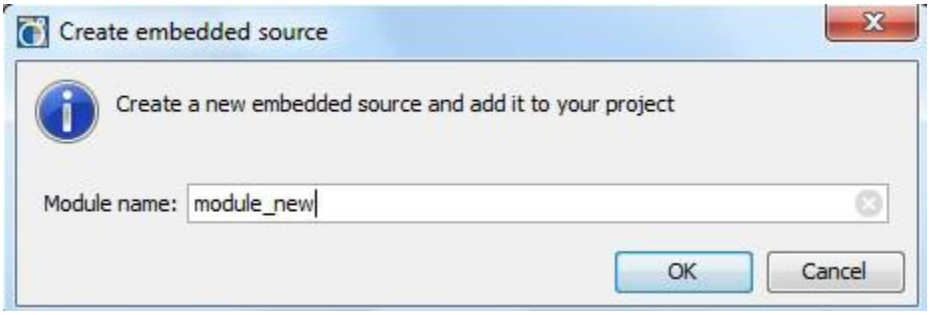
Σχήμα 4.13- Add Sources

Αμέσως εμφανίζεται ένα παράθυρο στο οποίο ο χρήστης έχει δύο επιλογές. Ή να προσθέσει ένα σύστημα ή να δημιουργήσει καινούριο. Επειδή εμείς δεν έχουμε δημιουργήσει κάποιο ακόμη, θα επιλέξουμε Create Sub-Design.



Σχήμα 4.14- Add or Create Embedded Sources

Ένα μικρό παράθυρο ζητάει να ονομάσουμε το embedded source που θα δημιουργήσουμε:



Σχήμα 4.15- Module Name

Μέχρι αυτό το σημείο έχουμε επιλέξει όλα τα βασικά στοιχεία που απαιτούνται για τη δημιουργία της πραγματικής πλατφόρμας και πλέον αρχίζουμε να εργαζόμαστε στο Xilinx Platform Studio, όπως ακριβώς θα περιγράψουμε στο κεφάλαιο 5.

Κεφάλαιο 5

Embedded Development Kit (EDK)

5.1 Εισαγωγή στο Embedded Development Kit (EDK)

Το EDK, Embedded Development Kit, αποτελεί μία συλλογή σχεδιαστικών εργαλείων, καθώς και πολλών περιφερειακών, με τα οποία είναι δυνατόν να χτιστεί ένα ενσωματωμένο σύστημα επεξεργαστή, χρησιμοποιώντας τον MicroBlaze ή τον PowerPC. Τα σχεδιαστικά εργαλεία χωρίζονται σε δύο μεγάλες κατηγορίες, μία που αφορά τα εργαλεία σχεδιασμού του υλικού (hardware) και μία που αφορά το λογισμικό (software). Όλα τα εργαλεία λειτουργούν σε συνεργασία με το XPS, Xilinx Platform Studio. Το κομμάτι του σχεδιασμού του υλικού περιλαμβάνει την αυτόματη δημιουργία μιας πλατφόρμας υλικού, hardware platform, και την μετέπειτα τροποποίηση και επέκταση αυτής, ώστε να περιλαμβάνει τις επιθυμητές hardware λειτουργίες του χρήστη. Στο σημείο αυτό, το EDK διαθέτει επίσης ενσωματωμένους πυρήνες πνευματικής ιδιοκτησίας (Intellectual Property) συμπεριλαμβανομένων επεξεργαστών και περιφερειακών. Τέλος, στο κομμάτι που αφορά το λογισμικό, ο χρήστης προσθέτει τις δικές του software εφαρμογές μέσω του Software Development Kit (SDK).

Συνοπτικά, το EDK περιλαμβάνει τα εξής εργαλεία και λειτουργίες:

Hardware Development and Verification

•Xilinx Platform Studio: Ένα ολοκληρωμένο σχεδιαστικό περιβάλλον στο οποίο μπορεί να δημιουργηθεί ένα ενσωματωμένο hardware σχέδιο.

•Base System Builder Wizard: Επιτρέπει την γρήγορη δημιουργία ενός 24λειτουργικού ενσωματωμένου συστήματος χρησιμοποιώντας όλα τα διαθέσιμα χαρακτηριστικά της αναπτυξιακής πλακέτας.

•Create and Import Peripheral Wizard: Βοηθάει τον σχεδιαστή να προσθέσει τα δικά του περιφερειακά στο σύστημα.

•Coprocessor Wizard: Επιτρέπει την προσθήκη ενός coprocessor σε μια CPU.

•Platform Generator (PlatGen): Δομεί το προγραμματιζόμενο σύστημα στο τσιπ στη μορφή μιας HDL και συνθέτει τα αρχεία netlist.

•XPS Command Line: Επιτρέπει την εκτέλεση των ροών ενσωματωμένων σχεδίων από το command line.

•Bus Functional Model: Βοηθάει στην απλοποίηση της επαλήθευσης ενός περιφερειακού δημιουργώντας ένα μοντέλο του περιβάλλοντος του διαύλου.

•Simulation Model Generator (Simgen): Δημιουργεί το μοντέλο προσομοίωσης του υλικού.

•Simulation Library Compiler (Compxlib): Συνθέτει τις βιβλιοθήκες προσομοίωσης του EDK για τον προσομοιωτή.

Software Development and Verification

•Software Development Kit: Είναι ένα ολοκληρωμένο σχεδιαστικό περιβάλλον που επιτρέπει την ανάπτυξη software εφαρμογών.

•Library Generator (Libgen): Κατασκευάζει ένα BSP (Board Support Package) που περιλαμβάνει μια προσαρμοσμένη συλλογή βιβλιοθηκών λογισμικού, drivers και ΛΣ.

•GNU Compiler Tools: Δημιουργεί μια software εφαρμογή βασιζόμενη στις πλατφόρμες που δημιουργήθηκαν από το Libgen.

•Xilinx Microprocessor Debugger: Χρησιμοποιείται για download του λογισμικού και αποσφαλμάτωση.

•GNU Debugger: Είναι ένα ολοκληρωμένο σχεδιαστικό περιβάλλον που χρησιμοποιείται για την αποσφαλμάτωση του λογισμικού.

•Bitstream Initializer (Bitinit): Ανανεώνει ένα Bitstream για να αρχικοποιήσει την on-chip μνήμη εντολών με το εκτελέσιμο του software.

•Debug Configuration Wizard: Αυτοματοποιεί τις διαδικασίες αποσφαλμάτωσης τόσο της πλατφόρμας υλικού όσο και λογισμικού.

•System ACE File Generator (GenACE): Δημιουργεί ένα αρχείο Xilinx System ACE που βασίζεται στο Bitstream και το εκτελέσιμο του software.

•Flash Memory Programmer: Επιτρέπει τη χρήση του επεξεργαστή για τον προγραμματισμό της on-board Common Flash Interface (CFI).

•Format Revision Tool and Version Management Wizard: Ανανεώνει τα αρχεία του project στο τελευταίο format. Ο Version Management Wizard βοηθάει στη μετατροπή παλαιότερων project στην παρούσα έκδοση.

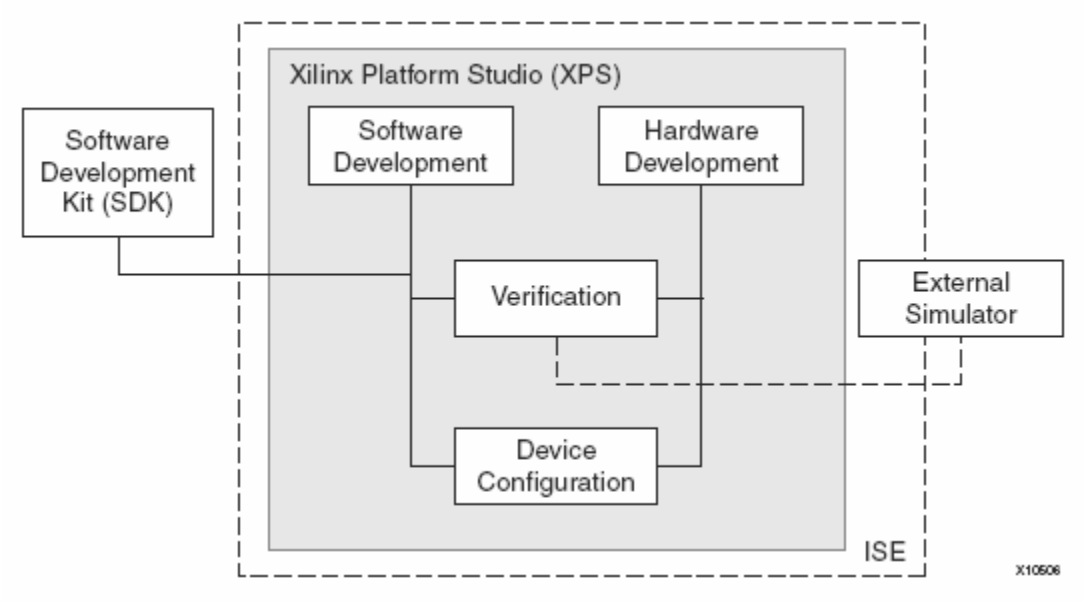
•Platform Specification Utility (PsfUtility) and PSF2Edward Program: Η PsfUtility επιτρέπει την αυτόματη δημιουργία Microprocessor Peripheral Definition (MPD) αρχείων.

•Microprocessor Peripheral Definition Translation tool (MPDX): Το MPDX είναι ένα εργαλείο μετάφρασης.

Από τα παραπάνω εργαλεία που παρουσιάστηκαν, θα επεκταθούμε σε εκείνα που χρησιμοποιήθηκαν για τη σχεδίαση και την υλοποίηση του συστήματος μας.

5.1.1 Ροή Σχεδίασης Ένθετου Συστήματος

Το παρακάτω διάγραμμα δείχνει τη γενική ροή σχεδίαση ενός ολοκληρωμένου ένθετου συστήματος.



Σχήμα 5.1 – Ροή Σχεδίασης με το EDK

Όπως φαίνεται και στο σχήμα τα βασικά στάδια κατά τη διαδικασία της σχεδίασης είναι τα εξής:

- Ανάπτυξη του Hardware τμήματος της σχεδίασης
- Ανάπτυξη του Software τμήματος της σχεδίασης
- Έλεγχος της λειτουργικότητας της σχεδίασης
- Διαμόρφωση της διάταξης που θα “φορτωθεί” η σχεδίαση

5.2 Xilinx Platform Studio (XPS)

Σχήμα 5.2 – Xilinx Platform Studio (XPS)

5.2.1 Εισαγωγή

Το XPS είναι ένα ολοκληρωμένο περιβάλλον για την προδιαγραφή των ροών του λογισμικού και του υλικού ενός ενσωματωμένου συστήματος με επεξεργαστή. Προσφέρει αρχικά τη δυνατότητα να προσθέσουμε πυρήνες όπως επεξεργαστές και περιφερειακά, να μεταβάλουμε τις παραμέτρους αυτών των πυρήνων και να κατασκευάσουμε συνδέσεις διαύλων και σημάτων για την δημιουργία ενός αρχείου MHS. Επιτρέπει επίσης τη δημιουργία και την παρακολούθηση του διαγράμματος block και της αναφοράς του συστήματος. Τέλος, επιτρέπει την εξαγωγή των αρχείων προδιαγραφών του υλικού στο SDK.

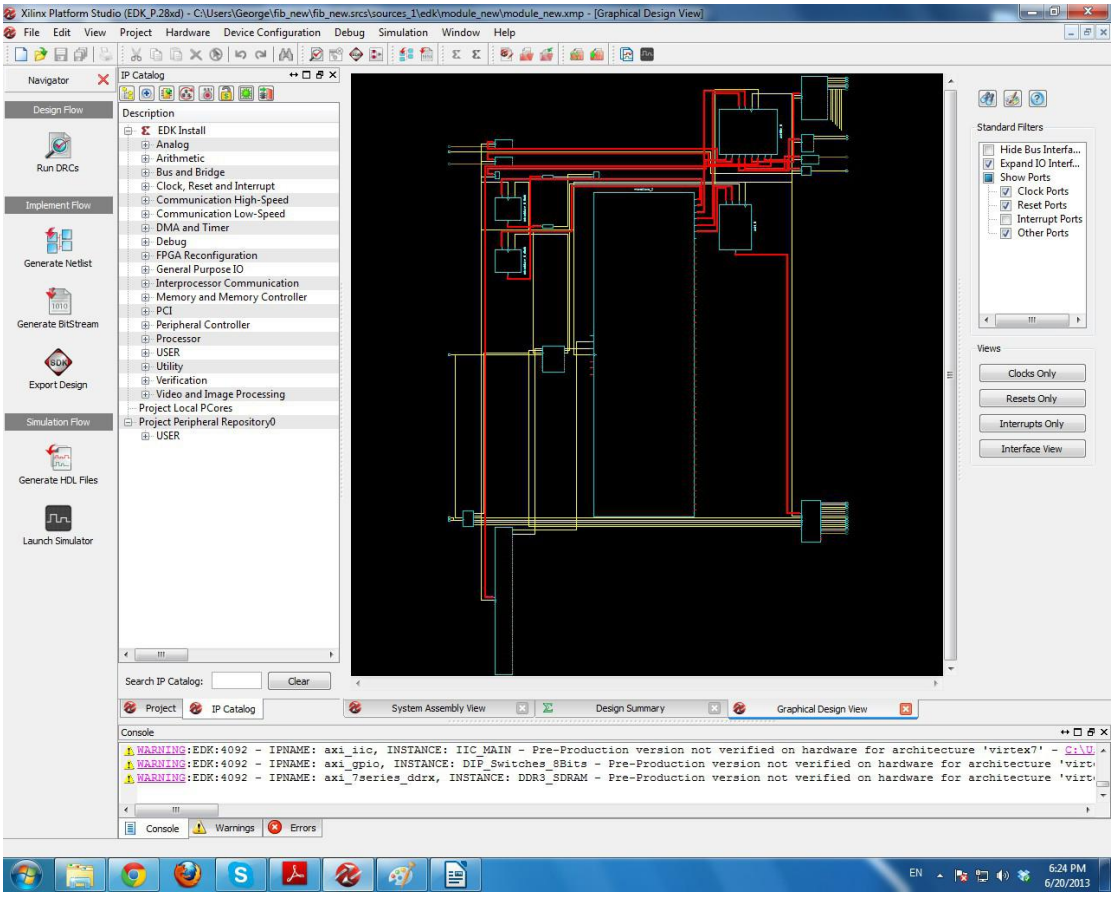
5.2.2 Διαχείριση του Project

Ένα νέο project μπορεί να ξεκινήσει με δύο τρόπους, όσες και οι επιλογές του υπό-μενού New Project του μενού File. Επιλέγοντας Base System Builder, εμφανίζεται ένας βοηθός που παρέχει καθοδήγηση βήμα προς βήμα για την εκκίνηση του project, για περιορισμένο όμως αριθμό καρτών μεταξύ των οποίων είναι και η κάρτα Virtex 7 VC707

Evaluation Kit, ενώ με την επιλογή Platform Studio το νέο project πρέπει να συσταθεί από τον χρήστη μέσω του XPS ή να φορτωθεί από ένα ήδη υπάρχων αρχείο MHS. Ένα αρχείο XMP κρατάει πληροφορίες του project για λογαριασμό του XPS, όπως για παράδειγμα την θέση των πηγαίων αρχείων κώδικα μιας εφαρμογής. Εάν η υλοποίηση πρόκειται να γίνει σε κάρτα διαφορετική από αυτές που υποστηρίζει ο βοηθός υπάρχει και η δυνατότητα να αλλάξει είτε η συσκευή, είτε το πακέτο επιλέγοντας Project Options στο μενού Options.

5.2.3 Η διεπαφή του XPS

Στο σχήμα φαίνεται ότι το XPS χωρίζεται σε τέσσερα παράθυρα.



Σχήμα 5.3 – Περιβάλλον Σχεδίασης στο XPS

- Στο αριστερό μέρος του παραθύρου του XPS, το flow είναι χωρισμένο σε τρεις κατηγορίες. Το Design Flow με επιλογή το Run DRCs, το Implement Flow, όπου μπορούμε να δημιουργήσουμε το Netlist, το BitStream ή να κάνουμε Export το SDK και το Simulation Flow, όπου υπάρχει η δυνατότητα για Generate HDL Files και Simulator.
- Δίπλα ακριβώς, υπάρχει μία καρτέλα που ονομάζεται IP Catalog , όπου είναι δυνατόν να προστεθούν και να αφαιρεθούν cores (IPs) από το σύστημα, να ορισθούν οι συνδέσεις των επιλεχθέντων cores στους αντίστοιχους διαδρόμους δεδομένων, να ορισθούν οι θύρες των cores και τα χαρακτηριστικά τους, και τέλος να ορισθούν οι παράμετροι λειτουργίας για τα ίδια τα cores. Αξίζει να σημειωθεί

ότι τα Pcores που βρίσκονται σε αυτόν τον κατάλογο μπορεί είτε να είναι έτοιμα και να παρέχονται από το ίδιο το εργαλείο είτε να τα έχει δημιουργήσει ο ίδιος ο χρήστης, ανάλογα με τις απαιτήσεις και τις ανάγκες που έχει.

- Το κεντρικό και μεγαλύτερο παράθυρο ουσιαστικά απεικονίζει το όλο το ενσωματωμένο σύστημα μαζί με τα βασικά χαρακτηριστικά του και λεπτομέρειες , όπως τα Bus Interfaces, τα Ports και τις Addresses όλων των components, με Design Summary και με Graphical Design View.
- Το τρίτο από τα παράθυρα του XPS είναι αυτό που βρίσκεται στο κάτω μέρος, και το οποίο χρησιμεύει για την εμφάνιση των ενεργειών του XPS και μηνυμάτων που αφορούν λάθη ή ειδοποιήσεις του XPS ή των εργαλείων που αυτό καλεί.

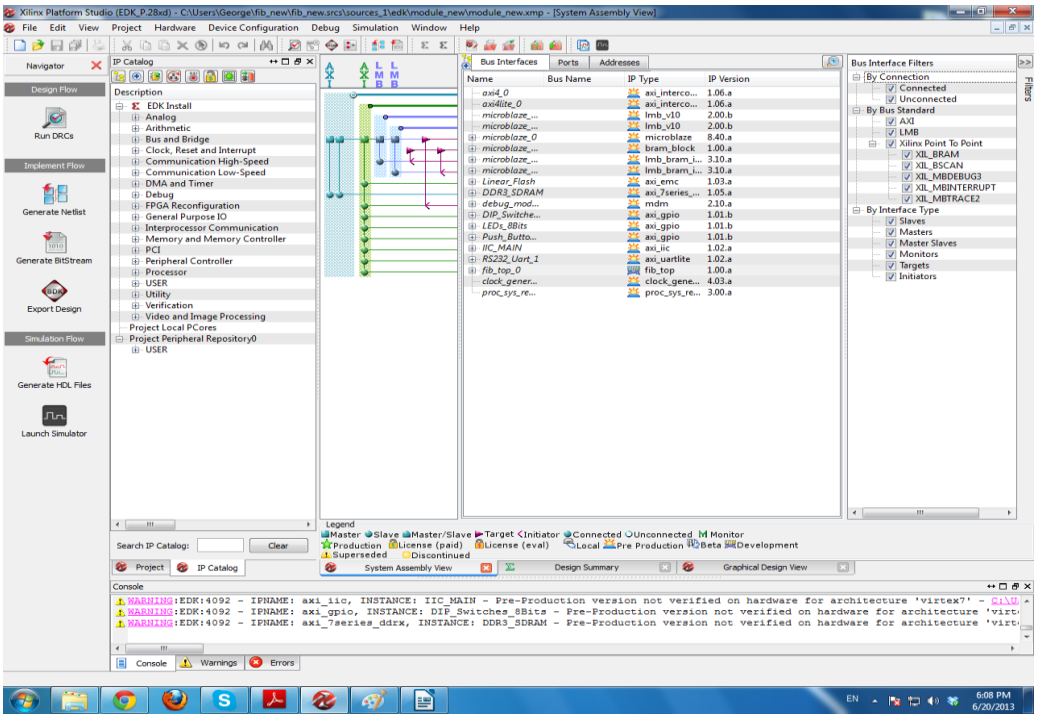
5.2.4 Διαχείριση των πλατφόρμων υλικού, λογισμικού και προσομοίωσης

Προκειμένου ο χρήστης να τροποποιεί της πλατφόρμες υλικού, λογισμικού, και προσομοίωσης, το XPS διαθέτει μια σειρά από λειτουργίες που αλλάζουν τα χαρακτηριστικά και τις παραμέτρους του συστήματος.

Στο αρχείο System Assembly εμφανίζονται τρεις καρτέλες :

- Bus Interface

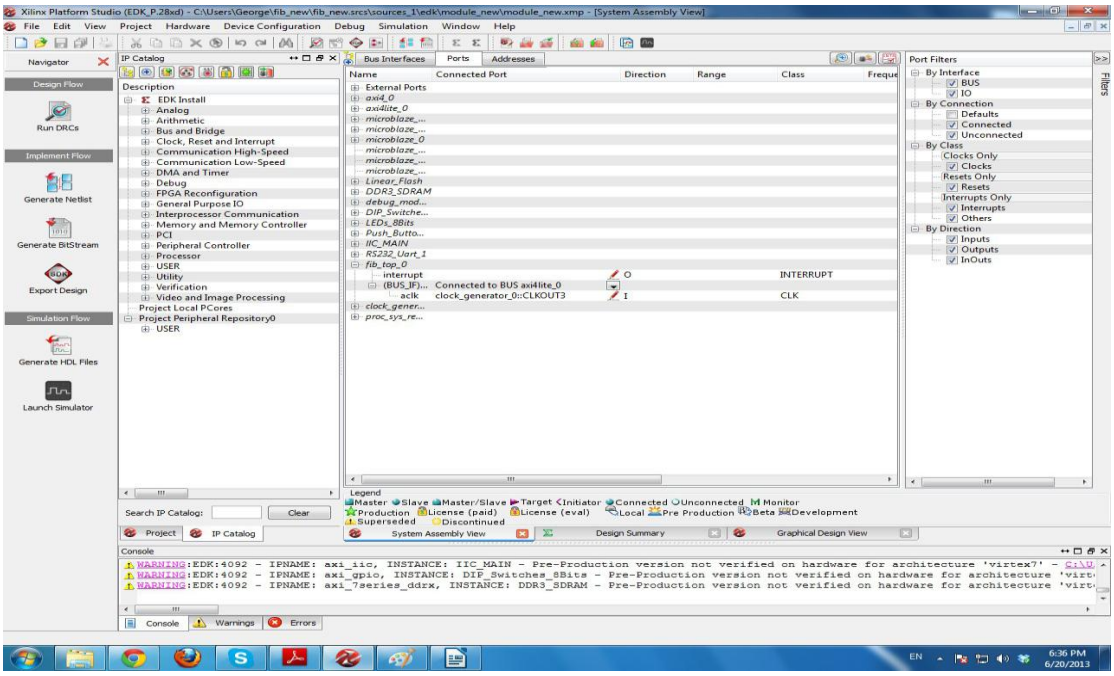
Στην καρτέλα αυτή βρίσκονται τα περιφερειακά που αποτελούν το σύστημα.Είναι δυνατόν να προστεθούν και να αφαιρεθούν cores (IPs) από το σύστημα, να ορισθούν οι συνδέσεις των επιλεχθέντων cores στους αντίστοιχους διαδρόμους δεδομένων και τέλος να ορισθούν οι παράμετροι λειτουργίας για τα ίδια τα cores.



Σχήμα 5.4 – Bus Interface

- Ports

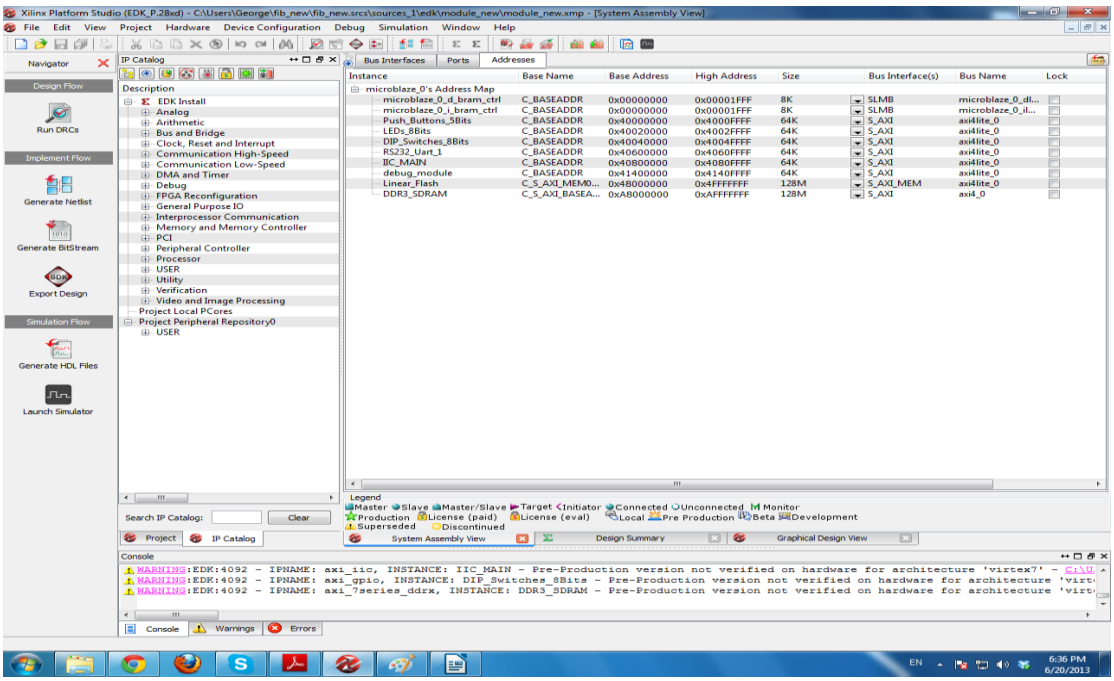
Εδώ ορίζονται οι θύρες όλων των cores και τα χαρακτηριστικά τους, και μπορούμε να κάνουμε οποιεσδήποτε αλλαγές με δεξί κλικ πάνω στην κάθε θύρα.



Σχήμα 5.5 – Ports

- Addresses

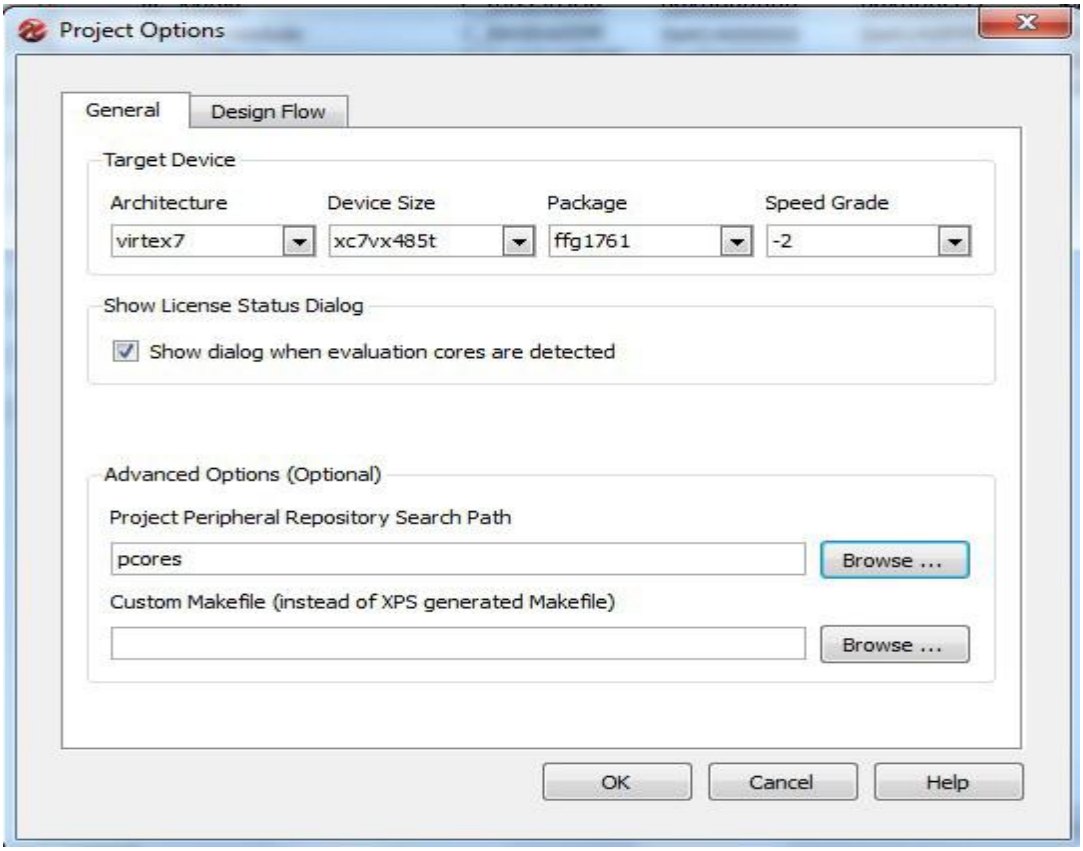
Σε αυτήν την καρτέλα φαίνεται η διεθυνσιοδότηση του κάθε περιφερειακού. Μπορούμε να αλλάξουμε τη διεύθυνση μνήμης που καταλαμβάνει το κάθε περιφερειακό αλλάζοντας την high και base address του.



Σχήμα 5.6 – Addresses

5.2.5 Ιδιότητες του Project

Η επιλογή από το μενού Options του XPS, Options → Project Options, ανοίγει ένα παράθυρο διαλόγου που επιτρέπει στον χρήστη να ορίσει διάφορες επιλογές του project.



Σχήμα 5.7 – Project Options

5.3 Βοηθός δημιουργίας συστήματος, Base System Builder

Ο βοηθός Base System Builder (BSB), οδηγεί στην εύκολη και γρήγορη δημιουργία του project του XPS ενός συστήματος, το οποίο απευθύνεται σε μία συγκεκριμένη αναπτυξιακή κάρτα. Βασιζόμενο στην επιλογή της κάρτα, ο βοηθός BSB προσφέρει μια σειρά από επιλογές σχετικές με αυτό, που οδηγούν στην δημιουργία ενός βασικού και στοιχειώδους συστήματος. Με το πέρας του βοηθού, έχει δημιουργηθεί ένα αρχείο MHS του συστήματος αυτού, που περιέχει τις ιδιότητές του, το οποίο φορτώνεται σε ένα project του XPS. Από εκείνο το σημείο, είναι στην ευχέρεια του χρήστη να βελτιώσει περαιτέρω το σύστημα πριν την υλοποίηση.

Ο βοηθός ξεκινάει με την επιλογή File → New Project → Base System Builder.

Η φύση του βοηθού BSB είναι τέτοια ώστε να επιτρέπει την δημιουργία απλών και βασικών συστημάτων, χωρίς προχωρημένες αρχιτεκτονικές και ειδικές διαμορφώσεις, όπως χρήση άνω του ενός επεξεργαστή ή επεξεργασία του χάρτη διευθύνσεων. Παρά το γεγονός αυτό μετά την ολοκλήρωση του βοηθού υπάρχει η δυνατότητα επέκτασης και βελτίωσης του

συστήματος μέσω του XPS. Επομένως το σύστημα που παράγει ο βοηθός BSB μπορεί να θεωρηθεί σαν την αφετηρία για την κατασκευή ενός πιο πολύπλοκου συστήματος.

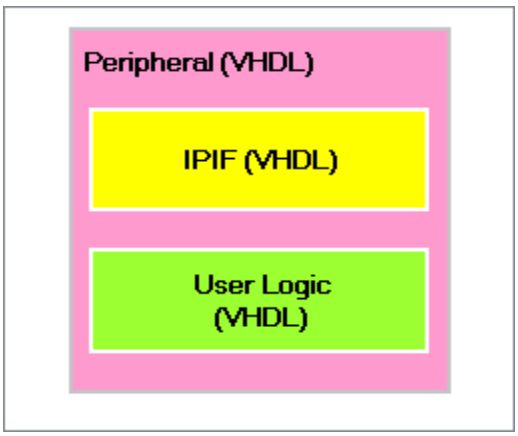
5.4 Βοηθός δημιουργίας / εισαγωγής περιφερειακού

5.4.1 Εισαγωγή

Ο Create and Import Peripheral Wizard (CIP) βοηθάει στη δημιουργία περιφερειακών και στην εισαγωγή τους στο συμβατό project του XPS. Δημιουργεί επίσης οδηγούς, που βοηθούν στην υλοποίηση του περιφερειακού χωρίς να απαιτείται λεπτομερής κατανόηση των πρωτοκόλλων διαύλων,των συμβάσεων της ονοματολογίας ή των μορφών των ειδικών αρχείων διεπαφών που χρησιμοποιούνται στο XPS. Στην λειτουργία εισαγωγής περιφερειακού, το συγκεκριμένο εργαλείο δημιουργεί τα αρχεία διεπαφής και τις δομές καταλόγου που είναι απαραίτητα ώστε το περιφερειακό να είναι ορατό στα διάφορα εργαλεία του XPS. Μετά την εισαγωγή του, το περιφερειακό είναι διαθέσιμο στη βιβλιοθήκη των περιφερειακών του XPS. Με την εισαγωγή ή την δημιουργία ενός περιφερειακού, δημιουργούνται αυτόματα τα αρχεία Microprocessor Peripheral Definition (MPD) και Peripheral Analyze Order (PAO). Το αρχείο MPD καθορίζει την διεπαφή για το περιφερειακό ενώ το αρχείο PAO, ορίζει στο Platgen και το Simgen ποια αρχεία HDL απαιτούνται για τη σύνθεση ή την προσομοίωση του περιφερειακού καθώς και τη σειρά αυτών των αρχείων.

Για τις ανάγκες των εφαρμογών που σχεδιάστηκαν και υλοποιήθηκαν, ο βοηθός χρησιμοποιήθηκε με σκοπό την δημιουργία νέων περιφερειακών, όπως για παράδειγμα το περιφερειακό που χρησιμοποιούμε και υπολογίζει έναν αριθμό Fibonacci. Η διαδικασία της δημιουργίας ενός περιφερειακού μέσω του βοηθού αποτελείται από διαδοχικά παράθυρα διαλόγου τα οποία καθοδηγούν τον χρήστη. Στην καλύτερη περίπτωση το μόνο που απαιτείται από τον χρήστη είναι ο συγγραφή του σώματος ενός αρχείου HDL, στο οποίο περιγράφεται η λειτουργία του περιφερειακού. Ακόμα όμως και για την συγγραφή του κομματιού αυτού υπάρχουν κατευθύνσεις που δεν καθιστούν αναγκαία την πλήρη γνώση των σημάτων και της δομής που χρησιμοποιείται.

Κάθε περιφερειακό που είναι συμμορφώσιμο με τις συμβάσεις του EDK περιέχει τα τρία παρακάτω μέρη (components):



Σχήμα 5.8 – Peripheral Components

- Μία διεπαφή με τον διάδρομο στον οποίο συνδέεται. Η σύνδεση πραγματοποιείται μέσω των θυρών που διαθέτει το περιφερειακό, για να επικοινωνεί με το υπόλοιπο σύστημα
- Την διεπαφή IPIF, IP Interface. Μέσω της IPIF συνδέεται το περιφερειακό με το προηγούμενο component. Εκτελεί όλες τις βασικές λειτουργίες που χρειάζεται για την διαχείριση ενός περιφερειακού, όπως αποκωδικοποίηση διευθύνσεων, διαχείριση καταχωρητών, διαχείριση διακοπών, υποστήριξη DMA. Από όλες τις λειτουργίες που υποστηρίζει υλοποιούνται μονάχα όσες απαιτούνται από το περιφερειακό του χρήστη.
- Το τελευταίο component αποτελεί την ειδική λογική που χρησιμοποιεί το περιφερειακό για τις εφαρμογές του, και η οποία δεν καλύπτεται από την IPIF. Το κομμάτι αυτό αποκαλείται user-logic.

Η ένωση του user-logic με την IPIF γίνεται με την βοήθεια μιας σειράς από θύρες, που ονομάζονται IPIC, IP Interconnect, και απλουστεύουν την υλοποίηση του user-logic.

Συμπερασματικά, ένα από τα σημαντικότερα components κάθε νέου περιφερειακού, είναι η διεπαφή IPIF. Η μία πλευρά της συνδέεται με την διεπαφή ενός από τους διαθέσιμους διαδρόμους δεδομένων και η άλλη με την διεπαφή IPIC, την οποία υλοποιεί το περιφερειακό του χρήστη. Η IPIC δεν διαφοροποιείται ανάλογα με τον διάδρομο δεδομένων που χρησιμοποιείται και είναι ειδικά σχεδιασμένη για να συνεργάζεται με ευκολία με το περιφερειακό. Η διεπαφή IPIF προσφέρει μία σειρά από βασικές, και άλλες πιο σύνθετες, λειτουργίες οι οποίες είναι στην διάθεση του χρήστη για να τις επιλέξει. Μια περιγραφή των λειτουργιών που προσφέρονται από την διεπαφή IPIF δίνεται στο σχήμα 5.9.

Λειτουργίες IPIF	Περιγραφή
Μηδενισμός (reset) από εφαρμογή, και καταχωρητής πληροφορίας, Module Information Register (MIR)	Το περιφερειακό διαθέτει μία ειδική διεύθυνση, μόνο για γράψιμο, όπου όταν γράφεται μία συγκεκριμένη λέξη, η διεπαφή IPIF δημιουργεί ένα σήμα μηδενισμού για το περιφερειακό. Με τον τρόπο αυτό το περιφερειακό μπορεί να μηδενιστεί από μία εφαρμογή λογισμικού. Το περιφερειακό έχει επίσης έναν καταχωρητή, μόνο για ανάγνωση που προσδιορίζει την έκδοση του περιφερειακού
Μεταφορά δεδομένων κατά ριπές (burst transaction) και μεταφορές Cacheline	Η λειτουργία αυτή επιτρέπει, με μία μόνο αίτηση, την μεταφορά μεγάλου όγκου δεδομένων. Η μεταφορά Cacheline υποστηρίζεται μόνο για περιφερειακά που συνδέονται στον LMB
DMA	Η διεπαφή IPIF του περιφερειακού υποστηρίζει την απευθείας πρόσβαση της μνήμης, Direct Memory Access, χωρίς την παρέμβαση του επεξεργαστή Ισχύει μόνο για περιφερειακά συνδεδεμένα στον LMB
FIFO	Η διεπαφή IPIF του περιφερειακού υποστηρίζει την λειτουργία ουράς FIFO Ισχύει μόνο για περιφερειακά συνδεδεμένα στον LMB
Μηχανισμός διακοπών στο περιφερειακό	Το περιφερειακό διαθέτη μηχανισμό συλλογής και διαχείρισης διακοπών. Το user-logic και η διεπαφή IPIF, παράγουν μία γραμμή εξόδου διακοπών, προερχόμενη από το περιφερειακό
Καταχωρητές ελεγχόμενοι από το λογισμικό	Το περιφερειακό διαθέτει έναν αριθμό καταχωρητών, οι οποίοι είναι προσβάσιμοι μέσω των εφαρμογών λογισμικού, με δικές τους διευθύνσεις.
Λειτουργία του user-logic σαν κυρίαρχο (master) περιφερειακό	Περιλαμβάνει τα σήματα της διεπαφής IPIC που αφορούν τις λειτουργίες κυρίαρχου (master) περιφερειακού. Συνοδεύεται και από απλό παράδειγμα, σε HDL, για το πως ελέγχεται ένα κυρίαρχο user-logic Ισχύει μόνο για περιφερειακά συνδεδεμένα στον LMB
Πολλαπλές περιοχές διευθύνσεων στο user-logic	Γεννά σήματα επίτρεψης για κάθε περιοχή διευθύνσεων, σε αντίθεση με το σήμα επίτρεψης που γεννάτε για κάθε καταχωρητή πυο ελέγχεται από λογισμικό. Η λειτουργία είναι χρήσιμη για περιφερειακά που χρησιμοποιούν πολλές περιοχές διευθύνσεων

Σχήμα 5.9 – Οι λειτουργίες της διεπαφής IPIF

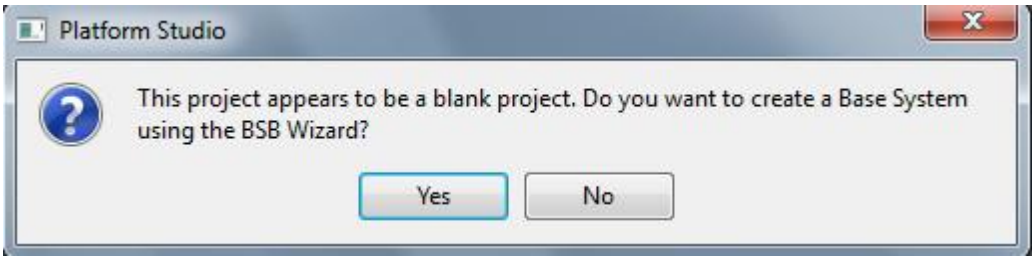
5.5 Software Development Kit (SDK)

Σχήμα 5.10 – Xilinx SDK

Το SDK παρέχει ένα αναπτυξιακό περιβάλλον για εφαρμογές λογισμικού και βασίζεται στο open-source standard του Eclipse. Υποστηρίζει την ανάπτυξη εφαρμογών λογισμικού σε συστήματα ενός ή δύο επεξεργαστών σε ομαδικό περιβάλλον (team environment). Εισάγει την περιγραφή της hardware πλατφόρμας που δημιουργεί το XPS. Επίσης το SDK έχει τη δυνατότητα να δημιουργεί και να τροποποιεί Board Support Packages (BSPs) για τρίτα ΛΣ παρέχοντας παράλληλα παραδείγματα για τον έλεγχο της λειτουργικότητας τόσο του hardware όσο και του software. Τέλος, διαθέτει ένα ολοκληρωμένο περιβάλλον για την απρόσκοπτη αποσφαλμάτωση και χαρακτηρισμό ενσωματωμένων συστημάτων.

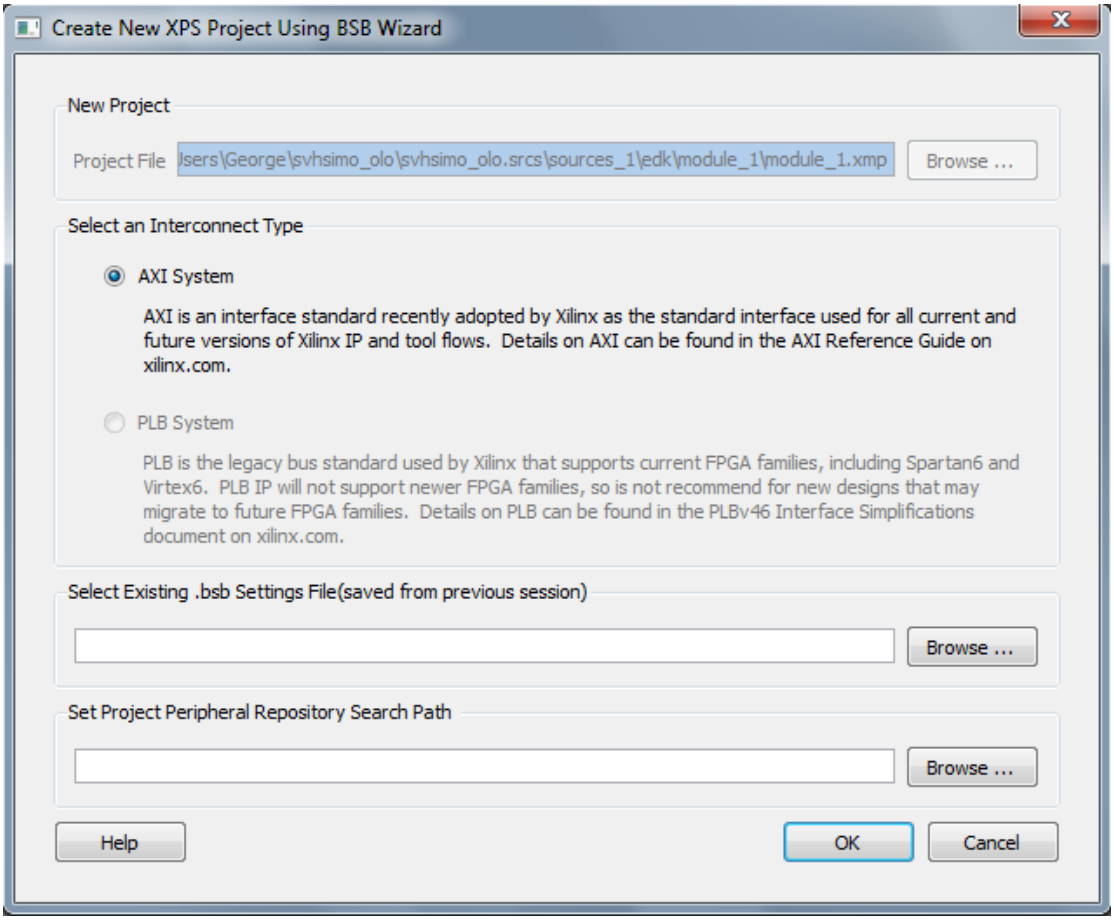
5.6 Συνέχεια Εφαρμογής

Στο προηγούμενο κεφάλαιο είχαμε διακόψει την περιγραφή του flow στο σημείο όπου αρχίζουμε να δουλεύουμε με το Xilinx Platform Studio. Ξεκινώντας θα επιλέξουμε να βοηθηθούμε από τον Base System Builder .



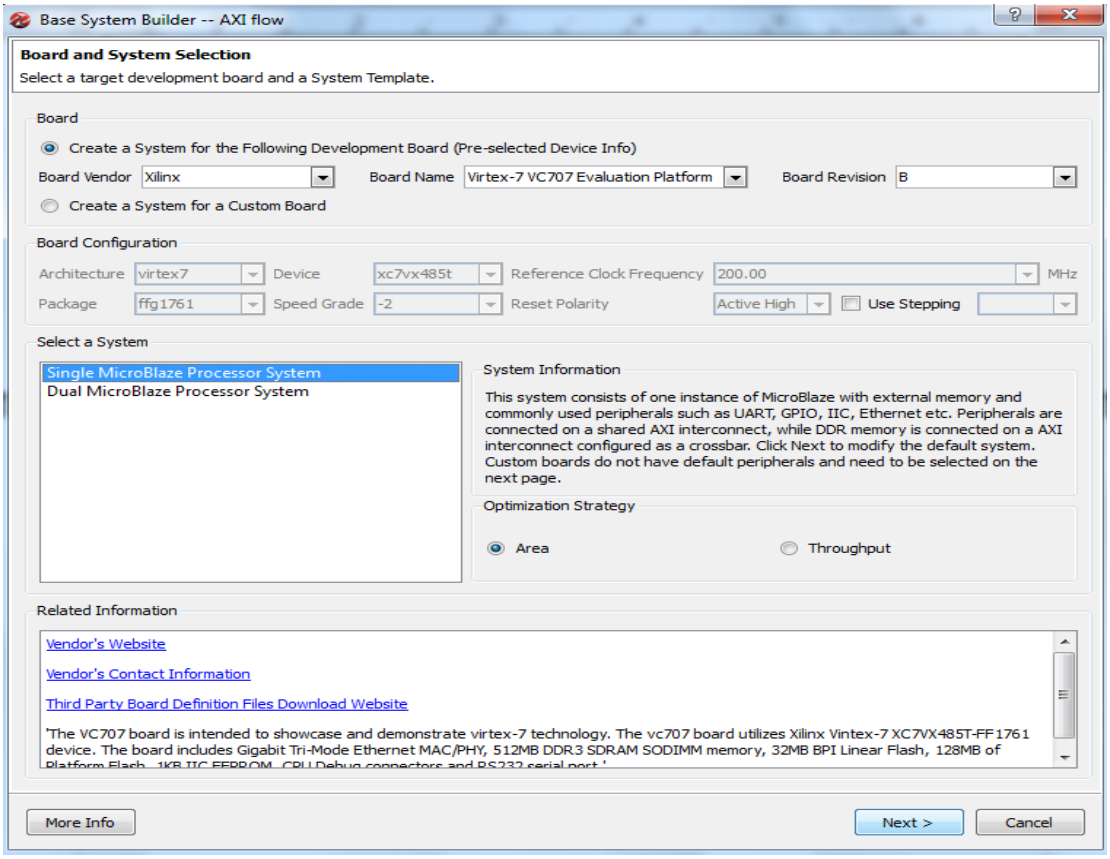
Σχήμα 5.11 – Base System Builder

Και σε νέα καρτέλα θα επιλέξουμε το σύστημα που θα δημιουργηθεί να χρησιμοποιεί το AXI System για να επικοινωνεί με τα περιφερειακά του. Στην ουσία είναι και η μόνη επιλογή αφού το Platform Studio δεν αφήνει να επιλεγεί κάτι διαφορετικό.



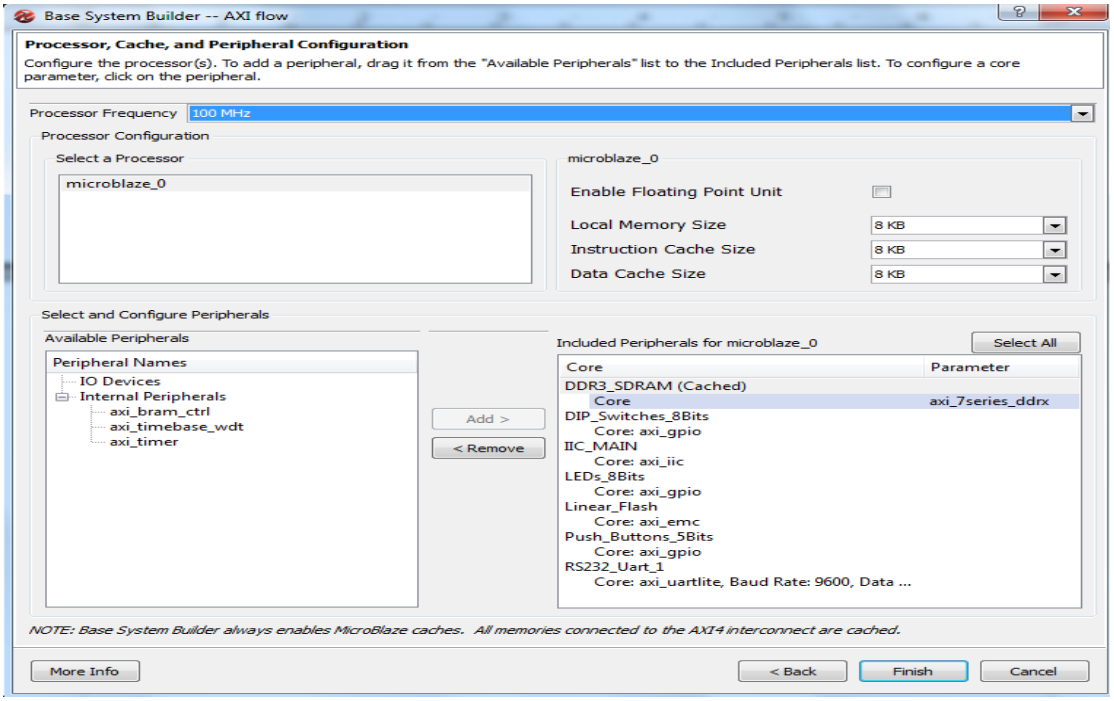
Σχήμα 5.12 – AXI System

Στην συνέχεια, μέσω του βοηθού δημιουργίας συστήματος και των στοιχείων που είχαμε δώσει στο Planahead, εντοπίζεται η πλατφόρμα που θα χρησιμοποιήσουμε και επίσης επιλέγουμε και τον επεξεργαστή που θα ενσωμματώσουμε.



Σχήμα 5.13 – Board and System Selection

Στο επόμενο βήμα διαλέγουμε απλώς κάποια by default περιφερειακά που θέλουμε να έχει το σύστημά μας και πατώντας Finish ολοκληρώνεται η δημιουργία του.

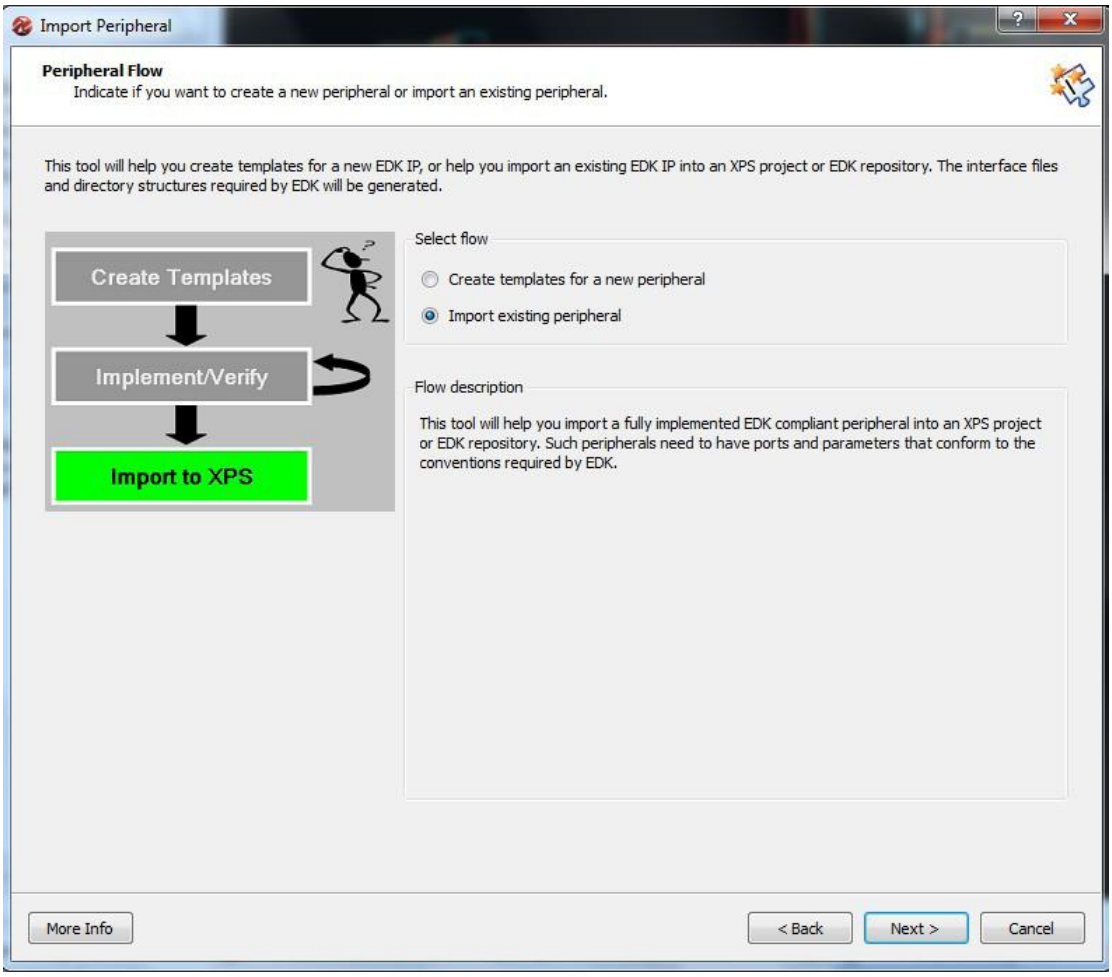


Σχήμα 5.14 – Processor, Cache and Peripheral Configuration

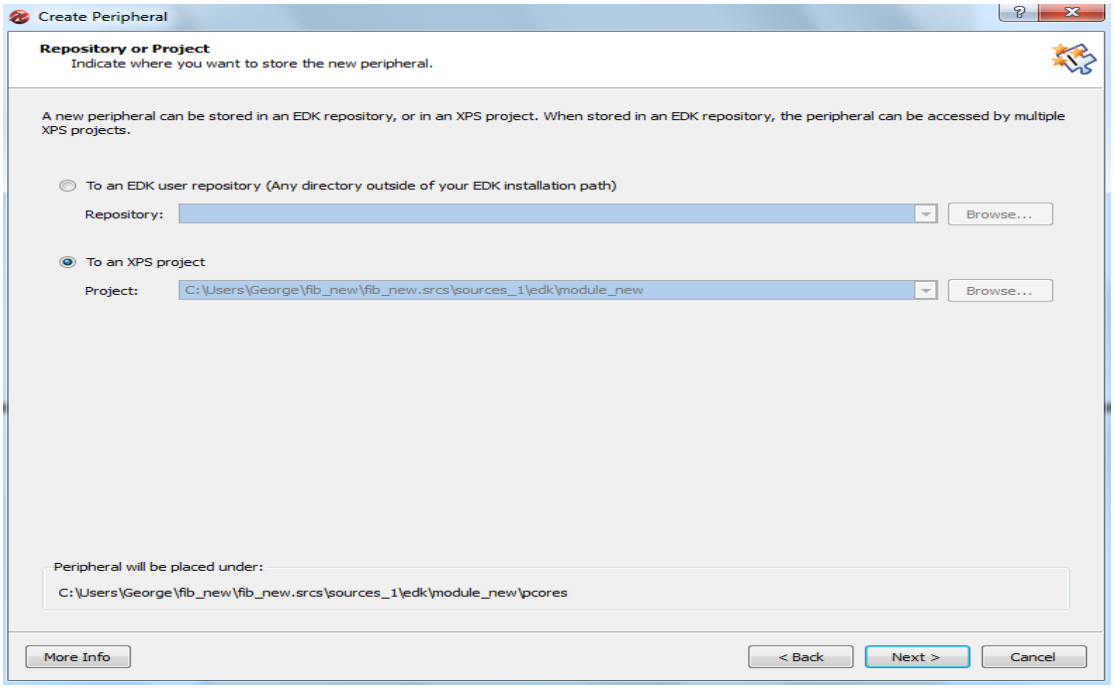
5.6.1 Εφαρμογή του Βοηθού για προσθήκη του περιφερειακού fib στον Microblaze

Στην συγκεκριμένη ενότητα θα περιγράψουμε αναλυτικά την ένωση του Pcore που δημιουργήθηκε από το εργαλείο VivadoHLS με την πλατφόρμα στην οποία έχουμε συνθέσει τον Microblaze.

Ο βοηθός ξεκινά επιλέγοντας Hardware → Create or Import Peripheral Wizard. Επιλέγοντας το Import Existing Peripheral, ο χρήστης καλείται να αποφασίσει εάν θα αποθηκεύσει τα αρχεία, και την ειδική δομή φακέλου που ακολουθούν, μέσα στους φακέλους του τρέχοντος project ή σε κάποια τοποθεσία ειδική για όλα τα περιφερειακά του χρήστη.

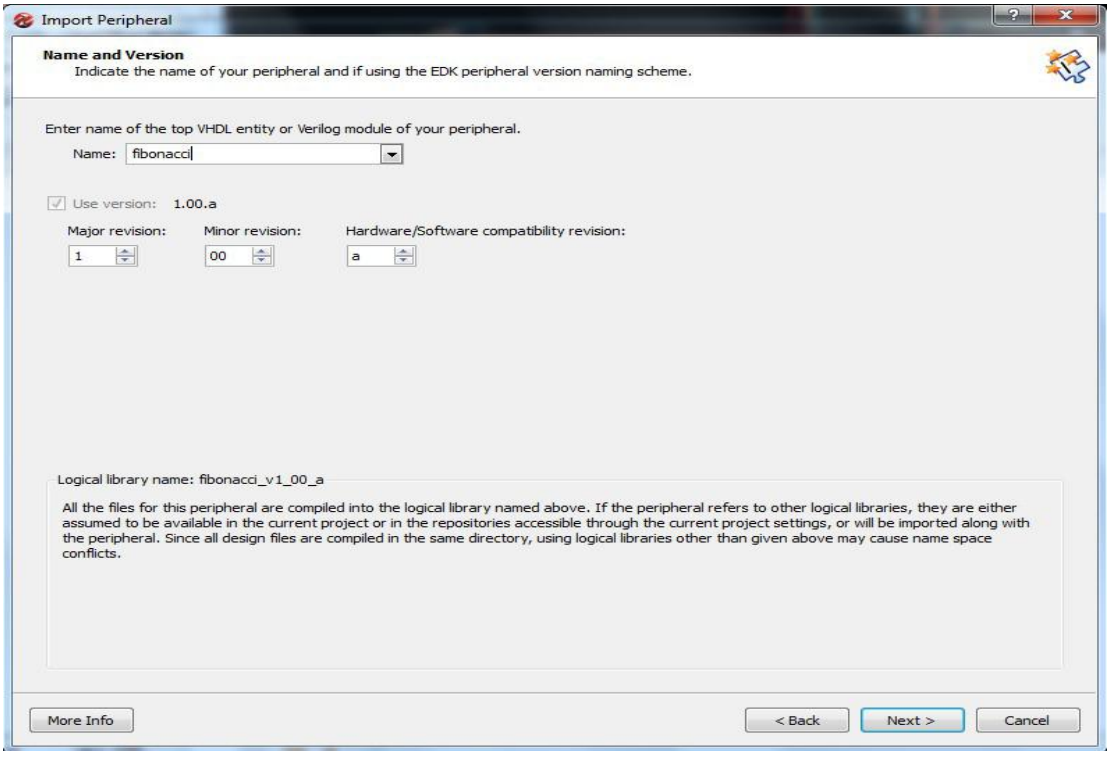


Σχήμα 5.15 – Peripheral Flow



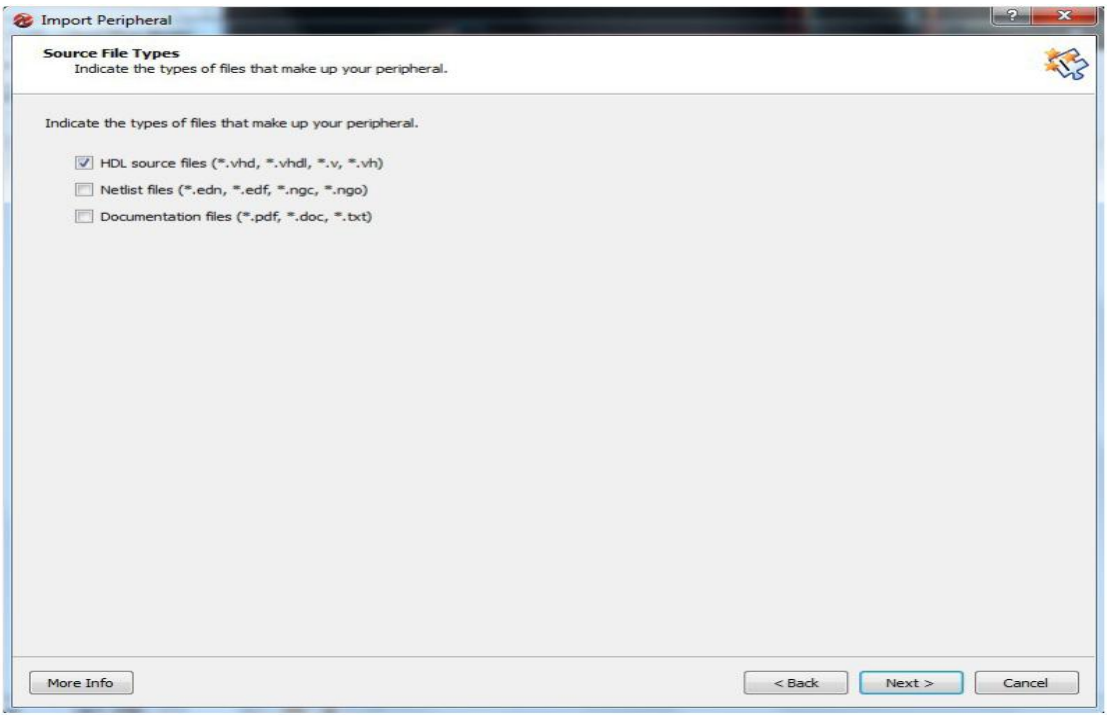
Σχήμα 5.16 – Repository or Project

Μετά την απόφαση για τον τόπο στον οποίο θα αποθηκευτούν τα αρχεία του νέου περιφερειακού, ο χρήστης πρέπει να δώσει την ονομασία του περιφερειακού που είναι ταυτόχρονα και αυτή του top-level HDL αρχείου στην ιεραρχία. Επίσης μπορούν να δοθούν αναγνωριστικά, όπως αριθμός κύριας και δευτερεύουσας αναθεώρησης και αναγνωριστικό συμβατότητας υλικού / λογισμικού.



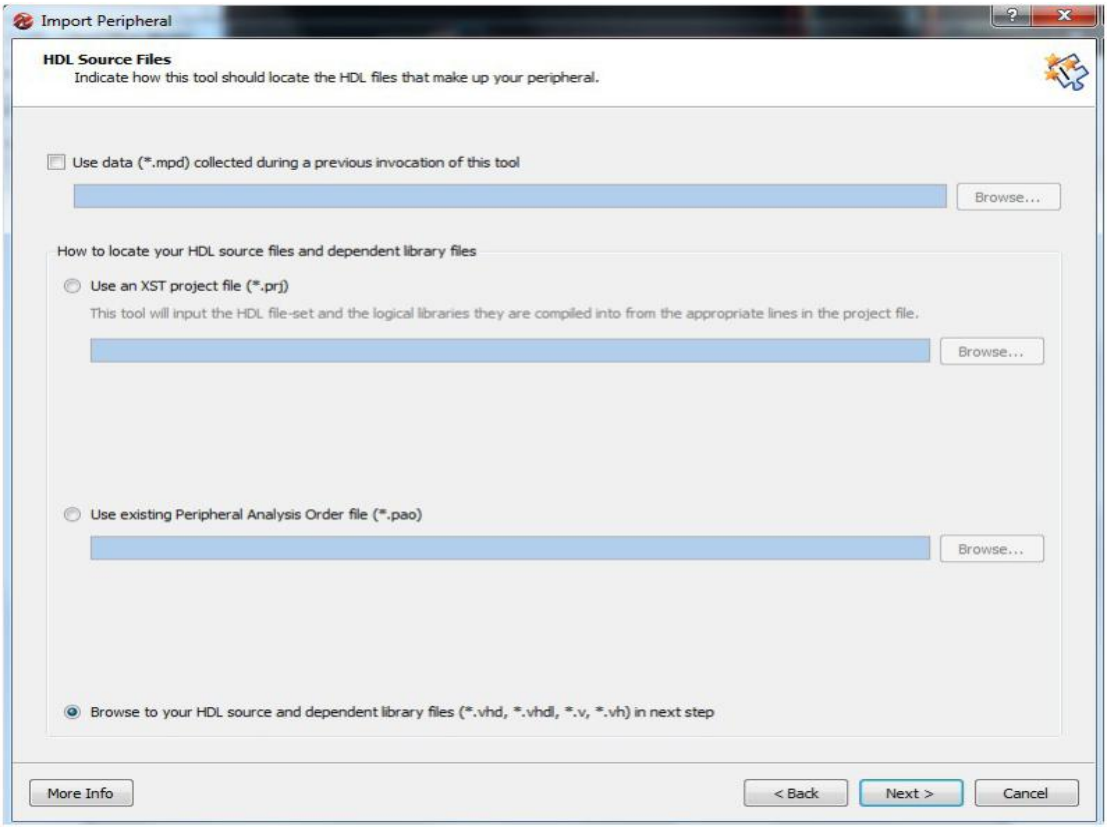
Σχήμα 5.17 – Name and Version

Επόμενο βήμα του χρήστη είναι η επιλογή του κώδικα σε Vhdl ή Verilog του Pcore που έχει δημιουργήσει και περιγράφουν τον τρόπο δημιουργίας του περιφερειακού.



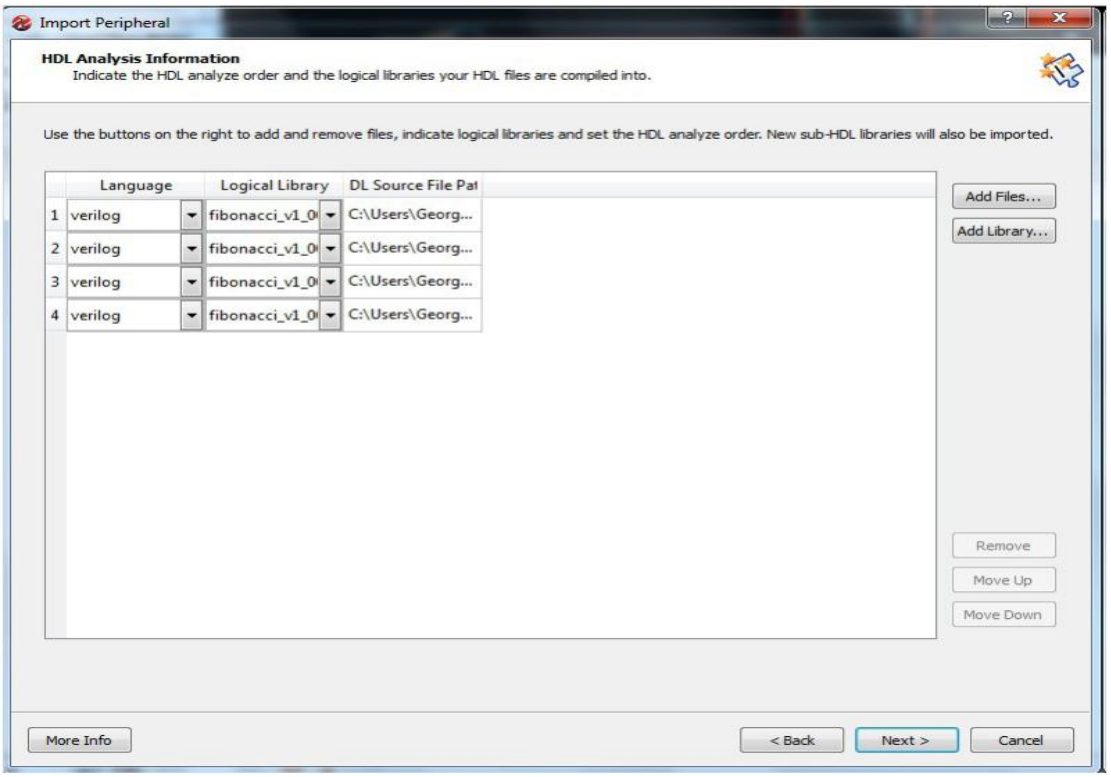
Σχήμα 5.18 – Name and Version

Μετά επιλέγουμε να ανεβάσουμε τα συγκεκριμένα αρχεία HDL στον βοηθό ώστε να χρησιμοποιηθούν για τη δημιουργία του περιφερειακού. Τα αρχεία αυτά δεν είναι άλλα από εκείνα που μας είχε η High Level Synthesis στο VivadoHLS.



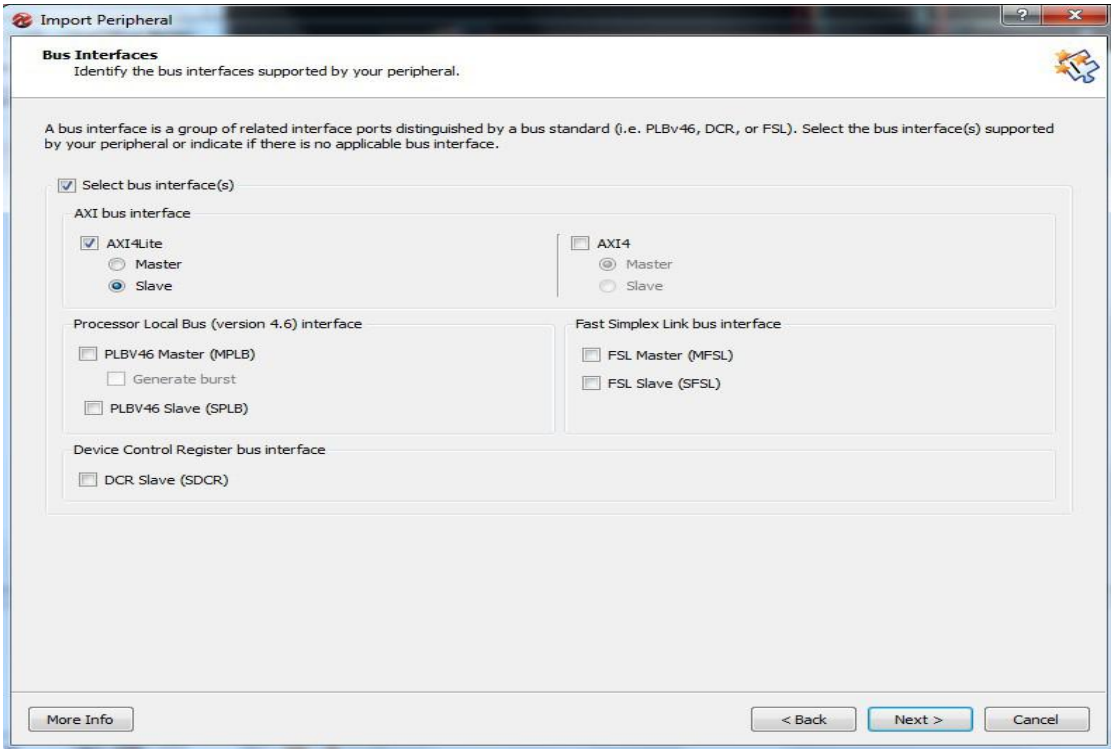
Σχήμα 5.19 – HDL Source Files

Και μόλις τα βρούμε τα προσθέτουμε πατώντας Add Files.



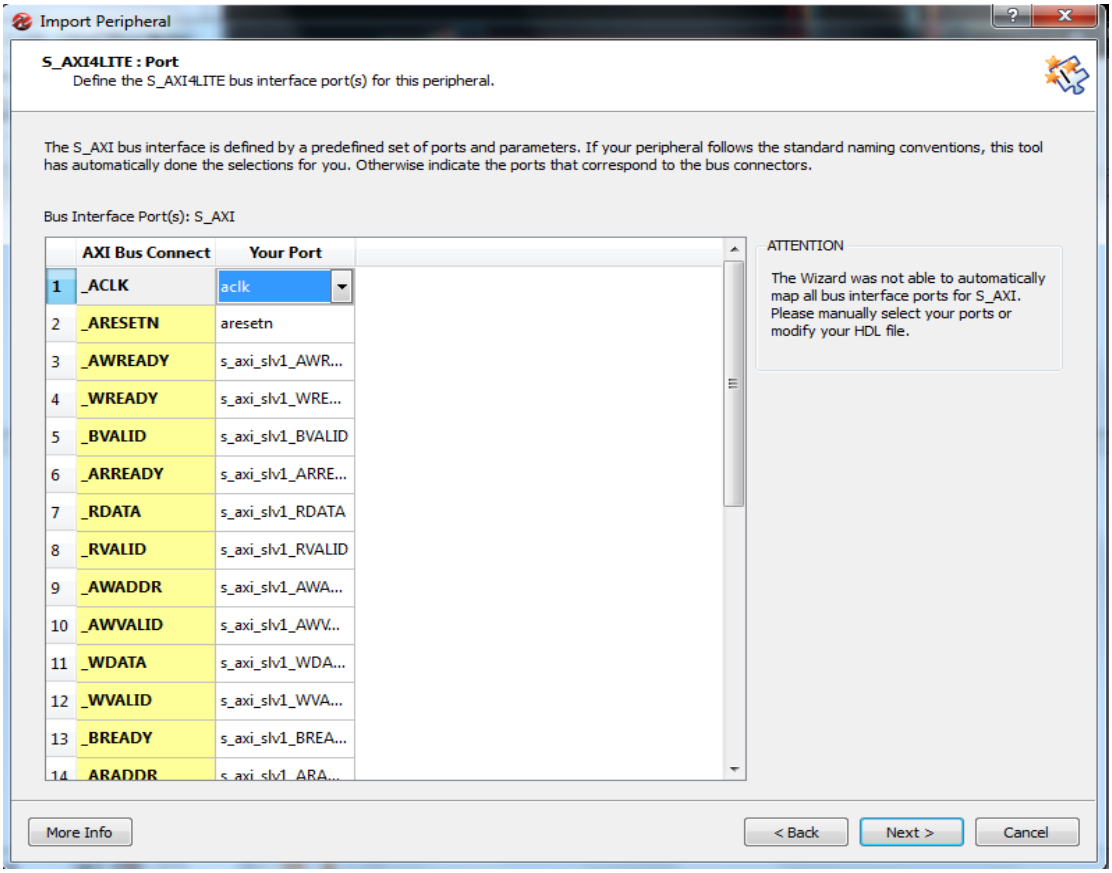
Σχήμα 5.20 – HDL Analysis Information

Επόμενο βήμα του βοηθού είναι η επιλογή του διαδρόμου δεδομένων στον οποίο θα συνδεθεί το περιφερειακό, προκειμένου να υλοποιηθεί η κατάλληλη διεπαφή. Εμείς στην σχεδίαση που υλοποιήσαμε ως Bus Interface χρησιμοποιήσαμε το AXI4Lite και μάλιστα με συμπεριφορά slave.



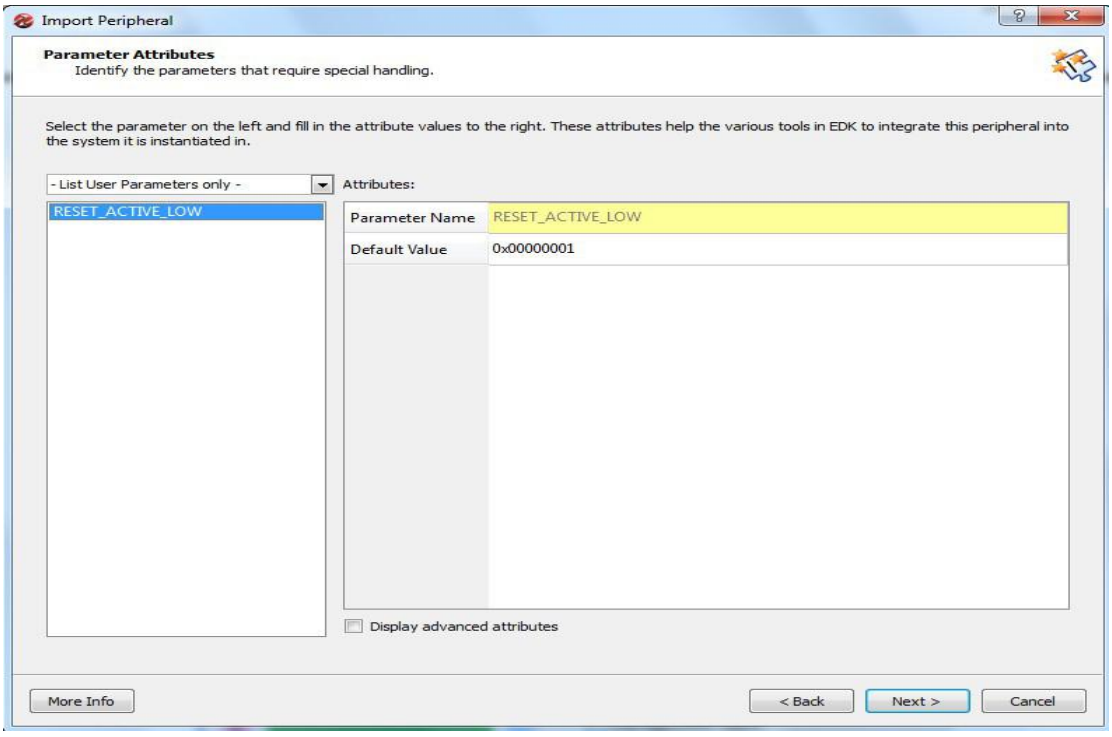
Σχήμα 5.21 – Bus Interferances

Ο βοηθός χρησιμοποιεί τον κώδικα HDL για να ορίσει τα bus interface ports για το AXI4Lite, ωστόσο κάποιοι ορισμοί πρέπει να γίνουν manually ή προσθέτοντας κάποιες παραπάνω γραμμές στον κώδικα, όπως περιγράψαμε στο τρίτο κεφάλαιο.



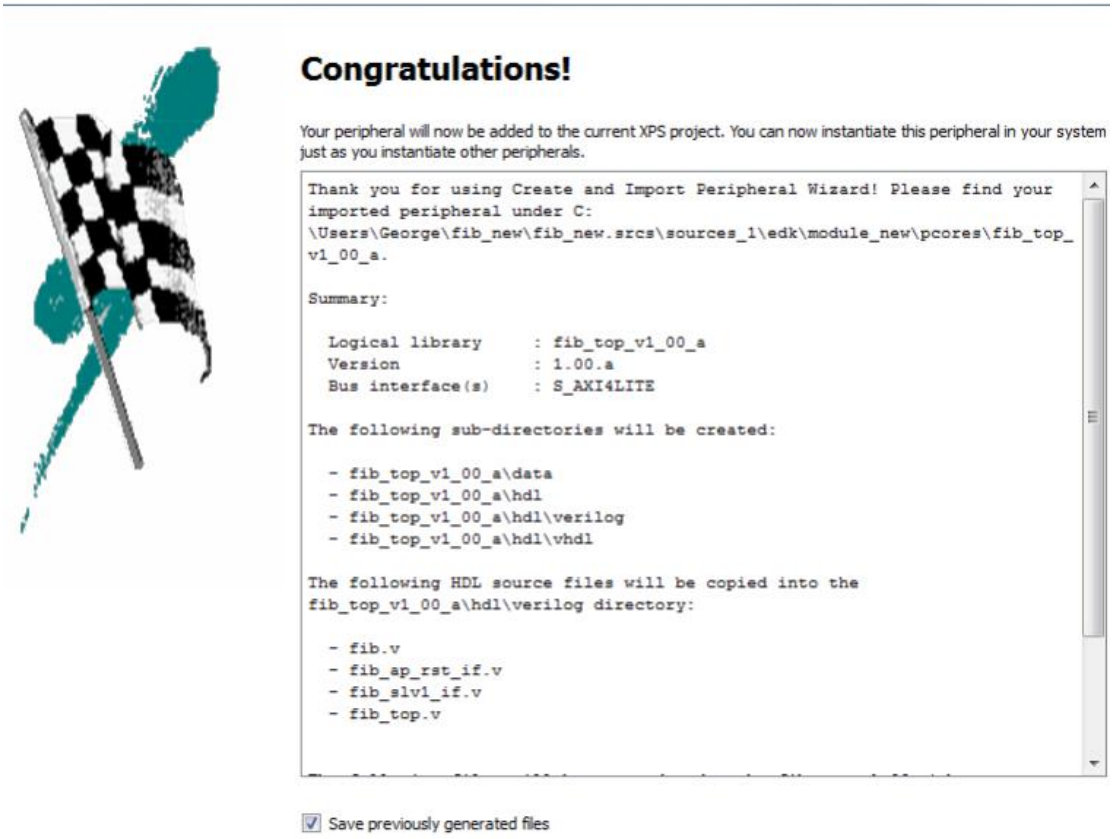
Σχήμα 5.22 – S_AXI4LITE PORT

Στην συνέχεια αρχικοποιούμε τις απαραίτητες παραμέτρους



Σχήμα 5.23 – Parameter Attributes

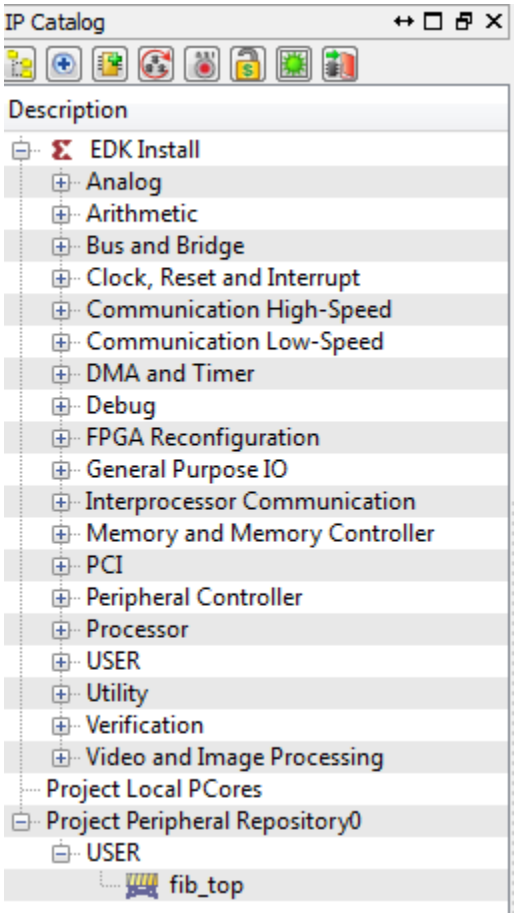
Ο βοηθός δεχόμενος σαν είσοδο όλες τις επιλογές που κάνει ο χρήστης, εισάγει την αντίστοιχη πληροφορία στα αρχεία που δημιουργεί. Μια περίληψη του περιφερειακού και των λειτουργιών του γίνεται στο τελευταίο παράθυρο διαλόγου του βοηθού, καθώς και μία αναφορά στα αρχεία που δημιουργούνται.



Σχήμα 5.24 – Peripheral Design Summary

Μόλις ολοκληρωθεί η δημιουργία του περιφερειακού τότε αυτό προστίθεται στον IP κατάλογο όπως φαίνεται και στο σχήμα.

Όπως αναφέρθηκε και πιο πάνω το περιφερειακό θα πρέπει να ενσωματωθεί στο υπόλοιπο σύστημα. Για να γίνει η προσθήκη από τον χρήστη αρκεί αυτός να επιλέξει με αριστερό click το περιφερειακό fib_top και να πατήσει add στο αναδυόμενο παράθυρο.



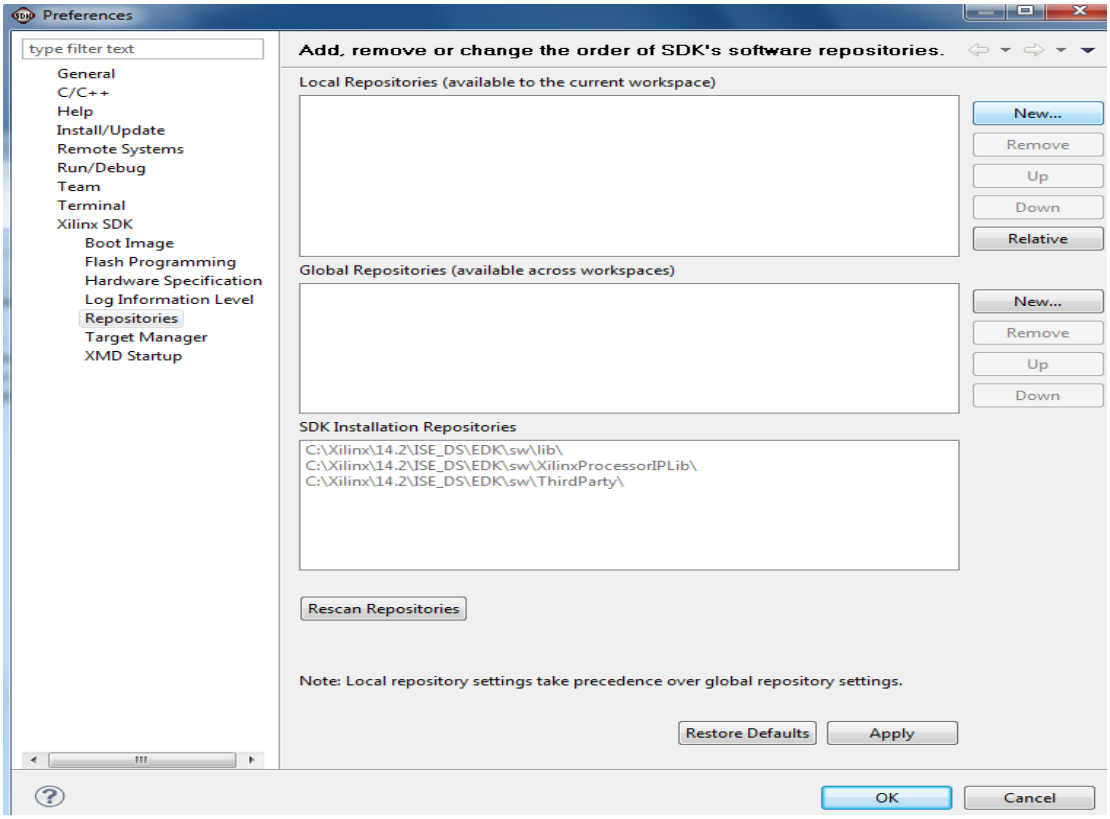
Σχήμα 5.25 – XPS IP Catalog

Τέλος, επιλέγουμε Generate BitStream και Export Design->Export & Launch SDK έτσι ώστε να δημιουργήσουμε και το software application που θα τρέξει στο ενσωματωμένο σύστημα που δημιουργήσαμε.

5.6.2 Δημιουργία software application στο SDK

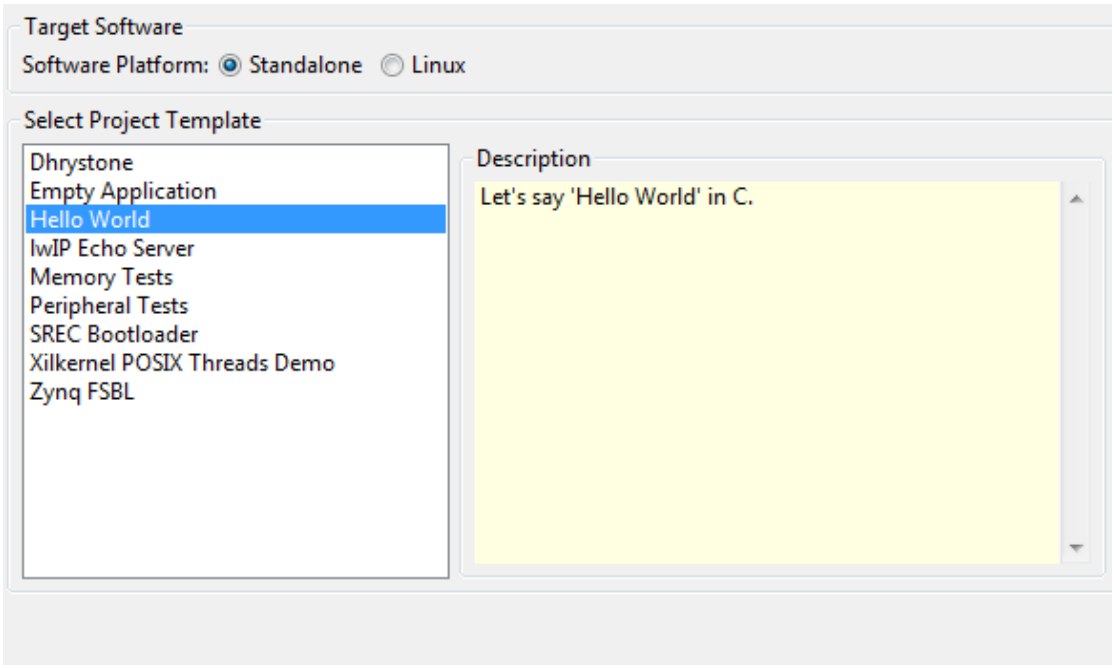
Στο SDK δεν μας ενδιαφέρει η εφαρμογή που θα δημιουργήσουμε, καθώς οι επιθυμητοί υπολογισμοί γίνονται στο hardware μέσω του περιφερειακού. Γι' αυτό και επιλέγουμε να τρέξουμε στο Virtex-7 έναν έτοιμο κώδικα C, το Hello World.

Από το αρχικό μενού του εργαλείου, λοιπόν, επιλέγουμε Xilinx Tools->Repositories και στο παράθυρο Local Repositories πατάμε New και τοποθετούμε το subdirectory solution1/impl κάτω από το directory του project του Vivado HLS, ώστε να χρησιμοποιηθεί ο σωστός driver για το περιφερειακό που φτιάξαμε.



Σχήμα 5.26 – SDK Environment

Στη συνέχεια επιλέγουμε File->New->Xilinx C Project και από την λίστα με τις έτοιμες εφαρμογές επιλέγουμε την Hello World. Επιλέγοντας Finish το SDK κάνει build project και δημιουργεί ένα .elf file .



Σχήμα 5.21 – Target Software Application

Το ενσωματωμένο σύστημα που σχεδιάσαμε είναι πλέον έτοιμο να τρέξει στην πραγματική πλακέτα με το Fpga. Συνεπώς, ανοίγουμε την πλακέτα, της παιρνάμε το bitsream.bit αρχείο και πλέον η σχεδίαση της πραγματικής μας πλατφόρμας έφτασε στο τέλος. Αφού λοιπόν, έχουμε ολοκληρώσει την σχεδίαση και της εικονικής πλατφόρμας μπορούμε να προχωρήσουμε στο έκτο και τελευταίο κεφάλαιο, όπου θα συγκεντρώσουμε όλα τα αποτελέσματά μας και θα τα σχολιάσουμε.

Κεφάλαιο 6

Σύνοψη, Συμπεράσματα και Μελλοντική Εργασία

6.1 Σύνοψη και συμπερασματα

Στην συγκεκριμένη ενότητα αυτό που κάνουμε είναι να συγκεντρώσουμε όλα τα αποτελέσματα από όλη την εργασία και να τα σχολιάσουμε τι πετύχαμε.

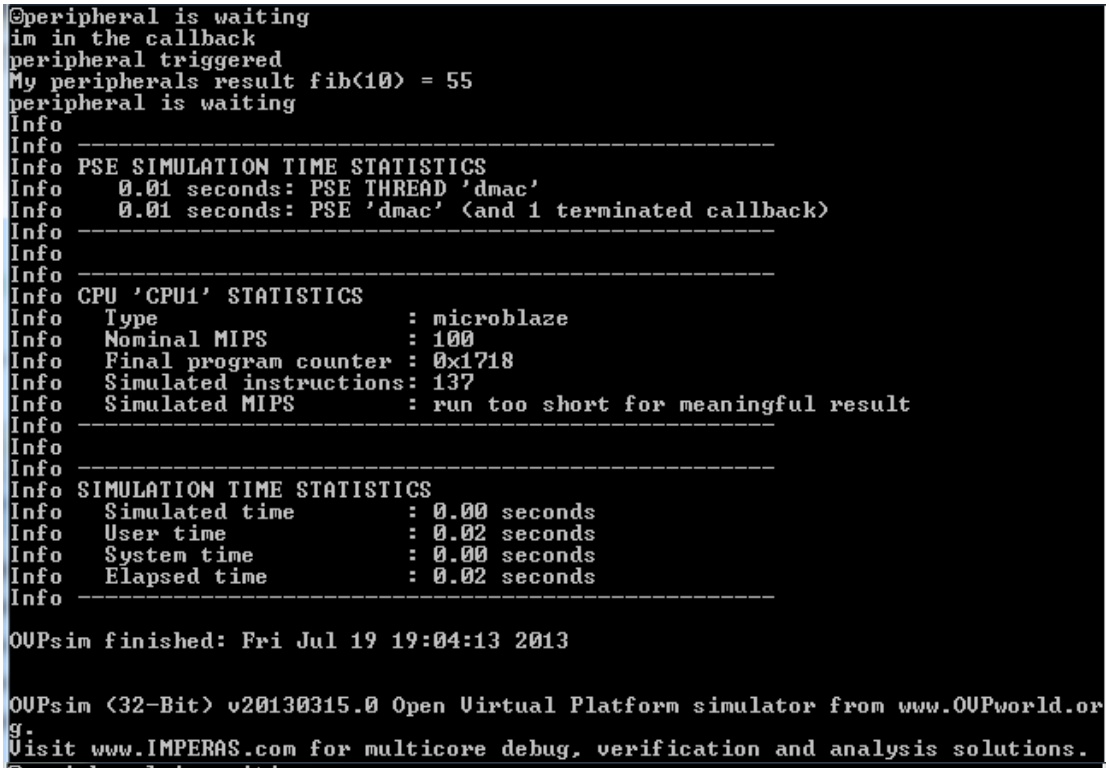
Ο συνηθισμένος τρόπος σχεδίασης ενσωματωμένων συστημάτων συμπεριλαμβάνει δύο ξεχωριστά μεταξύ τους στάδια: Την εικονική σχεδίαση και προσομοίωση του συστήματος, και την πραγματική του σχεδίαση. Αυτά τα δύο στάδια προσπαθήσαμε να συνδέσουμε και το πετύχαμε αυτό μέσω High Level Synthesis. Εμείς στην εργασία αυτή καταφέραμε να τα συνδέσουμε πετυχαίνοντας αυτοματοποίηση, βέλτιστη υλοποίηση και ταχύτητα στη σχεδίαση.

Τα βήματα και η ροή που ακολουθήσαμε για την ολοκλήρωση της εργασίας παρουσιάζονται παρακάτω:

Εικονική Σχεδίαση

Καταρχήν, ξεκινήσαμε με την σχεδίαση της εικονικής πλατφόρμας, η οποία χρησιμοποιούσε τον επεξεργαστή Microblaze της Xilinx και ένα περιφερειακό.

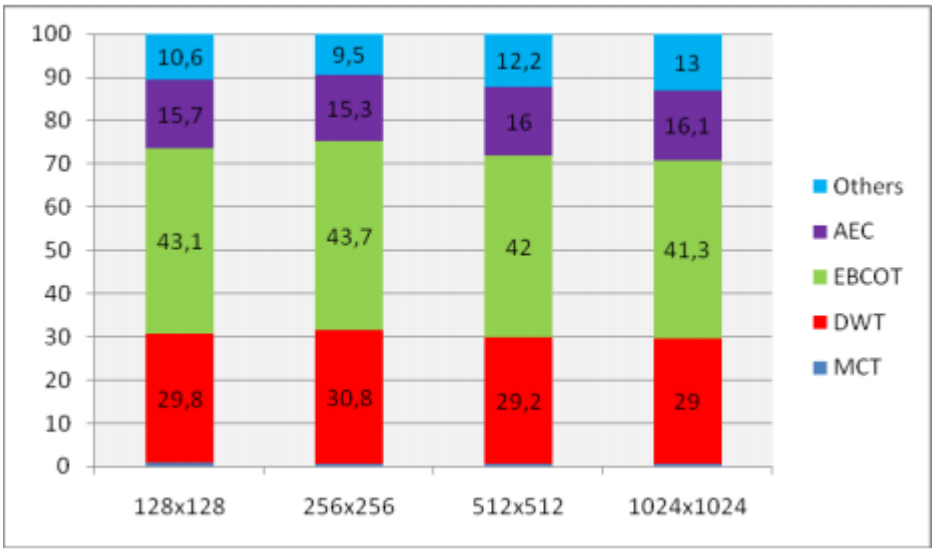
Η εφαρμογή που επιλέξαμε να υλοποιήσουμε υπολογίζει και εμφανίζει έναν αριθμό Fibonacci. Για να διαλέξουμε τι υπολογισμούς θα εκτελεί το περιφερειακό δοκιμάσαμε δύο περιπτώσεις. Στην πρώτη, το περιφερειακό δεν κάνει κάποιο υπολογισμό και όλα γίνονται στο software. Στη δεύτερη περίπτωση όλοι οι υπολογισμοί γίνονται από το περιφερειακό και το software το μόνο που κάνει είναι να το ενεργοποιεί. Ύστερα από προσομοίωση και των δύο περιπτώσεων καταλήξαμε, όπως ήταν φυσικό ότι όταν οι υπολογισμοί υλοποιούνατι με hardware τότε η ταχύτητα εκτέλεσης της εφαρμογής μειώνεται σημαντικά.



Σχήμα 6.1 – FIBONACCI H/S Partitioning

Όμως επειδή η εφαρμογή Fibonacci είναι αρκετά μικρή, κάναμε hardware/software partitioning σε μία μεγαλύτερη εφαρμογή, την JPEG2000. Όπως αποδείχτηκε από την προσομοίωση τα μέρη που κατανάλωναν περισσότερο χρόνο για να εκτελεστούν είναι το EBCOT(Coefficient Bit Modeling stage) και το DWT (Discrete Wavelet Transform). Συνεπώς το επόμενο βήμα είναι να υλοποίσουμε σε hardware τις βασικότερες συναρτήσεις αυτών και να ελέγξουμε κατά πόσο μείωνεται ο χρόνος εκτέλεσης της εφαρμογής. Μετά περνάμε

με HLS στον σχεδιασμό της πραγματικής πλατφόρμας με τον ίδιο τρόπο με την εφαρμογή Fibonacci.



Σχήμα 6.2 – JPEG2000 Parts

High Level Synthesis

Επόμενο βήμα ήταν να βρούμε το συνδετικό εργαλείο που θα ενώνει την virtual με την real platform. Το εργαλείο αυτό ήταν το VivadoHLS μέσω του οποίου μετατρέψαμε τον κώδικα που βάλαμε στο περιφερειακό από C σε Verilog. Η σημασία αυτού του εγχειρήματος είναι τεράστια αν αναλογιστεί κανείς ότι η σύνδεση αυτή που πετύχαμε αποτελεί το επόμενο βήμα στην μεθοδολογία σχεδιασμού SoC, όπως μάλιστα δήλωσε ο Simon Davidmann, πρόεδρος και διευθύνων σύμβουλος, Imperas.

Area Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	-	-	-	-
Expression	-	-	0	103	-
FIFO	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	98	-
Register	-	-	162	-	-
Total	0	0	162	201	0
Available	2060	2800	607200	303600	75900
Utilization (%)	0	0	~0	~0	0

Power Estimate

Summary

	Power
Component	-
Expression	10
FIFO	-
Memory	-
Multiplexer	9
Register	16
Total	35

Σχήμα 6.3 – VivadoHLS Results (1)

Interface Summary

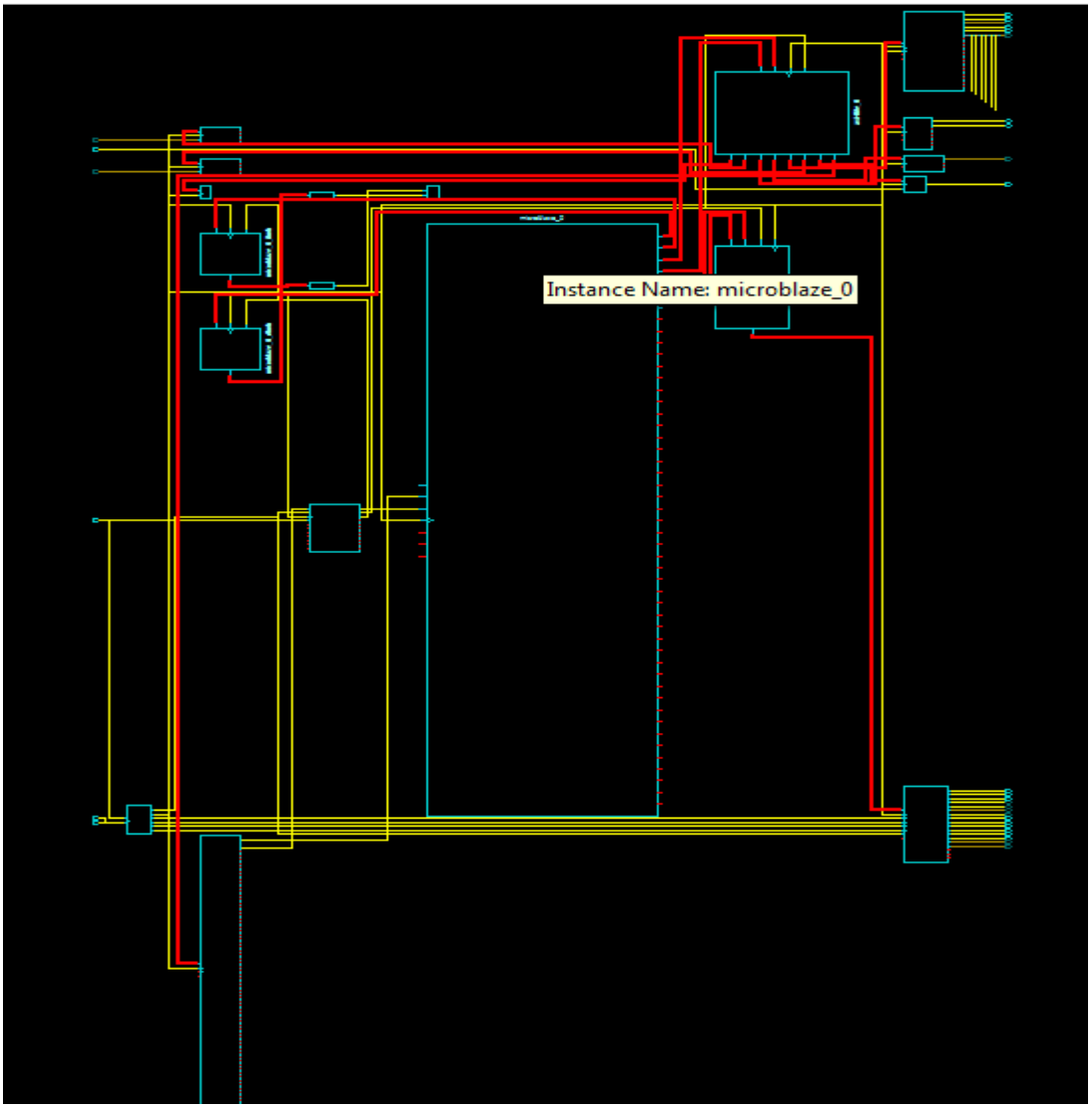
Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	fib	return value	-	ap_ctrl_hs	register	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_return	-	-	-	-	-	out	32
i	i	scalar	-	ap_hs	-	in	32
i_ap_vld	-	-	-	-	-	in	1
i_ap_ack	-	-	-	-	-	out	1

Σχήμα 6.4 – VivadoHLS Results (2)

Πραγματική Σχεδίαση

Από τη στιγμή που μετατρέψαμε τον κώδικα του περιφερειακού με το Fibonacci σε HDL Language περάσαμε αυτόματα στη σχεδίαση της πραγματικής πλατφόρμας και μέσω των εργαλείων της Xilinx καταφέραμε να ολοκληρώσαμε επιτυχώς το implementation στο FPGA.



Σχήμα 6.5 – XPS Results

6.2 Μελλοντική Εργασία

Προφανώς, υπάρχουν περισσότερες ιδέες για να βελτιωθεί ή να διερευνηθεί περισσότερο αυτή η εργασία. Μελλοντικά, αυτό που μπορεί να γίνει είναι

- Η ολοκλήρωση της εφαρμογής JPEG2000 ώστε να επιτευχθεί η σύνδεση της εικονικής με την πραγματική πλατφόρμα σε μία εφαρμογή που είναι σαφώς πιο απαιτητική από αυτήν με τον Fibonacci.
- Η προσθήκη περισσοτέρων του ενός hardware accelerators. Όπως είναι λογικό όσο περισσότερα περιφερειακά υπάρχουν και περιέχουν τις πιο βαριές συναρτήσεις της εφαρμογής JPEG2000, τόσο ταχύτερη θα είναι και η εκτέλεσής της.

Παράρτημα

Α. Κώδικες OVP

A.1.platform.c

Παρατίθεται ο κώδικας που δημιουργεί έναν Microblaze επεξεργαστή πάνω στην πλατφόρμα του συστήματος μας.

```
/*
 * creation of platform with Microblaze processor
 *
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "icm/icmCpuManager.h"

#define SIM_ATTRS (ICM_ATTR_DEFAULT)


//
// Perform platform creation and application simulation using OVPsim
//
static Bool simulate(const char *appName) {

    // initialize OVPsim, enabling verbose mode to get statistics at end
    // of execution
    icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC, 0, 0);

       // select library components
       icmAttrListP cpu1_attr = icmNewAttrList();
       icmAddStringAttr(cpu1_attr, "endian", "big");
    icmAddDoubleAttr(cpu1_attr, "mips", 100.000000);



    const char *microblazeModel    = icmGetVlnvString(NULL,
"xilinx.ovpworld.org", "processor",  "microblaze",        "1.0", "model");
    const char *microblazeSemihost = icmGetVlnvString(NULL,
"xilinx.ovpworld.org", "semihosting", "microblazeNewlib", "1.0", "model");

    // create the processor bus
    icmBusP bus = icmNewBus("busMain", 32);


////////////////////////////////////////////////////////////////////////////
    // Memory

////////////////////////////////////////////////////////////////////////////

    // create two memory regions mapping all memory except the DMAC
registers
    icmMemoryP mem1 = icmNewMemory("mem1", ICM_PRIV_RWX, 0x3fffffff);
    icmMemoryP mem2 = icmNewMemory("mem2", ICM_PRIV_RWX, 0x7fffffff);

    // connect memories to the bus
    icmConnectMemoryToBus(bus, "sp", mem1, 0xc0000000);
```

```
    icmConnectMemoryToBus(bus, "sp", mem2, 0x00000000);


////////////////////////////////////////////////////////////////////////////
    // Processor
////////////////////////////////////////////////////////////////////////////

   // create a processor instance
    icmProcessorP cpu1_c = icmNewProcessor(
        "cpu0",             // CPU name
        "microblaze",       // CPU type
        0,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        microblazeModel,    // model file
        "modelAttrs",       // morpher attributes
        SIM_ATTRS,          // attributes
        cpu1_attr,          // user-defined attributes
        microblazeSemihost, // semi-hosting file
        "modelAttrs"        // semi-hosting attributes
    );

    // connect the processor instruction and data busses to the bus
    icmConnectProcessorBusses(cpu1_c, bus, bus);

    // load the application executable file into processor memory space
    if(!icmLoadProcessorMemory(cpu1_c, appName, False, False, True)) {
        return False;
    }


////////////////////////////////////////////////////////////////////////////
    // DMAC Peripheral
////////////////////////////////////////////////////////////////////////////

    // instantiate the peripheral
    icmPseP dmac = icmNewPSE("dmac", "pse.pse", NULL, NULL, NULL);

    // connect the DMAC slave port on the bus and define the address range
it occupies
    icmConnectPSEBus(dmac, bus, "DMACSP", False, 0x80000000, 0x8000013f);



////////////////////////////////////////////////////////////////////////////
    // Simulation of Platform
////////////////////////////////////////////////////////////////////////////

    // simulate the platform
    icmProcessorP final = icmSimulatePlatform();

    // was simulation interrupted or did it complete
    if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
        icmPrintf("*** simulation interrupted\n");
    }

    icmTerminate();

    return True;

}

//
// Main routine
//
```

```
int main(int argc, char **argv) {

    Bool        promptAtExit = False;
    char        appChars[1024];
    const char *appName;

    // Check arguments for application to load
    if(argc==1) {
        // prompt for application to load

        promptAtExit = True;

        do {
            icmPrintf("Enter application elf file name > ");
            fgets(appChars, sizeof(appChars), stdin);

            // trim trailing '\n' and any trailing whitespace
            char *last = appChars + strlen(appChars) - 1;
            while((last>=appChars) && isspace(*last)) {
                *last-- = '\0';
            }

        } while(appChars[0] == '\0');

        appName = appChars;

    } else if(argc==2) {
        // application provided on command line

        appName = argv[1];

    } else {
        // incorrect arguments

        icmPrintf("Usage : %s <application name>\n", argv[0]);
        return -1;
    }

    // call OVPsim platform creation and run simulation
    Bool result = simulate(appName);

    // wait for key press before terminating
    if(promptAtExit) {
        icmPrintf("Press enter to exit demo > ");
        fgets(appChars, sizeof(appChars), stdin);
    }

    return result ? 0 : -1;
}
```

A.2.peripheral.user.c

Εδώ παρατίθεται ο κώδικας που δημιουργεί το περιφερειακό, το οποίο εκτελεί του υπολογισμό του αριθμού Fibonacci.

```
/*
 * peripheral creation
 *
 */


#include <stdio.h>
#include <string.h>
```

```
#include "dmacModel.h"
#include "dmacRegisters.h"

#define THREAD_STACK     (8*1024)
#define NUM_CHANNELS     1
#define BYTES_PER_ACCESS  8

// Give a 'nice' name to the default generated
#define controlRegs DMACSP_ab8_dataT
#define channelRegs DMACSP_ab32ch0_dataT

controlRegs *control;
channelRegs *ch[2];

typedef struct {
   bhmThreadHandle      thread;
   bhmEventHandle       start;
   Bool                 busy;
   char                 stack[THREAD_STACK];
} channelState;

typedef struct {
   ppmAddressSpaceHandle readHandle;
   ppmAddressSpaceHandle writeHandle;
   ppmNetHandle         intTCHandle;
   Bool                 intTCAsserted;
   Bool                 inReset;
   channelState         ch[NUM_CHANNELS];
} dmaState;

static dmaState DMAState;

static inline Uns32 byteSwap(Uns32 data){
#ifdef ENDIANBIG
   return
      ((data & 0xff000000) >> 24) |
      ((data & 0x00ff0000) >>  8) |
      ((data & 0x0000ff00) <<  8) |
      ((data & 0x000000ff) << 24);
#else
   return data;
#endif
}


static void writeAndStart(Uns8 channel, Uns32 data)
{
   ch[channel]->config.value = byteSwap(data);
  // if(!ch[channel]->config.bits.halt && ch[channel]->config.bits.enable && !DMAState.ch[channel].busy) {
      bhmTriggerEvent(DMAState.ch[channel].start);
   //}
}

//////////////////////////////// Callback stubs ////////////////////////////////

PPM_REG_WRITE_CB(TCclearWr) {
   // YOUR CODE HERE (TCclearWr)
   *(Uns8*)user = data;
}

PPM_REG_WRITE_CB(configCh0Wr) {
            printf("im in the callback data= %d\n",data);
   writeAndStart(0, data);
```

```
            *(Uns32*)user =  byteSwap(data);
}

PPM_REG_WRITE_CB(configCh1Wr) {
   writeAndStart(1, data);
}

PPM_REG_WRITE_CB(configWr) {
   control->config.value = byteSwap(data);
}

PPM_REG_WRITE_CB(errClearWr) {
   control->intErrStatus.value = 0;
}

PPM_REG_READ_CB(regRd32) {
   return byteSwap(*(Uns32*)user);
}

PPM_REG_READ_CB(regRd8) {
   // YOUR CODE HERE (regRd8)
   return *(Uns8*)user;
}

PPM_REG_WRITE_CB(regWr32) {
   *(Uns32*)user =  byteSwap(data);
}

static inline void writeReg32(Uns32 address, Uns32 offset, Uns32 value)
{
   *(volatile Uns32*) (address + offset) = value;
}
static inline Uns32 readReg32(Uns32 address, Uns32 offset)
{
   return *(volatile Uns32*) (address + offset);
}



static Uns32 dmaBurst(Uns32 ITERATIONS){

            int prev = -1;
   int result = 1;
   int sum;
            int i;
            for (i=0; i<=ITERATIONS; ++i) {
                        sum = result + prev;
      prev = result;
      result = sum;
            }


            printf ("reg32 %d\n",readReg32(DMA_BASE,DMA_C0_CONFIGURATION));
                        //writeReg32(DMA_BASE,  DMA_C0_SRC_ADDR, 9);
            return result;
}

//
// Thread for each channel
// When it starts, it runs to the first 'wait' which is in this case bhmWaitEvent()
//
static void channelThread(void *user)
{
```

```
            Uns32 ITERATIONS=10;
   Uns32 ch = (Uns32) user;
   for (;;) {
      if (diagnosticLevel >= 2) bhmMessage("I", "DMAC", "ch %u waiting\n", ch);
                        printf("peripheral is waiting\n");
      DMAState.ch[ch].busy = False;

      bhmWaitEvent(DMAState.ch[ch].start);
      {       printf("peripheral triggered\n");
         DMAState.ch[ch].busy = True;
         if (diagnosticLevel >= 2) bhmMessage("I", "DMAC", "ch %u running\n", ch);

         // Perform DMA burst
        // dmaBurst(ITERATIONS);
                                    printf("My peripherals result fib(%d) = %d\n", ITERATIONS, dmaBurst(ITERATIONS));

         control->rawTCstatus.value |= (1 << ch);
         if (diagnosticLevel >= 2) {
            bhmMessage("I", "DMAC",
               "ch %u status=0x%x\n",
               ch,
               control->rawTCstatus.value
            );
         }
      }
   }
}

void userInit(void)
{
   Uns32 i;
   char threadName[32];

   control = (controlRegs *)&DMACSP_ab8_data;
   ch[0] = (channelRegs *)&DMACSP_ab32ch0_data;
   ch[1] = (channelRegs *)((void *)&DMACSP_ab32ch1_data);

   DMAState.intTCAsserted = False;

   // Create threads for the channels
   for (i=0; i<NUM_CHANNELS; i++) {

      // Event to start the thread
      DMAState.ch[i].start = bhmCreateEvent();
      DMAState.ch[i].busy  = False;

      // create the thread
      printf(threadName, "ch%u", i);
      DMAState.ch[i].thread = bhmCreateThread(
         channelThread,
         (void*) i,
         threadName,
         &DMAState.ch[i].stack[THREAD_STACK] // top of downward growing stack
      );
   }
}


PPM_CONSTRUCTOR_CB(constructor) {
   // YOUR CODE HERE (peripheral constructor)
   periphConstructor();
   // YOUR CODE HERE (post constructor)
   userInit();
```

```
}

PPM_DESTRUCTOR_CB(destructor) {
   // YOUR CODE HERE (destructor)
}
```

A.2.1.peripheral.c

Ο κώδικας αυτός ουσιαστικά δημιουργεί το παράθυρο μνήμης στο περιφερειακό και δημιουργεί του καταχωρητές που θα χρησιμοποιήσει.

```
/*
 *create registers in peripheral ‘s memory
 */



#include "dmacModel.h"
////////////////////////////// Port Declarations //////////////////////////////

DMACSP_ab8_dataT DMACSP_ab8_data;

DMACSP_ab32ch0_dataT DMACSP_ab32ch0_data;

DMACSP_ab32ch1_dataT DMACSP_ab32ch1_data;

handlesT handles;

////////////////////////////// Diagnostic level ///////////////////////////////

// Test this variable to determine what diagnostics to output.
// eg. if (diagnosticLevel > 0) bhmMessage("I", "dmac", "Example");

Uns32 diagnosticLevel;

////////////////////////// Diagnostic level callback //////////////////////////

static void setDiagLevel(Uns32 new) {
   diagnosticLevel = new;
}

////////////////////////////// MMR Generic callbacks //////////////////////////

static PPM_VIEW_CB(view8) {  *(Uns8*)data = *(Uns8*)user; }

static PPM_VIEW_CB(view32) {  *(Uns32*)data = *(Uns32*)user; }

////////////////////////////// Bus Slave Ports ////////////////////////////////

static void installSlavePorts(void) {
   handles.DMACSP = ppmCreateSlaveBusPort("DMACSP", 320);
   if (!handles.DMACSP) {
      bhmMessage("E", "PPM_SPNC", "Could not connect port 'DMACSP'");
   }

}

////////////////////////// Memory mapped registers ////////////////////////////

static void installRegisters(void) {
```

```
ppmCreateRegister("intStatus",
   "internal status",
   handles.DMACSP,
   0,
   1,
   regRd8,
   0,
   view8,
   &(DMACSP_ab8_data.intStatus.value),
   True
);
ppmCreateRegister("intTCstatus",
   "internal TC status",
   handles.DMACSP,
   4,
   1,
   regRd8,
   TCclearWr,
   view8,
   &(DMACSP_ab8_data.intTCstatus.value),
   True
);
ppmCreateRegister("intErrStatus",
   "internal error status",
   handles.DMACSP,
   12,
   1,
   regRd8,
   errClearWr,
   view8,
   &(DMACSP_ab8_data.intErrStatus.value),
   True
);
ppmCreateRegister("rawTCstatus",
   "raw TC status",
   handles.DMACSP,
   20,
   1,
   regRd8,
   0,
   view8,
   &(DMACSP_ab8_data.rawTCstatus.value),
   True
);
ppmCreateRegister("rawErrStatus",
   "raw error status",
   handles.DMACSP,
   24,
   1,
   regRd8,
   0,
   view8,
   &(DMACSP_ab8_data.rawErrStatus.value),
   True
);
ppmCreateRegister("enbldChns",
   "enabled channels",
   handles.DMACSP,
   28,
   1,
   regRd8,
   0,
```

```
    view8,
    &(DMACSP_ab8_data.enbldChns.value),
    True
);
ppmCreateRegister("config",
    "configuration",
    handles.DMACSP,
    48,
    1,
    regRd8,
    configWr,
    view8,
    &(DMACSP_ab8_data.config.value),
    True
);
ppmCreateRegister("srcAddr",
    "channel 0 source address",
    handles.DMACSP,
    256,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch0_data.srcAddr.value),
    True
);
ppmCreateRegister("dstAddr",
    "channel 0 dest address",
    handles.DMACSP,
    260,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch0_data.dstAddr.value),
    True
);
ppmCreateRegister("LLI",
    "channel 0 LLI",
    handles.DMACSP,
    264,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch0_data.LLI.value),
    True
);
ppmCreateRegister("control",
    "channel 0 control",
    handles.DMACSP,
    268,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch0_data.control.value),
    True
);
ppmCreateRegister("config",
    "channel 0 configuration",
    handles.DMACSP,
    272,
```

```
    4,
    regRd32,
    configCh0Wr,
    view32,
    &(DMACSP_ab32ch0_data.config.value),
    True
);
ppmCreateRegister("srcAddr",
    "channel 1 source address",
    handles.DMACSP,
    288,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch1_data.srcAddr.value),
    True
);
ppmCreateRegister("dstAddr",
    "channel 1 dest address",
    handles.DMACSP,
    292,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch1_data.dstAddr.value),
    True
);
ppmCreateRegister("LLI",
    "channel 1 LLI",
    handles.DMACSP,
    296,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch1_data.LLI.value),
    True
);
ppmCreateRegister("control",
    "channel 1 control",
    handles.DMACSP,
    300,
    4,
    regRd32,
    regWr32,
    view32,
    &(DMACSP_ab32ch1_data.control.value),
    True
);
ppmCreateRegister("config",
    "channel 1 configuration",
    handles.DMACSP,
    304,
    4,
    regRd32,
    configCh1Wr,
    view32,
    &(DMACSP_ab32ch1_data.config.value),
    True
);
```

```
}

//////////////////////////////////// Constructor ////////////////////////////////

PPM_CONSTRUCTOR_CB(periphConstructor) {
   installSlavePorts();
   installRegisters();
}

/////////////////////////////////////// Main /////////////////////////////////////

int main(int argc, char *argv[]) {
   diagnosticLevel = 0;
   bhmInstallDiagCB(setDiagLevel);
   constructor();

   bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
   destructor();
   return 0;
}
```

A.2.2.peripheral.atts.c

Ο κώδικας αυτός ορίζει ό, τι χρειάζεται για την επικοινωνία του περιφερειακού με το υπόλοιπο σύστημα.

```
/*
 * This file defines the peripheral interface in a form that other tools
 * can read without connecting the model to a platform.
 * It defines:
 *    net ports,
 *    bus ports,
 *    parameters (formally known as attributes)
 *    documentation nodes
 *    intended library location
 *
 *    It is compiled with the PSE cross compiler and linked with the other
 *    files of the peripheral model.
 */



#ifdef _PSE_
#    include "peripheral/impTypes.h"
#    include "peripheral/bhm.h"
#    include "peripheral/ppm.h"
#else
#    include "hostapi/impTypes.h"
#endif


static ppmBusPort busPorts[] = {
   {
      .name          = "DMACSP",
      .type          = PPM_SLAVE_PORT,
      .addrHi        = 0x13fLL,
      .mustBeConnected = 1,
      .remappable     = 0,
      .description    = "DMA Registers Slave Port",
   },
```

```
    { 0 }
};

static PPM_BUS_PORT_FN(nextBusPort) {
    if(!busPort) {
        return busPorts;
    }
    busPort++;
    return busPort->name ? busPort : 0;
}

static ppmNetPort netPorts[] = {
    { 0 }
};

static PPM_NET_PORT_FN(nextNetPort) {
    if(!netPort) {
        return netPorts;
    }
    netPort++;
    return netPort->name ? netPort : 0;
}

static ppmParameter parameters[] = {
    { 0 }
};

static PPM_PARAMETER_FN(nextParameter) {
    if(!parameter) {
        return parameters;
    }
    parameter++;
    return parameter->name ? parameter : 0;
}

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,

    .busPortsCB    = nextBusPort,
    .netPortsCB    = nextNetPort,
    .paramSpecCB   = nextParameter,

    .vlnv          = {
        .vendor  = "ovpworld.org",
        .library = "peripheral",
        .name    = "dmac",
        .version = "1.0"
    },

    .family = "ovpworld.org"

};
```

A.2.3.peripheral.h, peripheral.macro.h

Για την εκτέλεση του περιφερειακού που δημιουργήθηκε κρίνεται απαραίτητη η εισαγωγή των συγκεκριμένων αρχείων.

- peripheral.h: ορισμός καταχωρητών και call back functions.

```
/*
 *peripheral.h
*/

#ifndef DMACMODEL_H
#define DMACMODEL_H

#ifdef _PSE_
#    include "peripheral/impTypes.h"
#    include "peripheral/bhm.h"
#    include "peripheral/ppm.h"
#else
#    include "hostapi/impTypes.h"
#endif

////////////////////////////////////// Externs //////////////////////////////////////

    extern Uns32 diagnosticLevel;

/////////////////////////// Register data declaration ///////////////////////////

typedef struct DMACSP_ab8_dataS {
    union {
        Uns8 value;
    } intStatus;
    union {
        Uns8 value;
    } intTCstatus;
    union {
        Uns8 value;
    } intErrStatus;
    union {
        Uns8 value;
    } rawTCstatus;
    union {
        Uns8 value;
    } rawErrStatus;
    union {
        Uns8 value;
    } enbldChns;
    union {
        Uns8 value;
        struct {
            unsigned burstSize : 2;
        } bits;
    } config;
} DMACSP_ab8_dataT, *DMACSP_ab8_dataTP;

typedef struct DMACSP_ab32ch0_dataS {
    union {
        Uns32 value;
    } srcAddr;
    union {
        Uns32 value;
    } dstAddr;
    union {
        Uns32 value;
    } LLI;
    union {
        Uns32 value;
    } control;
    union {
```

```
        Uns32 value;
        struct {
            unsigned enable : 1;
            unsigned __pad1 : 17;
            unsigned halt : 1;
        } bits;
    } config;
} DMACSP_ab32ch0_dataT, *DMACSP_ab32ch0_dataTP;

typedef struct DMACSP_ab32ch1_dataS {
    union {
        Uns32 value;
    } srcAddr;
    union {
        Uns32 value;
    } dstAddr;
    union {
        Uns32 value;
    } LLI;
    union {
        Uns32 value;
    } control;
    union {
        Uns32 value;
        struct {
            unsigned enable : 1;
            unsigned __pad1 : 17;
            unsigned halt : 1;
        } bits;
    } config;
} DMACSP_ab32ch1_dataT, *DMACSP_ab32ch1_dataTP;

////////////////////////////// Port Declarations //////////////////////////////

extern DMACSP_ab8_dataT DMACSP_ab8_data;

extern DMACSP_ab32ch0_dataT DMACSP_ab32ch0_data;

extern DMACSP_ab32ch1_dataT DMACSP_ab32ch1_data;

#ifdef _PSE_
////////////////////////////////// Port handles //////////////////////////////////

typedef struct handlesS {
    void            *DMACSP;
} handlesT, *handlesTP;

extern handlesT handles;

////////////////////////////// Callback prototypes //////////////////////////////

PPM_REG_WRITE_CB(TCclearWr);
PPM_REG_WRITE_CB(configCh0Wr);
PPM_REG_WRITE_CB(configCh1Wr);
PPM_REG_WRITE_CB(configWr);
PPM_REG_WRITE_CB(errClearWr);
PPM_REG_READ_CB(regRd32);
PPM_REG_READ_CB(regRd8);
PPM_REG_WRITE_CB(regWr32);
PPM_CONSTRUCTOR_CB(periphConstructor);
PPM_DESTRUCTOR_CB(periphDestructor);
PPM_CONSTRUCTOR_CB(constructor);
PPM_DESTRUCTOR_CB(destructor);
```

```
#endif

#endif
```

- **peripheral.macro.h**: θέσεις μνήμης των macros.

```
/*
 *peripheral.macro.h
 */



#ifndef DMACMODEL_MACRO_H
#define DMACMODEL_MACRO_H
//////////////////////////////////// Description ////////////////////////////////////

// DMAC peripheral model example 'creatingDMAC'

// Before including this file in the application, define the indicated macros
// to fix the base address of each slave port.
// Set the macro 'DMACSP' to the base of port 'DMACSP'
#ifndef DMACSP
   #error DMACSP is undefined.It needs to be set to the port base address
#endif
#define DMACSP_AB8_INTSTATUS    (DMACSP + 0x0)

#define DMACSP_AB8_INTTCSTATUS    (DMACSP + 0x4)

#define DMACSP_AB8_INTERRSTATUS    (DMACSP + 0xc)

#define DMACSP_AB8_RAWTCSTATUS    (DMACSP + 0x14)

#define DMACSP_AB8_RAWERRSTATUS    (DMACSP + 0x18)

#define DMACSP_AB8_ENBLDCHNS    (DMACSP + 0x1c)

#define DMACSP_AB8_CONFIG    (DMACSP + 0x30)

#define DMACSP_AB8_CONFIG_BURSTSIZE   0x3
#define DMACSP_AB32CH0_SRCADDR    (DMACSP + 0x100)

#define DMACSP_AB32CH0_DSTADDR    (DMACSP + 0x104)

#define DMACSP_AB32CH0_LLI    (DMACSP + 0x108)

#define DMACSP_AB32CH0_CONTROL    (DMACSP + 0x10c)

#define DMACSP_AB32CH0_CONFIG    (DMACSP + 0x110)

#define DMACSP_AB32CH0_CONFIG_ENABLE   0x1
#define DMACSP_AB32CH0_CONFIG_HALT   (0x1 << 18)
#define DMACSP_AB32CH1_SRCADDR    (DMACSP + 0x120)

#define DMACSP_AB32CH1_DSTADDR    (DMACSP + 0x124)

#define DMACSP_AB32CH1_LLI    (DMACSP + 0x128)

#define DMACSP_AB32CH1_CONTROL    (DMACSP + 0x12c)

#define DMACSP_AB32CH1_CONFIG    (DMACSP + 0x130)
```

```
#define DMACSP_AB32CH1_CONFIG_ENABLE   0x1
#define DMACSP_AB32CH1_CONFIG_HALT   (0x1 << 18)


#endif
```

A.3.application.c

Ο κώδικας της εφαρμογής που τρέχει στο σύστημα μας:

```
/*
 *application code
 */


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "dmacRegisters.h"
typedef unsigned int Uns32;

static inline void writeReg32(Uns32 address, Uns32 offset, Uns32 value)
{
   *(volatile Uns32*) (address + offset) = value;
}

static inline Uns32 readReg32(Uns32 address, Uns32 offset)
{
   return *(volatile Uns32*) (address + offset);
}


int main(int argc, char **argv)
{

   writeReg32(DMA_BASE, DMA_C0_CONFIGURATION, 10);

   return 1;
}
```

Registers.h: θέσεις μνήμης των καταχωρητών του application.

```
/*
 *peripheralRegisters.h
 *
 */

////////////////////// Registers for DMA Controller based on ARM PL081 ////////////////////////////

#ifndef DMAC_REGISTERS_H
#define DMAC_REGISTERS_H

#define DMA_BASE              0x80000000

#define DMA_INT_STATUS          0x00
#define DMA_INT_TC_STATUS       0x04
#define DMA_INT_TC_CLEAR        0x04
#define DMA_INT_ERROR_STATUS    0x0c
```

```
#define DMA_INT_ERROR_CLEAR      0x0c
#define DMA_RAW_INT_TC_STATUS    0x14
#define DMA_RAW_INT_ERROR_STATUS 0x18
#define DMA_EMBLD_CHNS           0x1c
#define DMA_CONFIGURATION        0x30

#define DMA_C0_SRC_ADDR          0x100
#define DMA_C0_DST_ADDR          0x104
#define DMA_C0_LL1               0x108
#define DMA_C0_CONTROL           0x10c
#define DMA_C0_CONFIGURATION     0x110

#define DMA_CHANNEL_STRIDE       0x20

#define DMA_C1_SRC_ADDR          (DMA_C0_SRC_ADDR     + DMA_CHANNEL_STRIDE)
#define DMA_C1_DST_ADDR          (DMA_C0_DST_ADDR     + DMA_CHANNEL_STRIDE)
#define DMA_C1_LL1               (DMA_C0_LL1          + DMA_CHANNEL_STRIDE)
#define DMA_C1_CONTROL           (DMA_C0_CONTROL      + DMA_CHANNEL_STRIDE)
#define DMA_C1_CONFIGURATION     (DMA_C0_CONFIGURATION + DMA_CHANNEL_STRIDE)

// Bit manipulation
#define MASK(_BITS)                 ((_BITS) == 32 ? -1U : (1U<<MIN((_BITS),31))-1)
#define PART_SELECT(_V,_LEFT,_RIGHT)   (((_V)>>(_RIGHT)) & MASK((_LEFT)-(_RIGHT)+1))
#define BIT_SELECT(_V,_BIT)          (((_V)>>(_BIT)) & 1)

// Control Bits
#define DMA_CONTROL_I(_R)       BIT_SELECT(_R,31)
#define DMA_CONTROL_TRAN_SIZE(_R) PART_SELECT(_R,11,0)

// Configuration bits
#define DMA_CONFIG_HALT(_R)      BIT_SELECT(_R,18)
#define DMA_CONFIG_ACT(_R)       BIT_SELECT(_R,17)
#define DMA_CONFIG_ITC(_R)       BIT_SELECT(_R,15)
#define DMA_CONFIG_ENA(_R)       BIT_SELECT(_R,0)

#endif
```

Α. Κώδικες Pcore

Εδώ παραθέτουμε τους κώδικες που δημιουργήσαμε από το VivadoHLS σε Verilog:

fib.v

```
// ==============================================================
// RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2013.1
// Copyright (C) 2013 Xilinx Inc. All rights reserved.
//
// ==============================================================

`timescale 1 ns / 1 ps

(*
CORE_GENERATION_INFO="fib,hls_ip_2013_1,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HL
S_INPUT_PART=xc7vx485tffg1761-
2,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=1.630000,HLS_SYN_LAT=-
1,HLS_SYN_TPT=none,HLS_SYN_MEM=0,HLS_SYN_DSP=0,HLS_SYN_FF=128,HLS_SYN_LUT=199}" *)

module fib (
      ap_clk,
      ap_rst,
```

```
    ap_start,
    ap_done,
    ap_idle,
    ap_ready,
    i,
    i_ap_vld,
    i_ap_ack,
    ap_return
);

input   ap_clk;
input   ap_rst;
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [31:0] i;
input   i_ap_vld;
output   i_ap_ack;
output  [31:0] ap_return;

reg ap_done;
reg ap_idle;
reg ap_ready;
reg i_ap_ack;
reg   [0:0] ap_CS_fsm = 1'b0;
reg   [31:0] i_read_reg_91;
reg    ap_sig_bdd_27;
wire   [30:0] j_1_fu_79_p2;
wire   [31:0] sum_fu_85_p2;
wire   [0:0] tmp_fu_74_p2;
reg   [31:0] prev_reg_34;
reg   [31:0] prev_1_reg_45;
reg   [30:0] j_reg_58;
wire   [31:0] j_cast_fu_70_p1;
reg   [0:0] ap_NS_fsm;
parameter    ap_const_logic_1 = 1'b1;
parameter    ap_const_logic_0 = 1'b0;
parameter    ap_ST_st1_fsm_0 = 1'b0;
parameter    ap_ST_st2_fsm_1 = 1'b1;
parameter    ap_const_lv1_0 = 1'b0;
parameter    ap_const_lv32_FFFFFFFF = 32'b11111111111111111111111111111111;
parameter    ap_const_lv32_1 = 32'b1;
parameter    ap_const_lv31_0 = 31'b0000000000000000000000000000000;
parameter    ap_const_lv31_1 = 31'b1;
parameter    ap_true = 1'b1;



/// the current state (ap_CS_fsm) of the state machine. ///
always @ (posedge ap_clk)
begin : ap_ret_ap_CS_fsm
   if (ap_rst == 1'b1) begin
      ap_CS_fsm <= ap_ST_st1_fsm_0;
   end else begin
      ap_CS_fsm <= ap_NS_fsm;
   end
end

/// assign process. ///
always @(posedge ap_clk)
begin
```

```
    if (((ap_ST_st1_fsm_0 == ap_CS_fsm) & ~ap_sig_bdd_27)) begin
        i_read_reg_91 <= i;
    end
end

/// assign process. ///
always @(posedge ap_clk)
begin
    if (((ap_ST_st2_fsm_1 == ap_CS_fsm) & (tmp_fu_74_p2 == ap_const_lv1_0))) begin
        j_reg_58 <= j_1_fu_79_p2;
    end else if (((ap_ST_st1_fsm_0 == ap_CS_fsm) & ~ap_sig_bdd_27)) begin
        j_reg_58 <= ap_const_lv31_0;
    end
end

/// assign process. ///
always @(posedge ap_clk)
begin
    if (((ap_ST_st2_fsm_1 == ap_CS_fsm) & (tmp_fu_74_p2 == ap_const_lv1_0))) begin
        prev_1_reg_45 <= sum_fu_85_p2;
    end else if (((ap_ST_st1_fsm_0 == ap_CS_fsm) & ~ap_sig_bdd_27)) begin
        prev_1_reg_45 <= ap_const_lv32_1;
    end
end

/// assign process. ///
always @(posedge ap_clk)
begin
    if (((ap_ST_st2_fsm_1 == ap_CS_fsm) & (tmp_fu_74_p2 == ap_const_lv1_0))) begin
        prev_reg_34 <= prev_1_reg_45;
    end else if (((ap_ST_st1_fsm_0 == ap_CS_fsm) & ~ap_sig_bdd_27)) begin
        prev_reg_34 <= ap_const_lv32_FFFFFFFF;
    end
end

/// ap_done assign process. ///
always @ (ap_CS_fsm or tmp_fu_74_p2)
begin
    if (((ap_ST_st2_fsm_1 == ap_CS_fsm) & ~(tmp_fu_74_p2 == ap_const_lv1_0))) begin
        ap_done = ap_const_logic_1;
    end else begin
        ap_done = ap_const_logic_0;
    end
end

/// ap_idle assign process. ///
always @ (ap_start or ap_CS_fsm)
begin
    if ((~(ap_const_logic_1 == ap_start) & (ap_ST_st1_fsm_0 == ap_CS_fsm))) begin
        ap_idle = ap_const_logic_1;
    end else begin
        ap_idle = ap_const_logic_0;
    end
end

/// ap_ready assign process. ///
always @ (ap_CS_fsm or tmp_fu_74_p2)
begin
    if (((ap_ST_st2_fsm_1 == ap_CS_fsm) & ~(tmp_fu_74_p2 == ap_const_lv1_0))) begin
        ap_ready = ap_const_logic_1;
    end else begin
        ap_ready = ap_const_logic_0;
    end
```

```
end

/// i_ap_ack assign process. ///
always @ (ap_CS_fsm or ap_sig_bdd_27)
begin
    if (((ap_ST_st1_fsm_0 == ap_CS_fsm) & ~ap_sig_bdd_27)) begin
        i_ap_ack = ap_const_logic_1;
    end else begin
        i_ap_ack = ap_const_logic_0;
    end
end
always @ (ap_CS_fsm or ap_sig_bdd_27 or tmp_fu_74_p2)
begin
    case (ap_CS_fsm)
        ap_ST_st1_fsm_0 :
            if (~ap_sig_bdd_27) begin
                ap_NS_fsm = ap_ST_st2_fsm_1;
            end else begin
                ap_NS_fsm = ap_ST_st1_fsm_0;
            end
        ap_ST_st2_fsm_1 :
            if (~(tmp_fu_74_p2 == ap_const_lv1_0)) begin
                ap_NS_fsm = ap_ST_st1_fsm_0;
            end else begin
                ap_NS_fsm = ap_ST_st2_fsm_1;
            end
        default :
            ap_NS_fsm = 'bx;
    endcase
end
assign ap_return = prev_1_reg_45;

/// ap_sig_bdd_27 assign process. ///
always @ (ap_start or i_ap_vld)
begin
    ap_sig_bdd_27 = ((i_ap_vld == ap_const_logic_0) | (ap_start == ap_const_logic_0));
end
assign j_1_fu_79_p2 = (j_reg_58 + ap_const_lv31_1);
assign j_cast_fu_70_p1 = $unsigned(j_reg_58);
assign sum_fu_85_p2 = (prev_1_reg_45 + prev_reg_34);
assign tmp_fu_74_p2 = ($signed(j_cast_fu_70_p1) > $signed(i_read_reg_91)? 1'b1: 1'b0);


endmodule //fib
```

fib_top.v

```
// ==============================================================
// File generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
// Version: 2013.1
// Copyright (C) 2013 Xilinx Inc. All rights reserved.
//
// ==============================================================

`timescale 1 ns / 1 ps
module fib_top (
s_axi_slv1_AWADDR,
s_axi_slv1_AWVALID,
s_axi_slv1_AWREADY,
s_axi_slv1_WDATA,
```

```
s_axi_slv1_WSTRB,
s_axi_slv1_WVALID,
s_axi_slv1_WREADY,
s_axi_slv1_BRESP,
s_axi_slv1_BVALID,
s_axi_slv1_BREADY,
s_axi_slv1_ARADDR,
s_axi_slv1_ARVALID,
s_axi_slv1_ARREADY,
s_axi_slv1_RDATA,
s_axi_slv1_RRESP,
s_axi_slv1_RVALID,
s_axi_slv1_RREADY,
interrupt,
aresetn,
aclk
);

parameter C_S_AXI_SLV1_ADDR_WIDTH = 5;
parameter C_S_AXI_SLV1_DATA_WIDTH = 32;
parameter RESET_ACTIVE_LOW = 1;

input [C_S_AXI_SLV1_ADDR_WIDTH - 1:0] s_axi_slv1_AWADDR ;
input s_axi_slv1_AWVALID ;
output s_axi_slv1_AWREADY ;
input [C_S_AXI_SLV1_DATA_WIDTH - 1:0] s_axi_slv1_WDATA ;
input [C_S_AXI_SLV1_DATA_WIDTH/8 - 1:0] s_axi_slv1_WSTRB ;
input s_axi_slv1_WVALID ;
output s_axi_slv1_WREADY ;
output [2 - 1:0] s_axi_slv1_BRESP ;
output s_axi_slv1_BVALID ;
input s_axi_slv1_BREADY ;
input [C_S_AXI_SLV1_ADDR_WIDTH - 1:0] s_axi_slv1_ARADDR ;
input s_axi_slv1_ARVALID ;
output s_axi_slv1_ARREADY ;
output [C_S_AXI_SLV1_DATA_WIDTH - 1:0] s_axi_slv1_RDATA ;
output [2 - 1:0] s_axi_slv1_RRESP ;
output s_axi_slv1_RVALID ;
input s_axi_slv1_RREADY ;
output interrupt ;

input aresetn ;

input aclk ;


wire [C_S_AXI_SLV1_ADDR_WIDTH - 1:0] s_axi_slv1_AWADDR;
wire s_axi_slv1_AWVALID;
wire s_axi_slv1_AWREADY;
wire [C_S_AXI_SLV1_DATA_WIDTH - 1:0] s_axi_slv1_WDATA;
wire [C_S_AXI_SLV1_DATA_WIDTH/8 - 1:0] s_axi_slv1_WSTRB;
wire s_axi_slv1_WVALID;
wire s_axi_slv1_WREADY;
wire [2 - 1:0] s_axi_slv1_BRESP;
wire s_axi_slv1_BVALID;
wire s_axi_slv1_BREADY;
wire [C_S_AXI_SLV1_ADDR_WIDTH - 1:0] s_axi_slv1_ARADDR;
wire s_axi_slv1_ARVALID;
wire s_axi_slv1_ARREADY;
wire [C_S_AXI_SLV1_DATA_WIDTH - 1:0] s_axi_slv1_RDATA;
wire [2 - 1:0] s_axi_slv1_RRESP;
wire s_axi_slv1_RVALID;
wire s_axi_slv1_RREADY;
```

```
wire interrupt;

wire aresetn;


wire [32 - 1:0] sig_fib_i;
wire sig_fib_i_ap_vld;
wire sig_fib_i_ap_ack;
wire sig_fib_ap_start;
wire sig_fib_ap_ready;
wire sig_fib_ap_done;
wire sig_fib_ap_idle;
wire [32 - 1:0] sig_fib_ap_return;

wire sig_fib_ap_rst;



fib fib_U(
   .i(sig_fib_i),
   .i_ap_vld(sig_fib_i_ap_vld),
   .i_ap_ack(sig_fib_i_ap_ack),
   .ap_start(sig_fib_ap_start),
   .ap_ready(sig_fib_ap_ready),
   .ap_done(sig_fib_ap_done),
   .ap_idle(sig_fib_ap_idle),
   .ap_return(sig_fib_ap_return),
   .ap_rst(sig_fib_ap_rst),
   .ap_clk(aclk)
);

fib_slv1_if #(
   .C_ADDR_WIDTH(C_S_AXI_SLV1_ADDR_WIDTH),
   .C_DATA_WIDTH(C_S_AXI_SLV1_DATA_WIDTH))
slv1_if_U(
   .ACLK(aclk),
   .ARESETN(aresetn),
   .I_i(sig_fib_i),
   .I_i_ap_vld(sig_fib_i_ap_vld),
   .I_i_ap_ack(sig_fib_i_ap_ack),
   .I_ap_start(sig_fib_ap_start),
   .O_ap_ready(sig_fib_ap_ready),
   .O_ap_done(sig_fib_ap_done),
   .O_ap_idle(sig_fib_ap_idle),
   .O_ap_return(sig_fib_ap_return),
   .AWADDR(s_axi_slv1_AWADDR),
   .AWVALID(s_axi_slv1_AWVALID),
   .AWREADY(s_axi_slv1_AWREADY),
   .WDATA(s_axi_slv1_WDATA),
   .WSTRB(s_axi_slv1_WSTRB),
   .WVALID(s_axi_slv1_WVALID),
   .WREADY(s_axi_slv1_WREADY),
   .BRESP(s_axi_slv1_BRESP),
   .BVALID(s_axi_slv1_BVALID),
   .BREADY(s_axi_slv1_BREADY),
   .ARADDR(s_axi_slv1_ARADDR),
   .ARVALID(s_axi_slv1_ARVALID),
   .ARREADY(s_axi_slv1_ARREADY),
   .RDATA(s_axi_slv1_RDATA),
   .RRESP(s_axi_slv1_RRESP),
   .RVALID(s_axi_slv1_RVALID),
   .RREADY(s_axi_slv1_RREADY),
```

```
   .interrupt(interrupt));

fib_ap_rst_if #(
   .RESET_ACTIVE_LOW(RESET_ACTIVE_LOW))
ap_rst_if_U(
   .dout(sig_fib_ap_rst),
   .din(aresetn));

endmodule
```

Βιβλιογραφία

Πηγές

[1] Ron Sass, Andrew G. Schmidt, 'Embedded Systems Design with Platform FPGAs, Principles and Practices', Morgan Kaufmann

[2] Volnei A. Pedroni, ‘’Σχεδιασμός κυκλωμάτων με τη VHDL’’, Επιστημονική επιμέλεια ελληνικής έκδοσης: Γεώργιος Θεοδωρίδης, Λέκτορας Α.Π.Θ., Εκδόσεις Κλειδάριθμος, Αθήνα 2007

[3] Douglas L. Perry, 'VHDL: Programming by Example', McGraw-Hill

[4] Neil H. E. Weste, David M. Harris**,** ‘’Σχεδίαση Ολοκληρωμένων Κυκλωμάτων, CMOS VLSI Design’’, Εκδόσεις Παπασωτηρίου, Αθήνα 2011

[5] Κ.Ζ. Πεκμεστζή Καθηγητής Ε.Μ.Π, ‘’Ψηφιακά Συστήματα VLSI’’ Αθήνα 2003

[6] Ron Sass, Andrew G. Schmidt, 'Embedded Systems Design with Platform FPGAs, Principles and Practices', Morgan Kaufmann

[7] UG111 (v14.1), 'Embedded System Tools Reference Manual, EDK', Xilinx

Ηλεκτρονικοί σύνδεσμοι

[8] https://www.google.gr/

[9]http://net.pku.edu.cn/~course/cs101/2008/resource/The_C_Programming_Language.pdf

[10] http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[11]http://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf

[12] http://www.ovpworld.org/

[13] http://www.ovpworld.org/documents/Imperas_Installation_and_Getting_Started.pdf

[14] http://www.ovpworld.org/documents/OVP_Guide_To_Using_Processor_Models.pdf

[15] http://www.ovpworld.org/documents/OVP_Processor_Modeling_Guide.pdf

[16] http://www.ovpworld.org/modeldocs/OVP_Model_Specific_Information_m16c_r8c.pdf

[17] http://www.ovpworld.org/documents/OVP_Peripheral_Modeling_Guide.pdf

[18] http://en.wikipedia.org/wiki/High-level_synthesis

[19] http://www.eetimes.com/document.asp?doc_id=1276220

[20] http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug871-vivado-high-level-synthesis-tutorial.pdf

[21] http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[22]http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/PlanAhead_Tutorial_Quick_Front-to-Back_Overview.pdf

[23]https://wiki.ittc.ku.edu/ittc/images/archive/4/40/20070821143241!Edk_baseSystemBuilder.pdf

[24] http://www.xilinx.com/tools/platform.htm

[25] https://wiki.ittc.ku.edu/ittc/images/4/4a/Edk_customCores.pdf

[26] http://www.ovpworld.org/necs-cyberworkbench-and-imperas-ovp-fast-processor-models-integrated-to-expand-hardware-software-co-verification-capabilities

[27] http://www.imperas.com/kazutoshi-wakabayashi-senior-manager

[28] http://www.cadence.com/Community/blogs/ii/archive/2011/06/29/q-amp-a-linking-virtual-prototypes-to-high-level-synthesis.aspx

[29] http://en.wikipedia.org/wiki/Software_development_kit

[30]http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/SDK_doc/getting_started/sdk_tutorial_intro.htm

[31]https://en.wikipedia.org/wiki/JPEG_2000

[32]http://www.barco-silex.com/ip-cores/jpeg-2000?gclid=CKbS3YPCwbgCFYWN3godLjwAgw

[33]http://www.stanford.edu/class/ee398a/handouts/lectures/10-JPEG2000.pdf

[34]http://qss.stanford.edu/~godfrey/wavelets/wave_paper.pdf

[35]http://ijcsi.org/papers/IJCSI-8-5-3-551-557.pdf

[36]http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=1273240&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1273240