ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Integrating High-Level Synthesis derived Hardware Accelerators on an FPGA-based SoC: Evaluation and Analysis of Design Alternatives

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνου Γ. Ραΐλη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Μάρτιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ
ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Integrating High-Level Synthesis derived Hardware Accelerators on an FPGA-based SoC: Evaluation and Analysis of Design Alternatives

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνου Γ. Ραΐλη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή επιτροπή την 30η Μαρτίου 2016.

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κιαμάλ Ζ. Πεσκμεστζή	Γεώργιος Οικονομάκος
Αναπληρωτής Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής

Αθήνα, Μάρτιος 2016

….....

ΚΩΝΣΤΑΝΤΙΝΟΣ Γ. ΡΑΪΛΗΣ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Σύντομη Περίληψη

Τα τελευταία χρόνια, ο σχεδιασμός επιταχυντών υλικού έχει καθιερωθεί σαν δεδομένο όταν στοχεύουμε σε βελτιστοποιήσεις αλγοριθμικών υλοποιήσεων. Συγκεκριμένα, οι επιταχυντές βασισμένοι σε FPGA έχουν κερδίσει το ενδιαφέρον των σχεδιαστών και του επιστημονικού κόσμου καθώς οι συσκευές FPGA προσφέρουν ταχύτατη ανάπτυξη του υλικού και δυνατότητες επαναδιαμόρφωσής του. Σε συνδυασμό με το επίπεδο του αφαιρετικού σχεδιασμού που προσφέρει η Σύνθεση Υψηλού Επιπέδου (High-Level Synthesis – HLS) σχηματίζουν μία σαφή λύση όταν επιθυμείται η γρήγορη σχεδίαση πρωτοτύπων για συστήματα. Προσφάτως, η κυρίαρχη τάση για μία συσκευή FPGA είναι να περιλαμβάνει τα πλεονεκτήματα που προσφέρουν οι ενσωματωμένοι επεξεργαστές σχηματίζοντας με αυτόν τον τρόπο ένα ολοκληρωμένο Σύστημα-σε-Ψηφίδα (System-on-a-Chip – SoC). Η συνύπαρξη επιταχυντών υλικού και ενσωματωμένων επεξεργαστών σε μία συσκευή έχει φέρει στο προσκήνιο τη διασύνδεσή τους σαν ένα στοιχείο ζωτικής σημασίας για την επίδοση ολόκληρου του συστήματος. Για ευκολία στη διασύνδεση ενός επιταχυντή και ενός επεξεργαστικού συστήματος έχει υιοθετηθεί σαν πρακτική η σχεδίαση σε μορφή Πνευματικής Ιδιοκτησίας (Intellectual Property – IP). Συνήθως ένα IP είναι εξοπλισμένο με διεπαφές ελέγχου και επικοινωνίας έτσι ώστε να είναι εύκολος ο συνδυασμός του με άλλα στοιχεία, τις περισσότερες φορές χωρίς να απαιτείται η προσθήκη πρόσθετου υλικού. Μια ευρέως διαδεδομένη διεπαφή επικοινωνίας είναι το πρωτόκολλο ARM AMBA Advanced eXtensible Interface (AXI). Οι σχεδιαστικές εναλλακτικές που παρέχονται από το πρωτόκολλο AXI μπορεί να κυμαίνονται από απλή, χαμηλού εύρους ζώνης επικοινωνία και μεταφορά δεδομένων μέχρι υψηλές τιμές εύρους ζώνης χρησιμοποιώντας διαθέσιμα χαρακτηριστικά όπως η Άμεση Πρόσβαση Μνήμης. Σε αυτή την εργασία επικεντρωνόμαστε στη ροή υλοποίησης ενός συστήματος για τη συσκευή Zynq-7000 AP SoC. Ξεκινώντας με την προσθήκη διαφορετικών διεπαφών επικοινωνίας δημιουργούμε επιταχυντές σε μορφή IP μέσω του HLS. Στη συνέχεια προχωρούμε στη διασύνδεση των IP με ένα επεξεργαστικό σύστημα βασισμένο στον ARM και δημιουργούμε το συνολικό σύστημα. Τέλος, ακολουθεί η δημιουργία ενσωματωμένων Linux διανομών για το σύστημά μας και η ανάπτυξη μιας εφαρμογής που θα εκτελεστεί στο επεξεργαστή. Οι επιταχυντές υλικού που χρησιμοποιήθηκαν για την αξιολόγηση και ανάλυση των εναλλακτικών σχεδίων ανήκουν σε διαφορετικά επιστημονικά πεδία. Ο πρώτος είναι μία υλοποίση του αλγορίθμου ανίχνευσης γωνιών Harris & Stephens. Ο δεύτερος είναι ένας ταξινομητής Μηχανών Διανυσμάτων Υποστήριξης (Support Vector Machines – SVM) για την καρδιακή αρρυθμία που χρησιμοποιεί τη βάση δεδομένων ΗΚΓ MIT-BIH. Διαφέρουν όχι μόνο στα επιστημονικά τους πεδία αλλά επίσης στο μέγεθος των δεδομένων εισόδου, στην πολυπλοκότητα του κώδικα και στη χρησιμοποίηση πόρων. Η ανάλυση μας παρουσιάζει την επίδραση των διαφορετικών διεπαφών επικοινωνίας στο χρόνο εκτέλεσης, στο εύρος ζώνης, στη χρησιμοποίηση πόρων του FPGA και στη συνολική επίδοση του συστήματος. Η διερεύνηση

των εναλλακτικών διεπαφών και διασυνδέσεων για μία συγκεκριμένη έκδοση ενός επιταχυντή κατέληξε σε κέρδος μέχρι και 20% στο χρόνο εκτέλεσης και σημαντικό κέρδος στο εύρος ζώνης.

Λέξεις-Κλειδιά: Σύνθεση Υψηλού Επιπέδου, AMBA AXI, AXI4-Lite, AXI4-Stream, Αναπτυξιακή Πλακέτα Zynq Evaluation and Development Board, Άμεση Πρόσβαση Μνήμης, Αλγόριθμος Ανίχνευσης Γωνιών Harris & Stephens, Μηχανές Διανυσμάτων Υποστήριξης, Ανάλυση ΗΚΓ

Abstract

In recent years, the design of hardware accelerators has been established as a standard practice when targeting to optimizations of algorithmic implementations. FPGA-based accelerators, in particular, have gained the interest of system architects and the scientific world due to the innate fast hardware development and reconfiguration capabilities that are offered by an FPGA device. These features, combined with the level of design abstractions of High-Level Synthesis (HLS) frame a definite solution when it comes to fast prototyping of system designs. Lately, the tendency for an FPGA device is to comprise the benefits of embedded processors, thus forming a whole system-on-a-chip (SoC). The coexistence of hardware accelerators and embedded processors on a single device have brought the interconnection of these components to the proscenium as an element of vital significance for the performance of the whole system. In order for the custom hardware to be readily interconnected to a processing system, the Intellectual Property (IP) design style has been adopted. Typically, an IP is equipped with control and communication interfaces so that it can be easily combined with other components, in most cases, without the utilization of additional hardware. A widely used communication interface for IP generation is the ARM AMBA Advanced eXtensible Interface (AXI) protocol. Design alternatives offered by the AXI might range from simple low-bandwidth communication and data transfers to higher values of bandwidth by employing the available Direct Memory Access features. In this work, we focus on the system implementation flow targeting to a Zynq-7000 AP SoC device. Beginning with the addition of different communication interfaces we generate custom accelerator IPs through HLS. Then we proceed to the interconnection of those IPs with an ARM-based processing system and generate the system design. The final steps include the generation of Embedded Linux distributions for our custom hardware and the development of a userspace application to be executed on the processing system of our design. The hardware accelerators that are employed for evaluation and analysis of design alternatives appertain to two distinct scientific fields. The first one is an implementation of the Harris & Stephens Corner Detection Algorithm. The second is a Support Vector Machine classifier for arrhythmia detection using MIT-BIH ECG signal database. The employed accelerators differ not only in their respective fields but also in the input data sizes, complexity of the code and resource needs. Our combined analysis shows the impact of different communication interfaces in latency, bandwidth, utilized FPGA resources and overall system performance. The exploration of different interface and interconnection configurations for a default accelerator lead to latency gains of up to 20% and significant bandwidth gains.

Keywords: High-Level Synthesis, AMBA AXI, AXI4-Lite, AXI4-Stream, ARM, Zynq Evaluation and Development Board, Direct Memory Access, Embedded Linux, HW/SW codesign, Harris and Stephens Corner Detection Algorithm, Support Vector Machines, ECG Analysis

This page is intentionally left blank.

Ευχαριστίες

Η εκπόνηση της παρούσας διπλωματικής εργασίας έγινε σε συνεργασία με το Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του ΕΜΠ. Θα ήθελα να εκφράσω της ειλικρινείς ευχαριστίες μου στον κύριο Δημήτριο Σούντρη, Αναπληρωτή Καθηγητή της ΣΗΜΜΥ για την εμπιστοσύνη που μου έδειξε ώστε να μου εμπιστευτεί ένα αρκετά απαιτητικό θέμα προς εκπόνηση, αλλά και για της εκπαιδευτικές ευκαιρίες και γνώσεις που μου προσέφερε, όχι μόνο στο πλαίσιο των μαθημάτων του εργαστηρίου αλλά και μέσω της προσωπικής μας επικοινωνίας.

Θα ήθελα στη συνέχεια να ευχαριστήσω ειλικρινά τον Βασίλη Τσούτσουρα, Υποψήφιο Διδάκτορα και δεύτερο πρωταγωνιστή στην εκπόνηση αυτής της διπλωματικής εργασίας. Με τα εποικοδομητικά του σχόλια, την καλή διάθεση και την αρκετά μεγάλη υπομονή και κατανόηση που έδειχνε στις συναντήσεις μας μου έδινε συνεχώς θάρρος να συνεχίζω και να επικεντρώνομαι στο στόχο μου.

Θα ήθελα επίσης να ευχαριστήσω τον Γιώργο Λεντάρη και τον Σωτήρη Ξύδη, Μεταδιδακτορικούς Ερευνητές για τις περιορισμένες μεν αλλά πάντα εποικοδομητικές συζητήσεις και για τα χρήσιμα σχόλιά τους.

Στη συνέχεια, δεν θα μπορούσα να μην ευχαριστήσω τον Γιάννη Γαλάνη, Διπλωματούχο Ηλεκτρολόγο Μηχανικό και Μηχανικό Υπολογιστών για την παροχή του πηγαίου κώδικα για την υλοποίηση του Αλγορίθμου Ανίχνευσης γωνιών Harris & Stephens. Στο ίδιο πλαίσιο αλλά σε πολύ μεγαλύτερο βαθμό θα ήθελα να εκφράσω τις ευχαριστίες και την ευγνωμοσύνη μου στην Κωνσταντίνα Κολιογεώργη, Διπλωματούχα Ηλεκτρολόγο Μηχανικό και Μηχανικό Υπολογιστών αφενός για την παροχή του πηγαίου κώδικα για το SVM, αφετέρου γιατί η εξαιρετική της εργασία και οργάνωση του κώδικα μου προσέφερε ένα μεγάλο κέρδος σε μονάδες χρόνου.

Θέλω επίσης να ευχαριστήσω όλους του ανθρώπους που συνυπάρχουν στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων για το εξαιρετικό κλίμα που δημιουργούν αλλά και για τη διαθεσιμότητα τους σε οποιαδήποτε απορία που μπορεί να είχε προκύψει, όπως επίσης και τους συμφοιτητές οι οποίοι εκπονούν και εκείνοι τη διπλωματική τους εργασία με τους οποίους περάσαμε ατελείωτες ώρες ο ένας δίπλα στον άλλον στο εργαστήριο.

Περισσότερο από όλους, θέλω να ευχαριστήσω την οικογένεια μου που με στήριξε σε κάθε μου επιλογή και μου προσέφερε από τη γέννησή μου όλα αυτά που μου έδωσαν τη δυνατότητα σήμερα να γράφω αυτό το ευχαριστήριο σημείωμα στις πρώτες σελίδες της Διπλωματικής μου εργασίας. Τέλος, να ευχαριστήσω την αδελφή μου και όλους μου τους φίλους, σχολικούς και “πανεπιστημιακούς” που με κάνουν να χαμογελώ και να ανταπεξέρχομαι σε οποιαδήποτε συγκυρία με αισιοδοξία, κάτι που θα ήταν αδύνατο χωρίς την ύπαρξή τους.

This page is intentionally left blank.

Acknowledgements

The present diploma thesis is the result of the work and collaboration with the Microprocessors and Digital Systems Laboratory (MicroLab) of National Technical University of Athens. I would like to express my sincere gratitude to Prof. Dimitrios Soudris for the trust he showed, granting me such a demanding thesis but also for the educational opportunities and knowledge he offered me, not only through the Lab's courses but also through our personal communication.

I would like to sincerely thank Phd Student Vasilis Tsoutsouras, who can be considered are a second protagonist during the work for this diploma thesis. His insightful comments, good mood, patience and understanding he showed during our meetings gave me courage to continue and focus on my target.

I would also like to thank Postdoctoral Researchers George Lentaris and Sotiris Xydis for our limited but always full with constructive and insightful comments conversations.

Thanks to Giannis Galanis, Graduate Student of the School of Electrical and Computer Engineering for the provision of the Harris & Stephens Corner Detection Algorithm source code. Special thanks to Konstantina Koliogeorgi for the provision of the SVM classifier source code but also for her excellent work.

I would also like to thank all the people that coexist in MicroLab for the great environment that they create and their availability whenever I had a question and fellow undergraduate students with whom we worked together, side by side for long hours in the Lab.

Most of all, I would like to thank my family which has been supporting every choice I have made and since my birth have offered me all these that today give me the opportunity to write this thank you note in the first pages of my diploma thesis. Finally, I would like to thank my sister and my friends for making me smile and cope with any situation that has occurred with optimism, something which would be impossible without their existence.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Θεωρητικό Υπόβαθρο

Αλγόριθμος Ανίχνευσης Γωνιών Harris & Stephens

Ο Αλγόριθμος Ανίχνευσης γωνιών Harris & Stephens είναι ένας αλγόριθμος ο οποίος όπως υποδηλώνει το όνομά του, έχει στόχο την ανίχνευση γωνιών σε εικόνες. Τα βασικά στοιχεία του αλγορίθμου αυτού όπως υλοποιήθηκε στη συνέχεια της διπλωματικής εργασίας είναι τα εξής:

- Ο αλγόριθμος παίρνει επικαλυπτόμενα παράθυρα της εικόνας και τα μετακινεί προς όλες της κατευθύνσεις ώστε να εντοπίσει τις μεταβολές στην ένταση της εικόνας. Αρχικά η συνάρτηση που δίνει την ένταση της εικόνας σε κάθε σημείο δίνεται από την εξής σχέση:

$$I(x+u, y+v) \approx I(u,v) + x I_x(u,v) + y I_y(u,v)$$

  Στη συνέχεια υπολογίζεται το άθροισμα των τετραγώνων των διαφορών ως εξής

$$E(x,y)=\sum_{u,v} w(u,v)(I(u,v)+x I_x(u,v)+y I_y(u,v))^2 \quad \text{ή} \quad E(x,y)=\begin{bmatrix} x & y \end{bmatrix} A \begin{bmatrix} x \\ y \end{bmatrix}$$

  όπου

$$A = \sum_{u,v} w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix} \cdot$$

- Ένα σημαντικό στοιχείο του αλγορίθμου Harris είναι η χρήση γκαουσιανού παραθύρου για την ομαλοποίηση της εικόνας που εξασφαλίζει μία μία θορυβώδη απόκριση.
- Στον αλγόριθμο του Harris οι γωνίες θεωρούνται πως παρουσιάζουν μεγάλη διακύμανση του αθροίσματος των τετραγώνων των διαφορών σε κάθε κατεύθυνση. Αν ένα σημείο ενδιαφέροντος/γωνία εξετάζεται τότε ο πίνακας A πρέπει να έχει δύο ιδιοτιμές με μεγάλη τιμή. Αν $\lambda_1 \approx 0,\ \lambda_2 \approx 0$ τότε το σημείο που εξετάζεται δεν είναι σημείο ενδιαφέροντος. Αν $\lambda_1 \approx 0$ και λ_2 τότε έχουμε ανίχνευση μιας ακμής. Τέλος αν και οι δύο ιδιοτιμές είναι μεγάλες θετικές τιμές τότε έχουμε εντοπίσει μία γωνία.

Θεωρία Μηχανών Διανυσμάτων Υποστήριξης

Οι Μηχανές Διανυσμάτων Υποστήριξης (Support Vector Machines – SVM) είναι μοντέλα επιβλεπόμενης μάθησης που εκπαιδεύονται με ένα μεγάλο σύνολο δεδομένων και είναι κατάλληλα για την ταξινόμηση των νέων εισόδων σε δύο υποψήφιες κλάσεις συμπληρωματικές μεταξύ τους. Το σύνολο εκπαίδευσης αποτελείται από διανύσματα με συγκεκριμένα χαρακτηριστικά και μία ετικέτα της κλάσης στην οποία ανήκει το κάθε διάνυσμα.

Τα SVM εφαρμόζουν αρχικά μία συνάρτηση πυρήνα που ανάγει τα διανύσματα σε έναν χώρο με περισσότερες διαστάσεις όπου ο διαχωρισμός είναι πιο εύκολος. Στο χώρο αυτό εντοπίζεται ένα υπερεπίπεδο που αποτελείται από διανύσματα που απέχουν μέγιστα από αυτά της κάθε κλάσης. Κάθε νέο διάνυσμα ανάγεται σε αυτόν το χώρο, υπολογίζεται η απόστασή του από το υπερεπίπεδο και αναλόγως ταξινομείται σε κάποια κλάση. Η συνάρτηση πυρήνα παίζει έναν πρωταγωνιστικό ρόλο στη ακρίβεια και την πολυπλοκότητα του μοντέλου. Στο πρόβλημα που θα εξετάσουμε προτιμούμε μη γραμμική συνάρτηση πυρήνα και συγκεκριμένα εκθετικής φύσης για τον διαχωρισμό των παλμών της καρδιάς.

Η μαθηματική εξίσωση που περιγράφει τον υπολογιστικό πυρήνα του ταξινομητή είναι η παρακάτω:

$$Class = sgn\left(\sum_{i=1}^{N_{sv}} \left(y_i * a_i * \exp\left(-\gamma \| x - \text{sup\_vector}_i \|^2\right)\right) - b\right)$$

όπου x είναι το διάνυσμα του παλμού προς ταξινόμηση, sup_vector(i) είναι το I-οστό διάνυσμα υποστήριξης και y_i , a_i είναι τιμές διαφορετικές για κάθε διάνυσμα υποστήριξης και προέκυψαν κατά την εκπαίδευση του SVM.

Ροή Εργασίας για Υλοποιήσεις στο ZedBoard

ZedBoard

Το ZedBoard είναι μία αναπτυξιακή πλακέτα χαμηλού κόστους. Είναι ένα σύστημα που έχει υλοποιηθεί σε ολοκληρωμένο κύκλωμα (SoC) που ανήκει στην οικογένεια Zynq-7000 AP SoC της Xilinx. Συνδυάζει την ύπαρξη ενός υπολογιστικού συστήματος με δύο επεξεργαστές ARM με την ύπαρξη επαναπρογραμματιζόμενης λογικής. Υποστήζει υλοποίηση Linux, Android και RTOS εφαρμογών. Τα κύρια χαρακτηριστικά του ZedBoard είναι τα εξής:

- **Μνήμη:** δυναμική (DDR3) και στατική μνήμη (SPI Flash, Διεπαφή κάρτας SD)
- **USB:** USB-to-UART σύνδεση, λειτουργικότητα JTAG, προστασία κυκλωμάτων
- **Οθόνη και Ήχος:** HDMI πομπός, Analog Device Audio Codec, OLED dislpay
- **Clock Sources:** 33.3333 MHz ρολόι για το υπολογιστικό σύστημα και παροχή έως και τεσσάρων ρολογιών για το επαναπρογραμματιζόμενο μέρος.
- **Reset Sources:** εξωτερικοί διακόπτες για επανεκκίνηση της πλακέτας και επαναπρογραμματισμό
- **User I/O:** 7 user GPIO push buttons, 8 user dip switches, 8 LEDS
- **10/100/1000 Ethernet PHY**
- **PS και PL I/O επεκτάσεις**

Στόχος της παρούσας εργασίας είναι η προσθήκη στο ZedBoard επιταχυντών υλικού που έχουν σχεδιαστεί και παραχθεί με τη βοήθεια της σύνθεσης υψηλού επιπέδου και η μελέτη της επικοινωνίας τους με το διαθέσιμο επεξεργαστικό σύστημα.

Δημιουργία IP με Σύνθεση Υψηλού Επιπέδου

Το πρώτο βήμα για μία υλοποίηση ενός αλγορίθμου ή ενός επιταχυντή στην αναπτυξιακή πλακέτα ZedBoard είναι το βήμα της σύνθεσης υψηλού επιπέδου (High-Level Synthesis – HLS). Κατά τη διαδικασία της σύνθεσης υψηλού επιπέδου επιλέγουμε τις απαραίτητες διεπαφές επικοινωνίας και ελέγχου. Στην παρούσα εργασία, οι διεπαφές που χρησιμοποιήθηκαν ήταν τα πρωτόκολλα AXI4-Lite και AXI4-Stream. Και τα δύο μπορούν να προστεθούν πολύ εύκολα από την καρτέλα Directives του Vivado HLS. Στην περίπτωση του AXI4-Lite το εργαλείο προσθέτει αυτόματα εκτός από το AXI4-Lite πρωτόκολλο στη συνάρτηση και ένα πρωτόκολλο επιπέδου-μπλοκ για τον έλεγχο του συγκεκριμένου επιταχυντή, δηλαδή την εκκίνηση των υπολογισμών,

τον έλεγχο ολοκλήρωσης των υπολογισμών και άλλα. Επιπλέον για IP με AXI4-Lite διεπαφές γίνοται αυτόματη δημιουργία ενός οδηγού υλικού για τη συγκεκριμένη συσκευή μέσω του οποίου μπορούμε να έχουμε πρόσβαση στη μνήμη της. Αντιθέτως όταν προσθέτουμε AXI4-Stream διεπαφές επιλέγουμε ο έλεγχος της λειτουργίας να μην γίνεται με πρωτόκολλα επιπέδου-μπλοκ. Αντίθετα, τοποθετούμε τον έλεγχο εντός του επιταχυντή. Στις stream υλοποιήσεις οι επιταχυντές μας πρώτα συλλέγουν τις τιμές που απαιτούνται για τον υπολογισμό και στη συνέχεια εκτελούν τον υπολογισμό ενώ επιπλέον δεδομένα που μπορεί να βρίσκονται στην είσοδο δεν διαβάζονται μέχρι να συλλεχθούν όλα και να εκκινήσει ο υπολογισμός.

Δημιουργία του Συνολικού Συστήματος

Μετά τη δημιουργία των επιταχυντών υλικού σε μορφή IP σειρά έχει διασύνδεση του με το επεξεργαστικό σύστημα (PS) του ZedBoard και η δημιουργία του συνολικού συστήματος. Στην περίπτωση των AXI4-Lite πρωτοκόλλων η διασύνδεση γίνεται κυριολεκτικά με το πάτημα ενός κουμπιού. Αντιθέτως στην περίπτωση των AXI4-Stream πρωτοκόλλων η διασύνδεση δε γίνεται αυτόματα. Ο χρήστης πρέπει να προσθέσει ένα AXI DMA μπλοκ για τη μεταφορά των δεδομένων. Στη συνέχεια εκτελείται η σύνθεση και η υλοποίηση του συστήματος και εξάγεται το αρχείο bitstream που χρησιμοποιείται για τον προγραμματισμό του FPGA.

Δημιουργία Linux Διανομών

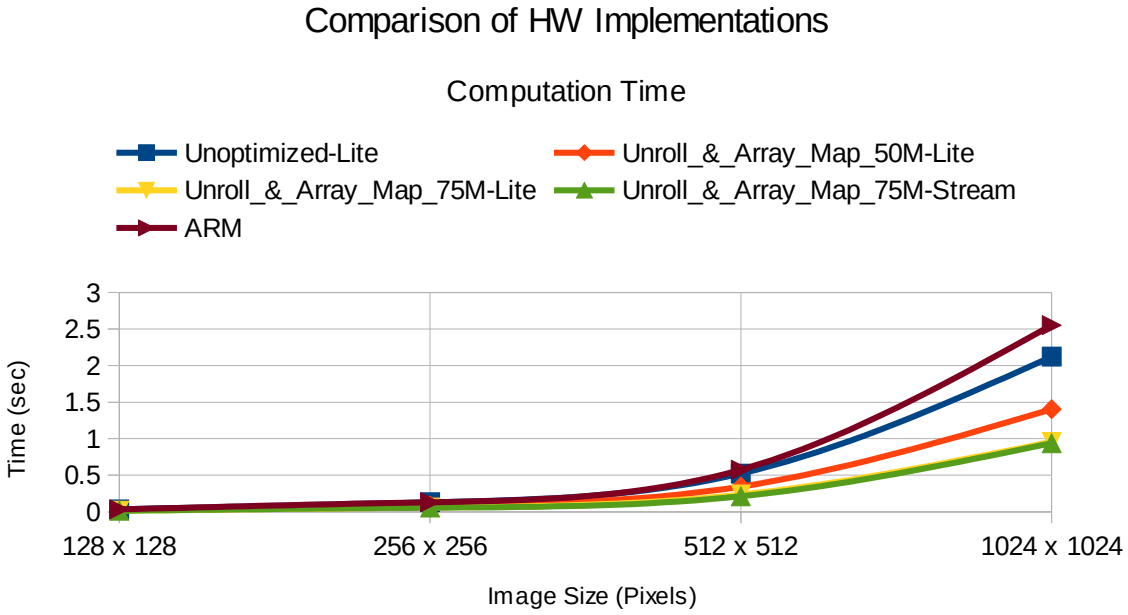
Έπειτα από την υλοποίηση του συστήματος το επόμενο βήμα είναι η δημιουργία μιας ενσωματωμένης Linux διανομής για το σύστημά μας. Για το σκοπό αυτό χρησιμοποιούμε τα Petalinux Tools της Xilinx. Με τα Petalinux δημιουργούμε μία νέα πλατφόρμα Linux για το ZedBoard και στη συνέχεια από την περιγραφή υλικού που έχει εξαχθεί προηγουμένως χτίζουμε μία νέα διανομή για το δικό μας σύστημα. Φορτώνουμε την εικόνα της διανομής στην κάρτα SD και στη συνέχεια μπορούμε να συνδεθούμε μέσω της σειριακής θύρας και του προγραμμάτος GtkTerm με τη συσκευή.

Ανάπτυξη Εφαρμογών στο ZedBoard

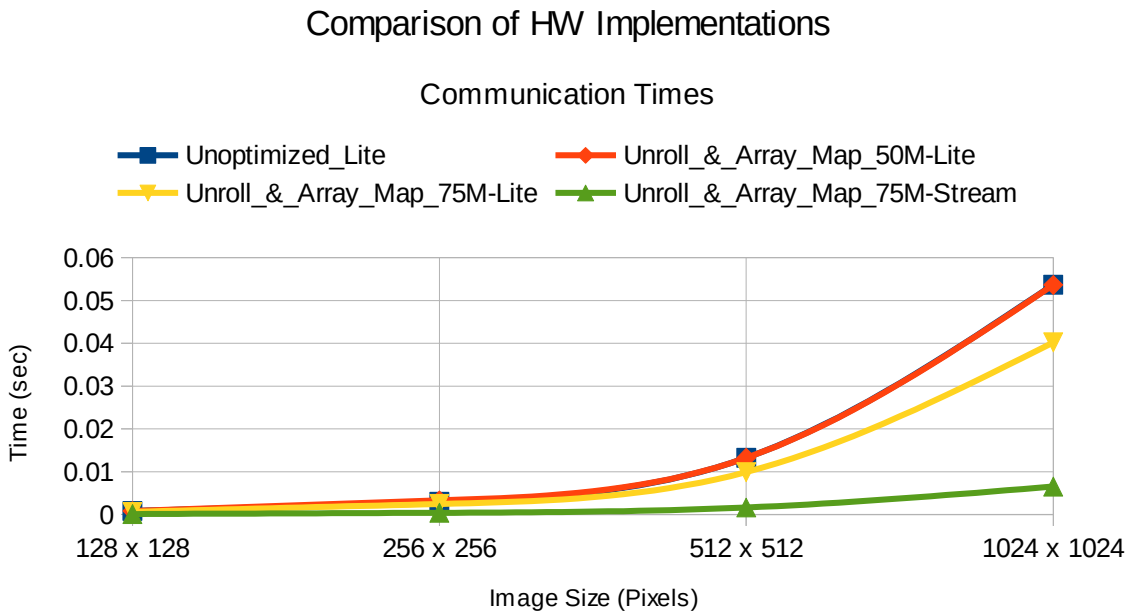
Αφού έχουμε δημιουργήσει την πλατφόρμα που τρέχει στο επεξεργαστικό σύστημα στη συνέχεια πρέπει να αναπτύξουμε μία εφαρμογή που τρέχει στο χώρο χρήστη, αποκτά πρόσβαση και ελέγχει τον επιταχυντή. Οι συσκευές που διαθέτουν AXI4-Lite πρωτόκολλο μπορούν να απεικονιστούν στο χώρο χρήστη μέσω του Linux UIO οδηγού. Αντίθετα για την ανάπτυξη εφαρμογών για συσκευές με AXI4-Stream πρωτόκολλα η διαδικασία είναι διαφορετική καθώς απαιτείται ένας πιο πολύπλοκος οδηγός. Η ανάπτυξη για αυτή την περίπτωση έγινε με τη βοήθεια του zynq-xdma driver [https://github.com/bmartini/zynq-xdma].

Αξιολόγηση Ροής Εργασίας για τον Αλγόριθμο Ανίχνευσης Γωνιών Harris & Stephens

Κατά την εφαρμογή της προτεινόμενης ροής εργασίας στον αλγόριθμο ανίχνευσης γωνιών Harris προχωρήσαμε σε πέντε διαφορετικές υλοποιήσεις στο ZedBoard. Η πρώτη υλοποίηση δεν περιελάμβανε καμία βελτιστοποίηση, οι επόμενες περιελάμβαναν την προσθήκη της ντιρεκτίβας UNROLL και ARRAY_MAP με ρολόι 50 MHz και 75 MHz. Προχωρήσαμε σε υλοποιήσεις χρησιμοποιώντας τα πρωτόκολλα AXI4-Lite και AXI4-Stream. Στις υλοποιήσεις αυτές κατεφέραμε να πετύχουμε ένα εύρος ζώνης μέχρι και 154 MB/s . Παρακάτω μπορούμε να δούμε συγκριτικά διαγράμματα για τον απαιτούμενο χρόνο επικοινωνίας και υπολογισμού σε κάθε υλοποίηση.



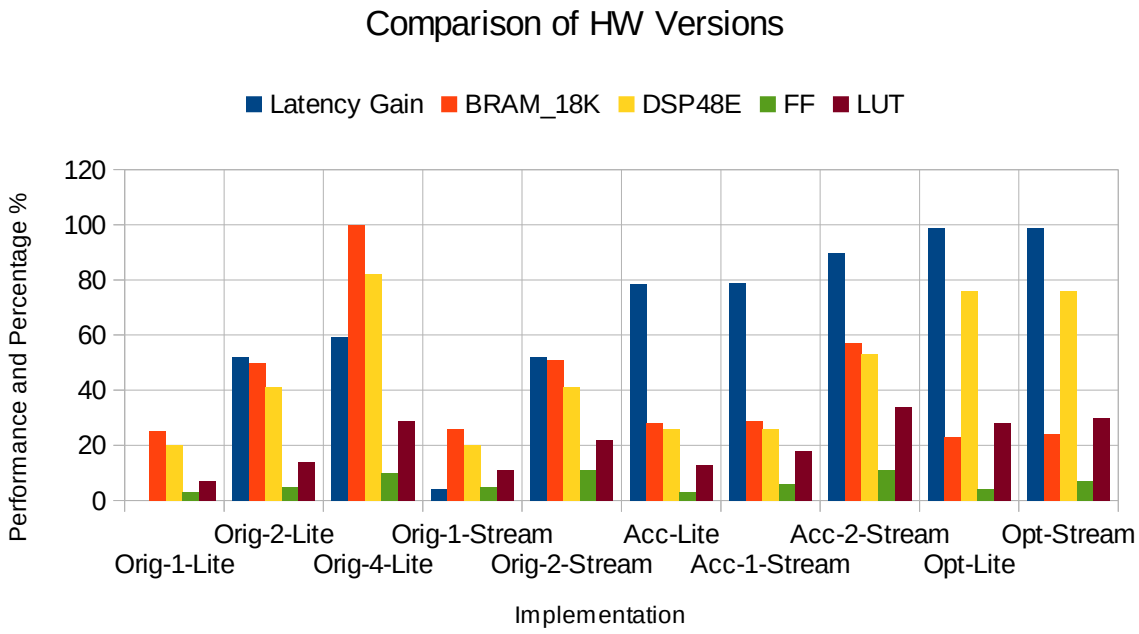
Σχήμα 1: Χρόνου υπολογισμού για της υλοποιήσεις του Harris



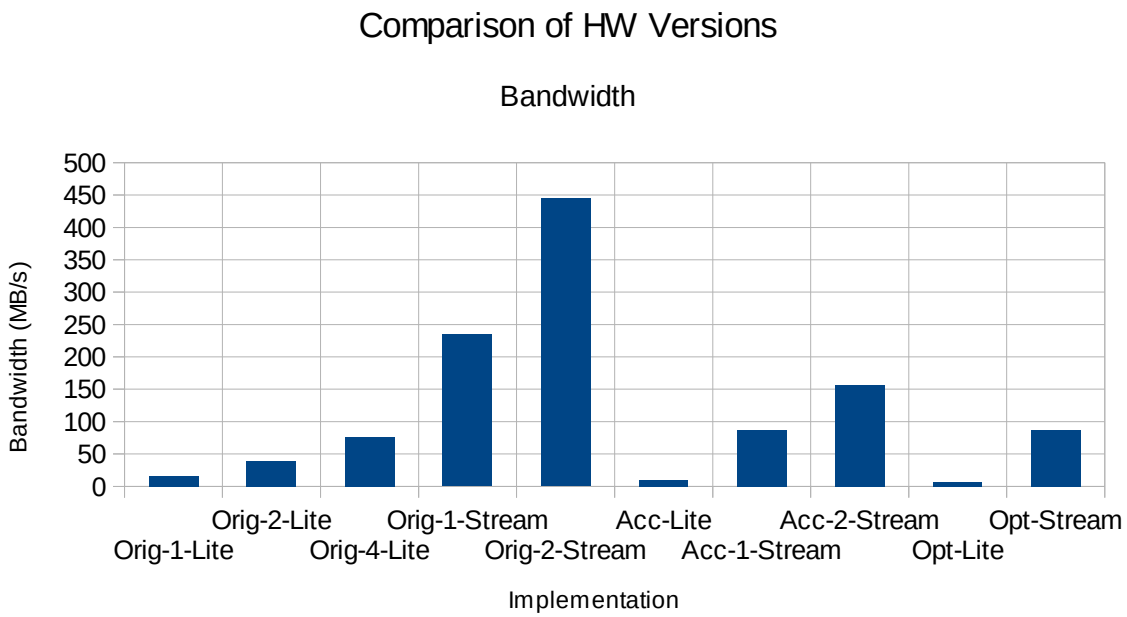
Σχήμα 2: Χρόνοι Επικοινωνίας για τις υλοποιήσεις του Harris

Αξιολόγηση Ροής Εργασίας για τον Ταξινομητή SVM

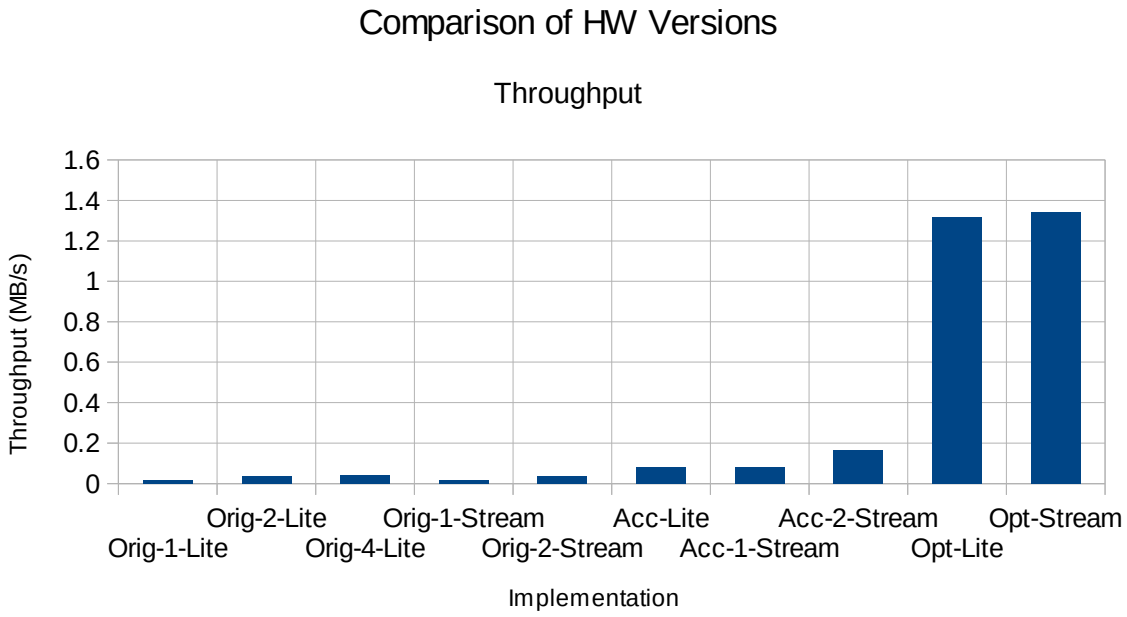
Κατά την ενασχόλησή μας με τον ταξινομητή SVM προχωρήσαμε σε εξαγωγωγή μέσω του HLS έξι διαφορετικών εκδόσεων του κώδικα. Οι πρώτες έκδοσεις είναι ο γνήσιος κώδικας χωρίς βελτιστοποιήσεις με AXI4-Lite και AXI4-Stream. Η 3η και 4η έκδοση είναι ένας επιταχυμένος κώδικας και πάλι με AXI4-Lite και AXI4-Stream υλοποιήσεις. Τέλος, υλοποιούμε και δύο εκδόσεις της βέλτιστης εκδοχής του κώδικα. Συνολικά οι υλοποιήσεις για τον ταξινομητή ήταν πέντε για τον γνήσιο κώδικα, με χρήση και περισσοτέρων του ενός IP, τρεις για τον ενδιάμεσο κώδικα και δύο για τον βέλτιστο. Το εύρος ζώνης που καταφέραμε να πετύχουμε ήταν στα 444 MB/s ενώ για τη βέλτιστη έκδοση του επιταχυντή η ίδια ακριβώς υλοποίηση με AXI4-Stream προσφέρει ένα 20% κέρδος σε σχέση με την αντίστοιχη AXI4-Lite. Παρακάτω μπορούμε να δούμε σχετικά διαγράμματα.



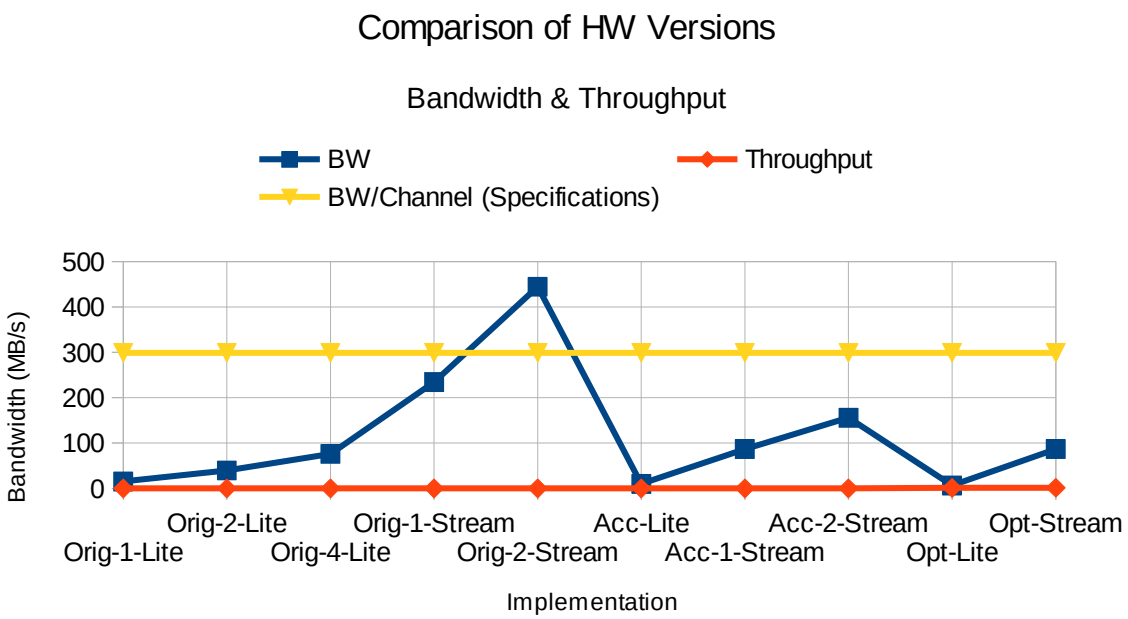
Σχήμα 3: Επίδοση και Κέρδος για διαφορετικές SVM υλοποιήσεις



Σχήμα 4: Εύρος Ζώνης για διαφορετικές SVM υλοποιήσεις



Σχήμα 5: Throughput για διαφορετικές SVM υλοποιήσεις



Σχήμα 6: Εύρος Ζώνης και Throughput για διαφορετικές SVM υλοποιήσεις

This page is intentionally left blank.

List of Figures

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

1.1 Introduction to FPGA

A Field-Programmable Gate Array (FPGA) is an integrated digital circuit (IC) which is constituted of a number of Configurable Logic Blocks interconnected with programmable connections. The term "field" denotes the fact that the FPGA is programmable on the spot in comparison to other integrated circuits whose functionalities cannot be altered after integration.

The reconfigurability that FPGAs offer is an element that enhances flexibility and makes them a very good platform for quick implementations and prototyping of system designs. The correction of errors is made easy and bears a very low cost in comparison with Application-Specific Integrated Circuit (ASIC) implementations which require a large amount of time and bear a higher cost.

FPGAs can be configured for various applications. In addition, almost every computational algorithm can be implemented on an FPGA. Applications in which FPGAs are widely used include Digital Communications, Image Processing, Digital Signal Processing and others. Moreover, an FPGA is capable of implementing a System on a Chip (SoC), a fact which gives the ability of a unified hardware-software approach to the design and implementation of applications.

1.1.1 History

Fixed logic devices, a name which implies devices that cannot be reprogrammed, were the first approach to system designs. Although they were widely used, the large amount of time requirements for the transition from a design to a prototype along with the fact that error correction would demand a new design and implementation led the way to fabrication of Programmable Logic Devices (PLDs).

One of the first attempts in the PLD field were Programmable Logic Arrays (PLAs). PLAs consisted of a set of AND gates and another set of OR gates (Fig. 1.1) which could be

conditionally complemented to produce an output. PLAs were mainly used for implementing combinatorial logic circuits [1]. Programmable Array Logic (PAL) was an evolution of PLA. PAL devices consisted of a small programmable read-only memory (PROM) core and additional output logic used to implement various logic functions with a few components [2].

The beginning of a new technology and market occurred in 1985 when Xilinx co-founders R. Freeman and B. Vonderschmitt introduced the first FPGA which was the first device which had programmable gates and interconnects. Since then, and especially during the 1990s, FPGA production grew explosively. Various vendors entered the market and the competition increased. In the early 1990s FPGAs were primarily used in telecommunications and networking, however, by the end of the decade and lately FPGA usage expanded to consumer, automotive, and industrial applications [3].

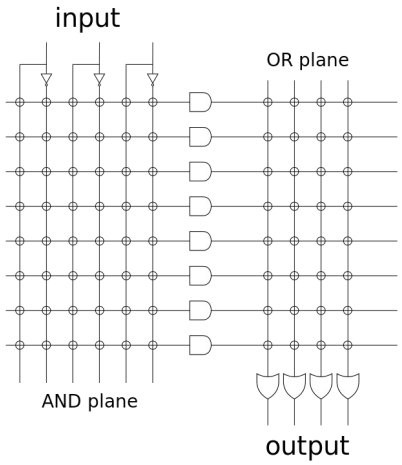


Figure 1.1: A PLA schematic paradigm [1]

1.1.2 FPGA Attributes and Advantages

The appearance of FPGAs in the market was accompanied by low speeds and high power consumptions. Additionally, early FPGAs carried a finite number of functionalities. The above mentioned are some of the reasons that made ASIC implementations preferable. Nowadays, FPGAs have drastically evolved and are capable of providing solutions which overpower the equivalent ASIC ones. A list of reasons which led to the proliferation of FPGA production and use can be seen below:

- **Increased Speeds.** When the origination of FPGAs occurred they were used for lower speed designs, however, current FPGAs easily push the 500MHz performance barrier and readily support higher speed designs.

- **Low Power Consumption.** FPGA vendors are constantly pursuing the minimization of power consumptions. With approaches like a triple-oxide process technology for

transistors to reduce their static power consumption or a shift to coarse-grained logic architectures for more compact designs and minimization of dynamic power consumption, FPGA vendors along with FPGA programmers have managed to decrease power consumption through time.

- **Declining Cost per Unit.** The competition among various FPGA vendors has been proven beneficial to users. Today, customers are able to purchase 1 million-gate FPGAs for much less than $100 in low volumes and for tens of dollars in higher volumes.

- **Reconfigurability.** As already mentioned an FPGA's configuration is easily alterable offering high flexibility during the development of applications. On top of that, the latest trend is for an FPGA to partially alter its configuration while operating. More specifically, some regions of the FPGA can be reprogrammed while applications continue their executions in the remainder of the device.

- **Short Time-to-Market.** The relatively fast transition from a design to a prototype allows an FPGA – based product or application to enter the market in a shorter time compared to ASIC implementations (Fig. 1.2).

- **Low NRE Cost.** A consequence of rapid prototyping is the reduction of the non-recurring engineering (NRE) cost, which is defined as the one-time cost to research, develop, design and test a new product.

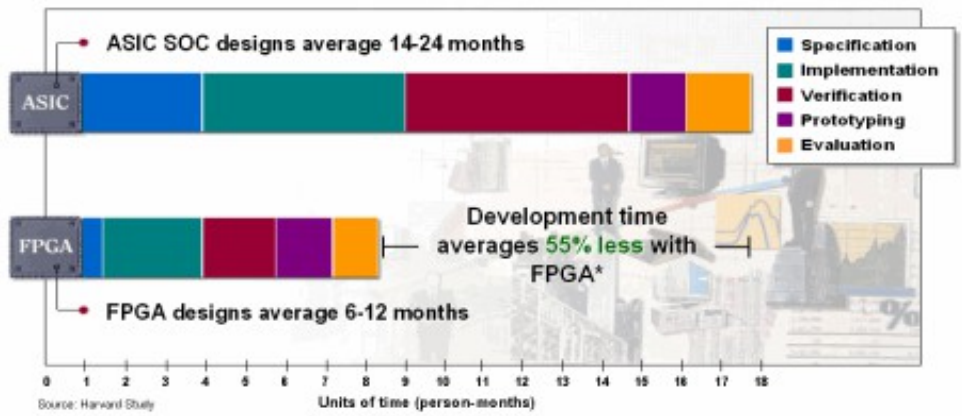


Figure 1.2: FPGA Vs. ASIC SOC design time [4]

The above mentioned are only a few of the features which have led to the prosperity of the FPGA market. Additionally, the short time-to-market combined with the constantly declining NRE costs is a factor which drops the FPGA unit costs below the ASIC ones for high volumes [4].

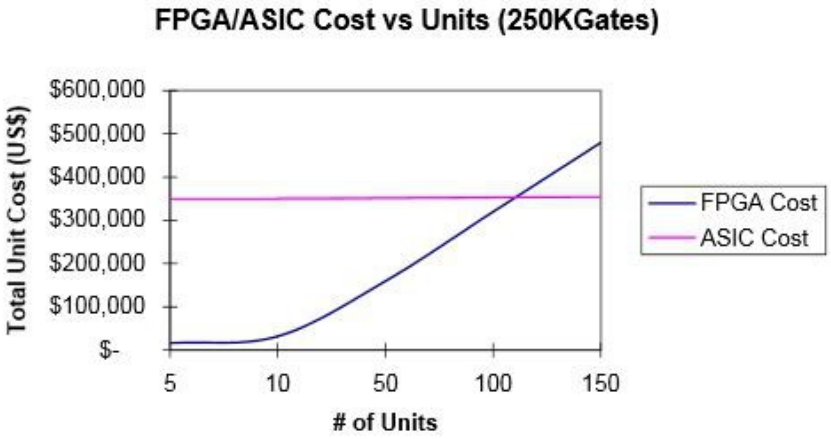


Figure 1.3: FPGA Vs. ASIC cost per unit

1.1.3 State of the Art

Since the dawn of the market in 1985, FPGAs have become increasingly important to the electronics industry, as innovative accomplishments have occurred in the FPGA field. Today's FPGAs are entire programmable systems on a chip (SoC) which are able to cover an extremely wide range of applications. Latest trends make FPGAs a highly flexible alternative to ASICs for a larger number of higher-volume applications, a fact which is mirrored in the growing number of FPGA design starts [5],[6]. For instance, in 2005 FPGA design starts were estimated around 80.000, however, the number had increased to 90.000 by 2008.

The flourishment of the FPGA market could not have been achieved, if a tremendous increment in the number of logic gates had not transpired. Back in 1982 the number of logic gates was 8.192 (Burroughs Advances System Group, integrated into the S-Type 24-bit processor for reprogrammable I/O) for it to rise up to 9.000 in 1987 by Xilinx. Since 1987, an explosive growth in the number of logic gates led to 600.000 gates in 1992 (Naval Surface Warfare Department) [3]. In early 2000s the number of logic gates had already increased to millions.

The latest tendency in the FPGA field is the combination of traditional logic blocks with embedded micro-processors and the essential peripherals to develop a SoC device. Such an innovation was introduced in 2010, when Xilinx presented Zynq®-7000 All Programmable SoC (AP SoC), the first SoC device that combined the features of a Dual-Core ARM® Cortex A9 Processing System (PS) with Programmable Logic (PL), or a dual-core processor with an FPGA core. The combination of the software programmability of an ARM®-based processor with the hardware programmability of an FPGA in a single device offers to developers the capability of applying a hardware-software unified approach to embedded system designs, with a conjunction of serial and parallel processing. On top of that, the Zynq®-7000 AP SoC is architected to deliver the lowest possible system power and system level performance through optimized architecture [7]. It should be mentioned that Zynq®-7000 AP SoC is going to be the target device for the application development in the present diploma thesis.

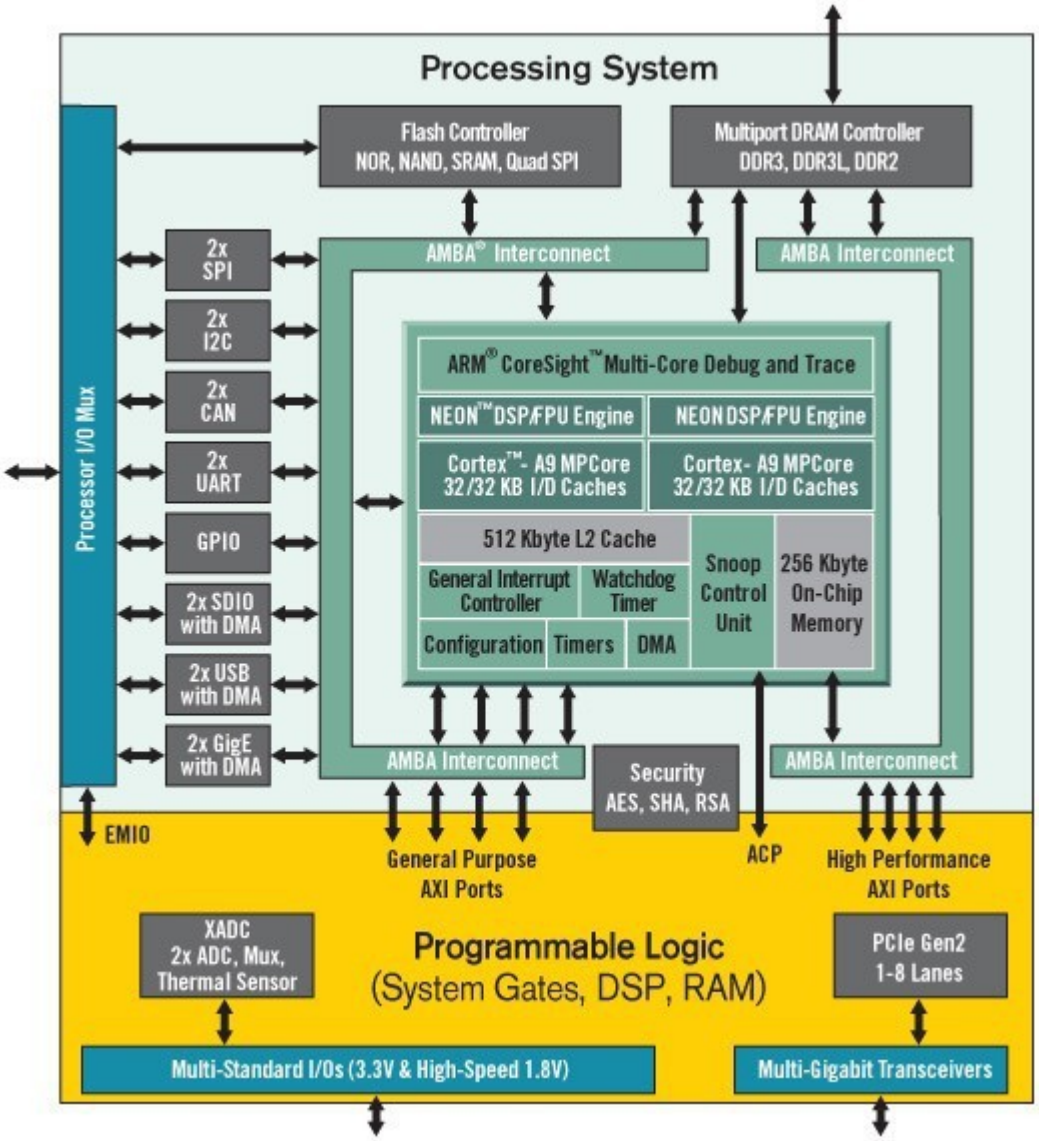


Figure 1.4: Zynq®-7000 All Programmable SoC

Lately, Xilinx launched new 16nm and 20nm UltraScale™ families based on the first all programmable architecture to span multiple nodes from planar through FinFET technologies and beyond, while also scaling from monolithic though 3D ICs. At 20nm Xilinx pioneered the first ASIC-class All Programmable architecture to enable multi-hundred gigabit-per-second levels of system performance with smart processing at full line rates, scaling to terabits and teraflops, while UltraScale+ families, at 16nm, combine new memory, 3D-on-3D, and multiprocessing SoC (MPSoC) technologies [8].

The latest innovations in the FPGA field add to its reconfigurable nature and make it an obvious choice when it comes to rapid prototyping of system designs, hardware accelerators, or even embedded system designs, as it offers a steadily dropping power consumption combined with a steadily increasing speed and data throughput.

1.2 FPGA fabric

An FPGA consists of a number of Configurable Logic Blocks (CLBs), I/O blocks and programmable routing. The CLB serves as the main functional unit of an FPGA. Each FPGA contains a large number of CLBs, which are organized in a two-dimensional array and are interconnected via horizontal and vertical routing channels (Fig. 1.5). A CLB consists of four slices and each slice is composed by two logic cells (Lcs) [9].

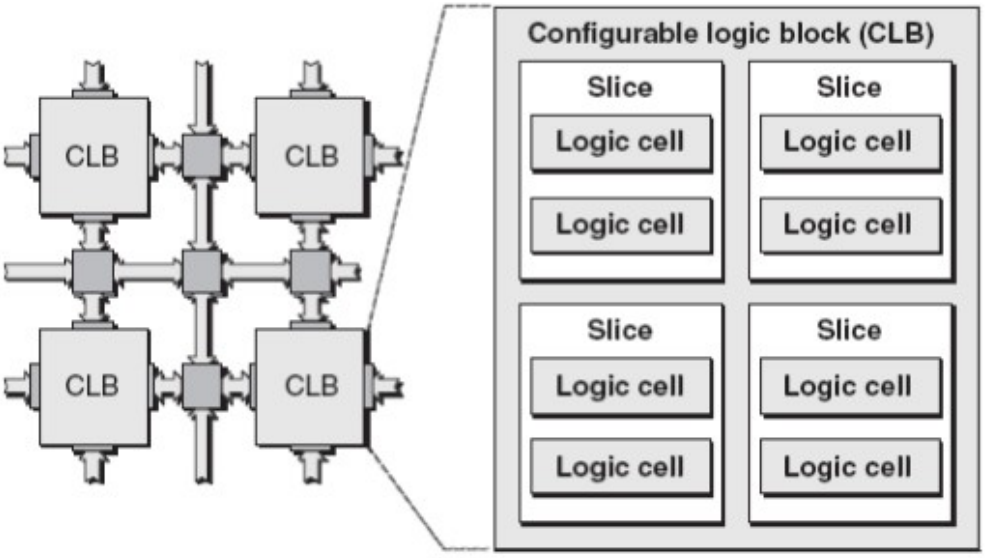


Figure 1.5: An array of CLBs composed by four slices and two logic cells per slice [9]

A logic cell consists of a Look Up Table (LUT) with 4 inputs, a multiplexer and a flip-flop. In addition, FPGAs contain hardwired memories, multipliers and DSP (Digital Signal Processing) Blocks interconnected with the CLBs. Last but not least, a number of I/O blocks, organized in banks, enables the FPGA to communicate with a variety of devices in the outside world, for instance, sensors and processors.

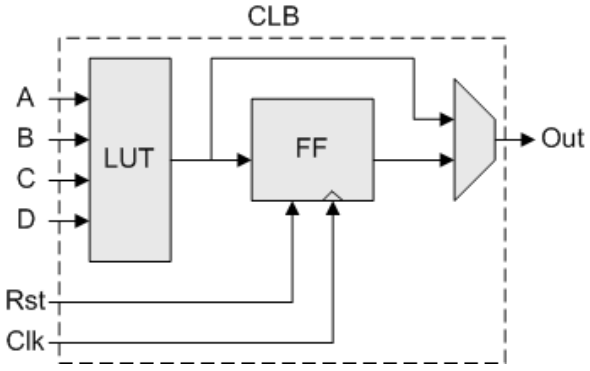


Figure 1.6: A simplified schematic of a logic cell

1.2.1 Look Up Table

A N-LUT is a functional unit capable of computing any function of N inputs. The operation of a LUT resembles the process of finding the value of a logical function via its truth table. Given the truth table of a function, the LUT is programmed accordingly. Then it is responsible for matching a pattern of the N inputs with one of the 2^N rows of the table and generate the corresponding output value. LUTs can be combined to implement more complex functionalities than a N-bit logical function. Specifically, a LUT is able to implement a logical function of N inputs, a N-bit shift register or, alternatively be used as N-bit distributed memory. A N-LUT is usually implemented as a column of 2^N SRAM bits which serve as inputs to a 2^N-to-1 multiplexer. The N inputs of the LUT are used as the select lines of the multiplexer. Additionally, there is a single-bit storage element in the basic logic block in the form of a D flip-flop. The output multiplexer selects either a result generated by the LUT or the bit stored in the D flip-flop.

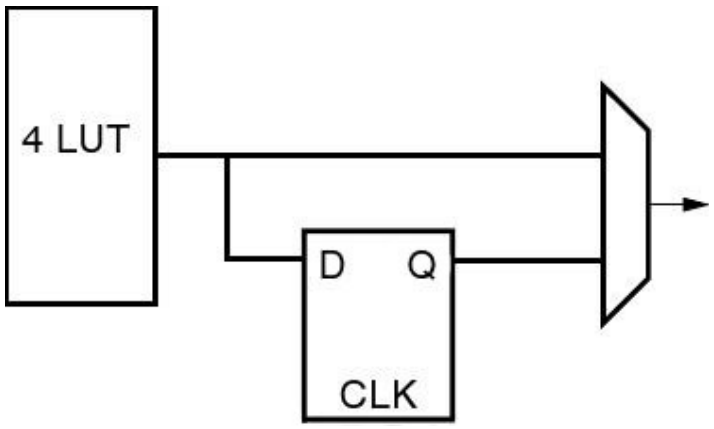


Figure 1.7: A simple 4-bit look up table logic block [10]

Through time, the LUT has been chosen to serve as the smaller computational unit in commercially available FPGAs. However, the size of the LUT in each logic block has been widely investigated. On the one hand, larger look up tables would allow more complex operations to be performed per logic block, thus reducing the wiring delay between blocks along with the number of needed logic blocks. Yet, a large LUT would introduce additional delays due to the requirement of larger multiplexers. On top of that, a larger LUT yields an increased probability of wasting resources if the implemented functionality has lower demands. On the other hand, small look up tables might lead to an increment of logic blocks consumption, thus increasing the wiring delay between blocks. Empirical studies have shown that the 4-LUT structure makes the best trade-off between area and delay for a wide range of benchmark circuits [10].

1.2.2 Hardwired Blocks

As already mentioned, configurable logic blocks serve as the main functional unit of an FPGA, with the look up tables playing an important role in their operation. However, it is currently the rule for an FPGA to have common functionalities embedded into the silicon, in order to reduce the required area and provide increased speed compared to building those functionalities from primitives. Examples of hardwired blocks include multipliers, generic DSP blocks, embedded processors, high-speed I/O logic and embedded memories. It should also be mentioned that, nowadays, it is more and more common for an FPGA to dispose high-speed transceivers, Ethernet MACs, PCI controllers and external memory controllers.

To begin with, FPGA boards are equipped with various memory elements that can be utilized as RAM, ROM or shift registers. One of these elements is the look up table which is discussed in the previous paragraph. Flip-flops also serve as a basic storage unit in an FPGA design. Another significant memory element is the BRAM (Block RAM). The BRAM is a dual-port RAM component which is embedded into the FPGA board and can achieve storage of a large set of data. The capacity of block RAMs usually instantiated is is 18KB and 32KB. Of course, each and every board comes with a specific number of embedded BRAMs [3]. A key element in BRAMs is the dual-port operation which is introducing a parallel behavior as it is providing access to different locations in the same clock cycle.

One of the most important and complex computational unit embedded into the FPGA fabric is the DSP (Digital Signal Processing) Block. The usage of embedded DSP blocks has been established in order to support the increasing amount of computational load. A DSP block is a combination of adders, subtractors and multipliers put together to compose an arithmetic logic unit (ALU). The adder or subtractor unit is connected to a multiplier which has a cascading connection to the final add/subtract/accumulator engine. Following, we can observe a schematic of a DSP block.

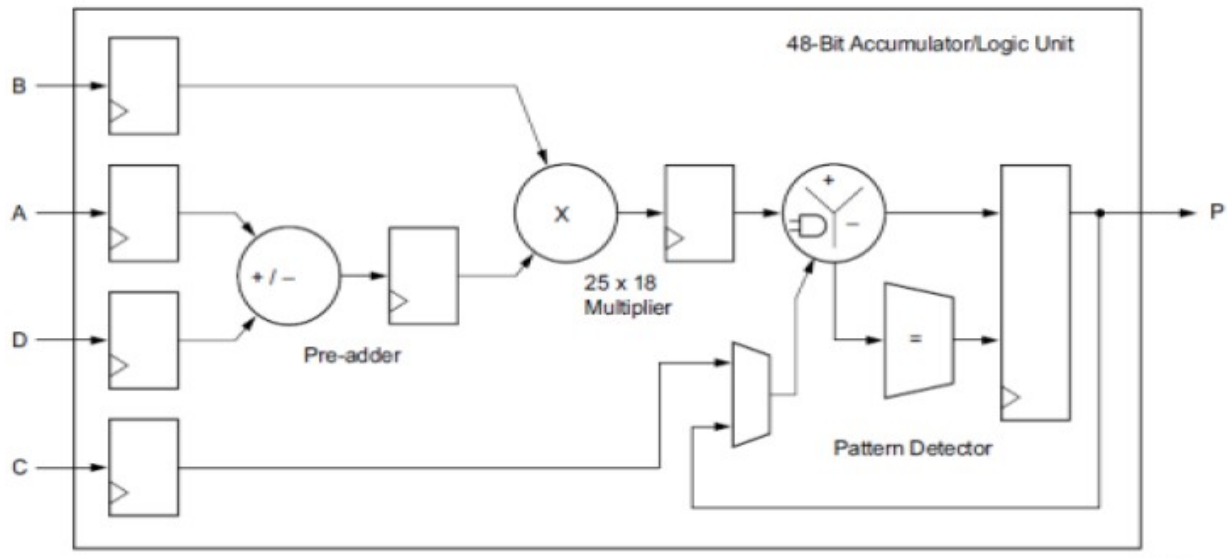


Figure 1.8: Structure of DSP Block

Finally, the programmable high speed I/O blocks are another essential element of an FPGA board. The I/O blocks are usually organized in banks and every bank can use a specific IO mechanism and protocol (e.g. Time-to-Live/TTL). By programming the I/O blocks we usually define the direction of data (input, output or input & output), or whether tri-state logic will be used [9].

1.2.3 Interconnection

Contemporary popular FPGAs implement what is often called island-style architecture. This specific architecture has logic blocks tiled in a two-dimensional array. The logic blocks form the islands and float in a sea of interconnect. With this array architecture, computations are performed spatially in the FPGA fabric [10]. Large computations are broken into 4-LUT pieces and mapped into physical logic blocks in the array. The interconnect is then configured to appropriately route the signals among the logic blocks.

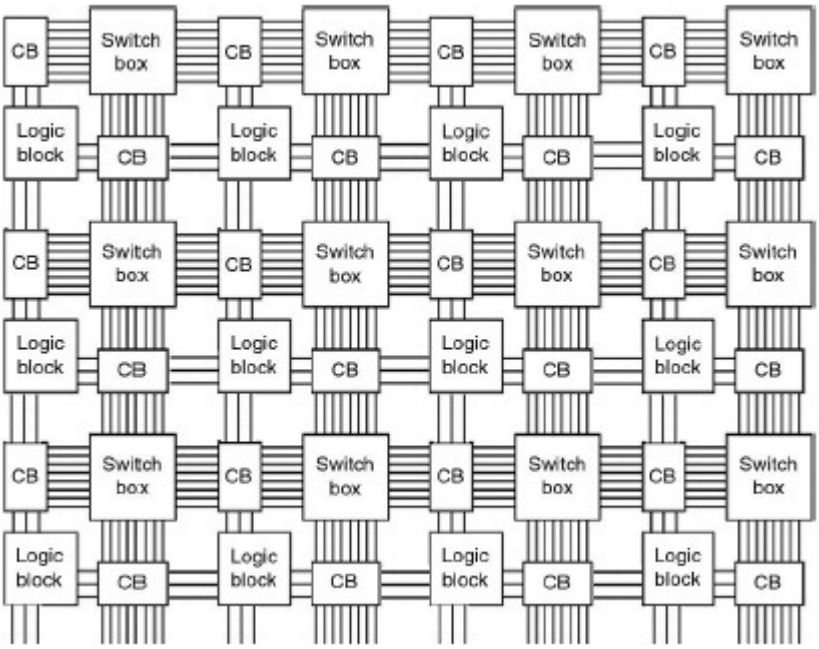


Figure 1.9: An island-style architecture with connect blocks and switch boxes [10]

In Figure 1.9 an island-style architecture is shown. A random logic block accesses nearby communication resources through a connection block. The connection block connects logic block input and output to routing resources with programmable switches and multiplexers. It allows logic block I/Os to be assigned to arbitrary horizontal and vertical tracks, increasing routing flexibility.

On the intersections of horizontal and vertical routing tracks a switch box makes its appearance. In general sense, the switch box is an array of programmable switches that allow a signal on one track to connect to another track. Depending on the design of the

switch box, a signal might turn right or left when it meets a corner or continue straight until it reaches another switch box or connection block.

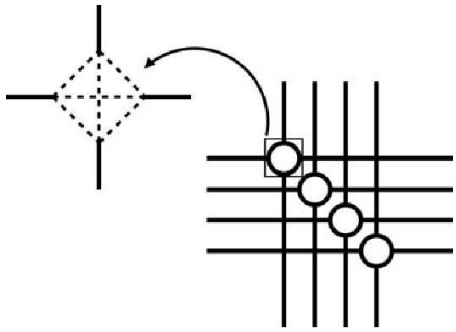


Figure 1.10: A switch box [10]

A key fact in this in this interconnect architecture is that the introduction of connect blocks and switch boxes separates the interconnect from the logic, allowing long-distance routing to be accomplished without consuming logic block resources.

1.2.4 Programming Technologies

Each configurable element in an FPGA requires 1 bit of storage to maintain a user-defined configuration. For a common LUT-based FPGA these programmable locations generally include the contents of the logic block and the connectivity of the routing fabric. The configuration of an FPGA is accomplished through programming the storage bits connected to these programmable locations according to user definitions [9]. For the look up table this translates into filling it with logic ones and zeros. For the routing fabric, programming enables and disables switches along routing tracks and channels. The most popular programming technologies for configuring an FPGA include SRAM, anti-fuse and Flash memory.

The most widely used method for storing the configuration information in commercially available FPGAs is volatile static RAM or SRAM. This specific method has gained popularity among FPGAs as it provides fast and infinite reconfiguration in a well-known technology. The drawbacks of SRAM include power consumption and data volatility [3]. Firstly, the SRAM cell size dissipates significant static power due to leakage current. Secondly, the FPGA is not configured at power-up and must be programmed using off-chip logic and storage. This could be accomplished with an additional non-volatile storage unit and a micro-controller to configure the FPGA. However, it adds to the component count and complexity of a design and prevents SRAM-based FPGAs from being a true single-chip solution [10].

Another method for FPGA programming, yet not very popular, is the usage of Flash memory for the maintenance of configuration information. The key difference between SRAM and Flash memory is that the second is non-volatile and can only be written a finite number of times. Since Flash memory is non-volatile, it is able to retain the FPGA configuration when

power turns off. In addition, the flash memory cell usually consists of fewer transistors compared to SRAM, a fact which reduces static power consumption. One major disadvantage of flash memory, as already mentioned, is that it can be written a finite number of times so it does offer an infinite reconfigurability.

A third approach to FPGA configuration is anti-fuse technology. As its name suggests, anti-fuse is a metal-based link that behaves in a way opposite to fuse. The anti-fuse link is normally open or unconnected. The programming in this case involves a laser or a high-current programmer melting the link to form an electrical connection. Although anti-fuse technology yields zero static power consumption as it does not consist of transistors, the fact that an anti-fuse link cannot be reprogrammed removes the most significant element of an FPGA which is reconfigurability and does not allow the use of anti-fuse FPGAs for prototyping of system designs [10].

1.3 CAD Tools and FPGA programming

CAD (Computer-Aided Design) tools are one of the three main factors that determine the performance of an FPGA design. The other two are the quality and efficiency of a specific FPGA architecture and the transistor-level design of the FPGA. Investigation of different architectures and implementations of an FPGA could not have been possible without the assistance of CAD tools. It might be obvious, that the implementation of a design in modern FPGAs requires thousands or millions of programmable switches and configuration bits set to proper state. Instead of that, a specific circuit can be described by the user at a higher level of abstraction by using a hardware description language, for instance, VHDL or Verilog, in general an RTL (Register-Transfer Language) or alternatively a design generated through high-level synthesis, which will be discussed in the next paragraph. Then the process of mapping a design on an FPGA is broken down to steps including Logic Synthesis, Technology Mapping, Placement, Routing and finally the generation of the bitstream file, the file according to which the FPGA is configured. Following, the steps for mapping a design on an FPGA are listed [10].

Logic Synthesis

It is the first step which includes the conversion of the circuit description, either in a hardware description language or a schematic form, into a netlist of basic gates. The next step is the conversion of the previously generated netlist to a netlist of FPGA logic blocks, such that the number of blocks is minimized while the speed of the circuit is maximized. Simplification and optimization of logic is made wherever possible.

Technology Mapping

In this step several LUTs and registers are packed into one logic block according to the limitations of the specific device on which the design is going to be implemented. The number of resources varies among different FPGA devices. The optimization goal in this phase is to pack LUTs so that the number of logic blocks and routed signals is minimized.

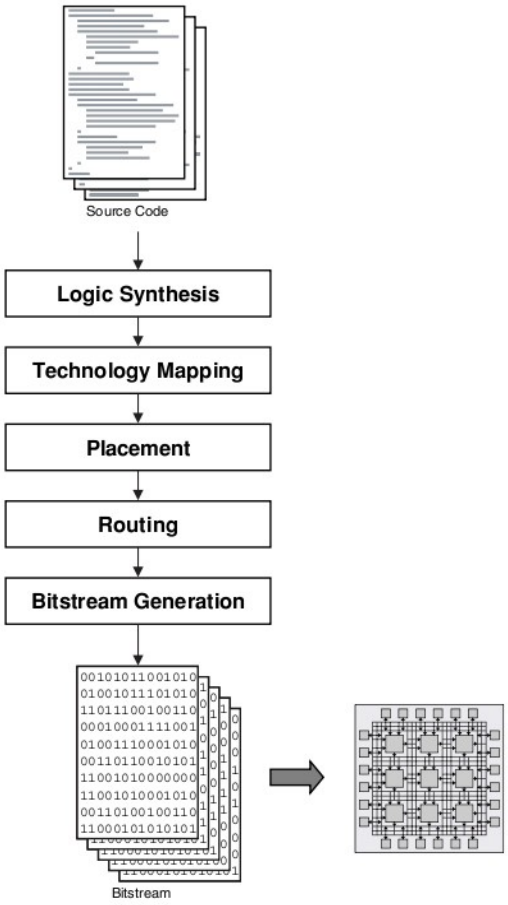


Figure 1.11: A typical FPGA mapping flow [10]

Placement

The step of placement includes the application of algorithms to determine which FPGA logic block should implement each of the logic blocks required by the circuit. The target is to place connected logic blocks together or in small distances in order to minimize the required wiring and delay, or in some cases, to balance the wiring density across the FPGA.

Routing

Once the locations for all logic blocks in a design have been chosen, a router determines which programmable switches should be turn on to connect all the logic block inputs and outputs throughout the circuit. Usually, the routing architecture is represented as a directed graph in which the nodes are the inputs and outputs of the logic blocks and potential connections are the edges of the graph. Of course, the target of this step is to interconnect the previously placed elements in the most efficient way, using short paths and fast routing connections. Since most of the delay in an FPGA design is due to programmable routing, most routers are timing-driven in the sense that an attempt to obtain good circuit speeds is made.

1.4 High-Level Synthesis

High-Level Synthesis (HLS), sometimes referred to as behavioral synthesis, is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements this behavior. The first generation of high-level synthesis tools made its appearance in the 1990s. While logic synthesis uses an RTL description of a design, high-level synthesis works at a higher level of abstraction, starting with an algorithmic description in a high-level language (HLL) such as ANSI C/C++.

Beginning with a specification of an application that is to be implemented as a custom processor, dedicated coprocessor or any other custom hardware unit, the user must provide a high-level description capture of the desired functionality using an HLL. This capture is a functional specification, sometimes referred to as untimed description, in which a function consumes all of its inputs simultaneously, performs all necessary computations without delay and provides its output data simultaneously [11]. In other words, the user is responsible for writing a function performing a desired computation as if it were to be included in a software project. At this level of abstraction variables and data types are related neither to the hardware design domain, nor to the embedded software. Thus, a realistic hardware implementation definitely requires the floating-point, integer or other data types to be converted to bit-accurate data types of specific length and acceptable computation accuracy. Then, an optimized hardware architecture should be generated.

At this point, HLS tools make their appearance, targeting to transformation of a given untimed or partially timed high-level specification into a fully timed implementation. HLS tools automatically or semi-automatically generate a custom architecture to efficiently implement the previously mentioned specification. In addition to the memory banks and communication interfaces, the generated architecture is described at the Register-Transfer Level and contains a data path and a controller as required by the given specifications and design constraints [11]. Except for the high-level description of the application, an RTL component library and specific design constraints are needed. Below, the steps from the high-level specification to the generation of RTL architecture are listed.

Compilation and Modeling

In this first step, the input description is transformed into a formal representation or model. Code optimizations, such as dead-code and false data dependency elimination, constant folding and loop transformations transpire. The formal model produced by the compilation exhibits the data and control dependencies between operations. Data dependencies are usually represented by a data flow graph (DFG) or a control and data flow graph (CDFG) which explicitly exhibit all the intrinsic parallelism of the specification. The main difference between DFGs and CDFGs is that CDFGs are more expensive in general because they take unbounded loops into account, a feature which DFGs miss.

Allocation

In this step, a definition of type and number of hardware resources needed to satisfy the design constraints transpires. The components are selected from the specific RTL

component library which is provided. At least one component for each operation in the specification model is selected. For example, if an addition is included in the specification then at least one adder will be selected from the RTL library. Depending on the HLS tool, some of the essential components might be added during later steps.

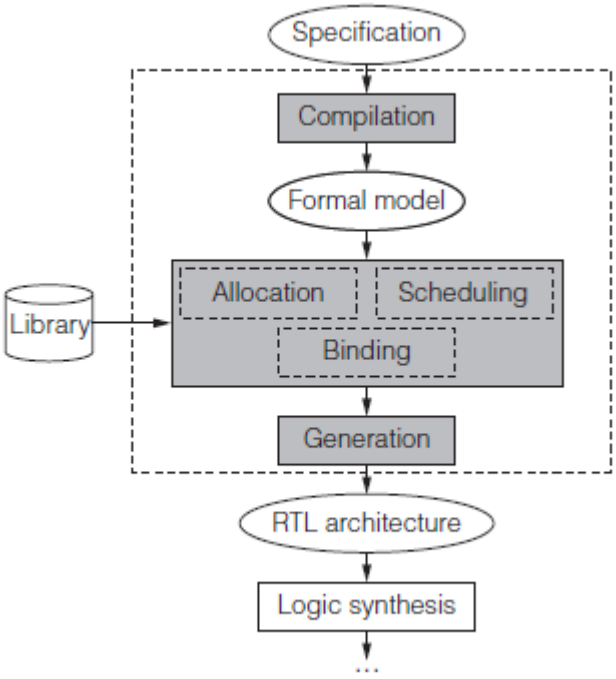


Figure 1.12: High-Level Synthesis design steps [11]

Scheduling

The step following allocation is scheduling. All operations required in the specification model must be scheduled into cycles. For each operation (e.g. +/-) variables must be read from their sources (i.e. storage or functional unit components), brought to the input of a functional unit which can perform the operation and the result must be brought to its destination storage or functional unit. Depending on the functional component to which the operation is mapped, it can be scheduled within one or several clock cycles. The operations can be chained and be scheduled to execute in parallel if there are no data dependencies between them and a sufficiency of available resources [11].

Binding

Each variable that carries values across cycles must be bound to a storage unit, while variables with non-overlapping or mutually-exclusive lifetimes can be bound to the same storage units. Additionally, every operation in the specification model must be bound to one of the functional units capable of executing it and in the case of plurality of such units the binding algorithm optimizes its selection. Finally, connectivity binding requires that each transfer from component to component be bound to a connection unit. Ideally, HLS estimates the connectivity delay and area as early as possible for better optimization.

Generation

Once decisions have been made in the preceding tasks of compilation and modeling, allocation, scheduling and binding, the goal of the RTL generation step is to apply all the design decisions made and generate an RTL model of the synthesized design. Given the generated RTL description, the steps that follow up are the ones mentioned in the previous paragraph, starting with logic synthesis and ending with the generation of the bitstream file.

1.5 Aims, Objectives and Organization of Chapters

The aim of the present diploma thesis is the exploration and evaluation of communication potentials between a processing system and custom hardware accelerators in the form of IP (Intellectual Property) cores. An IP core or IP block is a reusable unit of logic, cell or chip layout that is the intellectual property of one party. The target device of our implementations is Zynq®-7000 APSoC, and more specifically, Zedboard (Zynq Evaluation and Development Board). As already mentioned, Zynq®-7000 APSoC is composed of a Dual-Core ARM® Cortex A9 Processing System and additional Programmable Logic. For the purposes of this thesis, two different IP cores generated through Vivado HLS 2014.4 are going to be employed. The first one pertains to the field of Computer Vision, and more specifically, is an implementation of the Harris & Stephens Corner Detection Algorithm. The second resides in the Biomedical field and it is an implementation of an SVM (Support Vector Machine) classifier for arrhythmia detection. The natures of these IP cores differ not only in their corresponding applied fields but also, and most significantly for our thesis aims, in the size of their input and output data requirements. Both algorithms will be discussed further later. The rest of this thesis is organized as follows:

- Chapter 2 gives the theoretical background of the implemented algorithms along with information on the related work in Computer Vision and Bio-medicine.

- Chapter 3 focuses on more technical details of the implementation concerning the ARM Advanced Micro-controller Bus Architecture (AMBA), available interfaces and their characteristics, Direct Memory Access and the Linux UIO Driver.

- Chapter 4 presents the whole flow for ZedBoard implementations, beginning with the initial step of High-Level Synthesis, up to the development of the userspace application intended to control the hardware accelerators.

- Chapters 5 and 6 present all the implementations and corresponding results for the Harris & Stephens Corner Detector and the Support Vector Machine classifier for Arrhythmia Detection respectively.

- Finally, in Chapter 7 the conclusions of this thesis are recorded and proposals for further improvements and research are made.

This page is intentionally left blank.

Chapter 2

Theoretical Background

2.1 The Harris & Stephens Corner Detector

The role of this paragraph is to give us an overview of the field of Computer Vision and the implemented algorithm. The process of feature detection and, in particular, corner detection has, lately, gained a significant amount of interest from the scientific world. The theme of this paragraph is the study of evolution in Computer Vision and Feature Detection algorithms, beginning with the Canny Edge Detector and reaching the point where the Harris & Stephens Corner Detection algorithm was introduced.

2.1.1 Introduction to Computer Vision

Computer Vision (CV) is a term which denotes the scientific field which includes all the methods for acquiring, processing, analyzing and understanding data from the 3D world in order to generate numerical or symbolic information. Computer Vision emerged from the need to simulate human vision by electronically perceiving and understanding an image [12]. Since the analysis of the 3D world requires an interdisciplinary approach, it is presumed that the aid of scientific fields such as geometry, physics, statistics and learning theory is of vital importance.

Computer Vision algorithms have evolved rapidly in recent years, covering a wide range of applications. They play a dominant role in navigation of robots and vehicles, either ground or aerial. The applications in the automotive field might range from obstacle avoidance, autonomous robot navigation to space exploration, which is held by fully autonomous ground-based vehicles like the ESA ExoMars rover. Secondly, industrial CV applications provide vital information for the manufacturing process, such as the search for imperfections on a product or assisting robotic arms to perform pick-and-place operations in a manufacturing area. The contributions of CV in the medical field, and specifically in medical imaging, have facilitated the diagnosis for diseases or organ disorders and dysplasias by employing representations like x-ray images or CT and MRI scans for the measurement of organ dimensions, blood flow or even the structure of the brain [13].

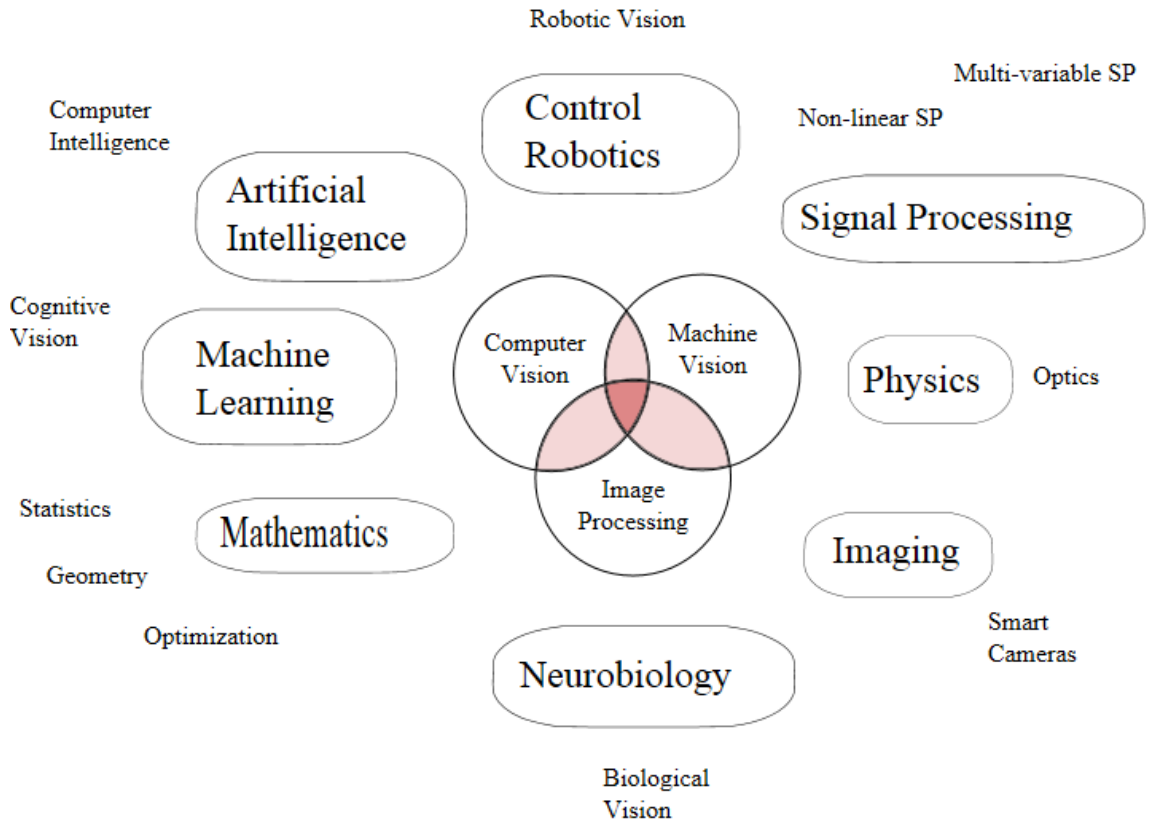


Figure 2.1 Scientific fields correlating with Computer Vision [12]

2.1.2 Feature Detection

One of the main and most significant targets of Computer Vision is the extraction of features from images in order to satisfy the requirements of a variety of systems concerning robotics, motion estimation and a number of other application kinds. In the field of Computer Vision, feature detection refers to essential methods and operations for the estimation, at every pixel of an image, of the presence or absence of a specific feature. In other words, feature detection is the process of estimating geometrical and physical properties of the surfaces of 3D world objects by using their image representations as inputs. The result of the feature detection process is a subset of the initial image which might contain points, continuous lines or connected regions depending on the kind of features one might need to extract. Occasionally, the definition of a feature type might be hazy, however, it is important to understand that any image pixel or region might be considered a feature if it holds a certain property that increases its interest in comparison with other image pixels or regions.

Lately, a variety of feature detection algorithms have been developed depending on the desired feature extraction. Following, in order to clarify what a feature might refer to let us enumerate those principally used in Computer Vision applications and systems:

- **Edges :** The term “edge” is used to describe the boundary between two or more different regions or surfaces of an image. Obviously, there is no predefined shape for an edge as it is the border between surfaces of any shape. Most edge detection algorithms rely on the fact that edges consist of pixels with a high gradient magnitude [14]. A well-known, yet not unique, edge detection algorithm is the *Canny Edge Detector* proposed by John F. Canny in 1983.

- **Corners/Interest Points :** The term “corner” describes the point of intersection between two or more edges. Initially the corner detection algorithms firstly detected the edges and afterwards the corners of an image by determination of the points with strong changes in direction. Lately, the corner detection algorithms search for high values of curvature in the image gradient. It was claimed that many of those algorithms occasionally misinterpreted non-corner points as corners due to contrast [14]. For example, a white dot on a black canvas would be characterized as a corner. For such points the term “interest points” is used.

- **Blobs/Regions of Interest or Interest points :** By the term “blob” an image region that differs in properties such as brightness or colour, compared to surrounding regions is described. Informally a “blob” is a region of an image in which some properties are constant or approximately constant [14]. In other words, a blob is a collection of points that, based on some criteria, are similar to each other. Commonly used blob detectors are the *Laplacian of Gaussian (LoG)*, *the Difference of Gaussians (DoG)* and the *Determinant of Hessian (DoH)*. As one might think, blob detection is a significant task especially in applications concerning Image Segmentation.

- **Ridges :** The “ridges” or “ridge set” of a smooth function of two variables are a set of curves whose points are local maximum points of the function in at least one dimension. In other words, a ridge could be thought as a one-dimensional curve that represents an axis of symmetry. In addition its width depends on the local ridge point. In general, the calculation of ridge points is much more computationally intense than the detection of edges, corners or blobs [13].

This diploma thesis focuses on corner detection which is commonly used in tasks like Motion Detection, Image Segmentation, Video Tracking, 3D Modeling and Object Recognition.

2.1.3 The Edge Tracking Problem

Edge detectors of some kind, particularly step edge detectors have been an essential part of many computer vision systems. The edge detection process serves as a simplification to the analysis of images by drastically reducing the amount of data to be processed, preserving useful structural information about about object boundaries inside an image [15].

One of the first attempts in edge detection was proposed by John F. Canny in 1983. The Canny operator was designed to be an optimal edge detector according to certain criteria:

- **Good Detection.** The probabilities of failing to mark real edge points and falsely marking non-edge points should be low. Both probabilities are monotonically decreasing functions of the output Signal-to-Noise Ratio (SNR) and so the first criterion is fulfilled if SNR is maximized [15].

- **Good Localization.** The points marked as edge points should be as close as possible to the center of the real edge.

- **Only one response to a single edge.** This is implicitly captured in the first criterion since when there are two responses to an edge, one of them must be considered false.

The mathematical background of the Canny operator will not be discussed in this diploma thesis. The role of this paragraph is the statement of the criteria that John F. Canny relied on during the design of the Canny operator as they are the criteria that were used, slightly different in some cases, for the design of later feature detection algorithms.

2.1.4 The Moravec Corner Detector

One of the first successful attempts in corner detection was Moravec's corner detector. It operates by considering a local window in the image and determining the average changes of image intensity that occur from shifting the window by a small amount in various directions [16]. In other words, the algorithm checks the similarity between a centered pixel with other local pixels. For this purpose, the sum of squared differences between the two sections is computed. There are three cases that need to be examined:

- **The windowed image patch is flat.** In this case all window shifts will result in small change, or in a low value of the sum of squared differences as the windowed image patch is approximately constant in intensity.

- **The window includes an edge.** In this case a shift in a parallel to the edge direction will result in a small change, however, a shift perpendicular to the edge will result in a large change, or a high value of the sum of squared differences.

- **The window includes a corner or an isolated point.** In this case shifts to any direction will result in large changes. Thus, a corner or an interest point is detected when the minimum change produced by any of the shifts is large.

Thereafter, we give a mathematical specification of the above statements. Denoting the image intensities by I, the change E produced by a shift (x, y) is given by

$$E(x,y)=\sum_{u,v} w(u,v)\left|I(x+u,y+v)-I(u,v)\right|^2$$

where w specifies the image window, which is unity within a specific rectangular region, and zero elsewhere. The directions (x, y) on which we compute the shifted intensity are {(1, 0), (1, 1), (0, 1), (-1, 1)}. Moravec's corner detector searches for local maxima in min{E}.

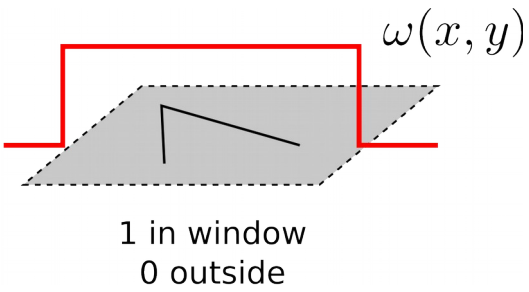


Figure 2.2 Binary Window Function

By consideration of the mathematical formula on which Moravec's corner detector depends, it is concluded that the specific detector suffers from a number of problems. Firstly, only a discrete set of shifts at every 45 degrees is considered. Secondly, the binary and rectangular window results in a noisy response [16]. Finally, only the minimum value of E is taken into account. The attempt to solve the above mentioned problems and the desire for a better performance in corner detection concluded in the Harris & Stephens corner detector.

2.1.5 The Harris & Stephens / Plessey / Shi–Tomasi Corner Detection Algorithm

Considering the drawbacks of Moravec's corner detector, Chris Harris and Mike Stephens proposed improvements by taking the differential value of a corner into account, regarding the direction directly and avoiding the usage of shifted regions. They applied corrective measures to overcome the above mentioned issues of Moravec's detector and defined the result as an “auto-correlation detector” [16].

One of the problems of Moravec's operator is that it generates an anisotropic response because only a discrete set of shifts at every 45 degrees is considered. The Harris & Stephens algorithm covers all possible small shifts by performing an analytic, Taylor series expansion in order to compute an approximation of I(x+u, y+v). Denoting I_x and I_y as the partial derivatives of the intensity of an image we write:

$$I(x+u, y+v) \approx I(u,v) + x I_x(u,v) + y I_y(u,v)$$

Thus, the expression for the computation of the sum of squared differences E, given in the previous paragraph becomes:

$$E(x,y)=\sum_{u,v} w(u,v)(I(u,v)+x I_x(u,v)+y I_y(u,v))^2 \quad \text{or} \quad E(x,y)=\begin{bmatrix} x & y \end{bmatrix} A \begin{bmatrix} x \\ y \end{bmatrix}$$

where

$$A = \sum_{u,v} w(u,v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix}$$

is the structure tensor.

Another significant improvement in comparison with Moravec's corner detector is that a smooth window function is employed, guaranteeing a less noisy response, a feature missing from Moravec's corner detector due to the choice of a binary window function [13],[16]. Instead, a Gaussian window can be used:

$$w_{u,v} = e^{-\frac{u^2+v^2}{2\sigma^2}}$$

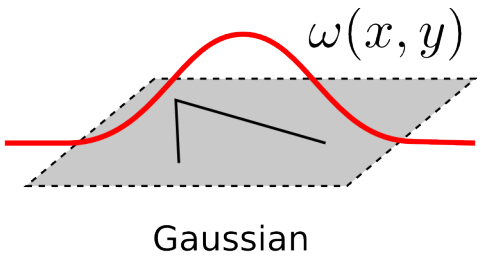


Figure 2.3 *Gaussian Window Function*

Finally, in Harris algorithm a corner is considered to have a large variation of the sum of squared differences in all directions of the vector (x, y) [13]. In mathematical form, this statement can be expressed in terms of the eigenvalues of matrix A. If an interest point is examined, then matrix A should have two eigenvalues with high magnitude. Considering the magnitudes of the eigenvalues, the following cases are determined:

- If $\lambda_1 \approx 0,\ \lambda_2 \approx 0$ then this point is of no interest.
- If $\lambda_1 \approx 0$ and λ_2 has a high positive value then an edge has been detected.
- If both λ_1 and λ_2 have high positive values then a corner has been detected.

The computation of eigenvalues bears a heavy workload, hence, Harris and Stephens proposed an alternative function which is

$$M_c = \lambda_1 \lambda_2 - \kappa(\lambda_1 + \lambda_2)^2 = \det(A) - \kappa \cdot \mathrm{trace}^2(A)$$

considering that $det(M) = \lambda_1 \lambda_2$ and $trace(M) = \lambda_1 + \lambda_2$. The factor κ is a chosen parameter whose value depends on the desired sensitivity [16].

2.1.6 Related Work

The high complexities of Computer Vision algorithms combined with the fact that these algorithms are mainly fed with images and videos, in other words, with large amounts of data, lead to greater demands of computational power, followed by greater power consumption and memory demands. General purpose CPUs are appropriate for low complexity applications, while GPUs perform a lot better. However, it is common with CV applications to demand non-linear optimizations for the sake of accuracy. The consequence is a computational load which might reach several millions of operations. Hence, another approach to design of CV applications should is recently considered.

The latest trend for a CV application is to be HW accelerated with an ASIC or an FPGA device. While ASICs are capable of meeting high performance expectations the high NRE costs, the long time-to-market and the lack of reconfigurability options lead the way to FPGA dominance. Recent improvements in FPGA technology manage to reach comparable to ASIC performances. The innate concurrent FPGA behavior proves as a great advantage for implementing CV applications. For instance, the convolution is a very common operation in CV and Image Processing systems, yet, it is computationally intensive and might require several millions of multiplications and additions. A convolution would be quite time consuming in a standard processor, however, it could be implemented simultaneously in an FPGA. On the other hand, FPGAs might introduce a major drawback when it comes to implementing CV applications. Floating-point operations consume a large amount of FPGA resources. This situation is worse when a floating-point operation needs to be performed repeatedly. Luckily, Xilinx FPGAs include DSP blocks embedded in the FPGA fabric which allows an application to perform operations like multiplications and additions more quickly, partially solving the floating-point operation issue.

In our work, a corner detection algorithm is accelerated and targeted to an FPGA device. The hardware accelerator is firstly implemented and generated through High-Level Synthesis. In our study case the accelerator is data intensive in both execution and communication time as it requires images as input data. Hence, not only should we add an interface for the accelerator to communicate with the processing system which feeds the input data, but also an effective way of communication should be considered by exploration of the available potential interfaces and interconnections. When we refer to the potential interfaces the available resources of the FPGA target device should be considered as some of them need to instantiate input ports, hence leading to even a 100% increase in resource utilization.

2.2 Support Vector Machine Classifier for Arrhythmia Detection

Electrocardiogram analysis has been established as a key factor for analyzing and assessing the health status of a person. The ECG Analysis flow is complex, relies on machine learning algorithms such as Support Vector Machine Classifiers and in an effort to be executed in real-time hardware acceleration is required [17]. In this paragraph an overview of the ECG analysis flow and Support Vector Machine classifiers is given.

2.2.1 Electrocardiogram Analysis Flow

Electrocardiography is an important tool in diagnosing the condition of the heart. The electrocardiogram (ECG) is the record of variation of bioelectric voltage with respect to time as the human heart beats. The state of cardiac health is generally reflected in the shape of ECG waveform and heart rate [18]. Due to its inherent relation to heart physiology the ECG is one of the most fundamental and crucial biological signals for monitoring and assessing the health status of a person [17]. Before proceeding to ECG Analysis flow description, we consider essential to give background information about the heart.

The heart is a four-chambered organ consisting of right and left valves. The upper two chambers, or in other words, the left and right atria, are entry-points into the heart, while the lower two chambers, or left and right ventricles, are responsible for contractions that send the blood through the circulation [18]. The role of the right ventricle is to pump deoxygenated blood to the lungs through the pulmonary trunk and pulmonary arteries, while the role of the left ventricle is to pump newly oxygenated blood to the body through the aorta.

The cardiac cycle refers to complete heartbeat from its generation to the beginning of the next beat. The first stage, defined as “diastole”, is when the semilunar valves (the pulmonary valve and the aortic valve) close, the atrioventricular (AV) valves (the mitral valve and tricuspid valve) open, and the whole heart is relaxed. The scond stage, defined as “atrial systole”, is when the atrium contracts, and blood flows from atrium to the ventricle. The third stage, defined as “isovolumic contraction” is when the ventricles begin to contract, the AV and semilunar valves close, and there is no change in volume. The fourth stage, "ventricular ejection", is when the ventricles are contracting and emptying, and the semilunar valves are open. Finally, the fifth stage, “isovolumic relaxation time”, is when pressure decreases, no blood enters the ventricles, the ventricles stop contracting and begin to relax, and the semilunar valves close due to the pressure of blood in the aorta [19].

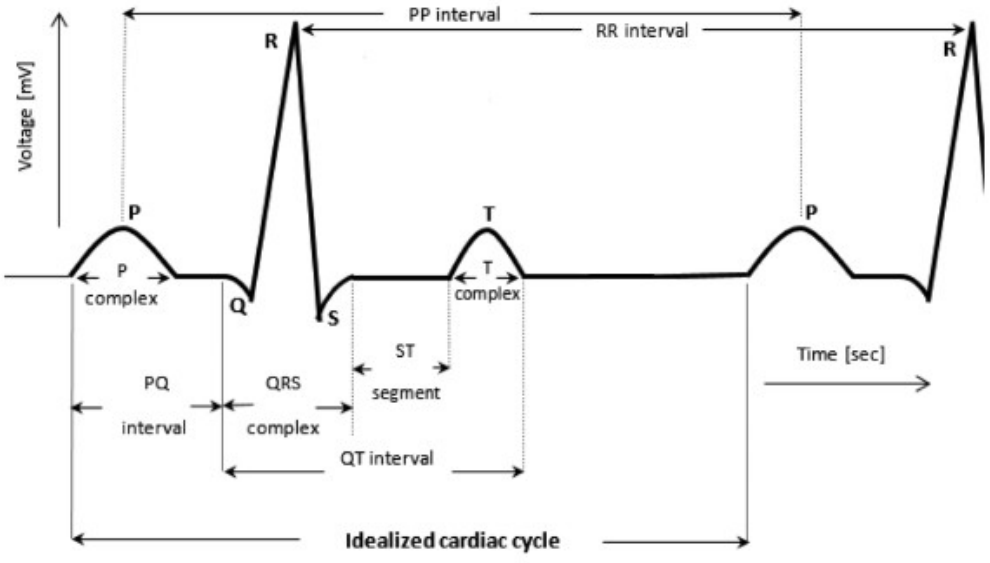


Figure 2.4: ECG Waveform Typical Morphology [20]

The cardiac cycle, which is described above is coordinated by a series of electrical impulses that are produced by specialized pacemaker cells. A typical ECG tracing is repeating cycle of three electrical entities: a P wave, a QRS complex that consists of three peaks, Q, R, and S, and finally a T wave. These waves are created by voltage fluctuations that depict the electrical activity of the heart and thus represent the cardiac cycle [18]. The phases of the cardiac cycle that each of the above signals are generated will not be discussed.

All the waves on the ECG and the intervals between them have a predictable duration, a range of acceptable amplitudes (voltages), and a typical morphology. This morphology is depicted in Figure 2.4. Any deviation from the normal tracing is potentially pathological and therefore of clinical significance. Arrhythmia is considered as one of the most commonly encountered heart malfunctions. Cardiac arrhythmia, also referred to as dysrhythmia, or irregular heartbeat, is a group of conditions in which the heartbeat is irregular, too fast, or too slow. Some arrhythmias do not cause symptoms, hence are not associated with increased mortality but this is not the typical case. Medical assessment of the abnormality using an electrocardiogram is a way to diagnose and assess the risk of any given arrhythmia [18].

Taking into account the critical condition of a person suffering from arrhythmia episodes, the field of depicting signs of arrhythmia in an ECG signal has been highly investigated. Arrhythmia incidents might occur at random in time scale because the ECG is not a stationary signal. Thus, the disease symptoms may not show up all the time, but manifest at certain irregular intervals during the day. Therefore, for an effective diagnosis, the study of the ECG pattern and heart rate variability signal may have to be carried out over several hours. This translates into an enormous data set that needs to be processed in order to reach a diagnosis. As a result, machine learning techniques are ideal for solving the diagnosis problem. The data set is used as a training set, and by the time the training is completed the system is ready to deliver a diagnosis. The training set could be formed from a number of databases of ECG signals that are available. Our choice was a rather commonly used database, the MIT-BIH Arrhythmia Database, which is a combined effort of MIT and Beth Israel Deaconess Medical Center. The heart beats included in this database have been verified by cardiologists, so this data base forms an ideal starting point for creating a training data set for the detection problem.

The process of acquiring and processing an ECG signal in order to extract the individual beats and their corresponding features is composed of various stages with distinct characteristics and requirements. It consists of three main stages: a preprocessing stage (noise removal), a processing stage (R peak detection, feature extraction), and a classification stage. A simplified overview of this processing flow can be seen in Figure 2.5. Our point of interest is the final step of diagnosis classification, or detecting whether the heart beat exhibits arrhythmia signs or not. This is performed using a classification algorithm, which detects the pattern of problematic beat. The classifier has been trained on the data set that includes the feature vectors of the isolated beats. Given a new feature vector the classifier can detect whether that corresponding beat displays signs of arrhythmia.

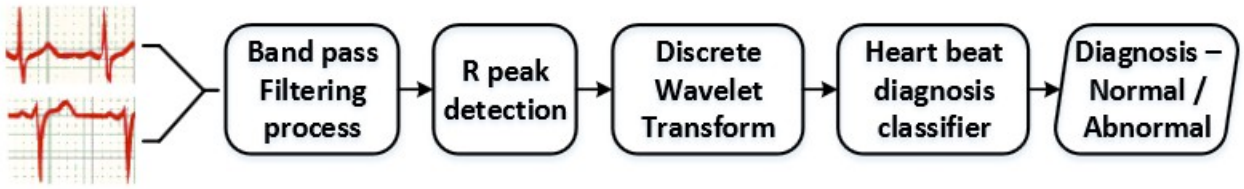


Figure 2.5: ECG Analysis Flow

2.2.2 SVM Classifier

In machine learning, Support Vector Machines (SVMs) are supervised learning models that are used for data-driver modeling and classification. They are suitable for binary classification problems. The classification process requires that the data is separated into training and testing set. Each of the instances in the training set has the form of a feature vector consisting of the attributes that are being observed and a label indicating the class of each instance. The instances in the training set consist solely of the attributes. The goal of the SVM classification technique, is to train a model that can predict the class of an instance of the training set given only the attributes of the corresponding instance [21].

This goal is accomplished through the ability of the SVM to find a hyperplane that divides samples into two classes with the widest margin between them. A mapping function is used to project each feature vector of the training set to a feature space of higher dimension where the classification of data will be easier. The SVM is used to find the optimal hyperplane for data classification according to their attributes. This optimal hyperplane maximizes the distance between itself and the feature vectors that belong to each class and are closest to the hyperplane. These feature vectors represent the decision boundary between the classes and are called support vectors. A new feature vector is classified by its distance from the support vector. The function used for computing the distance between a new feature vector and a support vector by firstly projecting them to a higher dimensional feature space is called kernel function. The hyperplane decision function for classifying a test feature vector x is of the following form:

$$Class = sgn\left(\sum_{i=1}^{N_{sv}} \left(y_i * a_i * K\left(x, \text{sup\_vector}_i\right)\right) - b\right)$$

where K is the kernel function, x is the feature vector, sup_vector_i is the i-th support vector and y_i , a_i are values related to it and result from the classifier training process. Coefficient b is a bias value, also a result of the training process and is constant for all support vectors. The kernel function is of great significance for the accurate prediction of testing data. Depending on the characteristics of a data set , different kernel functions are able to provide the desired classification accuracy.

In this work, we turn our attention to radial basis kernel function (RBF) since the complex correlations between the attributes of our feature vector and the physiological states of interest typically require the flexibility afforded by non-linear kernel functions. The

advantages of the RBF kernel over the other non-linear kernels is that RBF has fewer parameters and fewer numerical difficulties [21]. Following are the equations in case of the RBF kernel. The second is the final decision function that is implemented in HW.

$$K(x, \text{sup\_vector}_i) = \exp(-\gamma \| x - \text{sup\_vector}_i \|^2)$$

$$Class = sgn\left(\sum_{i=1}^{N_{sv}} (y_i * a_i * \exp(-\gamma \| x - \text{sup\_vector}_i \|^2)) - b\right)$$

2.2.3 Related Work

Most biomedical devices used for monitoring chronic patients and detection of abnormalities in biomedical signals aim to provide accurate results in real-time. This comes with processing an enormous amount of signal data with extremely complex correlations. On this ground, proposed methodologies include an algorithmic-driven architectural design space exploration of domain-specific medical-sensor processors. Data-driven modeling techniques are emerging as a powerful approach for overcoming the mentioned challenges. Additionally, most biomedical devices are wearable, hence application-specific architectures for low energy should be considered.

In our work a co-processor is build through High-Level Synthesis Design tools and is intended for arrhythmia detection study case. It is thus optimized for this case only. For that reason the application is fixed concerning the implementation of the kernel function. The design space exploration for this particular study case has already been made as part of the work of a former diploma thesis. The Pareto Design space has been granted to us and we had to choose different versions of the HW. Finally, three different versions were chosen and integrated in the target device in IP form. Of course, before integration different communication interfaces were added during the High-Level Synthesis step. Finally, implementations with one or more classifier IPs were made for those versions whose resource utilization allowed it.

This page is intentionally left blank.

Chapter 3

Technical Background

3.1 The Advanced Microcontroller Bus Architecture (AMBA)

The ARM® Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in SoC designs. It facilitates the development of multi-processor designs with large numbers of controllers and peripherals [22]. Since its inception, the scope of AMBA has, despite its name, gone fay beyond microcontroller buses. Today, it is widely used on a range of ASIC and SoC parts [23]. AMBA was introduced by ARM in 1996. Since then, AMBA protocols have become the de facto standard for 32-bit embedded processors because they are well documented and can be used without royalties.

The design principles of AMBA originate from the fact that an important aspect of a SoC is not only which components or blocks it utilizes but also the interconnection of these components. Hence, it is a clear solution for the blocks to interface with each other. The objectives of AMBA vary from facilitating right-first-time development of embedded microcontroller products with one or more CPUs, GPUs or signal processors to technology independence by allowing the re-use of IP cores, peripheral and system macrocells across diverse IC processes. Moreover, another objective is to encourage modular system design to improve processor independence and the development of re-usable peripheral and system IP libraries. Finally, the minimization of silicon infrastructure while supporting high performance and low power on-chip communication is of great importance. The AMBA 4 specifications define the following buses or interfaces [22]:

- AXI Coherency Extensions (ACE & ACE-Lite)
- Advanced eXtensible Interface (AXI4, AXI4-Lite & AXI4-Stream v1.0)
- Advanced Trace Bus (ATB v1.1)
- Advanced Peripheral Bus (APB4)

In this diploma thesis we focus on the characteristics and use of the Advanced eXtensible Interface (AXI) protocol.

3.2 The Advanced eXtensible Interface (AXI) Protocol

The AMBA AXI protocol supports high performance and frequency system designs. To begin with it is suitable for high-bandwidth and low latency designs providing high-frequency operation without using complex bridges. Secondly, it meets the interface requirements for a wide range of components. Additionally, it is suitable for memory controllers with high initial access latency. It provides flexibility in the implementation of interconnect architectures. Finally, it is backward-compatible with existing AHB and APB interfaces [24]. The key features of the AXI protocol are:

- Separate address/control and data phases
- Support for unaligned data transfers using data strobes
- Uses burst-based transactions with only the start address issued
- Separate read and write data channels that can provide low-cost Direct Memory Access (DMA)
- Support for issuing multiple outstanding addresses
- Support for out-of-order transaction completion
- Permits easy addition of register stages to provide timing closure

The above key features along with the fact that the AXI protocol includes optional extensions that cover signaling for low-power operation are what make AXI our first choice when it came to implementing the interconnection between the PS-side and the PL-side of the ZedBoard. We should now proceed to a further explanation of the architecture and operating principles of the AXI protocol. To begin with, it should be mentioned that the AXI protocol is burst-based and defines five independent transaction channels:

- read address
- read data
- write address
- write data
- write response

An address channel carries control information that describes the nature of data to be transferred. The data is transferred between the master and the slave using either a write data channel to transfer data from the master to the slave or a read data channel to transfer data from the slave to the master. It should be mentioned that in a write transaction, the slave uses write response channel to signal the completion of the transfer to the master. The AXI protocol permits address information to be issued before the actual data transfer, supports multiple outstanding transactions and out-of-order completion of transactions [24].

Each of the independent channels consists of a set of information signals and VALID and READY signals that provide a two-way handshake mechanism. The information source uses the VALID signal to show when valid address, data or control information is available on the channel. The destination uses the READY signal to show when it can accept the information. Both the read data channel and the write data channel also include a LAST signal to indicate the transfer of the final data item in a transaction. The read data channel carries both the read information and the read response from the slave to the master and includes the data

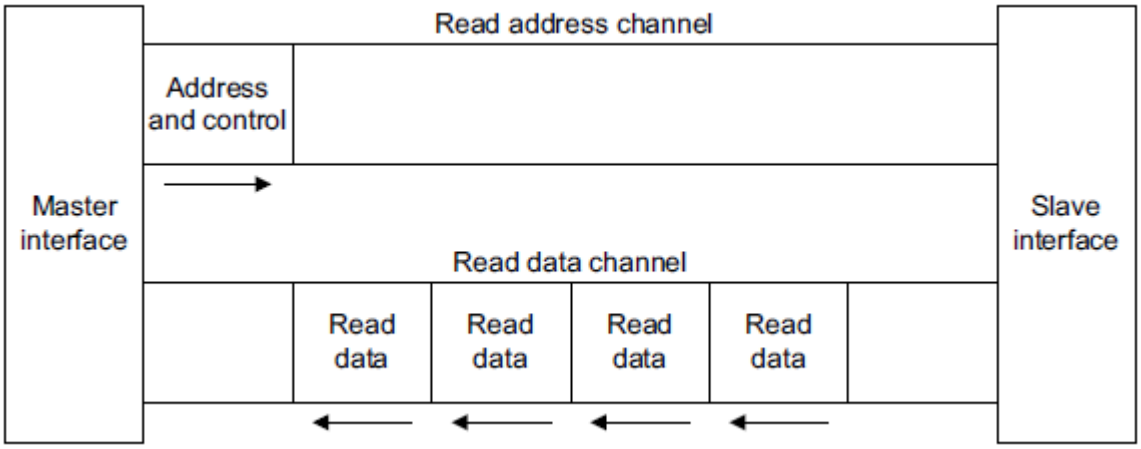


Figure 3.1: AXI Channel Architecture of Reads [24]

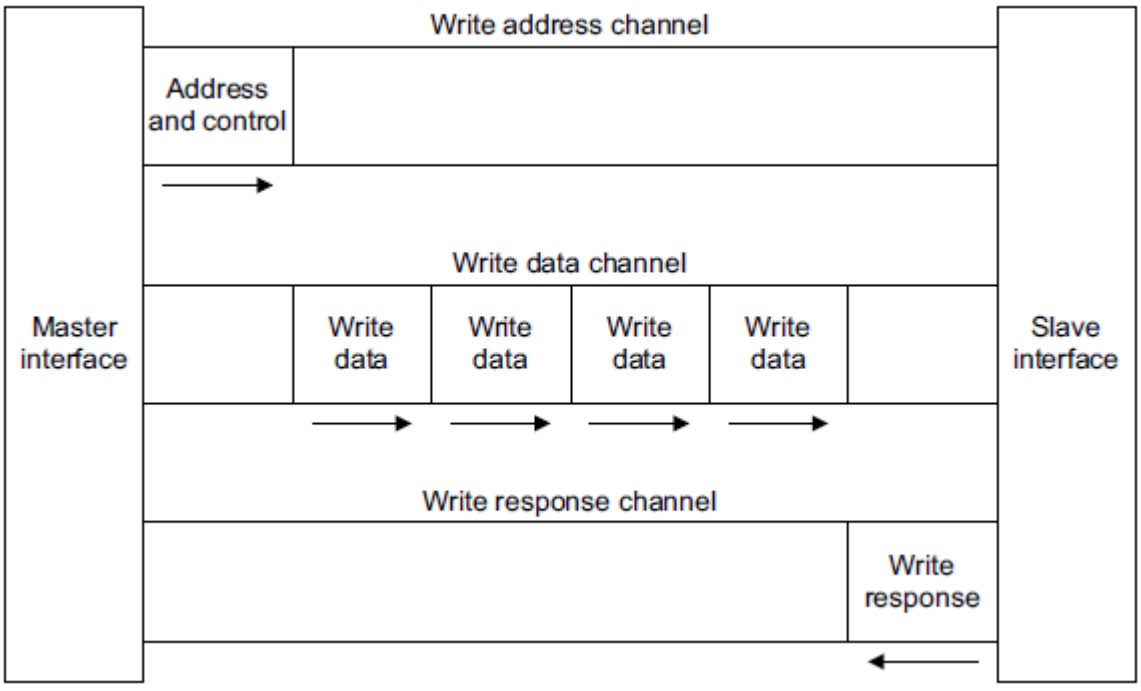


Figure 3.2: AXI Channel Architecture of Writes [24]

bus, that can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide, and a read response signal indicating the completion status of the read transaction. On the other hand, the write data channel carries the data from the master to the slave and includes a data bus of the same possible widths as the read channel's data bus, and a byte lane strobe signal for every data byte, indicating which bytes of the data are valid. A final notice is that a typical system consists of a number of master and slave devices connected together through some form of interconnect.

The AXI protocol provides a single interface definition for the interfaces between a master and the interconnect, between the slave and the interconnect and finally between a master and a slave. We now proceed to a further description and explanation of the AXI4-Lite and AXI4-Stream interface.

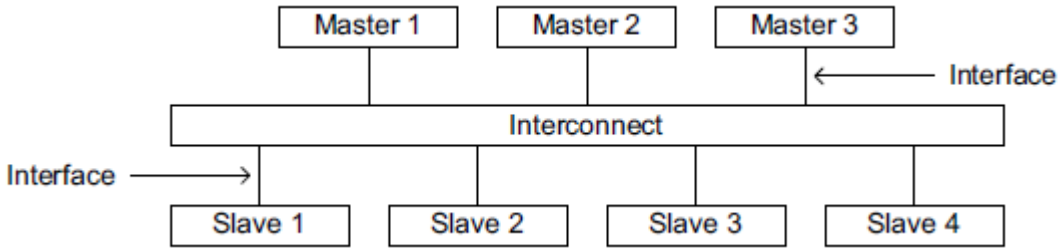


Figure 3.3: Interface and Interconnect [24]

3.2.1 The AXI4-Lite Interface

AXI4-Lite is an interface which is suitable for simple control register-style interfaces that do not require the full functionality of the AXI4 protocol. Of course, the potential transactions are compliant with general principles of the AXI protocol, however a subset of the signals offered by the AXI protocol are supported as AXI4-Lite refers to simpler transactions [24].

Lets now enumerate the key functionalities of AXI4-Lite interface. Firstly, all transactions are of burst length 1. This means that the maximal packet size that is transferred at once, can be either 32-bit or 64-bit depending on the data bus width. Secondly, all data accesses use the full width of the data bus. It should be mentioned that AXI4-Lite supports a data bus width of 32-bit of 64-bit. Thirdly, all accesses are non-modifiable and non-bufferable, and, finally exclusive accesses are not supported [24]. In Table 3.1 we might observe the signals that are supported by the AXI4-Lite interface for all kind of transactions. In this table we may notice some signals that were already mentioned before, like the VALID and READY signals. Other essential signals include the ADDR and DATA signals, which obviously refer to the address that we wish to read from or write to and the actual data transfer that is to be made. The PROT signal refers to protection type. The signal indicates the privilege and security level of the transaction and whether it is a data access or instruction access. The RESP signal refers to the response of either the write response or read data channel. Finally the STRB signal refers to the write strobes and indicates which byte lanes hold valid data. There is one write strobe bit for each byte of the write data bus.

Global	Write Address channel	Write Data Channel	Write Response Channel	Read Address Channel	Read Data Channel
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	-	ARPROT	RRESP

Table 3.1: AXI4-Lite Interface Signals [24]

The most important piece of information that should be kept in mind is that AXI4-Lite has a fixed data bus width and all transactions are the same width as the data bus which might be either 32-bit or 64-bit wide. On the one hand, this is a fact that combined with the burst length of 1, might limit bandwidth. On the other hand, if a data transfer requires less than 32 bits, the utilization of the data bus will be the same as if the data transfer requires 32 bits. Thus, power and data bus consumptions will be the same, independently of the actual needed data bus width. At this point, a basic explanation of the AXI4-Lite interface has been made and we now proceed to an overview of the AXI4-Stream interface.

3.2.2 The AXI4-Stream Interface

The AXI4-Stream Interface is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a single master that, that generates data, to a single slave, that receives data. The protocol can also be used when connecting larger numbers of master and slave components. The protocol supports multiple data streams using the same set of shared wires, allowing a generic interconnect to be constructed that can perform upsizing, downsizing and routing operations. The AXI4-Stream interface also supports a wide variety of different stream types [25].

Types of streams include byte streams, continuous aligned streams, continuous unaligned streams and sparse streams. For the purposes of this paragraph, the types of streams will not be discussed further. Additionally, AXI4-Stream interface applies a distinction of the data bytes that a data stream might consist of. A byte might be data byte, position byte or null byte. Data byte refers to a byte of data that contains valid information that is transmitted between the source and destination. The term position byte refers to a byte that indicates the relative positions of data bytes within the stream and performs as a placeholder that does not contain any relevant data values that are transmitted between the source and destination. Finally, a null byte is a byte that does not contain any data information or any information about the relative position of data bytes within a stream. In Table 3.2 a list of signals used in transactions with devices disposing AXI4-Stream interfaces is given.

Signal	Source	Description
ACLK	Clock Source	The global clock signal. All signals are sampled on the rising edge of ACLK.
ARESETn	Reset Source	The global reset signal. It is active-LOW
TVALID	Master	Indicates that the master is driving a valid master.
TREADY	Slave	Indicates the slave can accept a transfer in the current cycle.
TDATA [(8n-1):0]	Master	It is the primary payload that is used to transfer the data. The width of the data payload is an integer number of bytes.
TSTRB [(n-1):0]	Master	It is the byte qualifier that indicates whether the content of the associated type of TDATA is processed as a data byte of position byte.
TKEEP [(n-1):0]	Master	It is a byte qualifier that indicates whether the content of the associated byte of TDATA is processed as part of the data stream. Associated Bytes that have the TKEEP byte qualifier deasserted are null bytes and can be removed from the stream.
TLAST	Master	It indicates the boundary of the packet.
TID [(i-1):0]	Master	Data stream identifier that indicates different streams of data.
TDEST [(d-1):0]	Master	It provides routing information about the data stream.
TUSER[(u-1):0]	Master	User-defined sideband information that can be transmitted along the data stream.

Table 3.2: AXI4-Stream Interface Signals list [25]

3.3 The Linux UIO Driver

Userspace I/O (UIO) drivers are designed to handle devices like FPGAs found on embedded boards and are frequently used in embedded systems. The Linux UIO driver was introduced in Linux 2.6.23 and is suitable for devices that cannot fit into other kernel subsystems. It allows the programmer to develop a device driver almost entirely in userspace, using all standard application development tools and libraries. This is a feature that simplifies development, maintenance and distribution of device drivers [26].

Non-standard devices, for instance accelerators implemented on an FPGA, are commonly treated as character devices. A simple device might be easily handled by the `read()` and `write()` system calls, however, this is not the typical case. Such devices are usually more complex and the additional necessary functionalities are commonly implemented using the `ioctl()` system call. An important note for a conventional driver is that it is obliged to use many internal kernel functions and macros. For several reasons, kernel developers refuse to keep the internal API stable, causing a driver which might perfectly work with the current kernel to neither work nor compile anymore in a small amount of time. Although drivers designed for widely used devices will be updated by the Linux community, a non-standard device will require the programmer to maintain it throughout the whole lifetime of the product [26]. Therefore, to address this situation, the UIO framework was introduced.

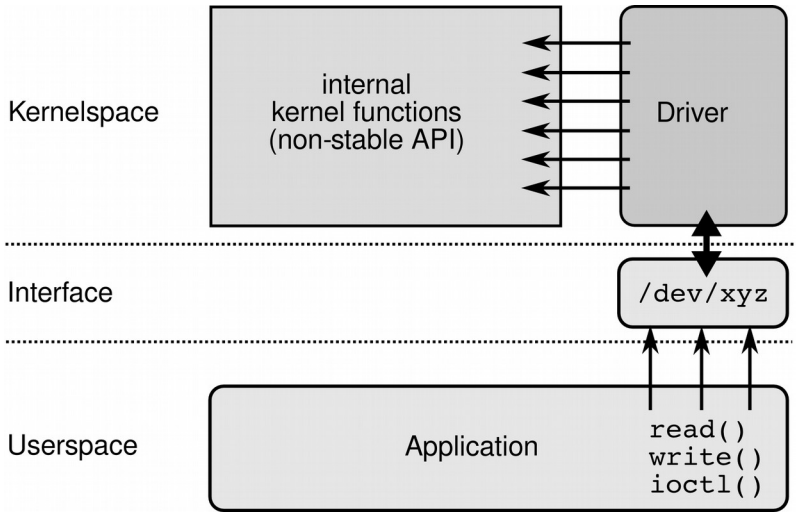


Figure 3.4: A Conventional Device Driver [26]

It is well-known that a device driver basically has two tasks to accomplish. The first one is to access the device memory. The second and more difficult task is to handle interrupts generated by the device. The first demand is easily fulfilled since Linux is capable of mapping physical device memory to an address accessible from userspace. This had already been possible by using `/dev/mem` and it is a fact that a lot of people used it for similar purposes leading to occurrences of security leaks and stability issues. The UIO framework prevents userspace from mapping memory that does not belong to the device, thus coping with the previously mentioned issues. Moreover, the framework itself offers an `mmap()` implementation able to perform the previous task for physical, logic and virtual memories.

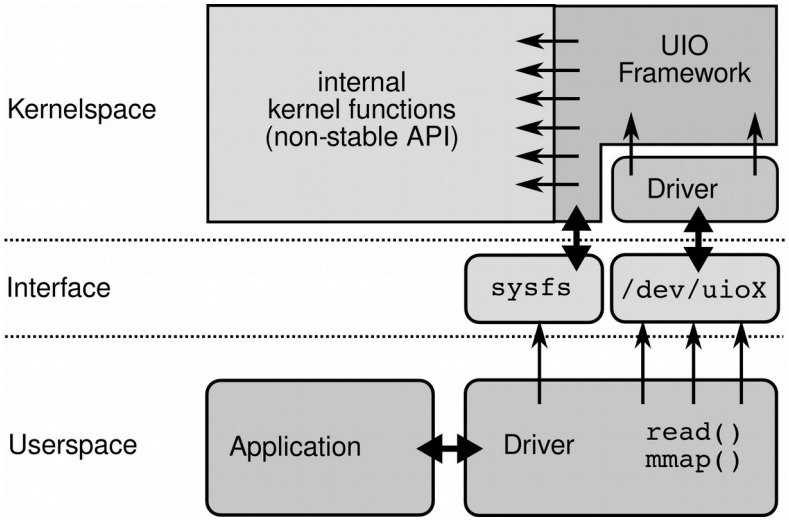


Figure 3.5: A UIO driver paradigm [26]

As mentioned above, a more difficult task concerning a device driver is interrupt handling. Interrupts need to be handled in kernel space. Current interrupts are level-triggered and the machine might hang if an interrupt is still active at the end of the interrupt service routine (ISR). Hence, the UIO framework will need to include a small kernel module containing a minimal ISR that only needs to acknowledge or disable the interrupt. Additionally, if the userspace part of the driver wills to wait for an interrupt, it simply does a blocking `read()` from `/dev/uioX`. The call returns immediately as soon as an interrupt occurs [26]. The following figure shows a small kernel driver that calls only a few kernel functions. The majority of the essential functionalities is handled in a generic way by the UIO framework, effectively protecting the author of a driver from the dirty sides of the kernel. A quick reference guide for development of userspace applications using the UIO driver is presented in Chapter 4.

As a final comment on the Linux UIO driver lets consider its performance. In real world drivers, ioctl() is commonly used to write a single value to a hardware register. As shown in Figure 3.6, this is not always as straightforward as one might think. In that system call, the Virtual File System needs to find the ioctl() implementation for the specific device and call it. Then, the ioctl() function will copy the value from userspace to kernel space. On the contrary, in a UIO driver the device memory is directly mapped into userspace. Hence, writing to a register might be as simple as an access to a regular array of integers. In addition, reading a result from the hardware is equally simple. This features are what make a UIO userspace driver code faster and easier to read [26]. Last but not least, the Linux UIO driver is employed in our AXI4-Lite implementations of our custom accelerators to map the device memory to userspace. This will be discussed further later.

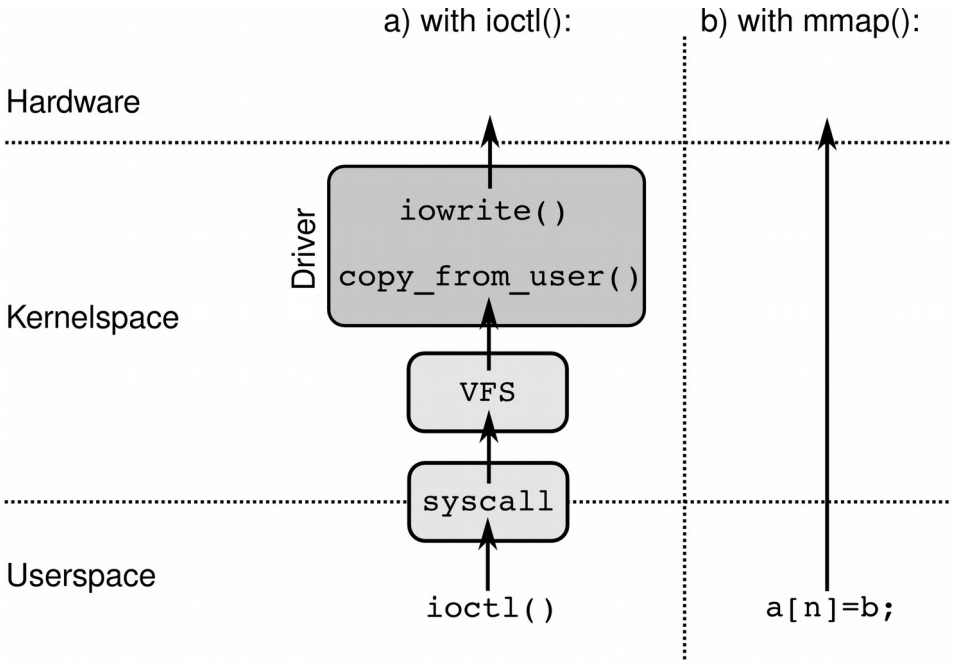


Figure 3.6: Ioctl() vs. Memory Access through UIO [26]

3.4 Direct Memory Access

Direct Memory Access (DMA) is a feature of computer systems that allows certain hardware subsystems to access the main memory (usually RAM), independently of the Central Processing Unit (CPU). In a system without DMA, the CPU is using programmed input/output and is typically fully occupied for the entire duration of read and write operations, thus it cannot perform any other tasks. However, this is not the case in a system disposing the DMA feature. In a system with DMA capabilities, a DMA controller is notified by the CPU that a data transfer should be made. The CPU initiates the transfer, then it performs other operations until it finally receives an interrupt from the DMA controller when the requested operation has finished. This feature is useful at any time the CPU cannot keep up with the rates of data transfer, or when it needs to perform useful tasks while waiting for a relatively slow I/O data transfer. We should now proceed to a basic explanation of the operation principles of DMA and DMA controllers.

A DMA controller is a device, usually a peripheral to a CPU, that is programmed to perform a sequence of data transfers on its behalf. The DMA controller can directly access the memory and make a transfer from a memory location to another, or from an I/O device to memory and vice versa [27]. A DMA controller manages several DMA channels each of which can be programmed to perform a sequence of data transfers. A DMA controller typically shares the system memory and I/O bus with the CPU and is able to perform as both master and slave. Depending on the manner that a DMA transfer is made there are different modes of operation that are presented below.

Burst Mode

In burst mode of operation an entire block of data is transferred in one contiguous sequence. Once the DMA controller is granted access to the system bus by the CPU, it transfers all bytes of data in the data block, also referred to as burst, before releasing control of the system buses back to the CPU, hence, the CPU might be inactive for relatively long periods of time depending on the burst size. It should be noted that the size of a data block depends not only on the burst size but also on the data bus width. If the width of a data bus is 32-bit then the size of a data block is 32-bit times the burst size. However, in a 64-bit data bus the block size would be 64-bit times the burst size.

Cycle Stealing Mode

The cycle stealing mode is used in systems in which the CPU should not be inactive for the length of time needed for a burst mode transfer to be completed. In this mode, the DMA controller gains access to the bus in the same way as before. However, in cycle stealing mode, after transferring one byte of data, the control of the buses returns to the CPU. If the transfer of the desired number of bytes has not been completed then the DMA controller requests the bus again to send another byte of data. This is repeated until the transfers are completed. On the one hand, in this mode of operation a data block is not transferred as quickly as in burst mode transfers. On the other hand, the CPU does not remain idle for long periods of time and can perform other operations even though the DMA transfer is not completed.

Transparent Mode

In the transparent mode of operation the transfer of a data block takes the longest time interval when compared to burst and cycle stealing mode, yet it is the most efficient mode in terms of overall system performance. In this mode, the DMA controller only transfers data when the CPU performs operations that do not utilize the system buses. The CPU never stops executing its programs and the DMA transfer is free in terms of time. A drawback of the transparent mode of operation is that the hardware needs to determine when the system buses are not utilized by the CPU.

DMA can lead to cache coherency problems. Lets imagine a CPU equipped with a cache and an external memory that can be accessed directly by devices using DMA. When a CPU access a location X in the memory, the location's value will be stored in the cache. Then the CPU performs subsequent operations on X, which will update the cached copy but not the external memory version of X, assuming a write-back cache. If cache is not flushed to memory before the next access of X by a DMA-based device, then the device will receive a stale value of X. This issue has been addressed either in a hardware method or in a software method [27]. ARM offers the Acceleration Coherency Port (ACP) on which a DMA controller can connect to, in order to deal with previously mentioned issue.

The AXI4-Stream versions of our implementations which are discussed further later take advantage of Direct Memory Access. Specifically, an AXI DMA block is used in order to make transfers from the memory to custom accelerators. The DMA block performs as both a master and a slave to the PS-side of the ZedBoard. A specific transaction is commanded by the Processing System, thus the AXI DMA block performs as a slave. When the transfer is to be done, the AXI DMA block accesses the memory through the High Performance (HP) slave ports of the PS, thus performing a master. The AXI DMA block and its operation will be discussed further in following chapters. At this point, a general description and explanation of the principles of DMA has been made and we are ready to proceed to the next chapter.

Chapter 4

Employed Work Flow for HW IP Integration on ZedBoard

4.1 Zynq Evaluation and Development Board Specifications

The purpose of this chapter is to capture a proposed framework and be an overall guide for implementations of various applications on ZedBoard. The whole flow of design tools utilized to produce and implement a system is described, starting with Vivado HLS, proceeding to Vivado Design Suite, Petalinux Tools and finally Xilinx SDK. To begin with, a description and listing of ZedBoard Zynq Evaluation and Development Board specifications is presented.

The ZedBoard [28] is a low-cost evaluation and development board based on the Xilinx Zynq®-7000 All Programmable SoC (AP SoC). ZedBoard combines a Dual-Core ARM® Cortex A9 Processing System (PS) with 85,000 Series-7 Programmable Logic (PL) cells and it can be targeted for a wide range of applications. The board includes everything necessary for Linux, Android, RTOS and other OS based designs. In addition, the processing system and programmable logic I/Os are exposed through several expansion connectors for easy user access. The features provided by ZedBoard [29] consist of the following:

- **Memory**: Zynq contains a hardened PS memory interface unit. The memory interface unit includes a dynamic memory controller (DDR3) and static memory interface modules (SPI Flash, SD card interface).

- **USB:** ZedBoard implements one of the two available PS USB OTG interfaces. Additionally a USB-to-UART bridge is connected to a PS UART peripheral providing JTAG functionalities and USB circuit protection.

- **Display and Audio:** An Analog Devices ADV7511 HDMI Transmitter provides a digital video interface to the ZedBoard. On top of that, the ZedBoard allows 12-bit video output through a through-hole VGA connector. An Analog Devices ADAU1761 Audio Codec provides integrated digital audio processing. Finally, An Inteltronic/Wisechip UG-2832HSWEG04 OLED Display is used on the ZedBoard.

- **Clock Sources:** The PS subsystem uses a dedicated 33.3333 MHz clock source, IC18, Fox 767-33.333333-12, with series termination. The PS infrastructure can generate up to foul PLL-based clocks for the PL system. An on-board 100 MHz oscillator, IC17, Fox 767-100-136, supplies the PL subsystem clock input.

- **Reset Sources:** The Zynq PS supports external power-on reset signals. The power-on reset is the master reset of the entire chip. A push button switch initiates reconfiguring the PL-subsection by the processor. Power-on reset erases all debug configurations.

- **User I/O:** The ZedBoard provides 7 user GPIO push buttons; five on the PL-side and two on PS-side. It has eight user dip switches, providing user input accompanied by eight user LEDs.

- **10/100/1000 Ethernet PHY:** The ZedBoard implements a 10/100/1000 Ethernet port for network connection using a Marvell 88E1518 PHY.

- **PS and PL I/O Expansion:** A single low-pin count (LPC) FMC slot is provided on the ZedBoard to support a large ecosystem of plug-in modules. The Zedboard has five Pmod compatible headers (2x6). The XADC header provides analog connectivity for analog reference designs, including AMS daughter cards.

- **Configuration Modes:** Zynq-7000 AP SoC devices use a multi-stage boot process that supports both non-secure and secure boot. The PS is the master of the boot and configuration process. Upon reset, the device mode pins are read to determine the primary boot device to be used: NOR, NAND, Quad-SPI, SD card or JTAG.

As already mentioned, the ZedBoard can be used for a wide range of applications varying from video processing, motor control, software acceleration, Linux/Android/RTOS development to Embedded ARM Processing and general Zynq-7000 AP SoC prototyping. The area of interest in this thesis is software acceleration by building an HLS-based IP on the PL-side of the device. To be more precise, our field of study is the exploration and evaluation of communication potentials between the PS and PL sides of the device. Hence, our interest focuses on the features of the PS and PL sides of the ZedBoard but mostly on their interconnection which is accomplished through High Performance ARM AXI interfaces (High Bandwidth AMBA interconnect), a solution providing scalable and effective communication. Table 4.1 contains essential information about the available resources of the ZedBoard.

Name	BRAM_18K	DSP48E	FF	LUT
Available	280	220	106400	53200

Table 4.1: ZedBoard Available Resources

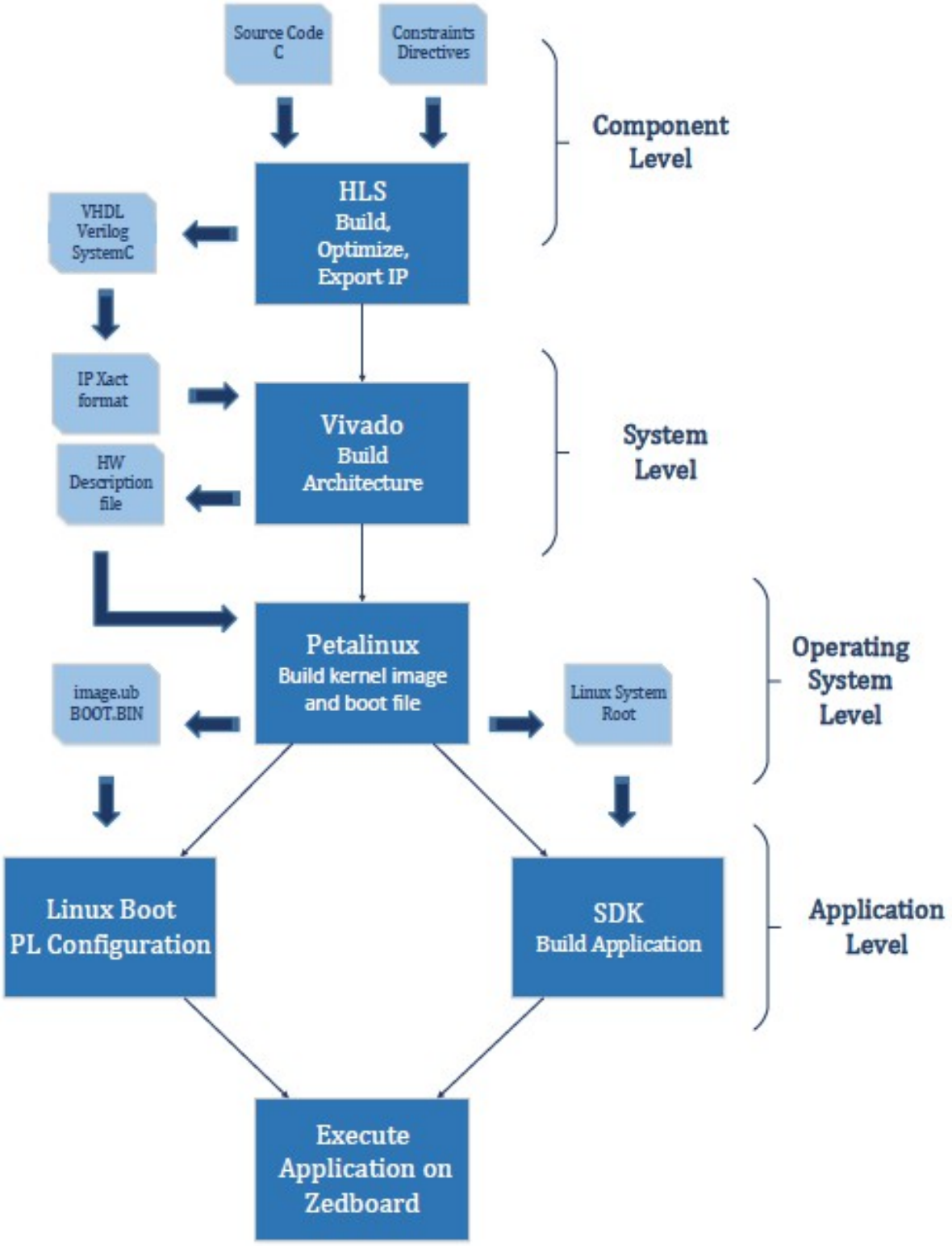


Figure 4.1: Implementation Work Flow [18]

4.2 IP Generation with High-Level Synthesis

Advanced algorithms used nowadays in wireless, medical, defense and consumer applications are more sophisticated than ever before. Vivado® High-Level Synthesis, a design tool launched by Xilinx, accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx All Programmable devices without the need to manually create RTL. Vivado HLS shows a fast path to IP creation. The abstraction of algorithmic description, data type specification and available interfaces (AXI4, AX4-Lite, AXI4-Stream) are a key element in what Vivado HLS offers. In addition, there are extensive libraries for arbitrary precision data types, video, DSP and more available. On top of that, the directives driven architecture-aware synthesis delivers the best possible quality of designs. Moreover, Vivado HLS offers an accelerated verification using C/C++ test bench simulation, automatic VHDL or Verilog simulation and test bench generation. It should be noted that for all IP generations that were demanded through the duration of this diploma thesis, Vivado HLS 2014.4 was employed.

A general capture of the steps of High-Level Synthesis given an abstract algorithmic description has already been presented in Chapter 1. As already mentioned, our field of interest is the evaluation of the available communication interfaces between the PS and PL sides of the ZedBoard, and not particularly in directives and optimizations during the High-Level Synthesis stage of the design flow. Lets consider a code file that is available in which a functionality has already been optimized, the directives to Vivado HLS which produce the optimal hardware are given in the code and the functionality is described in C programming language. For purposes of completion lets us enumerate and describe some of the basic HLS directives.

Directive	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop of function.
DATAFLOW	Enables task-level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations.
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block RAM resources
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
INTERFACE	Specifies how RTL ports are created from the function description.

Table 4.2: HLS Directives [18]

The given code might contain some of the above directives. Lets assume it may not contain the INTERFACE directive. This is where our job begins. Assuming the C code comes with no documentation whatsoever, there must be a clarification of the input and output data. Since this clarification transpires the INTERFACE directive should be used on the input data, output data and top function of the code. It is obvious that the input and output data have to be arguments of the top function. The interface refers to the type of I/O protocol that is used. Our solution of choice for specifying the type of I/O protocol is Interface Synthesis, where the port interface is created based on efficient industry standard interfaces. An alternative to Interface Synthesis would be a manual interface specification where the interface behavior is explicitly described in the input source code, a fact which allows any arbitrary I/O protocol to be used. The term Interface Synthesis refers to the process of the arguments of the top-level function being synthesized into RTL ports when the top-level function is synthesized. In general, Vivado HLS creates three types of ports on the RTL design: clock and reset ports, block-level interface protocols, port-level interface protocols.

Clock and Reset Ports

The `ap_clk` and `ap_rst` ports are automatically created in every synthesized design. The clock ports are created if a design requires more than one clock cycle of its completion. The input of the `ap_clk` port is applied to all existing functions of the design and it should be mentioned that only one clock can be applied to C or C++ designs. The operation of the reset is controlled by the config_rtl configuration.

Block-Level Interface Protocols

The block-level interface protocols are `ap_ctrl_none`, `ap_ctrl_hs` and `ap_ctrl_chain`. For the purposes of this diploma thesis only `ap_ctrl_none` and `ap_ctrl_hs` will be examined. The block-level interface protocols can only be specified on the function or the function return. Even if the function is of `void` type a block-level protocol may be specified on the function return.

The `ap_ctrl_hs` is the default protocol and generates ports that control the block independently of any port-level I/O protocols. The PS of the ZedBoard is later going to control an IP block through these ports. The generated ports control when the block can start processing data (`ap_start`), indicate when it is ready to accept new inputs (`ap_ready`), indicate if the design is idle (`ap_idle`) or has completed operation (`ap_done`). When using the AXI4-Lite interface, the previously mentioned ports are grouped in one bundle. The `ap_ctrl_none` mode implements the design without block-level I/O protocol and will be useful when implementing the AXI4-Stream versions of our accelerators.

Port-Level Interface Protocols

After the block-level protocol has been used to start the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block. The AXI4 Interfaces come under this category of protocols. Those supported by Vivado HLS are the AXI4-Stream (`axis`), AXI4-Lite (`s_axilite`), and AXI4 Master (`m_axi`) interfaces which will be discussed further. Another important mode is `ap_vld` which is set to 1 when an output port has a valid value and `ap_none` for input ports.

4.2.1 Setting AXI4-Lite Interfaces

Lets now assume that we are given a simple code and we are asked to add the necessary communication interfaces so that the produced IP could be added and interconnected in an AXI4-compliant system. In Listing 4.1 we may see a simple sample code. The specific code takes as inputs an array a of twenty integers and an integer number b and counts the occurrences of b in array a.

```
1     void count (int a[20], int b, int *c) {
2
3           int i = 0, temp_c = 0;
4
5           for (i = 0; i < 20; i++) {
6                 if (a[i] == b) temp_c++;
7           }
8           *c = temp_c;
9     }
```

Listing 4.1: A simple C-code paradigm

Although AXI4-Lite is not supposed to be used on arrays, it was in fact used in our implementations described in next chapters without occurring issues so we are going to employ it in this paradigm. We would like to group all interface ports in a bundle called "COUNT_IO". AXI4-Lite interfaces will be set to a and b, the return value c and the return port of the function. In Listing 4.2 we may notice the altered code using the INTERFACE directive.

```
1     void count (int a[20], int b, int *c) {
2
3     #pragma HLS INTERFACE s_axilite port=a bundle=COUNT_IO
4     #pragma HLS INTERFACE s_axilite port=b bundle=COUNT_IO
5     #pragma HLS INTERFACE s_axilite port=c bundle=COUNT_IO
6     #pragma HLS INTERFACE s_axilite port=return bundle=COUNT_IO
7
8           int i = 0; temp_c = 0;
9
10          for (i = 0; i < 20; i++) {
11                if (a[i] == b) temp_c++;
12          }
13          *c = temp_c;
14    }
```

Listing 4.2: A simple C-code with AXI4-Lite Interfaces

It should be mentioned that Vivado HLS automatically sets the top-level function's interface to `ap_ctrl_hs`, unless it is set manually by the user to another mode. The ports generated by this particular mode are bundled with the return port of the function. The input ports are also set to ap_none, and more important, the output ports are set to `ap_vld`. This will be useful when the hardware will be controlled by the PS-side of the ZedBoard, since `ap_vld` mode offers the ability to check if the output is valid, or, in other words, if the computation is completed and the output value is written and up-to-date. After the C-synthesizing the HW, the next step is the extraction of RTL in IP-XACT form. For IPs with AXI4-Lite interface ports a C driver is generated automatically so that the AXI4-Lite ports can be controlled through a Linux application. The generated device is memory-mapped. More details about the generated driver will be discussed further later, although an overview of the Linux UIO driver has already been presented in the previous chapter. In Figure 4.1 the classify IP with AXI4-Lite Slave that we generated for our implementations is presented.

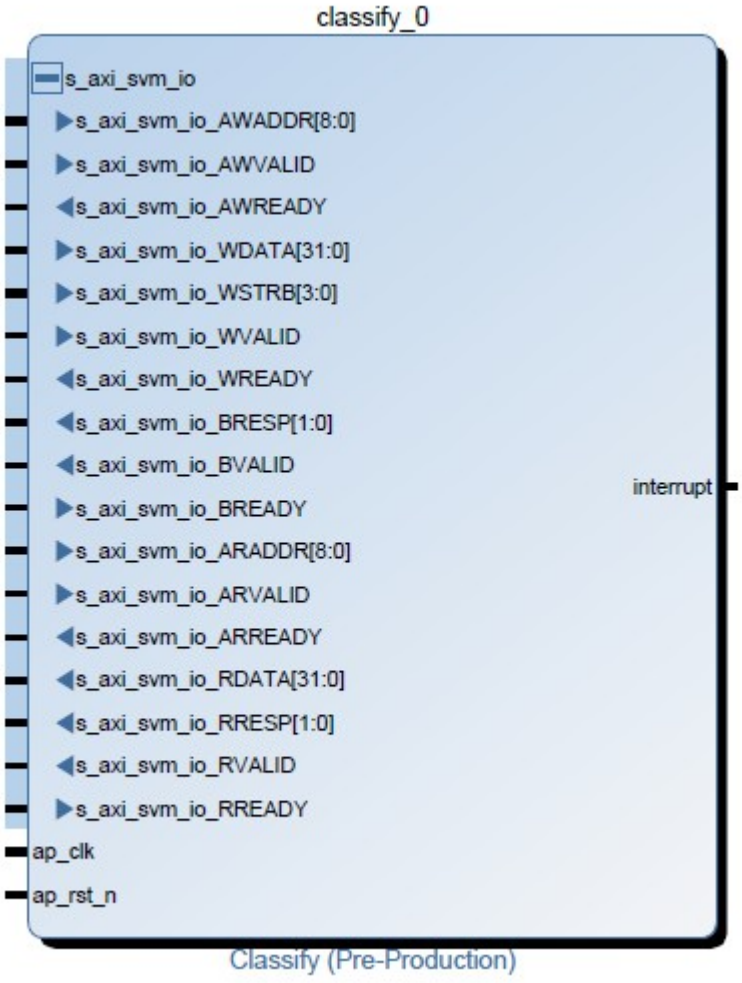


Figure 4.2: Classify IP with AXI4-Lite Interfaces

4.2.2 Setting AXI4-Stream Interfaces

Our next assumption is that we are given a simple code and our target is to add AXI4-Stream interfaces. A detail that should be paid attention to is the fact that if an AXI4-Stream interface is set on an array, then the accesses to the specific array have to made in a sequential order and no input values may be reused. In the Harris & Stephens Corner Detector, as much as in the Support Vector Machine Classifier code, accesses to input arrays are not made in a sequential order. Hence, if one wishes to take advantage of the speed that AXI4-Stream interfaces offer, then another approach should be considered, otherwise the AXI4-Stream interfaces cannot be set. Lets assume a kind of code like the one presented in Listing 4.3 is given and AXI4-Stream Interfaces should be added. The code is of no particular use but is employed in order to clarify the manner in which AXI4-Stream interfaces were used in our implementations during this thesis.

```
1     void dummy (int a[20], int *y) {
2
3           int sum = 0, i = 0, j = 0;
4
5           for (i = 0; i < 100; i++)
6                 for (j = 0; j < 20; j++)
7                       sum += a[j] – 1;
8           *y = sum;
9     }
```

Listing 4.3: Dummy C-code Paradigm Intended for addition of AXI4-Stream Interfaces

In the above listing, an array of twenty integer numbers is given as input to the top-level function. As we may notice, the accesses to each element of the array is reused a hundred times and a sum is computed. The final value of the sum is the output value of the function. Our implementations on Harris and SVM classifier might not be exactly like the above code, however, this code is adequate for the point that we need to address. In the next listing we may observe the transformed code with the AXI4-Stream interfaces added.

```
1     int dummy (int a[20]) {
2
3           int sum = 0, i = 0, j = 0;
4
5           for (i = 0; i < 100; i++)
6                 for (j = 0; j < 20; j++)
7                       sum += a[j] – 1;
8
9           return sum;
10    }
11
```

```
12    void top_dummy (int a[20], int *y) {
13
13    #pragma HLS INTERFACE axis port=a
15    #pragma HLS INTERFACE axis port=y
16    #pragma HLS INTERFACE ap_ctrl_none port=return
17
18          int i = 0, temp_a[20];
19
20          for (i = 0; i < 20; i++)
21                temp_a[i] = a[i];
22          *y = dummy(temp_a);
23    }
```

Listing 4.4 Transformed Dummy C-code with AXI4-Stream interfaces

The previous code was transformed in way so that the accesses to the input array seem to be sequential. In fact, the accesses to array `a` in `top_dummy` function are sequential. When all the values of input array `a` are collected to array `temp_a` then the array is passed as an argument to `dummy` function which is the one that performs the needed computations. The `dummy` function's type is changed to `int` and the `sum` is returned to the output value `y` of the `top_dummy` function. The block-level protocol of the top-level function is set to ap_ctrl_none because the necessary control is now transposed to the hardware. When twenty integer numbers are collected, the computation begins. Thus, when the PS-side of the ZedBoard needs to make a transfer to the hardware and get a result, it commands the AXI DMA block to perform the transfer. The values are streamed to the accelerator and the result is streamed back to the AXI DMA block which then forwards it to the PS. So, the PS controls the AXI DMA block and not the accelerator. If no data are sent to the hardware, then no computation is performed. Though this method for addition of AXI4-Stream interfaces might introduce additional latency, the function level handshakes are avoided, there is no need to initiate the computation and, finally, there is no need to check if the output signal is valid or not. These features combined with the extremely fast transfers that are achieved through the employment of DMA blocks, eventually end up with satisfactory latency gains as we have recorded in following Ch. 5 & 6.

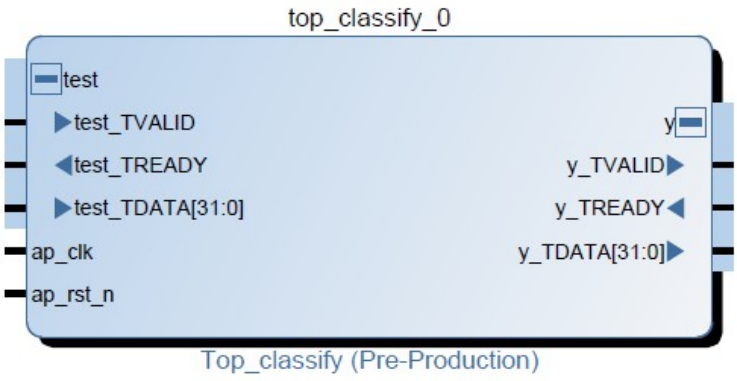


Figure 4.3: Classify IP with AXI4-Stream Interfaces

4.3 System Generation

At this point the IP, whether it contains AXI4-Lite interfaces or AXI4-Stream interfaces, has been generated. The next step is the creation and generation of the system architecture as a whole in Vivado Design Suite. It offers a new approach for ultra high productivity with next generation C/C++ and IP-based design. The RTL has already been produced during the C-synthesis of the HLS step. In Vivado Design Suite, a block design will be created by the addition of the ZYNQ7 Processing System and the previously generated IP. After the interconnection and validation of the design we proceed to the synthesis and implementation, and finally to bitstream file generation. In our work Vivado Design Suite 2014.4 was used.

To begin with, the first step is the creation of a block design and the addition of the ZYNQ7 Processing System IP. The IP should be re-customized to fit our needs. In clock configuration at least one PL Fabric Clock should be chosen. Up to four PL Fabric Clocks can be included with frequencies theoretically ranging from 0 to 250 MHz. The PL-PS fabric interrupts should be enabled creating an IRQ_F2P port on ZYNQ7 Processing System IP. The USB interface is not needed in our designs and should be disabled. After the necessary customizations the block design of ZYNQ7 Processing System looks like the one shown in Figure 4.3 where the DDR and FIXED_IO ports are made external when block automation is run. After the addition of the PS part in the block design, our generated IP should be added. The repositories should be edited so that our custom IP is included. A slightly different design process is then followed depending on the kind of interfaces of our IP ports. The cases of AXI4-Lite and AXI4-Stream interfaces are examined.

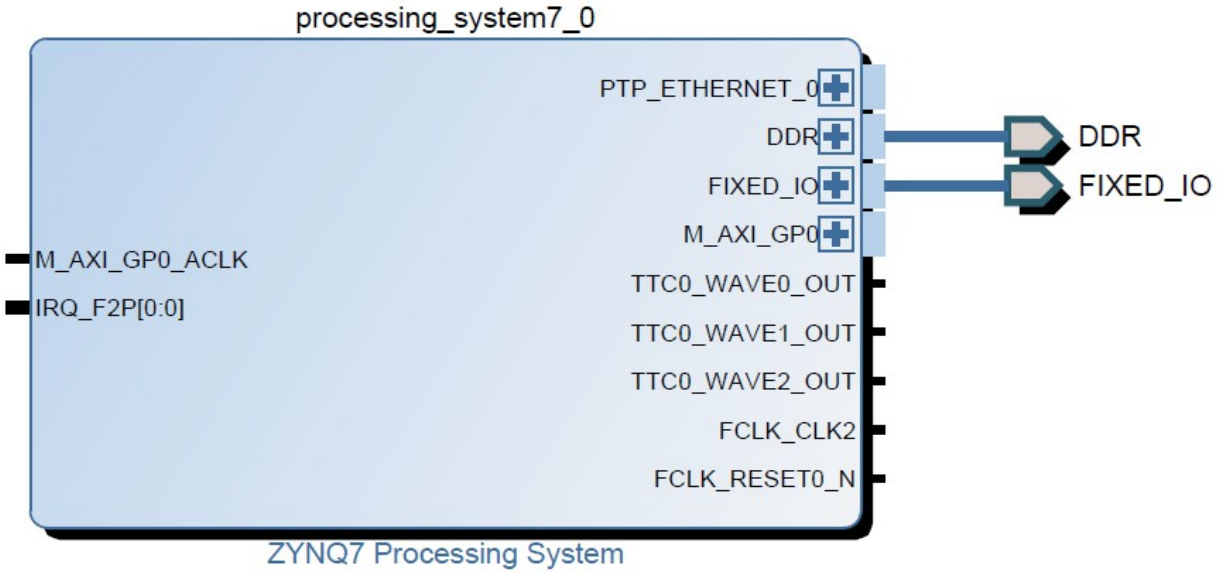


Figure 4.4: Re-customized ZYNQ7 Processing System

4.3.1 System Design with AXI4-Lite Interfaces

When a custom IP is equipped solely with AXI4-Lite interfaces the design of the system requires no effort whatsoever. On the IP addition the option “Run Connection Automation” is enabled and interconnects the AXI4-Lite custom IP with ZYNQ7 Processing System with addition of an AXI Interconnect Block and a Processor System Reset. The interrupt port of the Classify IP is connected to the IRQ_F2P port of the PS.

AXI Interconnect IP

The AXI Interconnect IP block connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices. The Interconnect IP is intended for memory-mapped transfers only and AXI4-Stream transfers are not applicable. It has the potential to connect 1 to 16 Master devices and 1 to 16 Slave devices. This means that if more than one Slave IPs are included in the same system design then only one AXI Interconnect IP will be utilized if the number of Slave devices is less than 16. The Slave port of the Interconnect IP is connected to the Master AXI General Purpose (M_AXI_GP) port of the ZYNQ7 Processing System, while one of the M_AXI Interconnect ports is connected to the AXI4-Lite Slave port of our custom accelerator. Obviously, the Interconnect IP and the custom IP have the same clock and reset port sources. No re-customization is needed for the nature of design we wish to implement.

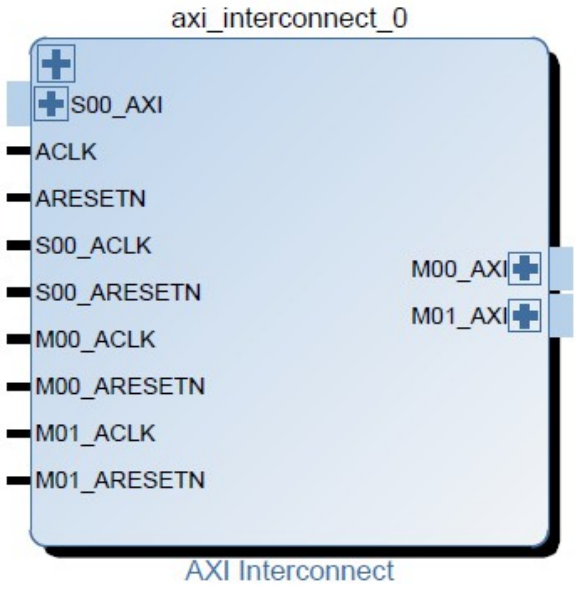


Figure 4.5: An AXI Interconnect IP Block

Processor System Reset

The Processor System Reset is another necessary component of our custom system architecture. It generally allows the users to tailor the design to suit their application by setting certain parameters to enable or disable features. It should be mentioned that the

asynchronous external and auxiliary external reset inputs are synchronized with clock. Needless to say that the application of proper reset signals is essential for an FPGA design to perform appropriately. The Processor System Reset is intended to implement a Power-on Reset (PoR) which detects the power applied to a the chip and generates a reset impulse that travels through the entire circuit placing it into a known state. No re-customization is needed for the Processing System Reset in our case.

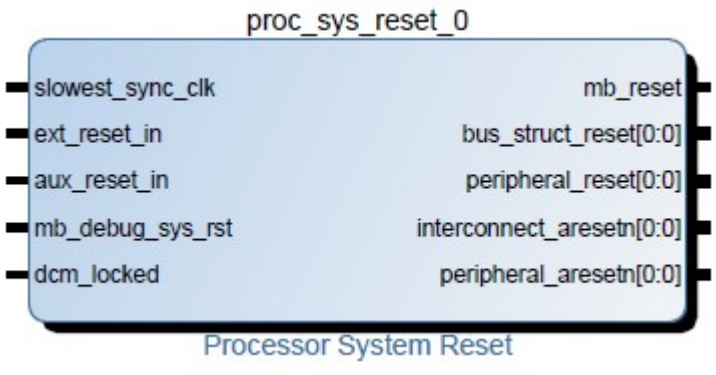


Figure 4.6: A Processor System Reset block

It should be mentioned that no specific block designs are presented in this chapter as various implementations along with their respective block designs are presented extensively in Ch. 5 & 6 of the present diploma thesis.

4.3.2 System Design with AXI4-Stream Interfaces

The work flow for the generation of our system architecture is not quite the same and straightforward when AXI4-Stream interfaces are added to our custom accelerators. In contrary to AXI4-Lite versions, in this case the IP is not automatically interconnected with the PS with the push of a button. In case of AXI4-Stream custom IPs the first step for system generation is the addition of an AXI DMA IP core. The existence of AXI Interconnect IP is essential once again for the AXI DMA block to be connected through its AXI4-Lite Slave port with the M_AXI_GP port of the processing system. An element absent in the AXI4-Lite version of the design is the Slave High Performance Ports of the ZYNQ7 Processing System. We proceed to an overview of the IP blocks and ports that were not present in the AXI4-Lite versions of the system and specifically AXI DMA.

AXI Direct Memory Access

The AXI DMA is utilized to provide high-speed data movement between system memory and an AXI4-Stream-based target IP, like the AXI4-Stream versions of our Harris_FindCorners and Classify IPs. The implementations will be discussed further in Ch. 5 & 6. The AXI DMA block is re-customized to fit to our specific needs. The Status/Control Stream and Scatter/Gather Engine are disabled because they are not needed in our applications. If we re-customize the AXI DMA block we will notice a number of parameters

that can be altered and may or may not affect the performance of the block and, consequently the speed of data transfers. Lets now have an overview of the parameters that an AXI DMA block is using:

- **Width of Buffer Length Register:** It refers to the length of the internal counter or register in the DMA which stores the length of DMA operation data. Its main impact is on maximal achievable frequency and has slight or no impact in the FPGA utilized resources. This parameter is set to 23 bits which is the largest possible value and is recommended by the utilized DMA driver.

- **Memory-Map Data Width:** It specifies the data width of AXI4 Interface. Data widths of 64 bits can significantly improve throughput when connected to the HP or ACP port of the ZYNQ7 Processing System. However, Vivado Design Suite does not leave us option for altering this value. This parameter should not be misinterpreted with AXI4 Stream data width.

- **Stream Data Width:** It represents the width of AXI4-Stream Interface. For instance, if an accelerator takes an input array of integers or floats then the width of the stream should be 32 bits. In our implementations, the Harris Corner Detector has an input image in the form of an array of unsigned chars (8-bit) so the stream width is set to 8 bits. On the other hand, the SVM Classifier receives an input array of floats so the stream width is set to 32 bits.

- **Max Burst Size:** Data on an AXI Interface can be transferred in bursts. Considering that the bus is 32-bits wide, if a burst size value of 8 is used then the size of a block to be transferred to a device would be 8*32 bits. Higher burst size leads to better throughput. This parameter should be set to at least 16. If the parameter is set to 256 the speedup in comparison with a burst size of 16 will be imperceptible. The PS AXI interfaces are AXI3-compliant so the burst size is limited to 16. Thus, the AXI Interconnect must split the AXI4 bursts to several AXI3 bursts. In our designs all possible burst sizes were used with almost no difference whatsoever in throughput.

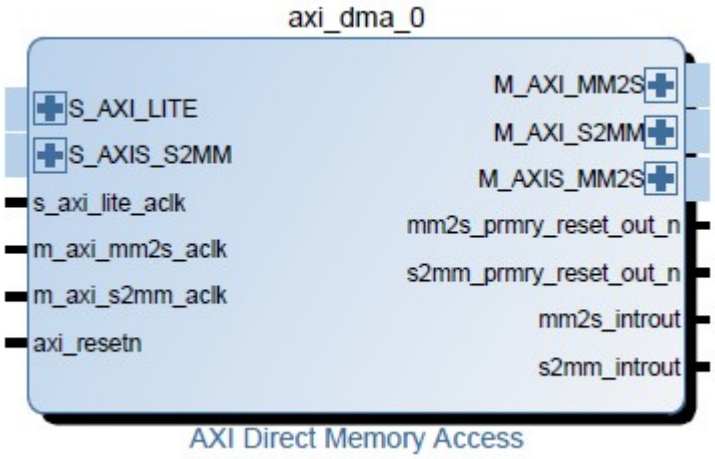


Figure 4.7: An AXI DMA Block

The AXI DMA block has several ports as we may notice in Figure 4.6. The S_AXI_LITE port is for the block to be interconnected to the PS. The M_AXI_MM2S and M_AXI_S2MM as their names might imply, are the memory-mapped to stream and stream to memory-mapped channels, in other words the channels that are used for reads from memory and writes to memory. This channels are connected to the Slave HP Ports of the PS-side through an AXI Memory Interconnect intended for use with non memory-mapped HW. The M_AXIS_MM2S port is connected directly to the input port of our accelerator and the accelerator's output is connected back to the S_AXIS_S2MM port of the block. It should be noted that Zynq Slave HP Ports are 64-bit wide and support the connection of both MM2S and S2MM channels to a single HP Port. In fact, different configurations were tested including the two channels being connected to one HP Port and the two channels being connected to two separate HP Ports with no occurring differences in execution times and data rates. Finally, the MM2S and S2MM interrupt ports are connected to a Concat IP which has a cascading connection to IRQ_F2P port of the ZYNQ7 Processing System. Afterwards, the design is ready for synthesis, implementation and generation of the bitstream file.

4.4 Generation of Embedded Linux Distributions

The next step of our design flow after the generation of the system's bitstream file and the hardware's exportation is the creation of an operating system that is executed on the PS-side of the ZedBoard, for a system which includes the hardware generated through Vivado Design Suite. The creation of an Embedded Linux distribution for our custom hardware is accomplished with the aid of Xilinx PetaLinux Tools which offer everything necessary to customize, build and deploy Embedded Linux solutions on Xilinx processing systems and especially on Zynq®-7000 All Programmable SoC. In our work PetaLinux Tools 2014.4 were used. The first step is the creation of a Linux platform in the form of an empty project template. The Linux platform is customized to precisely match the hardware system built in Vivado Design Suite. This is accomplished by copying and merging the platform configuration files generated through the hardware building phase into the newly created software platform. The tool configures the system by parsing the hardware description file (.hdf) to obtain the hardware information in order for the device-tree to be updated, as much as PetaLinux U-boot configuration files and kernel config files. During the configuration we set the SD card as the primary boot device. Then we confirm that the Userspace I/O drivers are included as built-in and, in case of a system including DMA transfers, that the Contiguous Memory Allocator is enabled under Generic Driver Options. If we take a look at the generated device-tree we will notice every block that is included in our design. In a design where a Dummy IP use AXI4-Lite interfaces, the device should be compatible with the UIO driver. For this reason the addition of Listing 4.5 in the device-tree is essential for every distinct IP and every instance of the same IP. It should be also noted that occasionally when building a new Linux platform, errors concerning the Ethernet device occurred. These errors were overcome by editing the device-tree once again and adding the code of Listing 4.5 concerning the ethernet device. The proposed alterations of the device-tree should be made in the `system-top.dts` file. When the configuration is finally completed we build the system image.

```
1     &ps7_ethernet_0_mdio {
2           phy-handle = <&phy0>;
3           mdio {
4                 #address-cells = <1>;
5                 #size-cells = <0>;
6                 phy0: phy@0 {
7                       compatible = “marvell,88e1510”;
8                       device-type = “ethernet-phy”;
9                       reg = <0>;
10                };
11          };
12    };
13
14    &dummy_0 {
15          compatible = “generic-uio”;
16    };
```

Listing 4.5: Linux device-tree necessary updates

Having built the system image, the next step is the creation of a boot image file that includes the Zynq FSBL (First Stage Boot Loader), the .BIT file for the configuration of the PL-side of the ZedBoard, U-boot and the Linux image for the SD card boot. The BOOT.BIN and image.ub files generated are copied to the SD card and Linux boots on ZedBoard. In this phase the PL has been configured and a USB-to-UART connection is made to our PC. Then, GtkTerm, which is a simple terminal used for communication with serial ports, is used. For our following implementations, the userspace application is cross-compiled on our machine and transferred to the ZedBoard through FTP.

At this point the PL-side of the ZedBoard has been configured with the hardware that was designed during the previous steps while a fully customized for our hardware Linux OS is running on the PS-side. If a device disposes AXI4-Lite interfaces then an entry will be created under the `/sys/class/uio`. In a system with more than one UIO devices, the developer is able to notice the name of a `uio0` device by executing `cat /sys/class/uio/name`. Normally, UIO devices should also have been created under the `/dev` directory. For instance, if there are three UIO compatible devices then uio0, uio1, and uio2 will be created. If the devices have not been added automatically then the `mdev -s` call should be executed and the UIO devices will be added. The previous comments refer to hardware with AXI4-Slave Lite interfaces. On the other hand, AXI4-Stream interfaces are not memory-mapped and no device is generated in the `/dev` directory.

For AXI4-Stream devices except for the Xilinx DMA driver, a complementary driver is utilized and performs as a wrapper for communication with the lower-level Xilinx DMA driver. The zynq-xdma [https://github.com/bmartini/zynq-xdma] has been developed by Berin Martini [https://github.com/bmartini] and generates a module that should be inserted in the system along with a library offering an API intended for use with the generated xdma module. The driver code should be built against the Linux Kernel that is intended to be used with. Minor adjustments were made for it to fit in our systems.

4.5 Userspace Application Development

At this point our system is up and running on the ZedBoard and our final task is the development of a userspace application. Xilinx SDK could be used but we preferred to develop our applications without it, cross-compile them in our machine and transfer the executable files to the implemented system through FTP (File Transfer Protocol). Applications targeted to AXI4-Lite implementations are not alike with applications targeted to AXI4-Stream implementations, so we will examine them separately.

4.5.1 Development of AXI4-Lite Targeted Application

The development of an application which controls an AXI4-Lite-based accelerator is based entirely on the Linux UIO driver which has already been presented in the previous chapter. Before proceeding to the development of the application one should first examine the automatically generated driver. As mentioned before, for AXI4-Lite-based devices a driver is automatically generated. Among the driver files we can find a header file where all the addresses for all signals of our accelerator are given. We keep this header file in mind. In Listing 4.6 a template for accessing an AXI4-Lite device is presented.

```
1     char *uiobf = “/dev/uio0”;
2     int *fd;
3     void *ptr;
4
5     fd = open(uiobf, O_RDWR);
6     if (fd < 1) {
7          printf(“UIO device error: %s.\n”, uiobf);
8          exit(EXIT_FAILURE);
9     }
10    ptr = mmap(NULL, MAP_SIZE, PROT_READ|PROT_WRITE,
11               MAP_SHARED, fd, 0);
12
11    /* Do Something */
12
13    munmap(ptr, MAP_SIZE);
```

Listing 4.6: Template code for accessing an AXI4-Lite device from userspace

In the above template code we may notice that the device is opened for reads and writes as a regular file, using the `open()` system call. Then, `mmap()` maps the device memory to userspace where it can be accessed regularly using the offsets from the previously mentioned header file. Specifically, `ptr` represents the beginning of the mapped memory and the offsets are used to read or write to specific addresses of the device memory. For instance, the `ap_start` port which is generated from the use of `ap_ctrl_hs` block-level protocol is usually the beginning of the device memory. After copying the necessary input

data to their corresponding addresses the initiation of the computation is accomplished by setting the `ap_start` signal to 1 for a brief moment and then again to 0. An `ap_vld` signal is set when the computation is completed, at which point we are ready to read the output values. If another computation is needed then the process is repeated. The device is finally unmapped from userspace when it is no longer needed. It is obvious that the UIO driver simplifies the process of accessing the device memory, consequently making application development faster when targeting AXI4-Lite devices.

4.5.2 Development of AXI4-Stream Targeted Application

In case of AXI4-Stream-based accelerators the device we wish to access is not memory-mapped. The task of the userspace application is to fill a buffer with input values, then call one of the high-level functions that are offered by the zynq-xdma API to send the data to the lower-level Xilinx DMA driver and perform the transfer. The essential function calls for making a DMA transfer and receiving a result are presented in the following table.

Function	Operation
`xdma_init()`	A function intending to initialize the AXI DMA blocks that are included in a design. Up to four devices are supported.
`xdma_alloc()`	Allocates the necessary input and output buffers for the transactions that are going to be performed. Returns a pointer to the address of the first element.
`xdma_num_of_devices()`	Returns the number of the active DMA devices. It should be called before a transaction to ensure the existence of at least an active DMA device.
`xdma_perform_transaction()`	It is responsible for sending the input buffer and receiving the output buffer. Arguments include the input and output buffers' addresses and sizes, the ID of the DMA device that we wish to perform the transfer and a flag (XDMA_WAIT_NONE, XDMA_WAIT_SRC, XDMA_WAIT_DST, XDMA_WAIT_BOTH) concerning the waiting or no-waiting of transfers.
`xdma_exit()`	It finalizes the DMA devices.

Table 4.3: Basic API of zynq-xdma driver library

The above functions are usually called in series of appearance in the table. The initialization of AXI DMA devices and the DMA engine is made through the `xdma_init()` call. After the initialization the DMA buffers should be allocated by the xdma_alloc() call. We usually allocate two buffers, one intended for input to the hardware and one intended for output of hardware. Before attempting to perform a transaction the number of DMA devices should be checked by the xdma_num_of_devices() call. The most critical part of the userspace

application is the transaction itself. The xdma_perform_transaction() call is used. It should be noted that we should be careful with the usage of available flags for this function call. Particularly, the flags refer to whether the application should wait for a transfer, either inward or outward, or not wait at all. In our view, the flag that made most sense to use was XDMA_WAIT_DST, which as its name implies, commands the driver to wait for the destination buffer, or output. So, after the issue for transfer of the source buffer, or input, there is no waiting for inward but only for outward transfers. Hence, the total time that is measured takes into account both communication and computation times. Finally xdma_exit() is called. At this point the application development for an AXI4-Stream device is finished.

Chapter 5

Evaluation of Work Flow on Harris & Stephens Corner Detector

5.1 General Description of HW Implementations

In this chapter the results of our implementations on Harris & Stephens Corner Detector are presented. This particular algorithm was our first attempt to explore and evaluate the communication potentials between the PS-side and the PL-side of the ZedBoard. The source code that was used for the High-Level Synthesis step of the implementation was provided by Ioannis P. Galanis, Graduate Student of the School of Electrical and Computer Engineering, NTUA, whose work during his diploma thesis[] led to the optimized code version that was implemented on the ZedBoard. However, it should be mentioned that the specific version had great possibilities at over-utilizing the ZedBoard. The target device during the development of this accelerator was Kintex-7 (xc7k325tffg900-2) which, in general, is a device with significantly more available resources than the ZedBoard (xc7z020clg484-1). Thus, the code should be transformed in order to give a realistic implementation for our target device. A comparison between the available resources of the above mentioned devices is shown in Table 5.1.

Device	BRAM_18K	DSP48E	FF	LUT
ZedBoard (xc7z020clg484-1)	280	220	106400	53200
Kintex-7 (xc7k325tffg900-2)	890	840	407600	203800

Table 5.1: Comparison of Available Resources between ZedBoard and Kintex-7

During development for a target device with more available resources one might think that a design is economical in the utilization of resources even though it consumes many of the available ones. In many cases percentages might be misleading. For instance if two-thousant BRAM_18K are available and their utilization is 40% it does not mean that not a lot of BRAMs are used. Since our target devices were different, in this chapter there is a paragraph

referring to the process of altering the code in order to fit to our target device, in terms of resources and utility. Then, after generating the HW we proceed to implementations of AXI4-Stream and AXI4-Lite versions. Harris & Stephens Corner Detector is an algorithm which consumes images as input data, hence, the communication between the PS and PL sides of the ZedBoard might be intense in terms of number of bytes that we need to transfer for a single execution of the algorithm. The communication and computation times of each implementation and image size have been measured. Communication time refers to the time that is needed for the input data to be transferred from the PS-side to the accelerator which lies on the PL-side of the ZedBoard, while computation time refers to the time that is needed for the necessary output data to be computed and written. In our case, the input data include the image in the form of a one-dimensional array and a struct of characteristics of the input image like height and width. The output data include the number of corners that were detected along with an array containing the coordinates of those corners. For purposes of comparison a software only version of the Harris & Stephens Corner Detector provided by Dr. Manolis Lourakis [http://users.ics.forth.gr/~lourakis] was executed on the PS-side of the ZedBoard.

Image Size (Pixels)	Communication Time (s)	Computation Time (s)
128 x 128	-	0.03313
256 x 256	-	0.13055
512 x 512	-	0.57567
1024 x1024	-	2.55367

Table 6.2: Time measurements for Harris SW version executed on ARM®

5.2 Code Transformations Targeting to a ZedBoard Implementation

As already mentioned, the optimized HW version of the Harris and Stephens corner detector was developed for a different target device with more available resources than the ZedBoard. In this paragraph the necessary code transformations leading to a ZedBoard implementation will be presented. Except for the addition of different communication interfaces, the input image's size was altered. In addition, we experimented with different memory cores, offered through Vivado HLS, that are essential for some parts of the algorithm's implementation. The final version that was implemented on the ZedBoard supports an input image size of 128 x 128 pixels. Of course, larger images were broken into pieces of 128 x 128 pixels through the userspace application that was developed and executed on the PS-side. The small input image size is a consequence of versions with larger input images over-utilizing the device. In fact, a version of 256 x 256 input image size would be a possibility if no additional resources were utilized by the essential communication interfaces. To begin with, we present the utilization for different 1024 x 1024 input image size versions. It should be mentioned that the initial version uses the memluv library for dynamic memory allocation which was not synthesizable, not only for our Vivado Design

Suite version but also for our target device, as many different versions of Vivado Design Suite were tried. The lack of this library might or might not create additional utilization of available resources.

To begin with, we employ a version of Harris and Stephens Corner Detector. The initial code processes an input image of 1024 x 1024 pixels and no synthesis directives are used except for the use of an asynchronous dual port RAM block to store some intermediate results needed through the computation of the image derivatives. The UNROLL and ARRAY_MAP optimizations are applied. In table 5.1 a comparison of the resource utilization is made in form of percentages.

	BRAM_18K	DSP48E	FF	LUT
None Utilization (%)	76	29	13	50
UNROLL Utilization(%)	76	42	19	80
ARRAY_MAP Utilization(%)	76	29	13	50
UNROLL & ARRAY_MAP Utilization(%)	76	42	19	80

Table 5.1: Utilized Resources for an Image Size of 1024 x 1024 for different directives

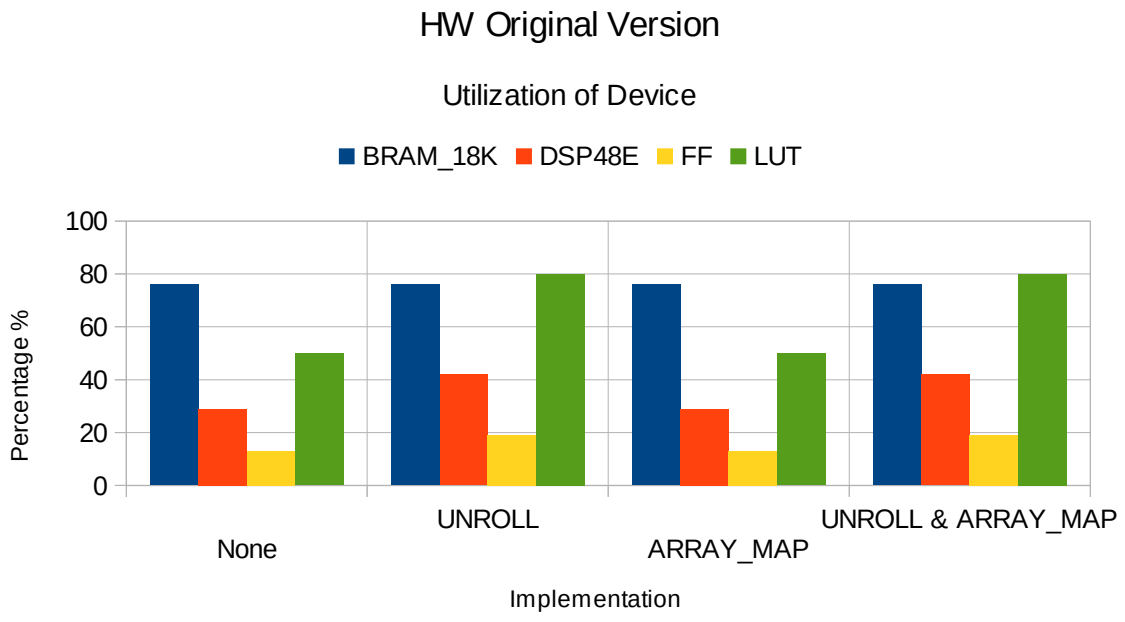


Figure 5.1: Utilization of Device for 1024 x 1024 Input Image Size

The above table, as mentioned, refers to a version of the algorithm using a dual port asynchronous RAM in one of its components and for an image size of 1024 x 1024. From aspect of estimated latency the version using the UNROLL directive on most loops with a factor of 4 has the lowest estimated latency, slightly below the UNROLL & ARRAY_MAP implementation. However, the later is used. We now assume that a basic default version of the code is acquired and we would now like to integrate a HW IP of this version in our target device. So, we decide to add the simplest interface possible, AXI4-Lite for an initial implementation and generate the HW. Nevertheless, an extremely high over-utilization of BRAM blocks is depicted. Then we decide to reduce the image size and see if it now fits in our device, and so we do, by reducing the image size to 512 x 512 pixels. Nonetheless the over-utilization of BRAM blocks still remains, yet with a lower value. The only option that is seen then is to reduce the image size even more and changing it to 256 x 256. Finally, the Harris_FindCorners IP fits in the device but with an extremely high utilization. So we then proceed to Vivado Design Suite for generation of our system. The IP is automatically interconnected through its AXI4-Lite ports. The synthesis is run, however an error occurs repeatedly, explaining that the dual port asynchronous RAM cannot be inferred. From the specifications of the product we find out that an asynchronous dual port RAM is not an option even though it is supported by the exactly same version of Vivado HLS. Our alternative option is to use a true dual port RAM either implemented as distributed memory or with BRAM blocks and of course single port RAMs. However, for every combination of memory cores there is an over-utilization of either the BRAM blocks or the LUTs. Hence, we decide to further reduce the input image size to 128 x 128 which is considered the smallest size that this computation makes sense because in smaller sizes it might be considered trivial, not define any memory core for this specific computation and let the tool decide which would be the best configuration. Lets now make another comparison of utilized resources up to this point. We have a 1024 x 1024 version with AXI4-Lite Interfaces and Dual Port asynchronous RAM, a 512 x 512 and 256 x 256 version of the same characteristics and finally a 128 x 128 version without the dual port asynchronous RAM core. In table 5.2 another comparison between the utilization is made.

	Dual Port RAM	BRAM_18K	DSP48E	FF	LUT
1024 x 1024 Utilization (%)	X	810	42	20	81
512 x 512 Utilization (%)	X	212	42	19	80
256 x 256 Utilization (%)	X	96	76	27	96
128 x 128 Utilization (%)	-	67	72	24	86

Table 5.2: Utilization of AXI4-Lite Version for Different Image Sizes

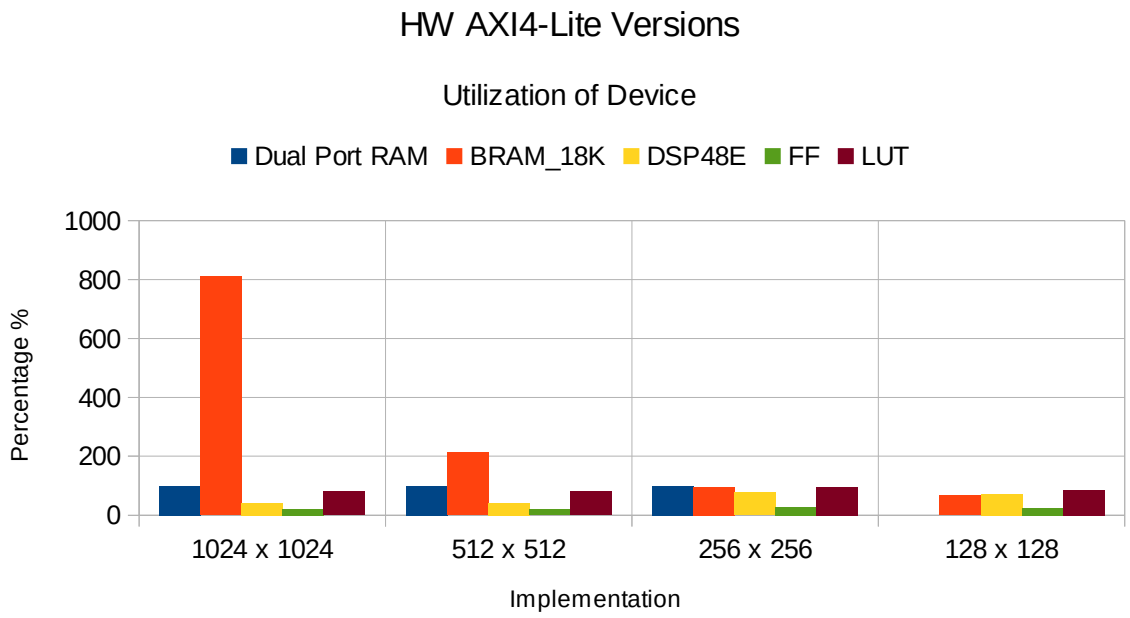


Figure 5.2: Utilization of Device for AXI4-Lite versions and different input image size

Finally, we proceeded to an AXI4-Stream Implementation of the 128 x 128, no dual port RAM version. AXI4-Stream in general does not utilize the device as much as AXI4-Lite which instantiates the input and output ports, but because of its operation principles needs no additional resources. So, we considered a 256 x 256 AXI4-Stream version to be viable only to discover that there was a really high unexpected utilization of BRAMs, hence only the AXI4-Stream 128 x 128 version was implemented and another version combining the AXI4-Lite and AXI4-Stream interfaces was created. In table 5.3 we may notice the utilized resources of our implementations.

	Lite	Stream	BRAM_18K	DSP48E	FF	LUT
128 x 128 Utilization (%)	X	-	67	72	24	86
128 x 128 Utilization (%)	-	X	57	82	24	85
256 x 256 Utilization (%)	-	X	173	72	24	86

Table 5.3: Utilization of Device for Different Interfaces

5.3 Implementation of AXI4-Lite Version

After reaching a viable solution for implementing the Harris & Stephens Corner Detector on the ZedBoard, it is now time that we present the results of the AXI4-Lite version of the HW. The AXI4-Lite version includes a bundle where all input and output values along with the block-level protocol ports are grouped together. The values of the struct harrisData are given through the device memory addresses mapped to userspace through the Linux UIO driver and the `memcpy()` function is used for the transfer of the image. Different AXI4-Lite versions were implemented. Beginning with the original unoptimized version and a clock of 50 MHz frequency, proceeding to the optimized version with 50 MHz and 75 MHz clocks. In the following tables we may notice the communication and computation times that were measured for different implementations. To begin with, in table 5.4 the time measurements for the not optimized version are presented. Of course, an important notice is that although the accelerator can process an image of only 128 x 128 pixels size per execution, larger images are broken into 128 x 128 pieces and are sent one after the other to the accelerator for processing. Hence, in the following time measurements we include the communication and computation times for all sizes of images.

Image Size (Pixels)	Communication Time (s)	Computation Time (s)
128 x 128	0.000825	0.032617
256 x 256	0.00297	0.129410
512 x 512	0.013271	0.522747
1024 x 1024	0.053690	2.121970

Table 5.4: Time measurements for Unoptimized HW Implementation (50 MHz Clock)

We may notice that the achieved bandwidth for this AXI4-Lite implementation is about 18.62 MB/s for the largest image size. We must notice that in the worst case we need to transfer 1 MB of data to the hardware accelerator. Lets now proceed to the optimized versions with different clocks of 50 MHz and 75 MHz. The results are depicted in the following table. The achieved bandwidth for the 75 MHz Clock was about 24.87 MB/s.

	Unroll & Array_Map (50 MHz)		Unroll & Array_Map (75 MHz)	
Image Size (Pixels)	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
128 x 128	0.000827	0.021395	0.000622	0.014588
256 x 256	0.003289	0.085210	0.002455	0.058099
512 x 512	0.013240	0.344918	0.009902	0.235177
1024 x 1024	0.053610	1.404327	0.040198	0.957514

Table 5.5: Time Measurements for Optimized HW Implementation with different clocks

Though expected, it should be noticed that the change of the clock in the optimized version from 50 Mhz το 75 MHz improves communication times to some extent and definitely improves the total latency of the algorithm as the computation time is decreased. Lets now make a comparison between computation times and communication times for all these implementations, including the software only implementation in the form of diagrams.

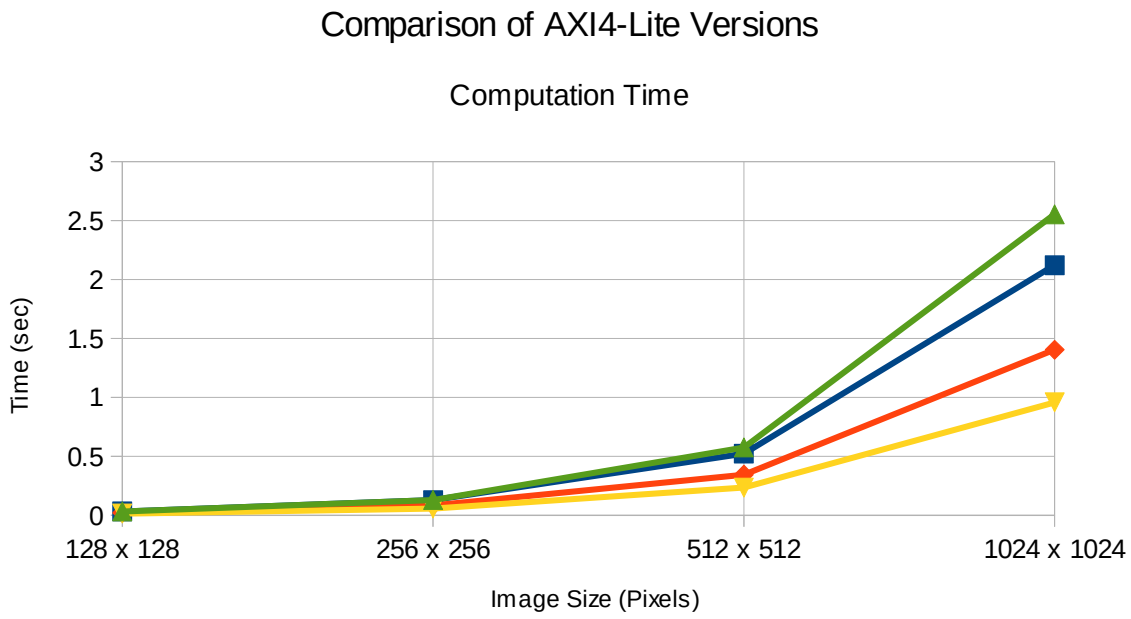


Figure 5.3: Computation Time for Different AXI4-Lite Implementations

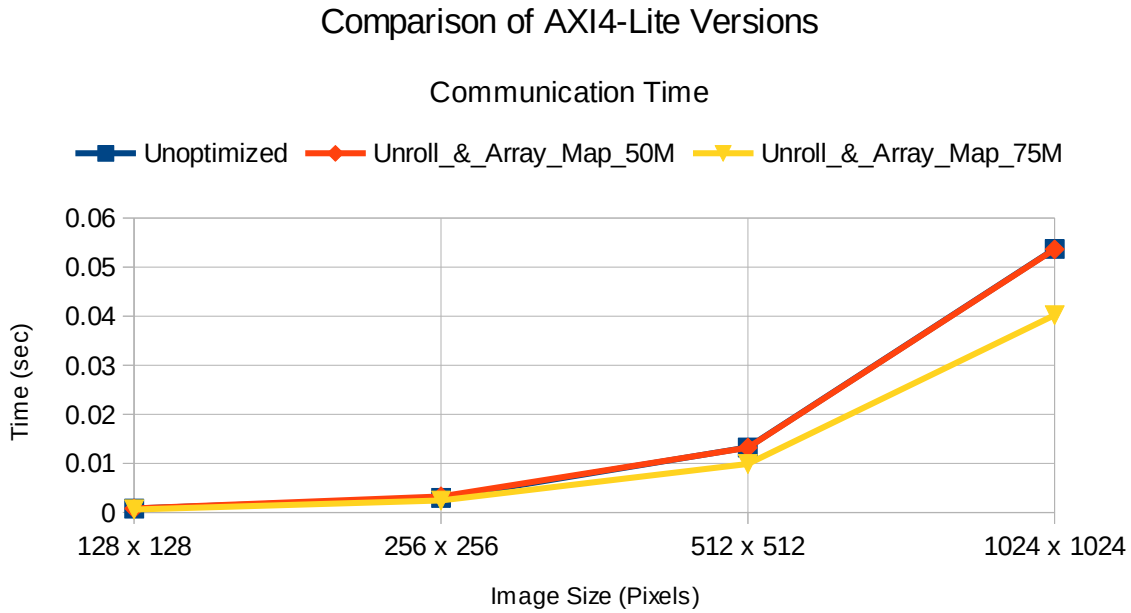


Figure 5.4: Communication Time for Different AXI4-Lite Implementations

5.4 Implementation of AXI4-Stream Version

We now proceed to an AXI4-Stream Implementation of the optimized unrolled and array mapped version of Harris_FindCorners IP. Considering that the elements of the input harrisData struct remain unaltered for all implementations, we proceeded to the integration of these data in the IP. Then, the only thing to have as an input would be the image which is streamed into the IP using the AXI4-Stream Protocol. Lets now proceed to the evaluation of the time measurements that were made during these implementations.

Image Size (Pixels)	Communication Time (s)	Computation Time (s)
128 x 128	0.0001018	0.012588
256 x 256	0.0004075	0.056099
512 x 512	0.001630	0.211797
1024 x 1024	0.006523	0.937865

Table 5.6: Time Measurements for AXI4-Stream Version

In the above measurements we clearly notice an incredible increment in bandwidth which now reaches values of up to 154.7 MB/s which is an incredible gain when compared to the previous implementations. In addition, the lack of a block-level protocol proves beneficial for the computation time as well as we may notice a slight decrement even though no additional optimizations where made to the algorithm.

5.5 Overall Comparison of HW Implementations

In this paragraph the HW Implementations of Harris & Stephens Corner Detection algorithm where examined. The different target device for which the specific implementation where developed introduced issues when moving the IP in another and, most significantly, smaller from the aspect of the device's available resources. In the following diagrams a comparison between communication and computation times is made. In addition the gain of bandwidth compared to the original, unoptized-50-MHz-clock implementation is made.

In the following charts it is made clear that the AXI4-Stream protocol is the best choice, offering not only the lowest communication times, and consequently highest bandwidth but also even a slight decrease in computation time, a fact which may have not been expected due to the lack of block-level protocols for the control of the device. The achieved bandwidth reached a climax of 154.7 MB/s when the AXI4-Stream Version was employed.

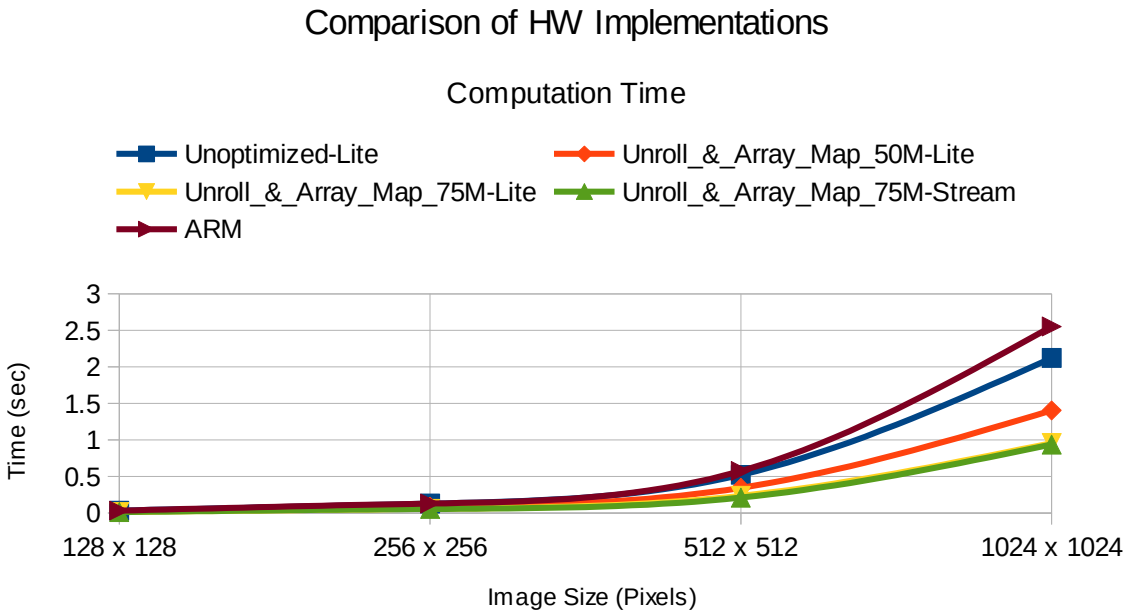


Figure 5.5: Computation Times for Different HW Implementations and ARM

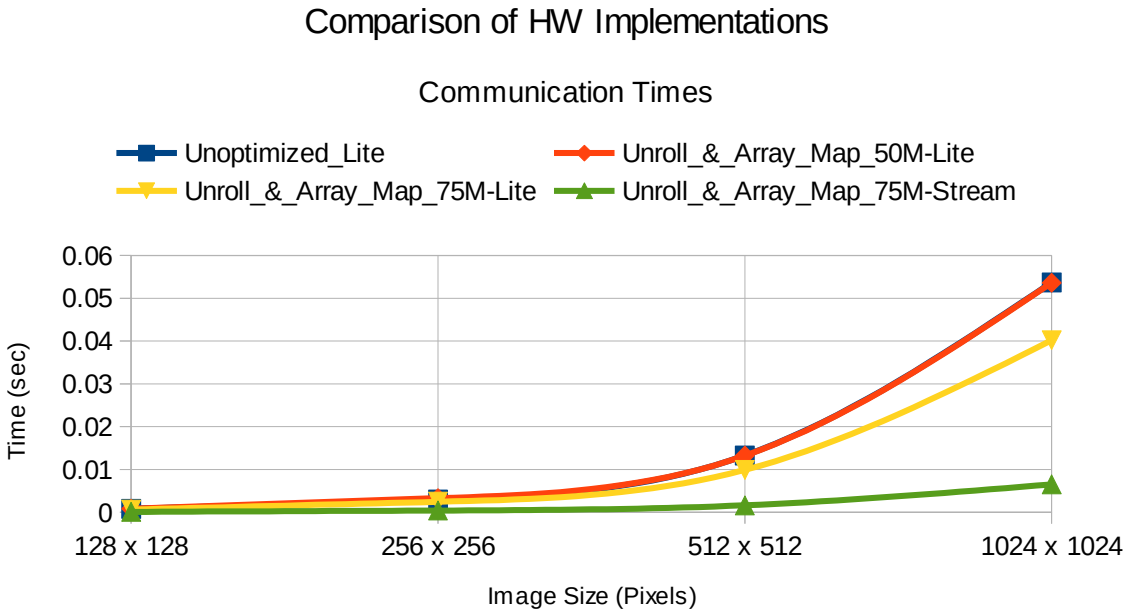


Figure 5.6: Communication Time for Different HW Implementations

This page is intentionally left blank.

Chapter 6

Evaluation of Work Flow on SVM Classifier

6.1 General Description of HW Implementations

The purpose of this chapter is to present and compare various hardware implementations of the Support Vector Machine classifier, mainly, from the aspect of communication between the PS and PL part of ZedBoard Zynq Evaluation and Development Board. High-Level Synthesis enabled us to produce Classify IPs with different communication interfaces. Particularly, AXI4 Slave Lite and AXI4 Stream Interfaces were utilized for the communication of the classify accelerator with the processing system. It should therefore be mentioned that the simplicity of the classifier code combined with the low utilization of device resources for each IP allowed us to experiment with the addition of more than one instance of the classifier accelerator and explore the multi-processing potentials that our target device offers. We implemented and explored three different hardware versions from the Pareto design space. The first is a HW original version of the algorithm. It has a relatively high execution latency combined with low demands of resources due to its simplicity. The second is a HW accelerated version with lower latency and higher resource demands, yet still low. The final HW version is the optimal one and has an extremely low latency, consequently combined with high utilization of the device. Of course, each HW version, based on its utilized resources, limits the possible alternative implementations. For example, if a specific HW configuration utilizes over 50% of the available resources, then it is impossible to add two instances of the specific HW in a system design.

The testing set for each of the implementations included 52291 test vectors which were read from a file. Time measurements were taken for the computation time per beat and for the time necessary for the test vectors to be transferred from the PS to the PL side. Additionally, the total transfer and total computation time were taken. The execution was repeated 10 times and the mean values were computed to eliminate potential mistakes. For comparison purposes, a software only implementation of the original classifier code, without any structural alterations was built and executed on the ARM® processing system of our target device. In the following table, the execution times of this implementation are presented. Obviously, for the software version of the classifier no communication time is measured.

	Communication Time (s)	Computation Time (s)
Per beat	-	0.002223635
Total	-	116.2761016

Table 6.1: Time measurements for SW version executed on ARM®

6.2 HW Original Version Implementations and Results

In the first approach to implementing the Classify IP, we employed a simple, not-accelerated version of the algorithm. No optimizations were made during the high-level synthesis of the hardware. We produced two different versions of the Classify IP. The first version includes an AXI4 Slave Lite Interface and the input, output and return values of the classify function are grouped in one bundle. In the second version of the Classify IP, AXI4 Stream Interfaces were used for transferring the input and output values. In the following table a comparison between the percentage of utilized resources is presented.

	BRAM_18K	DSP48E	FF	LUT
AXI4 Lite Util. (%)	25	20	3	11
AXI4 Stream Util. (%)	24	20	3	11

Table 6.2: Resource Utilization for the original HW implementation of the SVM classifier

As shown in the above table, the utilization of the device is almost identical for both AXI4 Slave Lite and AXI4 Stream Interfaces. However, a difference occurs in the utilization of BRAMs which derives from the fact that AXI4 Slave Lite Interface suffers from the need to instantiate the input and output ports of the classify IP, leading to an additional 1% in BRAM utilization. Given the above table, we proceeded to five different implementations of the SVM classifier on Zedboard by employing one or more instances of the classifier. The implementations are defined as 1-Lite, 2-Lite, 4-Lite, 1-Stream and 2-Stream. The Lite implementations include 1, 2 or 4 instances of the AXI4 Slave Lite version of while the Stream versions include 1 or 2 instances of the AXI4 Stream version of the classifier. Before proceeding to analysis of each version we present their final utilized ZedBoard resources.

	FF	LUT	Memory LUT	BRAM	DSP48	BUFG
1-Lite Ut.(%)	3	7	1	25	20	3
2-Lite Ut.(%)	5	14	2	50	41	3
4-Lite Ut.(%)	10	28	3	100	82	3
1-Stream Ut(%)	5	11	2	26	20	3
2-Stream Ut(%)	11	22	4	51	41	3

Table 6.3: Final Utilized Resources for HW Original ZedBoard Implementations

6.2.1 Original AXI4 Slave Lite Version with 1 Classify IP

The first approach to implementing the SVM classifier in ZedBoard Zynq Evaluation and Development Board was to employ one instance of the Classify IP and use the AXI4 Slave Lite interface for the communication of the processing system and the hardware (1-Lite Version). The userspace application is responsible for the initialization, input data transfer, output data collection and finalization of the device, to which the access is made through the Linux UIO driver that was discussed in a previous chapter. It should be noted that Vivado performs optimizations during the implementation phase of the design, leading to elimination of unused nets. Hence, the final resource utilization after synthesizing and implementing the design might be lower than the estimated during the high-level synthesis step of the implementation, a fact which is denoted in Table 6.2.

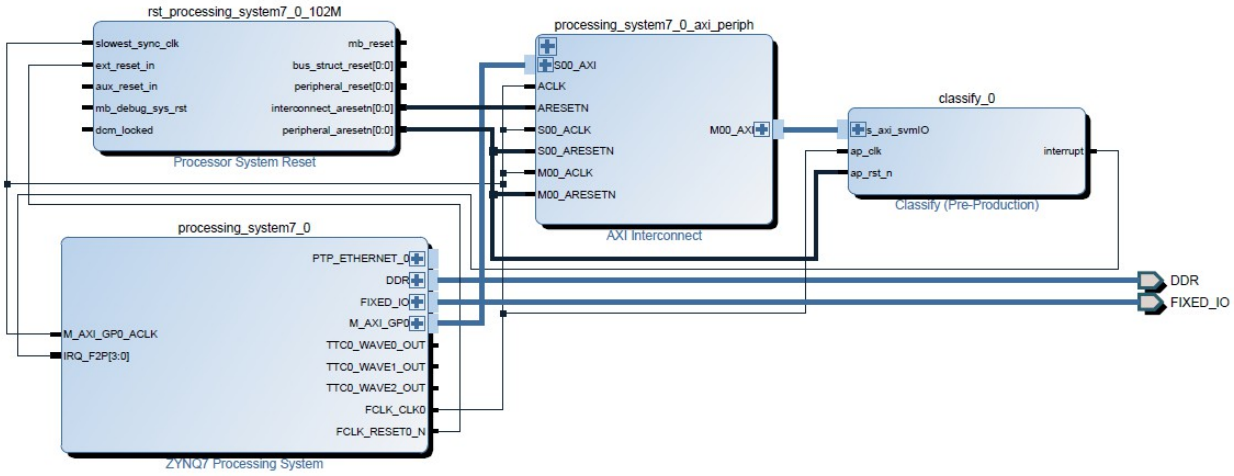


Figure 6.1: HW Original 1-Lite IP System Architecture

The above schematic of the implemented system architecture is constituted by four components. The ZYNQ7 Processing System, the Classify IP, an AXI Interconnect IP core and a Processor System Reset. The AXI Interconnect allows the ZYNQ7 Processing System to communicate through its AXI Master General Purpose port with the AXI Slave port of the Classify IP. It should be noted that only memory-mapped devices use the AXI Interconnect for communication and control purposes. The Processor System Reset is necessary for the operation of the whole system because the PS and PL parts of the device operate in different frequencies. Specifically, for this particular implementation a clock of 100MHz is used for the Classify IP core. In the following table the time measurements for this implementation are compared with the software version of the classifier. It can be observed that the not-optimized, original HW version of the SVM classifier presents a decelaration of 82% compared to the software only version. For our following implementations this HW version will be considered as a baseline as it is the simplest HW that can be created combined with the simplest interface, which is AXI4 Slave Lite.

	SW Version		HW Original Version	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	-	0.002223635	0.00000449943	0.004047181
Total	-	116.2761016	0.2352798	211.6311248

Table 6.4: Time Measurements for SW version and HW original 1-Lite version

6.2.2 Original AXI4 Slave Lite Version with 2 Classify IPs

The next approach to implementing the SVM classifier was to employ two instances of the Classify IP with AXI4 Slave Lite interfaces. The objective of this particular implementation was to take advantage of the multi-processing potentials that Zynq®-7000 offers. In the userspace application of this implementation two child processes are spawned Each of the child processes is granted the half beats of the testing set and controls its own Classify IP. Each child process is responsible for essential device initializations, input data transfers, output data collections and finalization of its corresponding HW accelerator. As already shown in Table 6.2, the final resource utilization of our target device is almost doubled in most cases, which, of course, is an expected outcome considering that the number of instantiated classifiers is doubled. The implemented system architecture is shown in Figure 6.2. We may notice two instances of the Classify IP connected with the ZYNQ7 Processing System through the same AXI Interconnect block. A Concat IP is an additional component which is utilized in order to connect the interrupt ports of the Classify IPs to ZYNQ7 Processing System which is able to support up to 16 interrupts.

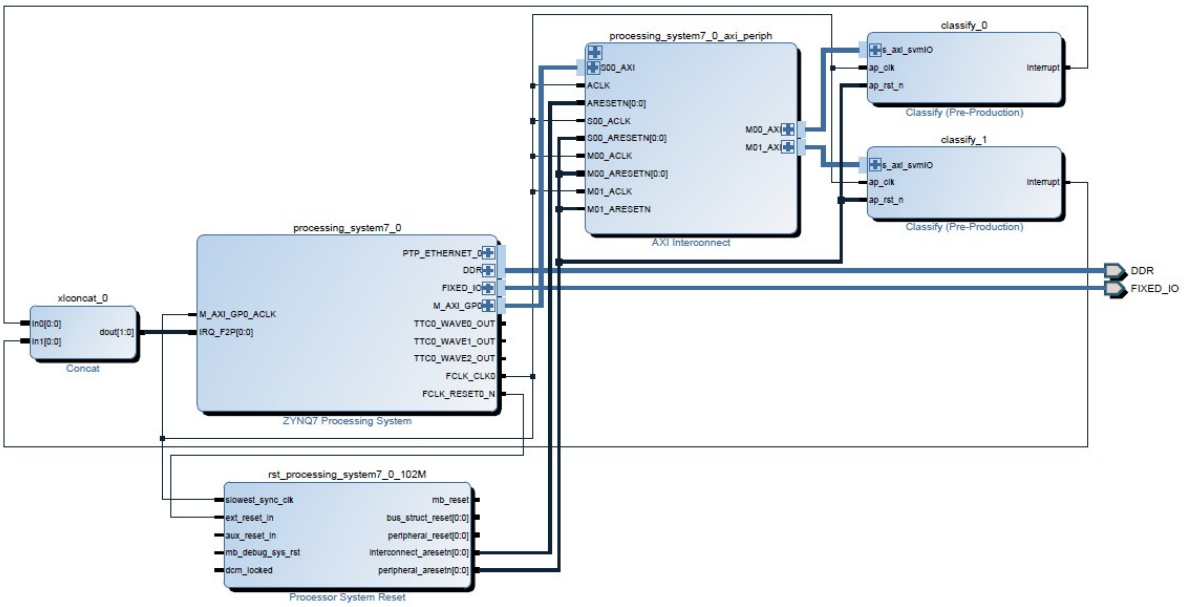


Figure 6.2: HW Original 2-Lite System Architecture

In the following table, the measurements for the necessary communication and computation time are compared with the HW original 1-Lite implementation.

	HW Original 1-Lite		HW Original 2-Lite	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.00000449943	0.004047181	0.0000034617	0.00388686
Total	0.2352798	211.6311248	0.090508	101.624144

Table 6.5: Time Measurements for HW Original 1-Lite and 2-Lite versions

We notice that the total computation and communication times measured for the classification of all 52291 beats are significantly lower than the ones measured for the HW original version. Actually, an over 50% decrease of the initial times is observed, leading to lower communication and computation times per beat. However, the new values are really close to the previous ones because it is not the processing time per beat that changes but the fact that the system is capable of processing two different beats simultaneously.

6.2.3 Original AXI4 Slave Lite Version with 4 Classify IPs

The third implementation on the SVM classifier follows a similar approach to the preceding one. In this implementation we employed four instances of the Classify IP. Four child processes are spawned and the operation is identical to preceding versions. The utilization of the device in this configuration is high as we may notice in Table 6.2 with the BRAMs and DSPs reaching utilizations of 100% and 82% correspondingly. The system architecture is presented below and is similar with previous ones.

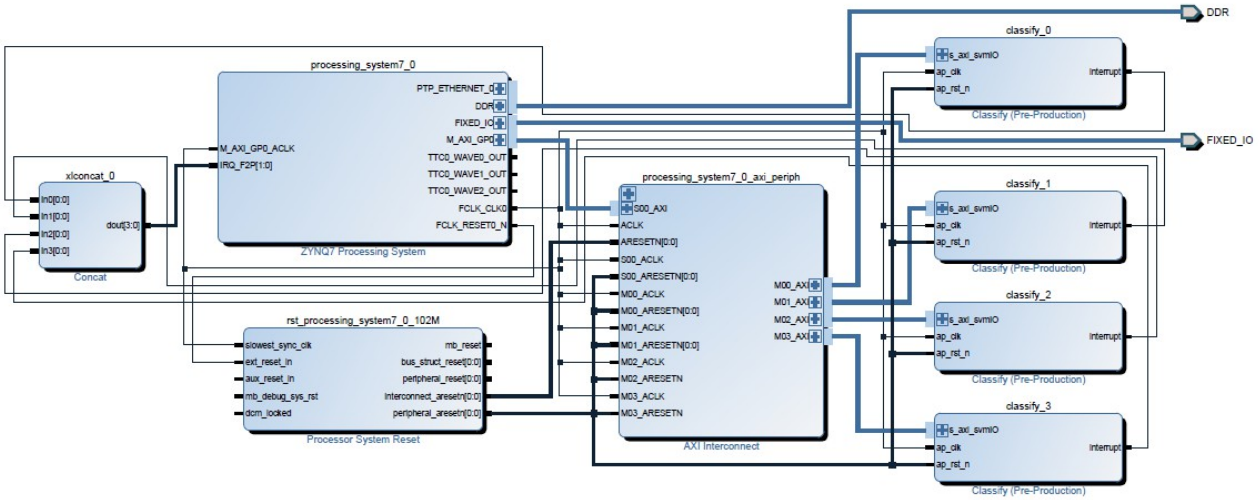


Figure 6.3: HW Original 4-Lite System Architecture

In the following table the the time results for the HW Original 4-Classify IP version are compared with the HW Original previously implemented versions.

	HW Original 1-Classify IP		HW Original 2-Classify IP		HW Original 4-Classify IP	
	Comm. Time (s)	Comp. Time (s)	Comm. Time (s)	Comp. Time (s)	Comm. Time (s)	Comp. Time (s)
Per Beat	0.000004499	0.00404718	0.000003461	0.00388686	0.000003609	0.00658754
Total	0.2352798	211.6311248	0.090508	101.624144	0.047185	86.117293

Table 6.6: Time Measurements for HW Original AXI4 Slave Lite Versions

As noticed, the total measured communication and computation times for the 4-Lite version are lower than both preceding ones. The 2-Lite and 4-Lite versions present latency gains of 52% and 62% correspondingly, while the latter utilizes an extremely high percentage of the available resources. A comparison between AXI4 Slave Lite implementations is shown in Figure 6.4. It should be noted that the computation time for the 4-Lite is increased compared to the 1-Lite version. The reason could be located in the PS part of our implemented system which includes a Dual-Core ARM® processor. In case of two processes handling their corresponding accelerators, each process can be executed solely on one of the processors. However, in case of four accelerators, a process might be stopped by the scheduler in a periodic or other manner for another one to be executed. Meanwhile, the output value of the accelerator might be ready, yet, it cannot be read because the process is stopped, leading to an increment in computation time per beat.

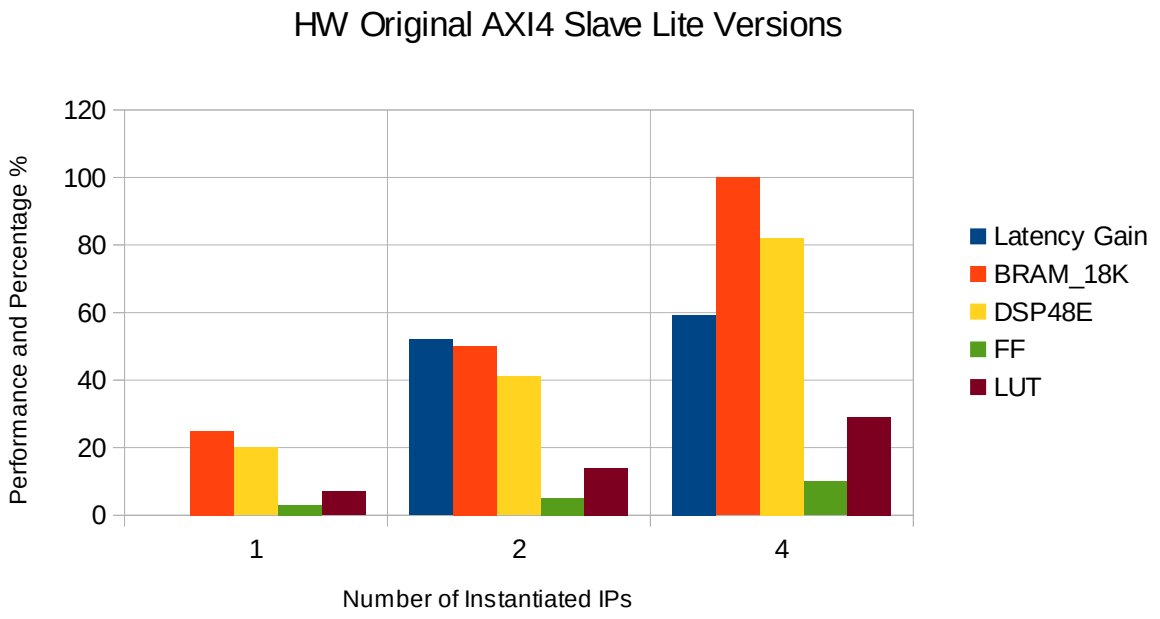


Figure 6.4: Performance and Gain of HW Original AXI4 Slave Lite Versions

The optimal choice out of the three HW Original AXI4 Slave Lite implementations from the aspect of latency gain and utilization of resources, as presented in the previous chart, would be the 2-Lite version as the gemination of utilized resources is accompanied by an over 50% latency gain in contrary to 4-Lite version in which the quadruplication of utilized resources does not induce analogous results. However, another area of interest beyond the latency gain is the bandwidth that can be reached in each of the versions. Hence, the following chart is presented.

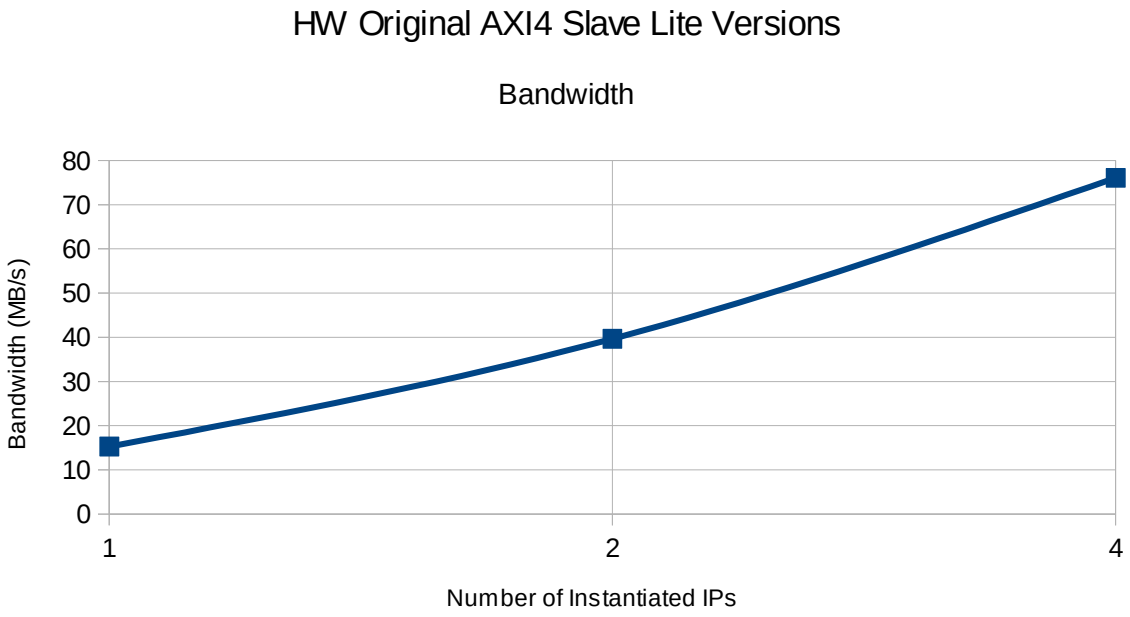


Figure 6.5: Bandwidth of HW Original AXI4 Slave Lite versions

As observed, the AXI4 Slave Lite implementations of the HW Original version reach a bandwidth of almost 80 MB/s. There is a clear increment in bandwidth that follows the addition of Classify IPs in contrary to total latency gain for which a similar observation cannot be made. Specifically, bandwidth is increased by 2.6 times for the 2-Lite version and by 5.2 times for the 4-Lite version. Yet, this increment is imperceptible as the total time needed for communication is less than 0.5% percent of the total execution time.

6.2.4 Original AXI4 Stream Version with 1 Classify IP

After implementing various AXI4 Slave Lite versions of the original HW we proceeded to implementations of AXI4 Stream versions. To begin with, we employed one instance of the HW Original AXI4 Stream Version. In this implementation the Classify IP is not memory-mapped and the data transfers are not performed by the PS part of the device. Instead an AXI Direct Memory Access (DMA) IP core is utilized and is responsible for input data transfers and output data collections. In order for the AXI DMA core to be controlled from

the userspace application, the original Xilinx DMA Linux driver was complemented by the zynq-xdma driver [https://github.com/bmartini/zynq-xdma], developed by Berin Martini [https://github.com/bmartini]. Minor alterations were made to the source code for the driver to fit in our situation. The userspace application is responsible for initializing the AXI DMA core and allocating the essential buffers for sending and receiving data. Then, it fills the buffer with the input data values and commands the AXI DMA to perform the transfer from memory to our accelerator. When the buffer is transferred, the AXI DMA waits for the computation to finish, so that the output buffer is written and transferred back to memory. In this particular implementation, the accelerator is responsible for collecting the necessary number of input values from the buffer. For instance, if a buffer of 180 float numbers is sent to the accelerator then it would extract the first 18 numbers, perform the computation and write the return value to the output buffer. Afterwards, it will extract the next 18 float numbers from the buffer and the process will repeat until the buffer is empty. If a buffer of length 18*N is sent, the length of the output buffer will be N.

In the following schematic we may notice the system architecture for the 1-Stream implementation. Firstly, as expected, we may notice the existence of an AXI DMA IP core. The ZYNQ7 Processing System and the AXI DMA communicate through an AXI Interconnect. The input values are fed to the Classify IP from the Master AXI Stream Port of the AXI DMA and the output values are fed back through the Slave AXI Stream Port of the AXI DMA. The Memory-Mapped to Stream (MM2S) and Stream to Memory-Mapped (S2MM) channels are connected to the Slave High Performance Ports of the PS through an AXI Memory Interconnect. A time comparison between this implementation and the HW Original 1-Lite version, which is considered as a baseline, is presented in Table 6.6

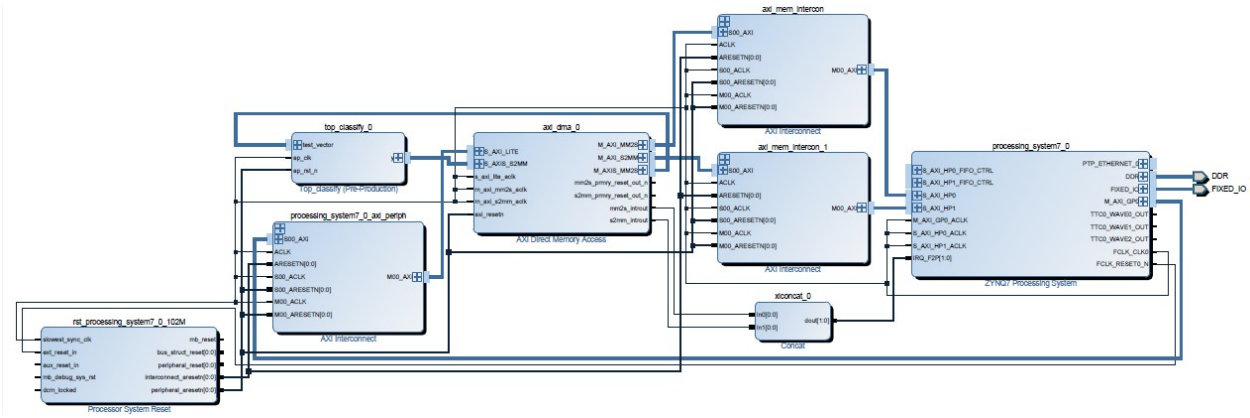


Figure 6.6: HW Original 1-Stream System Architecture

	HW Original 1-Lite		HW Original 1-Stream	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.00000449943	0.004047181	0.000000309078	0.00388195
Total	0.2352798	211.6311248	0.016162	202.995416

Table 6.7: Time Measurements for HW Original 1-Lite and 1-Stream Versions

In this implementation the execution was repeated for a variety of input and, consequently, output buffer sizes. Buffers of small sizes translated into a need for more transfers from the memory to the accelerator, thus, concluding to high communication times and low bandwidths. On the other hand, buffers of large sizes require a lower number of transfers, thus significantly reducing the communication times and increasing bandwidth. The following diagram presents the increment of bandwidth as a function of beats per transfer. It is noted that a beat is composed by 18 floating point numbers. This version's high computation latency combined with the operation of the Xilinx DMA driver did not allow us to try and send a buffer containing more than 768 beats or in other words 54 KB. As we may notice, the bandwidth varies from 1.24 MB/s up to 234.5 MB/s and increases as much as 100% when the buffer size is doubled.

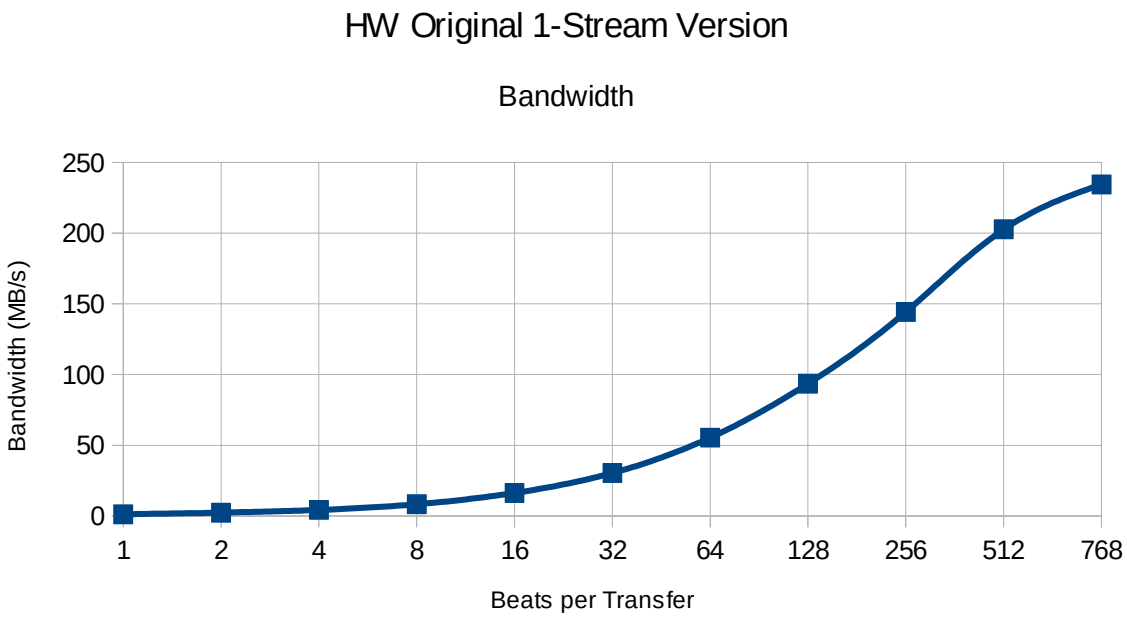


Figure 6.7: Bandwidth of HW Original 1-Stream Version for different buffer sizes

6.2.5 Original AXI4 Stream Version with 2 Classify IPs

The previous implementation includes one instance of the HW Original Classify IP and one instance of the AXI DMA IP core. In a manner similar to 2 and 4-Lite Versions, we proceeded to the addition of another instance of the Classify IP and another AXI DMA for the data transfers. The addition of the second AXI DMA block is essential because each AXI DMA block supports a channel dedicated to reads from memory and a channel dedicated to writes to memory. Two child processes are responsible for handling the DMA blocks. It should be mentioned that the userspace application does not have any control whatsoever on the Classify IP and are responsible for filling the input buffers and receiving the results. The control that exists in 1-Stream and 2-Stream implementations lies in the Classify IP, which internally extracts and collects the input data needed for a computation.

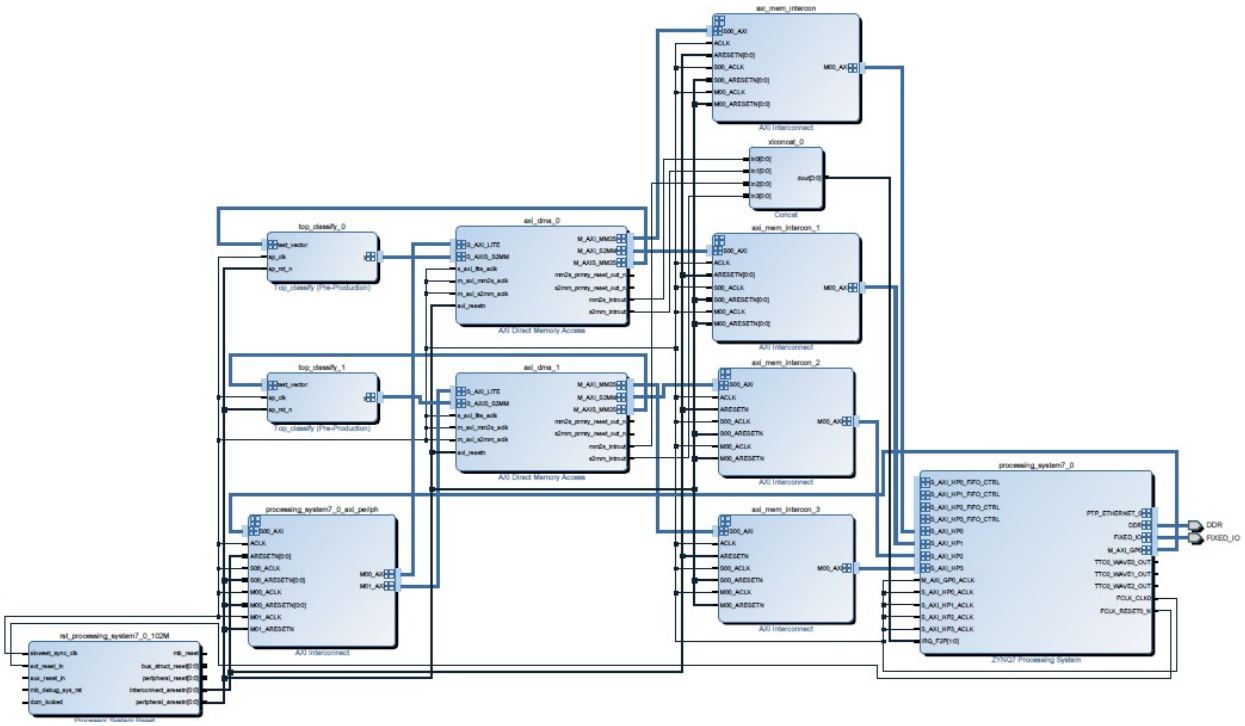


Figure 6.8: HW Original 2-Stream System Architecture

In Figure 6.8 a block design of the implemented system architecture is presented. It includes 2 instances of the AXI DMA IP core and 4 instances of the AXI Memory Interconnect IP. In this architecture we employed all available slave High Performance ports. The width of the High Performance ports is 64-bit, out of which, the 32 bits are destined for reads from memory and the other 32 bits are destined for writes. We implemented an extra version with the same number of Classify IP instances and the same number of AXI DMA blocks. Only 2 out of 4 High Performance ports were utilized. Both communication and computation times were identical in these implementations.

	HW Original 1-Stream		HW Original 2-Stream	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.000000309078	0.00388195	0.000000315733	0.00387699
Total	0.016162	202.995416	0.008255	101.36596

Table 6.8: Time Measurements for HW Original 1-Stream and 2-Stream Versions

The time measurements show a slight increment in communication time per beat and a slight decrement in computation time per beat. No significant changes are noticed in communication and computation times per beat, however, the system is able to process

twice as many beats in the same time, as does the 2-Lite implemented system. The execution was repeated for a variety of buffer sizes. The bandwidth varied from 2.9 MB/s and reached a value of 444.7 MB/s with approximately 222 MB/s per process and AXI DMA block. The bandwidth achieved for different number of beats per transfer is shown below.

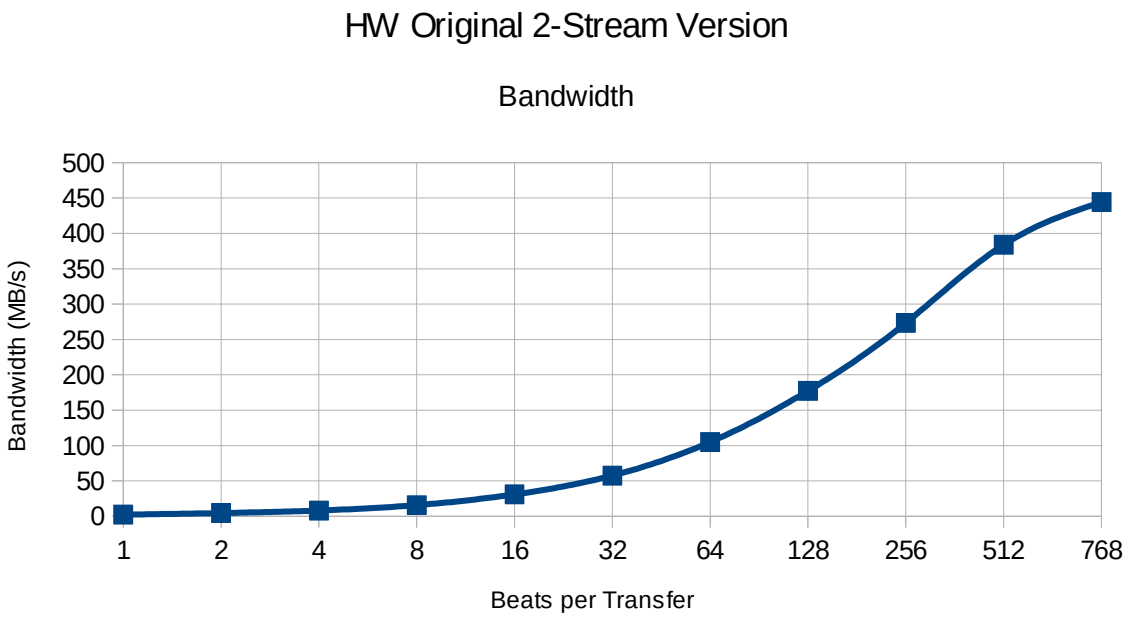


Figure 6.9: Bandwidth of HW Original 2-Stream Version for different buffer sizes

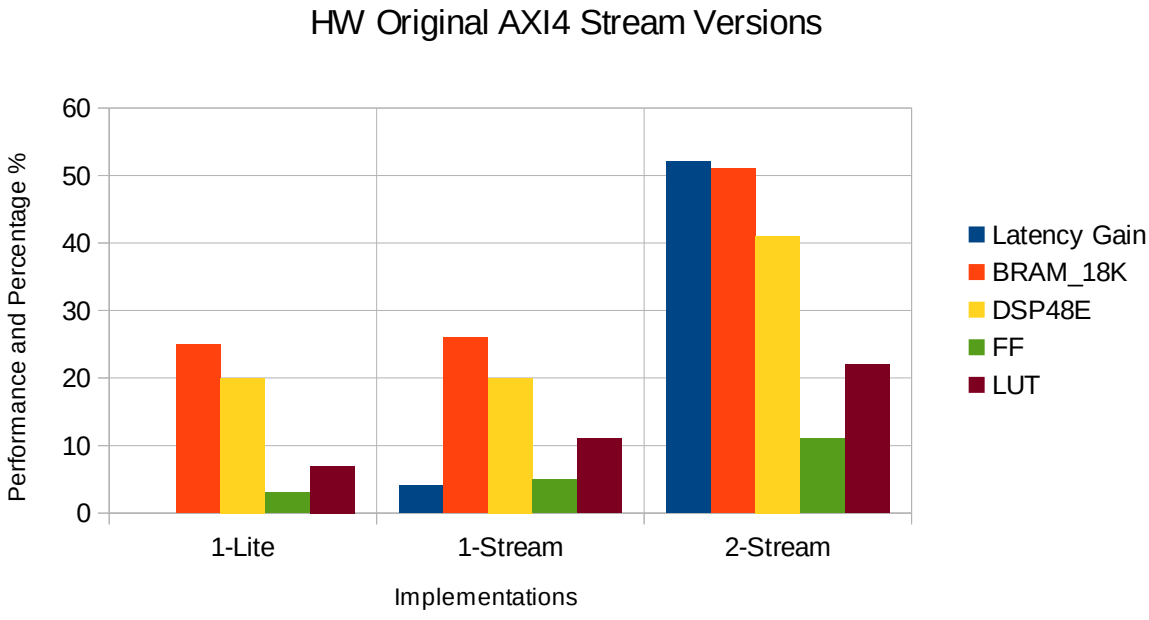


Figure 6.10: Performance and Gain for HW Original AXI4 Stream versions

As we may notice in Figure 6.10 a latency gain of 4.08% and 52.1% is measured for 1-Stream and 2-Stream versions respectively. It should be noticed that the gain of 1-Stream version might not be anticipated as the accelerators are almost identical. It is a fact that communication times are extremely lower compared to 1-Lite version. However, the communication times are not the principal components of latency. The measured improvements in computation times are due to our choice of different communication interfaces. Regarding the AXI4 Slave Lite interface, Vivado HLS automatically sets the interface to `ap_ctrl_hs`. This is a protocol which adds necessary signals to Classify IP so that it can be controlled from a processor. Among them are the ap_start and ap_return signals. It should be noted that the generated signals are exposed to memory and can be accessed and set by the processor through the Linux UIO Driver. In the AXI4 Lite versions of the original HW, the accelerator is memory-mapped and after using the `memcpy()` call to copy the input data to the intended device memory address, the userspace application sets the ap_start signal to 1 for a brief moment and then again to 0 in order to start the computation. The computation begins and when it is completed an ap_vld signal attached to the output of the accelerator is set to 1 and the output is read from its respective memory address. The `ap_ctrl_hs` protocol includes a function call handshake. On top of that, every time the userspace application triggers a computation, a very short initiation interval is required combined with an interval needed for the output of the accelerator to be valid and ready for reading. On the other hand, the AXI4 Stream versions of the classifier do not operate in the same manner. An `ap_ctrl_none` interface is manually set to the function so that ap_start and the rest of the signals are eliminated leading to elimination of the function call handshakes as well. The ap_ctrl_hs interface is not needed for the control of the device because the AXI4 Stream versions are designed in a manner that allows the Classify IP itself to control the incoming streams. No matter the size of input buffers, the accelerator counts and devides the input streams at every 18 values, which is the number of input values for a computation. This means that if more than 18 values are sent to the hardware, then only the first 18 will be used for the computation. If the remaining ones count to 18, then another computation is executed an another output value is written to the output buffer, otherwise, the accelerator does not perform another computation until the necessary number of input values is collected. This is a key element as the userspace application does not need to make any initializations or wait for a result. The only thing that it should do is fill the input buffer and read the results of the output buffer. Hence, a latency gain in computation time makes its appearance.

6.2.6 Comparison of HW Original Implementations

In this paragraph an overall comparison between HW Original implementations is made. To begin with, in Figure 6.11 we may notice the latency gain and utilization of the available target device resources for all implemented versions of the original HW. The 2-Lite and 2-Stream versions present significant latency gains without excessive utilization of the device. On the contrary, although the 4-Lite version presents slightly greater latency gains, the 100% and 82% utilization of BRAM and DSP blocks is prohibitive. It should be mentioned that though an attempt for a 4-Stream implementation was made, the BRAM utilization proved to be slightly above the available resources. Except for the latency gains and a

utilization comparison between the different implementations, we are also interested in the bandwidth that each implementation version achieves. It is noticed that the AXI4 Stream implementations which employ the DMA engine are able to achieve higher values of bandwidth (Figure 6.12).

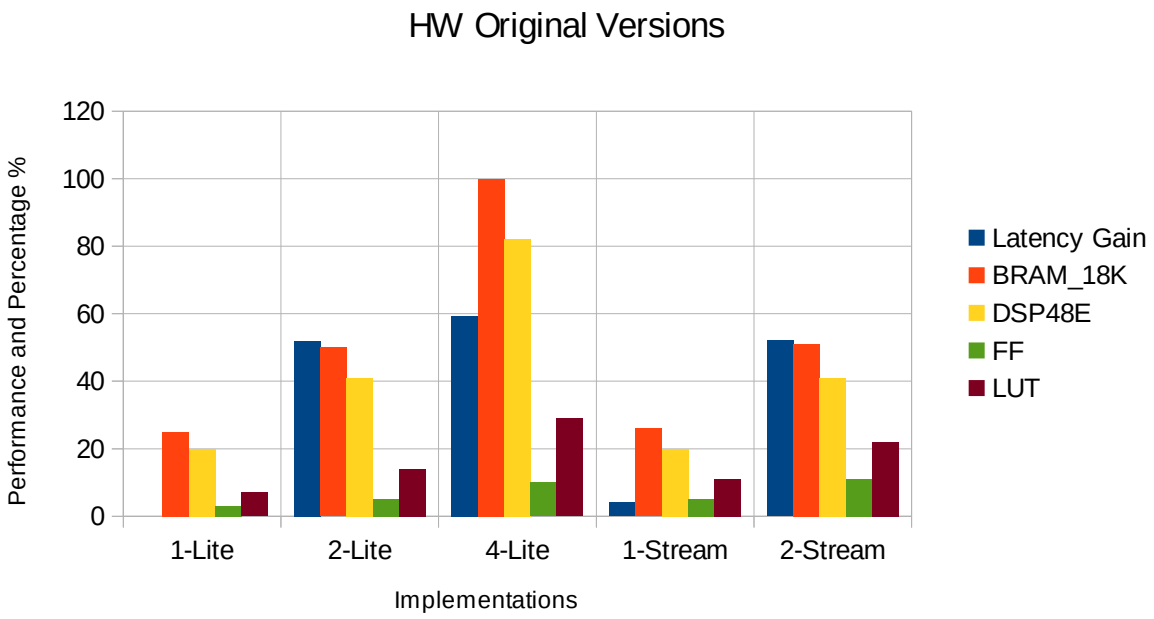


Figure 6.11: Performance and Gain for different HW Original Implementations

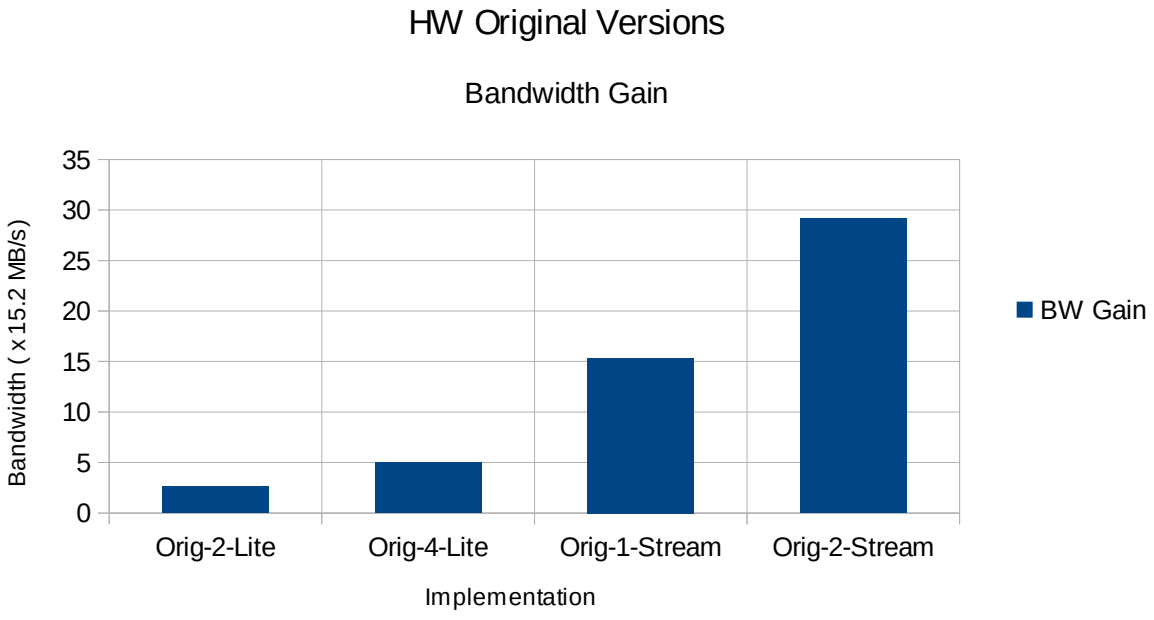


Figure 6.12: Bandwidth Gain for HW Original Implementations

6.3 HW Accelerated Implementations and Results

After implementing five different versions of original HW we proceed to various implementations of a HW Accelerated version picked from the Pareto design space. The chosen solution includes LOOP UNROLL, PIPELINE and ARRAY PARTITION directives to Vivado HLS. The estimated utilization of Vivado HLS, as seen in the table below, is slightly higher than the utilization of the original HW implementation while the estimated latency is 21416 cycles and the clock frequency is set to 25 MHz. The possible implementations on this accelerated HW, based on the utilized resources included 1-Lite, 2-Lite, 1-Stream and 2-Stream Version. However, Vivado Design Suite could not synthesize the 2-Lite version and the error stated that there are not enough RAMB blocks while the 2-Stream versions which also included two Classify IP Instances where synthesized and implemented regularly.

	BRAM_18K	DSP48E	FF	LUT
AXI4 Lite Util. (%)	27	26	3	16
AXI4 Stream Util. (%)	27	26	3	16

Table 6.9: Resource Utilization for the HW Accelerated Version of the SVM Classifier

The estimated through Vivado HLS utilized resources present no difference whatsoever between the AXI4 Slave Lite and AXI4 Stream versions in contrary to the HW Original version. We continue our analysis with a table of final utilized resources for all implementations of the HW Accelerated ZedBoard implementations.

	FF	LUT	Memory LUT	BRAM	DSP48	BUFG
1-Lite Ut. (%)	3	13	1	28	26	3
1-Stream Ut(%)	6	18	2	29	26	3
2-Stream Ut(%)	11	34	2	57	53	3

Table 6.10: Final Utilized Resources for HW Accelerated ZedBoard Implementation

6.3.1 Accelerated AXI4 Slave Lite Version

We proceeded to AXI4 Slave Lite implementation of the HW accelerated version of the classifier. The block design of the implementation is not presented as the system architecture of 1-Lite HW Accelerated version is identical to 1-Lite HW Original Version. Alterations are only made internally during the HLS process in the Classify IP. The utilization of the device (Table 6.9) is very close to HW Original 1-Lite version. In Table 6.13 the time measurements for the HW Accelerated Lite version are compared with our baseline. The HW Accelerated version offers a significant latency gain, which might not have been anticipated by the utilized resources. The communication time has increased as a consequence of the 25 MHz clock. The computation time per beat has dropped incredibly.

	HW Original 1-Lite		HW Accelerated Lite	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.00000449943	0.004047181	0.000007194272	0.0008590309
Total	0.2352798	211.6311248	0.376197	44.91959

Table 6.11: Time Measurements of HW Original 1-Lite and Accelerated Lite Version

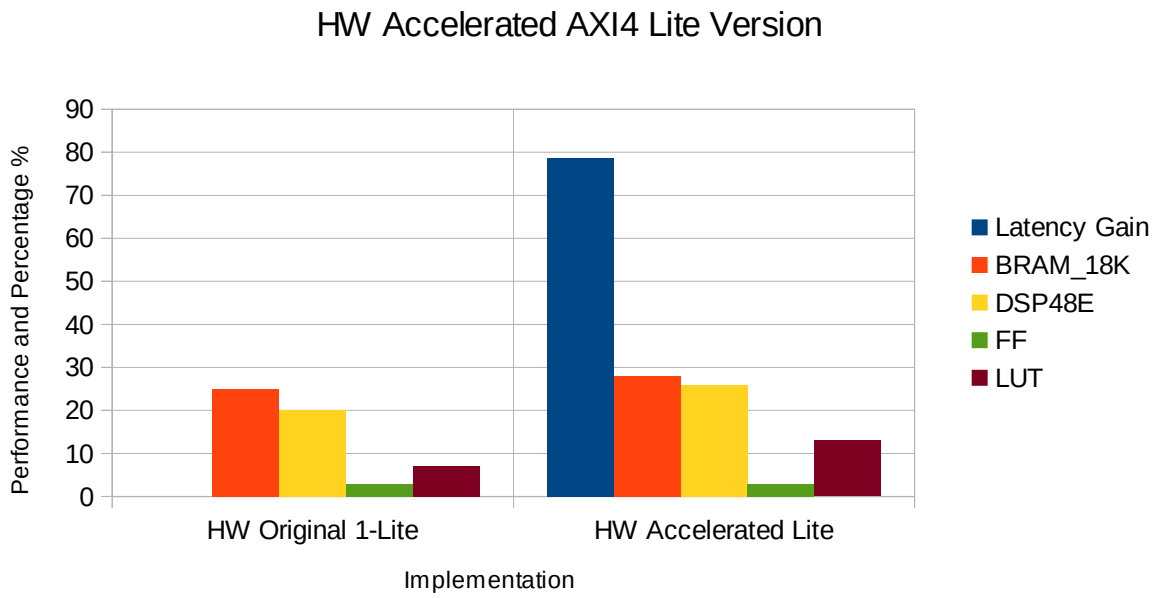


Figure 6.13: Performance and Gain for HW Original 1-Lite and Accelerated Lite

6.3.2 Accelerated AXI4 Stream Version with 1 Classify IP

In this version we employ an AXI4 Stream instance of the HW Accelerated version accompanied by an instance of the AXI DMA IP core. The system architecture is identical to this of the HW Original 1-Stream Version and it can be referred to in Figure 6.6. The final utilization of resources after design synthesis and implementation can be seen in the following table.

	HW Original 1-Lite		HW Accelerated 1-Stream	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.00000449943	0.004047181	0.000000832648	0.0008590744
Total	0.2352798	211.6311248	0.04354	44.921861

Table 6.12: Time Measurements for HW Original 1-Lite and Accelerated 1-Stream Versions

It can be observed that, similarly with the HW Accelerated Lite version of the classifier the total computation time has experienced an incredible decrease, hence, the computation time per beat has also declined. Furthermore, the communication time followed by the communication time per beat has decreased resulting in an increment, yet not sharp in bandwidth which now reaches 78.1 MB/s as it can be seen in the following diagram presenting the achieved bandwidth for the specific implementation as a function of the beats sent per transfer.

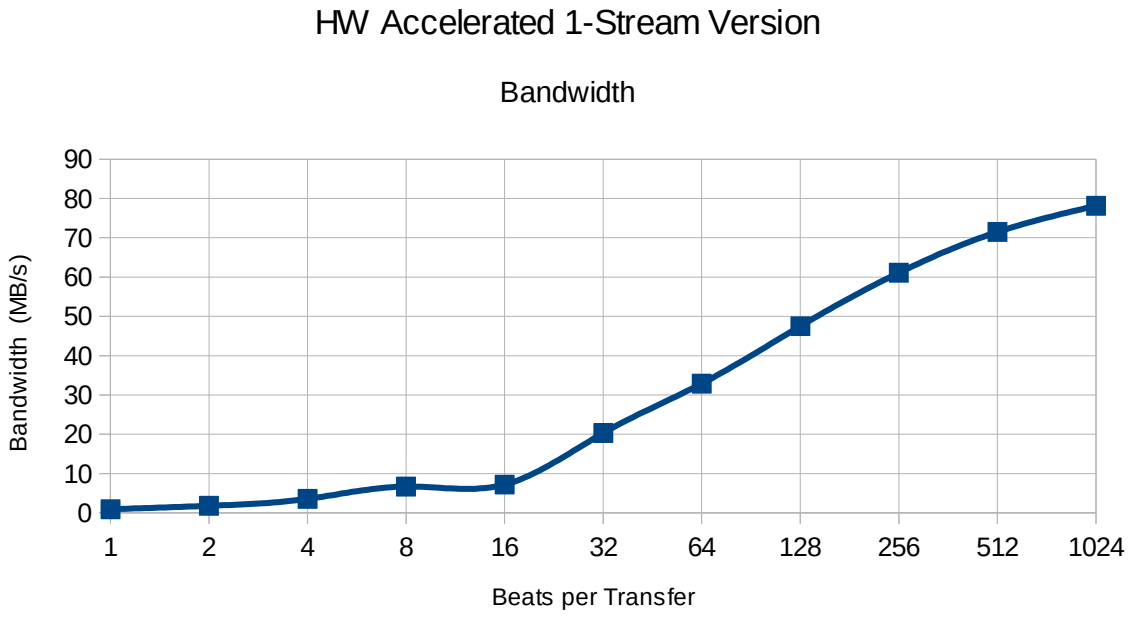


Figure 6.14: Bandwidth for HW Accelerated 1-Stream Version

6.3.3 Accelerated AXI4 Stream Version with 2 Classify IPs

The next implementation employs two instances of the HW Accelerated AXI4 Stream IP accompanied by two AXI DMA blocks. The system architecture is identical to HW Original 2-Stream implementation so it is not further discussed in this paragraph. The throughput of this implementation is doubled when compared to the preceding one, while computation time per beats remains the same with only a slight decrement. Following, we compare the 1 and 2-Stream implementations. The bandwidth is also presented as a function of the number of beats per transfer, or input buffer size. The achieved bandwidth was 157.4 MB/s.

	HW Accelerated 1-Stream		HW Accelerated 2-Stream	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.000000832648	0.0008590744	0.000000436403	0.0008410912
Total	0.04354	44.921861	0.02282	21.99075

Table 6.13: Time Measurements for HW Accelerated 1-Stream and 2-Stream Versions

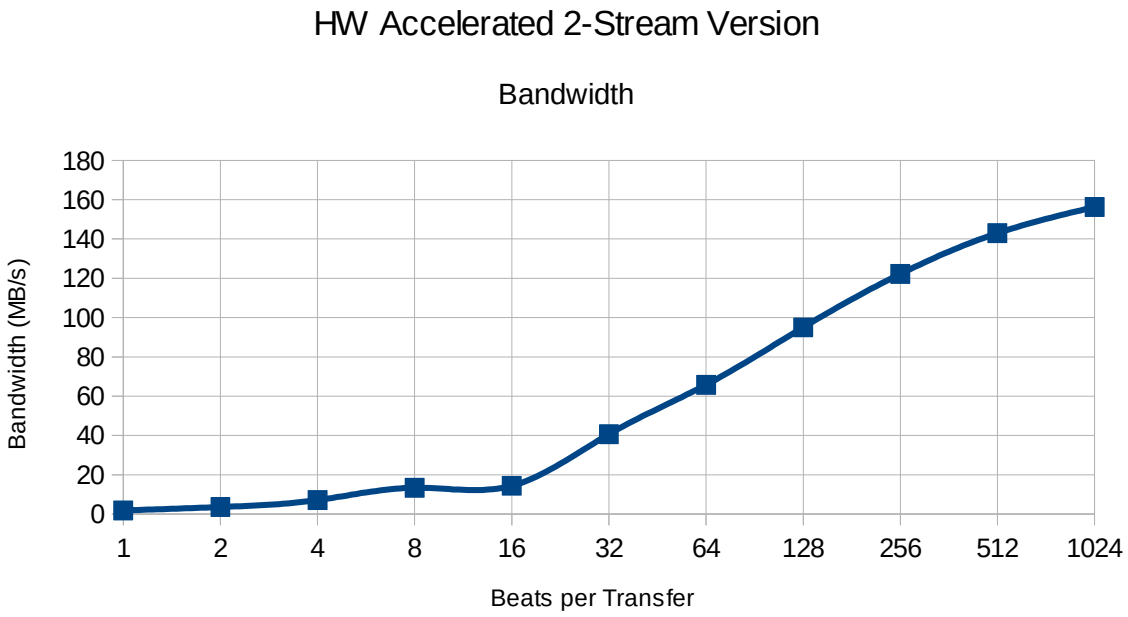


Figure 6.15: Bandwidth for HW Accelerated 2-Stream Version

An important technical fact should now be mentioned. In figure 6.9 we may see the achieved bandwidth for the HW Original 2-Stream version. A detail that should be observed is that the number of beats per transfer is limited to 768 for HW Original version while it reaches 1024 beats per transfer for the version we are currently examining. This derives from the Xilinx DMA Linux driver and the additional zynq-xdma driver. Specifically, the zynq-xdma driver, which, as already mentioned, is a high-level wrapper to communicate with the lower level Xilinx DMA driver. When sending an input buffer to our accelerator we have the option to wait for the input buffer to be transferred, wait for no transfer either input or output or wait for the output. The first option is to be chosen when the input buffer has a very large size and would not be transferred at once, or extremely fast as a small or medium sized buffer would. The second option is to wait neither for the input transfer to complete nor for the output. This would be an ideal option if the HW accelerator was able to process input data and instantly produce an output. A paradigm of this situation might be some kind of hardware that receives an input value, adds 1 to that value and then writes it to the output buffer. The previous job would be very fast and there might be no need to wait for the result. The third and final option would be to send the input buffer and wait for the output buffer which should be considered the safest and most obvious solution. In the HW Original Version buffers of all sizes up to 768 beats or 54 KB where regularly sent and the output was regularly received. A buffer of greater size would cause a timeout situation generated by the driver. On the contrary, in this implementation we were able to sent buffers of 72 KB or 1024 beats. The issues that occurred partially derive from the throughput of the accelerator and the operation principles of the driver. Particularly, the driver in fact waits for a very small interval, then it checks if the inward or outward transfer is completed. If not, then it refreshes the time but only to a certain multiple of the intervals, hence leading to the above mentioned issues.

6.3.4 Comparison of HW Accelerated Implementations

An overall comparison of different HW Accelerated implementations is following. With Lite and 1-Stream versions it is possible to achieve a high latency gain up to 80% accompanied by a very small increment in utilization of the device or 90% latency gain with 2-Stream Version and a double utilization of resources.

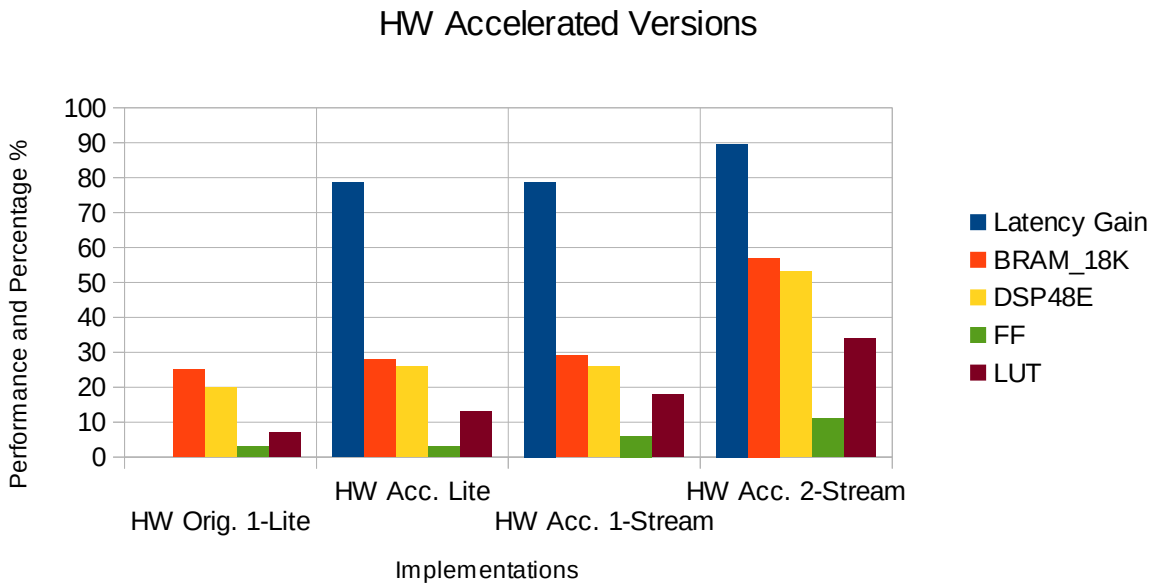


Figure 6.16: Performance and Gain for Different HW Accelerated Versions

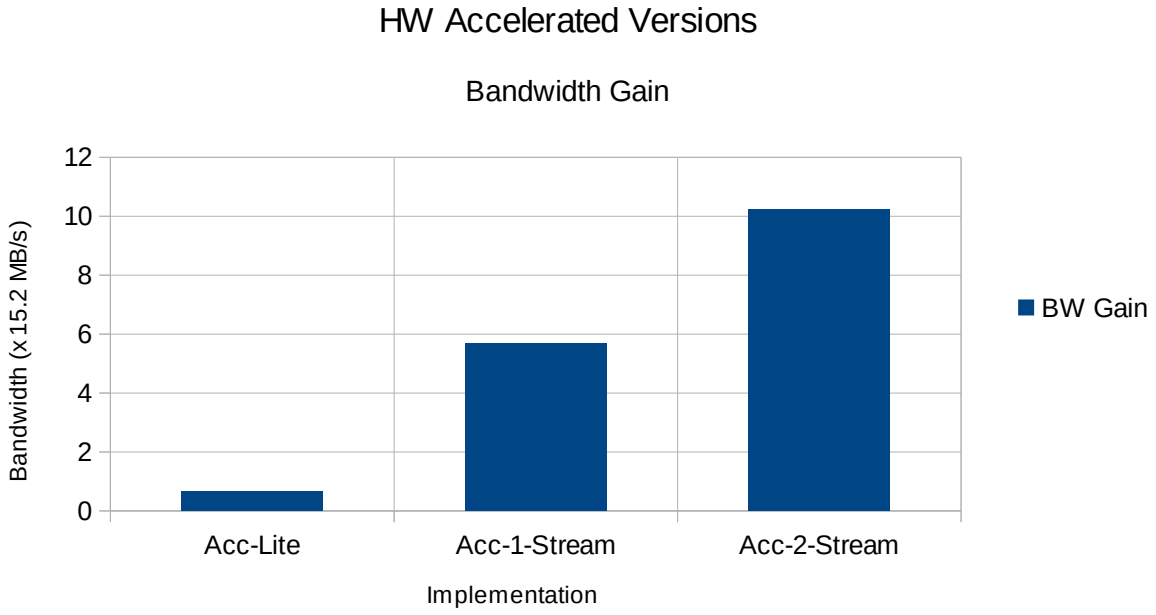


Figure 6.17: Bandwidth Gain for HW Accelerated Versions

6.4 HW Optimal Implementations and Results

The final choice for our Support Vector Machine classifier implementation is the HW Optimal from the Pareto design space. This specific implementation involves a full manual unroll of the inner loop of the original code and the addition of PIPELINE and ARRAY RESHAPE directives to Vivado HLS. It is expected to present the higher latency gain of all alternative HW versions. It should be noted that the clock for this implementations is set to 25 MHz. In the following table, the estimated HLS utilization is depicted.

	BRAM_18K	DSP48E	FF	LUT
AXI4 Lite Util. (%)	24	75	12	48
AXI4 Stream Util. (%)	24	75	12	46

Table 6.14: Resource Utilization for the Optimal HW implementation of the SVM classifier

We may notice that most components of the device are utilized with a percentage less than 50% except for DSP blocks where a 75% utilization makes its appearance. As already mentioned the accelerator in this version is pipelined which leads to the utilization of extra components and especially DSP blocks and LUTs for the creation of the hardware. Concerning different communication interfaces the utilization is almost identical with only a 2% difference in LUTs. Lets now proceed to the implemented versions. Obviously, the 75% utilization of DSP blocks is a constraint for different implementations, an option which existed in previous versions of the HW. So, the HW optimal implementation alternatives are an 1-Lite and 1-Stream where only an instance of the Classify IP is employed.

	FF	LUT	Memory LUT	BRAM	DSP48	BUFG
Lite Util. (%)	4	28	1	23	76	3
Stream Util. (%)	7	30	1	24	76	3

Table 6.15: Resource Utilization for HW Optimal Lite ZedBoard Implementation

6.4.1 Optimal AXI4 Slave Lite Version

The AXI4 Slave Lite system implementations are discussed extensively in the previous paragraphs. In this version we employ a single Classify IP and implement the system architecture of Figure 6.1. As we may notice in Table 6.15 there is an incredible decrease in total and per beat computation time. On the other hand, the 25 MHz clock that is used has increased total and per beat communication time. In fact, communication time presents an increase of 145% while computation time presents a decrease of 98.7%. Even with an increase in communication time, this particular implementation offers a total latency gain of 98.4%.

	HW Original 1-Lite		HW Optimal Lite	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.00000449943	0.004047181	0.000011064301	0.0000521259
Total	0.2352798	211.6311248	0.5785634	2.7257132

Table 6.15: Time Measurements for HW Original 1-Lite and Optimal Lite Version

6.4.2 Optimal AXI4 Stream Version

We now proceed to the final implementation of the Support Vector Machine classifier. This includes an AXI4 Stream implementation of the HW Optimal version of the code. An instance of the Classify IP is accompanied by an instance of the AXI DMA IP core. As we mentioned before, in the Lite implementation of Optimal HW, communication time is a principal component of total latency compared to former implementations. The HW Optimal Stream version not only copes with this issue but also slightly reduces computation time due the differences in the employed interfaces, as mentioned in paragraph 6.2.5.

	HW Original 1-Lite		HW Optimal Stream	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.00000449943	0.004047181	0.000000808647	0.0000502549
Total	0.2352798	211.6311248	0.042285	2.627881

Table 6.16: Time Measurements for HW Original 1-Lite and Optimal Stream Version

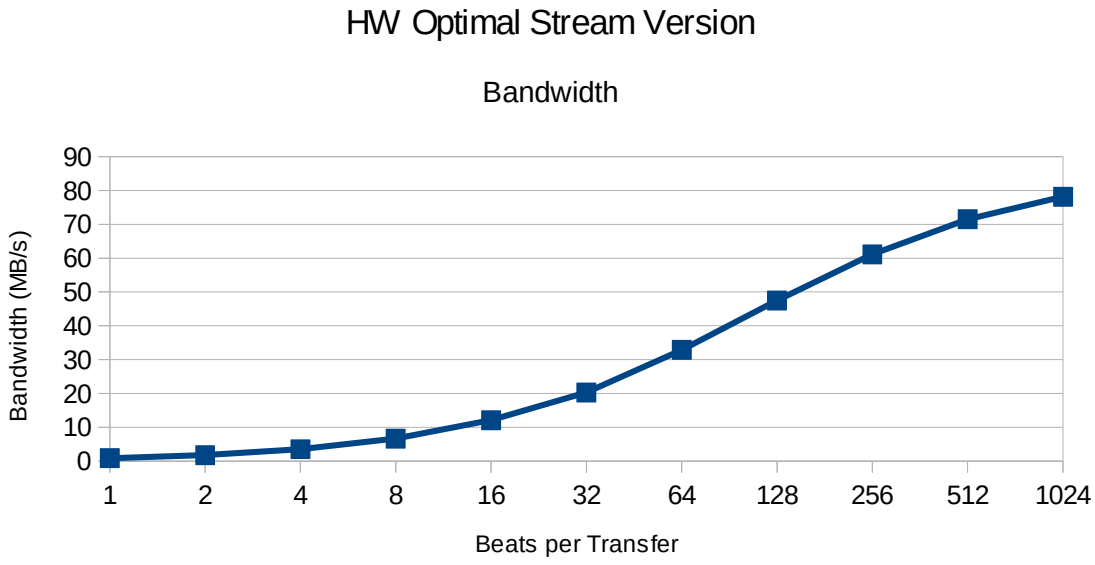


Figure 6.18: Bandwidth for HW Optimal Stream Version

6.4.3 Comparison of HW Optimal Implementations

In this paragraph a comparison and evaluation of HW Optimal versions is made. We notice that both Lite and Stream versions offer an extremely high latency gain compared to the HW Original version, reaching a value of almost 99%. The utilization of the device is low in general except for DSP blocks where a percentage of 75% is reached. Another instance of the Classify IP cannot be added to the system architecture, thus, leading to solely single-accelerator solutions. This translates to the fact that two heart beats cannot be processed at the same time in contrary to former 2-Lite, 4-Lite and 2-Stream implementations. However the computation time per beat is extremely low making it possible for a large number of beats be processed in a unit of time. For a real system this would translate to many processes being able to connect to the implemented system and send beats for classification with the accelerator being a critical part of the system which get locked during a computation and then unlocked and assigned to a different, thus implementing a resource sharing between different users.

	HW Optimal Lite		HW Optimal Stream	
	Communication Time (s)	Computation Time (s)	Communication Time (s)	Computation Time (s)
Per beat	0.000011064301	0.0000521259	0.000000808647	0.0000502549
Total	0.5785634	2.7257132	0.042285	2.627881

Table 6.17: Time Measurements for HW Optimal Versions

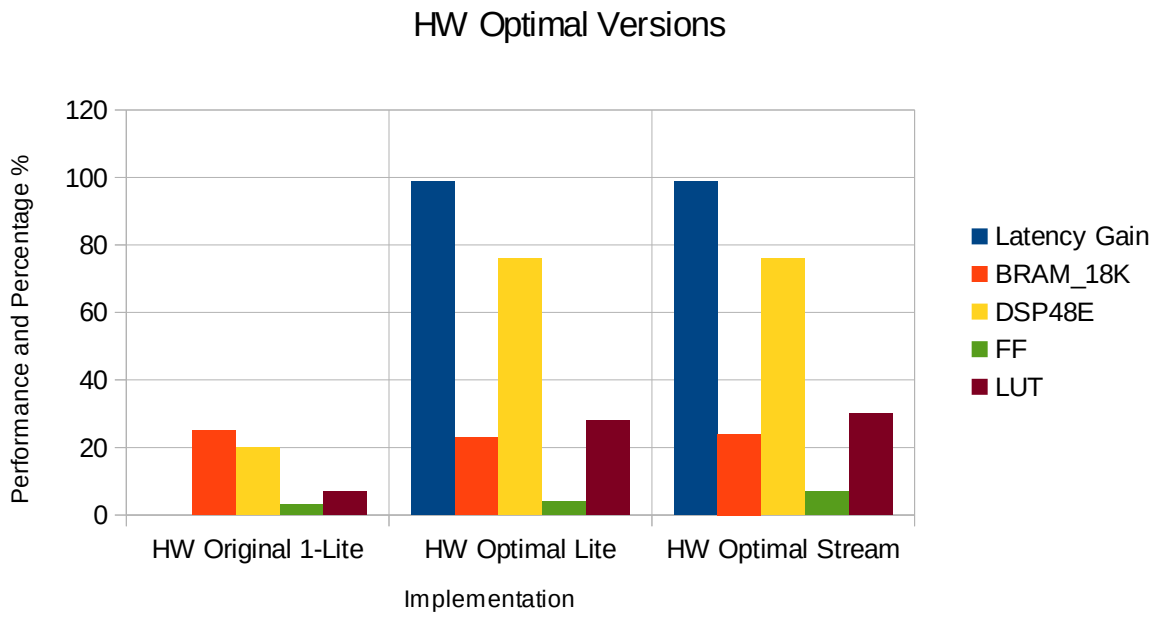


Figure 6.19: Performance and Gain for different HW Optimal Versions

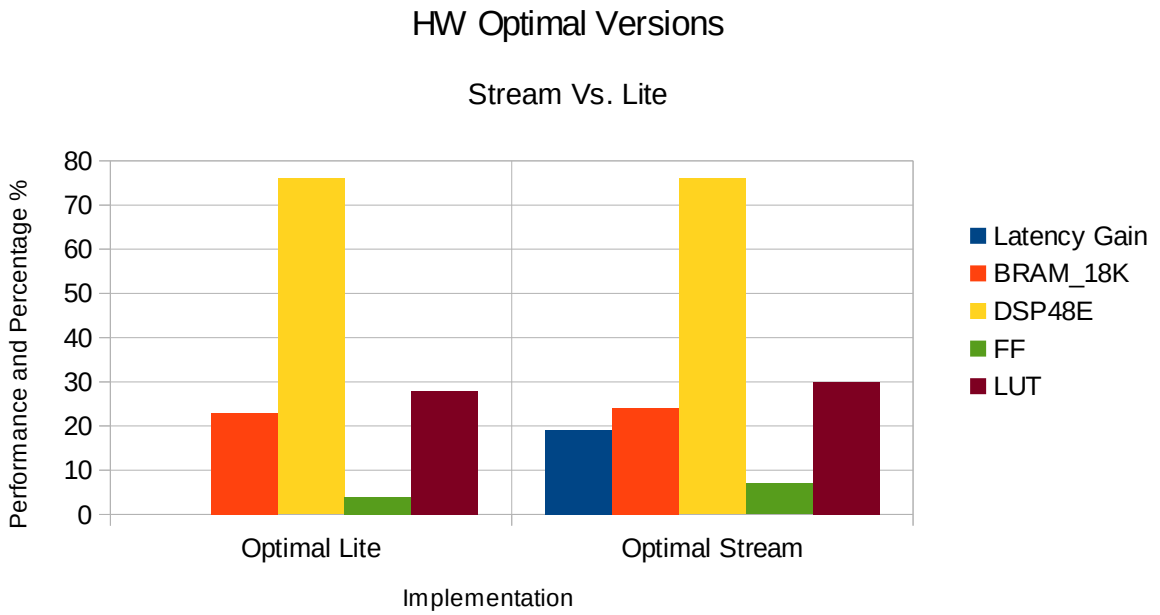


Figure 6.20: Performance and Gain of HW Optimal Stream vs. Optimal Lite version

Concerning the bandwidth of the HW Optimal implementations, an observation of loss of bandwidth was made in the Lite version, which is definitely due to the clock frequency of 25 MHz. An additional reason might be the use of ARRAY_RESHAPE directive for the input test vector. The directive technically breaks the input test vector in 18 parts, as many as the input values. This is a strategy which reduces latency because not the whole test vector must be loaded for the execution of a computation which needs only one of the values. However, the test vector is not necessarily stored in successive memory addresses and this might introduce an increment in communication time when copying the input values to the device memory. Given the above facts, the Lite implementation offered us a bandwidth of 0.4 times the HW Original 1-Lite bandwidth. Meanwhile, the Stream version increased the previous value to 5.6 times the original bandwidth.

6.5 Overall Comparison of HW Implementations

In this paragraph an overall comparison of all HW implementations is made. We present diagrams concerning latency gains of different versions, bandwidth gain and throughput. Moreover, a comparison between the impact of communication times in different implementations is made. To begin with, in Figure 6.20 a comparison of utilization and latency gain between all HW implementations is shown.

The HW Original Versions clearly offer the lowest possible utilization, at least when we refer to the 1-Stream and 1-Lite versions and are accompanied by high latency. When we proceeded to 2-Lite and 2-Stream implementations we acquired latency gains of over 50% compared to 1-Lite and 1-Stream versions and the utilization of the device was doubled. This was an expected outcome and no irregularities occurred. Then, the 4-Lite version was

implemented which lead to a 100% utilization of BRAMs and also high utilizations of the rest available components and resources. Theoretically, this implementation should have offered as a 75% latency gain, however this was not the fact. A latency gain of 60% occurred which was irregular but seems valid if we take the scheduling of the Linux operating system into account. The device disposes a dual-core processing system and scheduling issues occurred, thus leading to a low latency gain when compared to the HW Original 2-Lite version. A 4-Stream implementation was impossible as it required slightly more than the available BRAM blocks.

After the HW Original implementations, an optimized accelerated, yet not optimal, version of the classifier was employed. The utilization of the device obtained remained pretty low with a very satisfactory latency gain which reached 80% for the Lite and 1-Stream versions and 90% for 2-Stream version. The low utilization of all components combined with the high latency gains were compensatory. The only issue that occurred emerged during the synthesis phase of 2-Lite version in Vivado Design Suite where an error explaining the scarcity of RAMB blocks made its appearance, even though the respective 2-Stream version was successfully synthesized and implemented. We proceeded without implementing the 2-Lite version and recorded the error that occurred.

The HW Optimal versions clearly offer the highest latency gains while utilizing the 75% of DSP blocks. The utilization of the rest available components are definitely much lower. From the aspect of latency gain the two implementations are almost identical, however, if a comparison between them is made we will notice that the total computation times differ as much as 0.63 seconds or in terms of latency, the Stream version offers a 19% gain when compared to the Optimal Lite one, a result which emerges from the incredibly high gain in computation time.

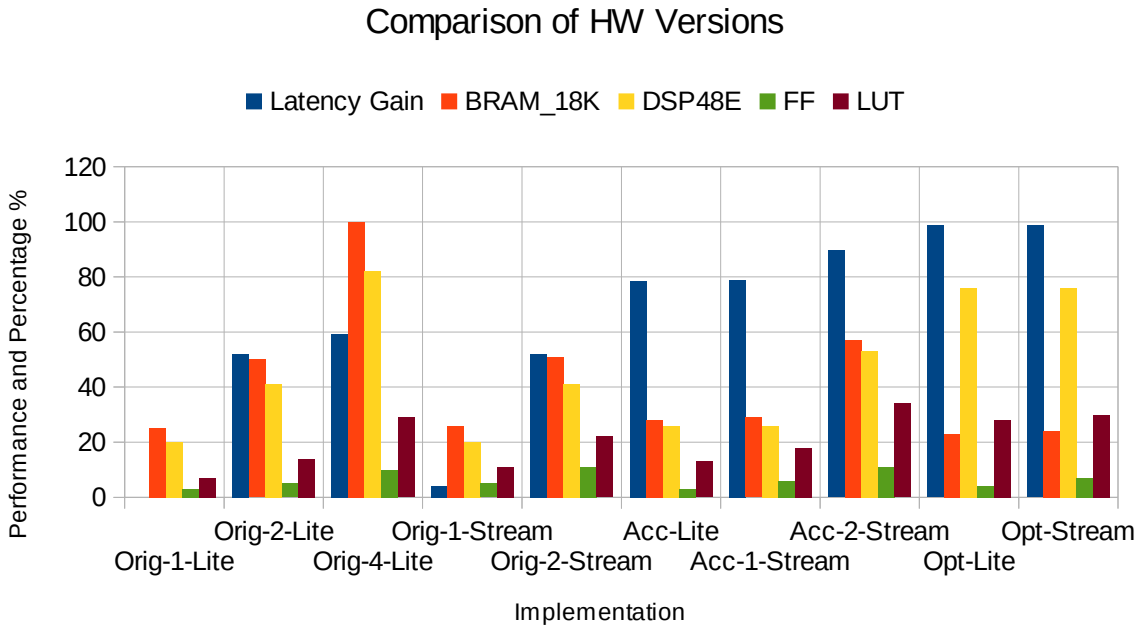


Figure 6.21: Performance and Gain for different HW versions

After latency gains, an evaluation of the achieved bandwidth of each implementation will be made but this time we are going to compare both bandwidth and throughput of our implementations. It is a fact that the achieved values of bandwidth were quite satisfactory and if the achieved throughput was even a little close to them we would be taking about an extremely fast system as a whole. However this is not the case as we will notice in the following diagrams which visualize the huge gap between bandwidth and throughput.

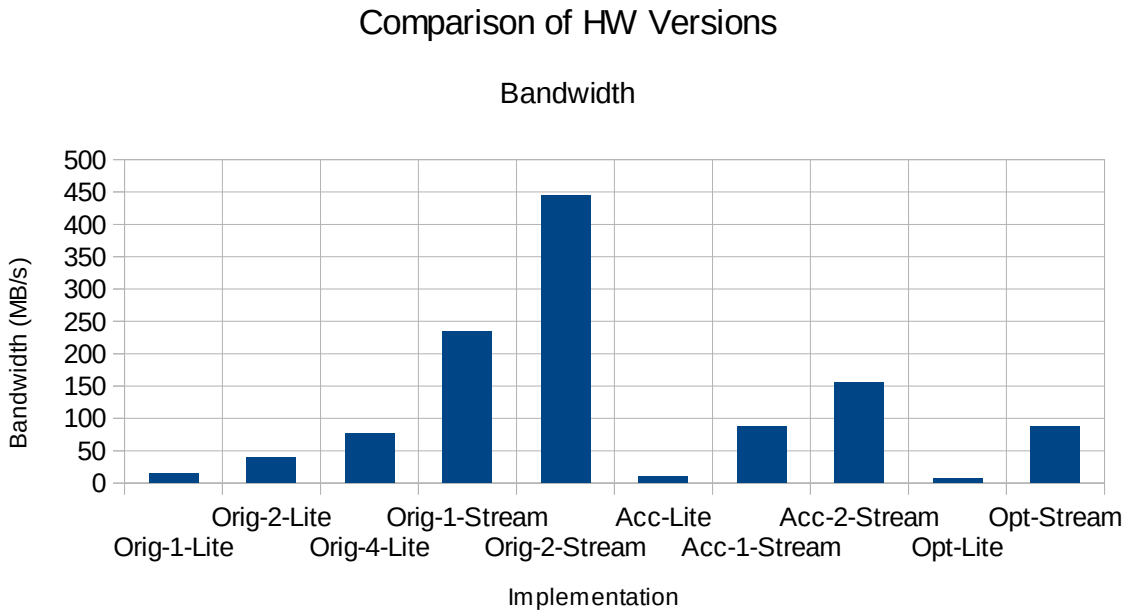


Figure 6.22: Bandwidth for different HW versions

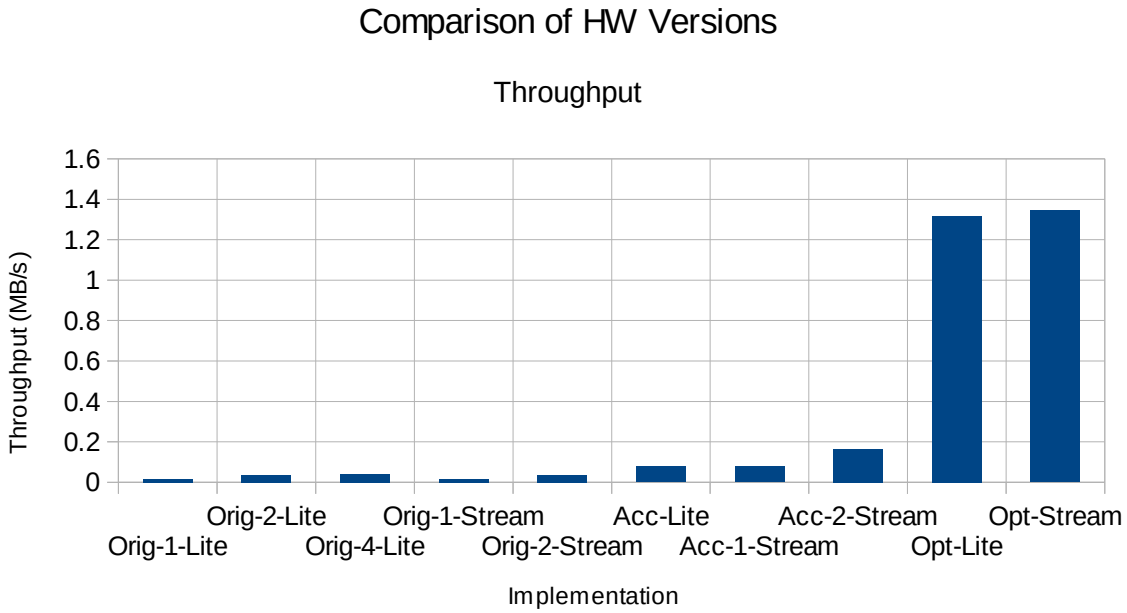


Figure 6.23: Throughput for different HW Versions

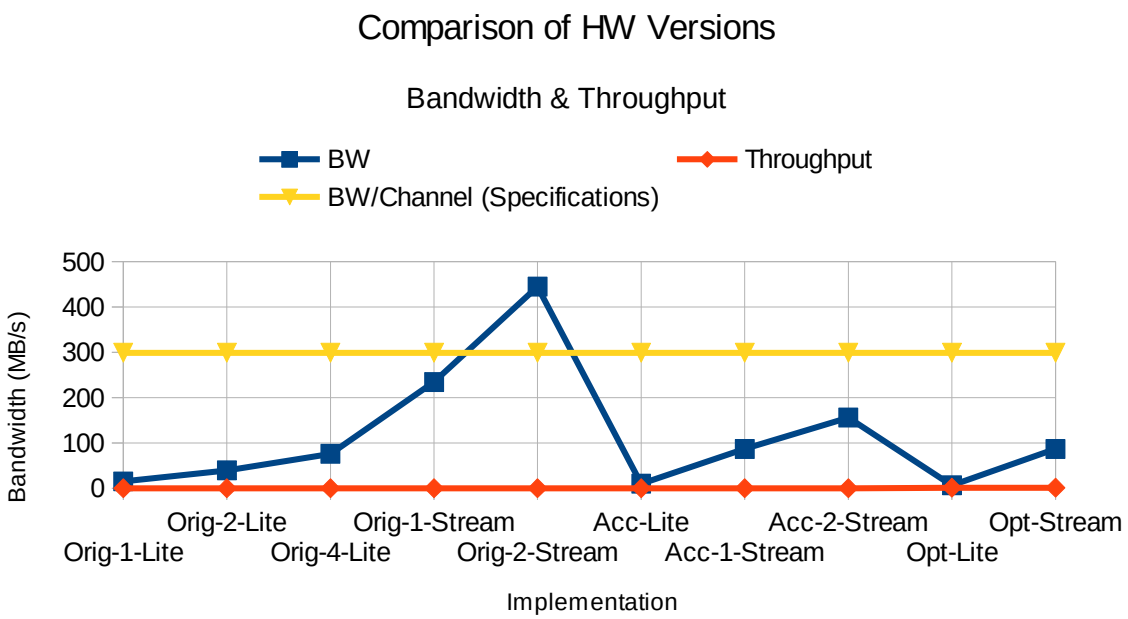


Figure 6.24: Bandwidth and Throughput for different HW Implementations

An additional comment on the impact of communication time in total execution time should be made. In most versions of the implemented SVM classifier the communication is an imperceptible component. The only version in which communication time plays an important role is the HW Optimal Lite version. For reference purposes two pie charts are presented to show the average and worst version from the aspect of communication.

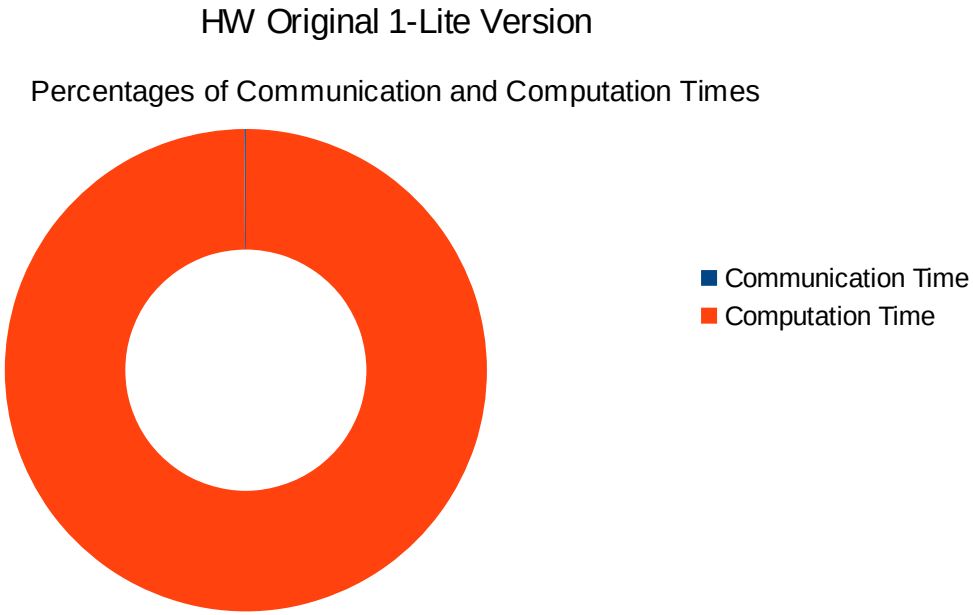


Figure 6.25: Communication and Computation Times for HW Original 1-Lite Version

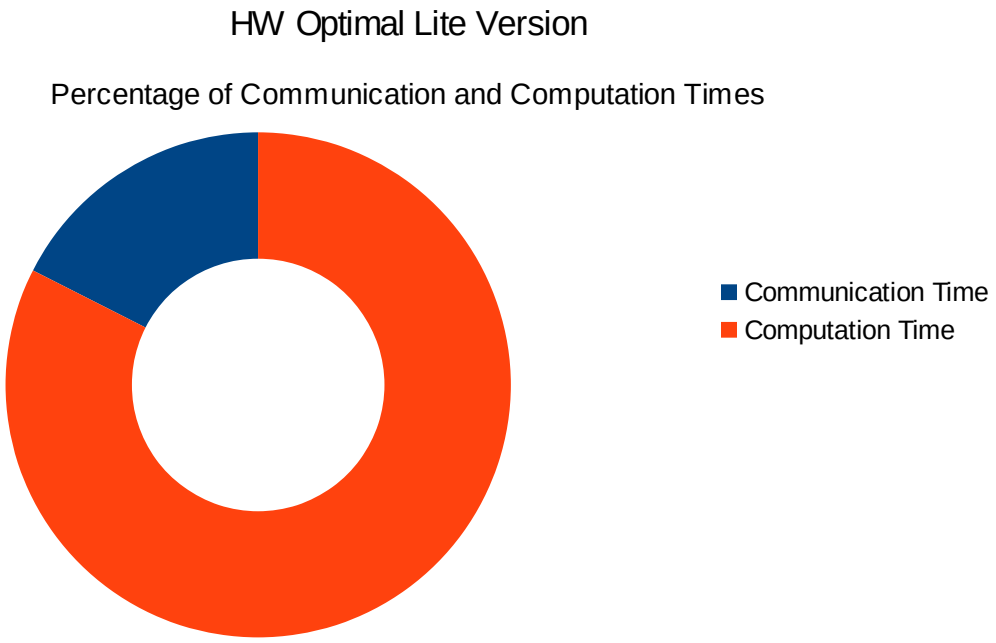


Figure 6.26: Communication and Computation Times for HW Optimal Lite Version

As one may notice, the impact of communication time in the average case is close to zero and its percentage in the pie chart is barely visible. On the other hand, in the HW Optimal Lite implementation communication time plays a more dominant role. Hence, the improvement through the employment of an AXI4 Stream version is more visible that in previous implementations.

At this point we have completed the exploration and evaluation of different Support Vector Machine classifiers. Three different versions of the Pareto design space were employed and tested. We proceeded to various different implementations using the previously mentioned versions. AXI4 Slave Lite and AXI4 Stream interfaces were used for the communication, control and data transfers between the PS and PL parts of the device. Additionally, versions of system architectures with more than one classifiers were implemented leading to higher bandwidth and throughput values. A conclusion of this exploration was that better communication times can be achieved by adding AXI4 Stream interfaces to custom accelerators and employing AXI DMA blocks for data transfers. Finally, talking about numbers, the latency gain of our final implementations was up to 99% and the achieved bandwidth reached values of as much as 445 MB/s.

Chapter 7

Conclusion

7.1 Summary

At this point the work for our diploma thesis has reached the end. Many things have been studied and implemented and so this text consisted of a lot of diagrams and results. The field of this thesis was the integration of custom hardware accelerators generated through High-Level Synthesis on an FPGA-based SoC device which is Zynq-7000 AP SoC. The algorithms that were studied and implemented are of vital significance for their corresponding fields as corner detection is the basis of most algorithms in Computer Vision and, on the other side, ECG is considered to be one of the most important biological signals.

For the integration of those custom hardware accelerators the widely used AMBA AXI protocol was utilized. Specifically, the its simplest form, AXI4-Lite and a more complex form, AXI4-Stream. The usage of AXI4-Lite was based on the Linux UIO driver which is usually built in distributions generated through Petalinux Tools. On the other, the AXI4-Stream protocol required the theoretical and technical background of Direct Memory Access, a method of accessing memory that does not utilizes the CPU and assists the fast data transfers that are needed.

After choosing the interfaces for our custom accelerators we proceeded to the system generations through Vivado Design Suite. Various AXI components were introduced, some of which were automatically interconnected by the tool while others needed our assistance. The most important of those components were the AXI Interconnect, which is used on every single design for interconnection with the PS-side of the ZedBoard, and the AXI DMA block which was utilized for fast transfers to AXI4-Stream-based accelerators.

The two distinct accelerators that we examined offered us different design alternatives. For starters, the Harris_FindCorners IP offered us design alternatives limited to the choice of AXI4-Lite or AXI4-Stream protocols because its utilization of resources accompanied with the fact that is was originally developed for another FPGA device restricted our options for diversity. On the other hand, the simplicity of the Classify IP, the low resource demands, combined with the fact that it was originally designed for the same target device offered a wide range of design alternatives, not limited to solely a choice of interfaces but also

expanded to the addition of multiple IPs controlled by different processors in both AXI4-Lite and AXI4-Stream Versions.

The tools that we used were in some cases a bit restrictive as the design options that were offered were usually not implemented. For instance, the AXI4 DMA block is supposed to support Burst Lengths of up to 256, however the AXI3-compliance restricts it to only 16, and setting it to higher values technically makes no difference.

Through our various implementations on both hardware accelerators we concluded that an addition of multiple IPs increases bandwidth and throughput only to a point because the PS-side of the ZedBoard includes a processor with only two cores, thus scheduling issues occurred slowing down the execution of four processes with no scaling in performance. AXI4-Stream has been evaluated as the fastest of the protocols that where used as the AXI DMA block can make transfers with speeds up to 300 MB/s per DMA channel. AXI4-Stream proved the best also in case of computation latency gains as its lack of block-level protocols lead to a total gain of as much as 20% compared to the equivalent AXI4-Lite solution for a default hardware accelerator. Simultaneously, a bandwidth of 444 MB/s was achieved.

7.2 Future Work

Devices combining a Processing System and Programmable Logic gain an increased interest as the next generation of FPGA-based devices. The innate nature of FPGA is fast development of system designs and reconfigurability. FPGAs have reached a point where a lot of research have proven them as top choices when it comes to implementing HW accelerators. On the other hand, the embedded processors that are added to Programmable Logic, due to their scarcity of time in the device might introduce scheduling issues as proven by our implementations. A nice idea for future work might be the study of the PS not in general but as part of an FPGA device for coping with issues concerning, for instance, scheduling.

Another idea would be the study and development of more possible design alternatives for a hardware accelerator when it is integrated in a SoC. Design tools offer many opportunities for development but the human factor has always proven beneficial. Even though, most of the design work is made by the tool, useful hints and directives from humans could be integrated to enhance automated design, not only to the extent of HLS but even more.

Finally another idea would be the exploration of the parameters of the AXI DMA blocks that are offered in Vivado Design Suite of other tools. Many of the parameters can be altered, such as burst size, but many of the theoretical utilities have no impact whatsoever due to compliance and other issues. If greater burst sizes could be supported along with wider DMA channels the AXI DMA block would be the must choice when it came to data transfers.

References

[1] “Programmable Logic Array”, https://en.wikipedia.org/wiki/Programmable_logic_array

[2] “Programmable Array Logic” https://en.wikipedia.org/wiki/Programmable_Array_Logic

[3] “FPGA”, https://en.wikipedia.org/wiki/Field-programmable_gate_array

[4] Karen Parnell, Roger Bryner, “*Comparing and Contrasting FPGA and Microprocessor System Design and Development”,* White Paper, Xilinx, 2004

[5] Dionysios Diamantopoulos, Ioannis Galanis, Kostas Siozios, George Economakos and Dimitrios Soudris, *“A Framework for Rapid System-Level Synthesis Targeting to Reconfigurable Platforms: A Computer Vision Study”,* Workshop in Reconfigurable Computing, Amsterdam, 2015

[6] Philip H.W. Leong, *“Recent Trends in FPGA Architectures and Applications”*, 4th IEEE International Symposium on Electronic Design, Test & Applications, February 2008

[7] “Zynq-7000 AP SoC”, http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[8] “Ultrascale Architecture”, http://www.xilinx.com/products/technology/ultrascale.html

[9] Κ.Ζ. Πεκμεστζή, *“ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ VLSI: ΕΡΓΑΣΤΗΡΙΑΚΕΣ ΑΣΚΗΣΕΙΣ”*, Αθήνα, 2014

[10] Χαράλαμπος Ν. Σιδηρόπουλος, “*Development of a Design Framework for Power/Energy consumption estimation in heterogeneous FPGA architectures”,* Diploma Thesis, National Technical University of Athens, July 2010

[11] Philippe Coussy, Michael Meredith, Daniel D. Gajski, Andres Takach, “*An Introduction to High-Level Synthesis”,* IEEE Design & Test of Computers, July/August 2009

[12] “Computer Vision”, https://en.wikipedia.org/wiki/Computer_vision

[13] Ιωάννης Π. Γαλάνης, “*High-Level Synthesis του αλγορίθμου Όρασης Υπολογιστών Harris σε FPGA”,* Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, Αθήνα, 2015

[14] “Feature Detection”, https://en.wikipedia.org/wiki/Feature_detection_%28computer_vision%29

[15] John F. Canny, *“A Computational Approach to Edge Detection”*, IEEE Transactions on Pattern Analysis and Machine Intelligence, VOL. PAMI-8, NO. 6, November 1986

[16] Chris Harris and Mike Stephens, *“A COMBINED CORNER AND EDGE DETECTOR”*, Plessey Research Roke Manor, United Kingdom, The Plessey Company plc. 1988

[17] Vasileios Tsoutsouras, Konstantina Koliogeorgi, Sotirios Xydis, Dimitrios Soudris, *“HLS code transformation strategies and directives exploration for FPGA accelerated ECG analysis”*, Workshop in Reconfigurable Computing, Prague, 2016

[18] Κωνσταντίνα Ι. Κολιογεώργη, “Optimizing ECG Signal Analysis by building FPGA-based accelerators using High-Level Synthesis”, Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, Αθήνα, Ιανουάριος 2016

[19] “Cardiac Cycle”, https://en.wikipedia.org/wiki/Cardiac_cycle

[20] A. Szczepanski and K. Saeed, *"A mobile device system for early warning of ecg anomalies"*, Sensors, vol. 14, no. 6, pp. 11031-11044, 2014. xxixxxix, 5

[21] C.-W. Hsu, C.-C. Chang, C.-J. Lin, et al., *"A practical guide to support vector classication"*, 2003.

[22] “AMBA Specifications”, http://www.arm.com/products/system-ip/amba-specifications.php

[23] “AMBA”, https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture

[24] ARM Ltd. , *“AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite”*, Copyright © 2003, 2004, 2010, 2011, 2013 ARM. All rights reserved.

[25] ARM Ltd. , *“AMBA 4 AXI4-Stream Protocol: Specification, Version: 1.0”*, Copyright © 2010 ARM. All rights reserved.

[26] Hans J. Koch, “Userspace I/O drivers in a real-time context”, Linutronix GmbH, Uhldingen, Germany

[27] A. F. Harvey and Data Acquisition Staff, *“DMA Fundamentals on Various PC Platforms”*, National Instruments, Application Note 011

[28] F. Digilent's ZedBoard Zynq, “Dev. Board documentation”

[29] “ZedBoard hardware user's guide”, http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf