ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Exploiting Partial Reconfiguration of SoC FPGAs: A Hardware-Software Co-design for Accelerating Cryptographic Systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΙΕΡΩΝΥΜΑΚΗ ΓΕΩΡΓΙΟΥ

Επιβλέπων : Κιαμάλ Πεκμεστζή
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2017

Η σελίδα αυτή είναι σκόπιμα λευκή.

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

Exploiting Partial Reconfiguration of SoC FPGAs: A Hardware-Software Co-design for Accelerating Cryptographic Systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΙΕΡΩΝΥΜΑΚΗ ΓΕΩΡΓΙΟΥ

Επιβλέπων : Κιαμάλ Πεκμεστζή
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 24η Μαρτίου 2017.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Κιαμάλ Πεκμεστζή	Δημήτριος Σούντρης	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π

Αθήνα, Μάρτιος 2017

(Υπογραφή)

....

ΙΕΡΩΝΥΜΑΚΗΣ ΓΕΩΡΓΙΟΣ

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

Σύντομη Περίληψη

Τα τελευταία χρόνια, η ολοένα αυξανόμενη ώθηση για την καλύτερη δυνατή υπολογιστική επίδοση έχει οδηγήσει στην υλοποίηση του ετερογενούς υπολογισμού και των ετερογενών πλατφόρμων. Τα συστήματα αυτά κερδίζουν υπολογιστική ισχύ και ενεργειακή απόδοση προσθέτοντας διαφορετικούς επιταχυντές ως συν-επεξεργαστές με εξειδικευμένες δυνατότητες επεξεργασίας, για να διαχειριστούν συγκεκριμένες εντατικές εργασίες. Τα FPGAs έχουν κερδίσει το ενδιαφέρον της αρχιτεκτονικής συστημάτων λόγω των δυνατοτήτων τους για γρήγορη προτυποποίηση επιταχυντών υλικού. Όπως υποδηλώνει το όνομά τους, τα FPGAs είναι προγραμματιζόμενα «στο πεδίο», με την έννοια ότι το εσωτερικό κύκλωμα μπορεί να διαμορφωθεί μετά την κατασκευή τους, καθώς και να τροποποιηθεί χωρίς να χρειάζεται η ανακατασκευή τους, όπως στα παραδοσιακά ASICs. Η Μερική Αναδιαμόρφωση (Partial Reconfiguration) οδηγεί αυτήν τους την ευελιξία ένα βήμα παραπέρα, δίνοντας τη δυνατότητα σε ένα FPGA που είναι ενεργό να τροποποιήσει ένα κομμάτι του όσο το υπόλοιπο σύστημα συνεχίζει να λειτουργεί κανονικά, χωρίς να βάζει σε κίνδυνο την ακεραιότητα των υπολογισμών που εκτελούνται στα τμήματα της συσκευής που δεν αναδιαμορφώνονται. Αυτή η τεχνική οδηγεί στην μείωση των πόρων που χρειάζονται για να υλοποιηθεί μια δεδομένη λειτουργία, με επακόλουθη μείωση στο κόστος και την ενεργειακή κατανάλωση, παρέχει ευελιξία στους αλγόριθμους/πρωτόκολλα που είναι διαθέσιμα σε μία εφαρμογή και επιταχύνει την υπολογιστική διαδικασία επιτρέποντας σε ένα σύστημα να είναι έτοιμο να ανταποκριθεί σε νέες απαιτήσεις γρηγορότερα. Αυτή η εργασία προσπάθησε να εξερευνήσει την τεχνολογία της Μερικής Αναδιαμόρφωσης σε FPGAs και να εφαρμόσει τη γνώση που αποκτήθηκε για να κατασκευάσει ένα σύστημα κρυπτογράφησης στη συσκευή Xilinx Zynq-7000 SoC. Το Zynq συνδυάζει τη συνύπαρξη προγραμματιζόμενης λογικής με ένα ενσωματωμένο επεξεργαστή αρχιτεκτονικής ARM στο ίδιο τσιπ, σχηματίζοντας έτσι ένα σύστημα-σε-τσιπ (system-on-a-chip, SoC), καθώς και επιτρέπει τη γρήγορη διασύνδεση με χαμηλή ενεργειακή κατανάλωση. Για τους σκοπούς αυτής της εργασίας διαλέξαμε τέσσερα κρυπτογραφικά modules (AES128, AES192, AES256 και SHA3-512). Αρχικά, κάναμε όλες τις απαραίτητες τροποποιήσεις που χρειάζονταν για να χρησιμοποιηθούν τα modules στο σύστημα και σχεδιάσαμε τις κατάλληλες, συμβατές με το πρωτόκολλο AXI4-Stream, διεπαφές ώστε να επιτραπεί η επικοινωνία μεταξύ των περιφερειακών και του επεξεργαστή, λαμβάνοντας υπ' όψιν την αρχιτεκτονική κάθε module και τους περιορισμούς του επεξεργαστικού συστήματος και της μερικής αναδιαμόρφωσης. Μετά, συνδέσαμε τα περιφερειακά με το επεξεργαστικό σύστημα μέσω ενός AXI DMA IP σε λειτουργεία Scatter/Gather. Η λειτουργία Scatter/Gather οδήγησε στη γρήγορη επικοινωνία και εφάρμοσε στρατηγική συνένωσης διακοπών (interrupt coalescing) για να μειώσει τον αριθμό των διακοπών στον ARM και έτσι, να χειριστεί τα περιφερειακά πιο αποδοτικά. Επίσης εφαρμόσαμε μέθοδο αποσύζευξης (decoupling) για να απομονώσουμε τα αναδιαμορφώσιμα modules κατά τη διάρκεια της μερικής αναδιάταξης και να αποτρέψουμε ανεπιθύμητα εξερχόμενα σήματα να επηρεάσουν το υπόλοιπο σύστημα. Τέλος, κάναμε μια εκτίμηση της δουλειάς και φτιάξαμε ένα τεστ αξιολόγησης για να δείξουμε τα πλεονεκτήματα επιτάχυνσης της μερικής αναδιαμόρφωσης. Σε αυτό το τεστ, το σύστημα είχε τη δυνατότητα να προσαρμόζεται στις υπολογιστικές απαιτήσεις και να αναδιαμορφώνει άεργα περιφερειακά με άλλα που χρειάζονταν, ώστε να κατανείμει τον υπολογιστικό φόρτο μεταξύ τους και έτσι, να μειώσει τον τελικό χρόνο υπολογισμού. Ως αποτέλεσμα, πετύχαμε σχεδόν full hardware utilization και προσεγγίσαμε το βέλτιστο speedup.

Λέξεις Κλειδιά: FPGA, Xilinx Zynq-7000, Μερική Αναδιαμόρφωση, Κρυπτογραφία, AES, SHA3, Scatter/Gather DMA, HW/SW co-design, Αναδιαμορφώσιμες Αρχιτεκτονικές, Ετερογενή Συστήματα

Η σελίδα αυτή είναι σκόπιμα λευκή

Abstract

In recent years, the continued push to gain the best computing performance possible has led to the realization of Heterogeneous computing and Heterogeneous platforms. These systems gain performance and energy efficiency by adding dissimilar accelerators as co-processors with specialized processing capabilities, to handle specific intensive tasks. Field Programmable Gate Arrays (FPGAs) have gained the interest of system architects due to their rapid prototyping and fast accelerator developing capabilities. As their name denotes, FPGAs are programmable "in the field", meaning that their internal logic can be configured after the fabrication process and modified, if needed, without going to re-fabrication process, as common ASICs. Partial Reconfiguration (PR) takes this flexibility one step further, by allowing an operating FPGA design to modify a part of itself, while the rest of the system continues to function normally, without compromising the integrity of the computation running on those parts of the device that are not being reconfigured. This technique leads to reduction of the amount of resources required to implement a given function, with consequent reductions in cost and power consumption, provides flexibility in the algorithms/protocols available to an application and accelerates computing by enabling a design to be ready to correspond to new computation requirements much faster. This thesis tried to explore the PR technology on FPGAs and apply the knowledge acquired to implement a cryptographic system on a Xilinx Zynq-7000 SoC device. Zynq combines the coexistence of programmable logic and an embedded ARM processor on a single chip, thus forming a system-on-a-chip (SoC), while enabling fast interconnection between them and power efficiency. For the purposes of this thesis we chose four cryptographic modules (AES128, AES192, AES256 and SHA3-512). Firstly, we made all the appropriate modifications needed to utilize the cryptographic modules in the SoC and designed the appropriate AXI4-Stream compliant interfaces to enable communication between the peripherals and the processor, with respective compromises to the different modules' architecture, the processing system's limitations and PR's restrictions. Then, we established connection between the peripherals and the processing system through an AXI DMA IP in Scatter/Gather mode. Scatter/Gather resulted in a high-speed communication and applied interrupt coalescing strategy to reduce the number of interrupts occupying the ARM, thus it allowed the processor to handle the peripherals more efficiently. We also applied decoupling strategy to isolate the reconfigurable modules during PR to avoid undesirable outcoming signals to affect the rest of the design. Finally, we made an evaluation of our work and constructed a benchmark to show the acceleration advantages of PR. In this benchmark, the system could adapt to computation requirements and reconfigured idle peripherals with others that were needed, to distribute the computational load between them and so, to reduce the total computation time. As a result, we achieved almost full hardware utilization and approximated the optimal speedup.

Keywords: FPGA, Xilinx Zynq-7000, Partial Reconfiguration, Cryptography, AES, SHA3, Scatter/Gather DMA, HW/SW co-design, AMBA, AXI4-Stream, Reconfigurable Computing, Heterogeneous Computing

Η σελίδα αυτή είναι σκόπιμα λευκή.

Σ’ ένα Μεγάλο Αύριο

Η σελίδα αυτή είναι σκόπιμα λευκή.

Contents

Θεωρητικό Υπόβαθρο

Field Programmable Gate Arrays (FPGAs)

Τα μοντέρνα συστήματα συχνά βασίζονται σε μικρο-επεξεργαστές για μια μεγάλη ποικιλία εφαρμογών, από μικρά κοινά συστήματα, μέχρι μεγάλα πολύπλοκα συστήματα με απαιτήσεις υψηλής απόδοσης. Σε αυτά τα συστήματα, διάφορων ειδών επεξεργαστές αναλαμβάνουν τους υπολογισμούς, από επεξεργαστές γενικού σκοπού μέχρι μονάδες ειδικού σκοπού, όπως οι μονάδες επεξεργασίας γραφικών και άλλες εξειδικευμένες μονάδες αφοσιωμένες για ειδικές εφαρμογές. Τα τελευταία χρόνια, οι απαιτήσεις πολλών εφαρμογών απαιτούν μονάδες μεγάλης υπολογιστικής ισχύς, χαμηλής κατανάλωσης ενέργειας και μικρού φυσικού μεγέθους. Μια τέτοια συσκευή είναι το FPGA, με την επιπρόσθετη ιδιότητα της αναδιαμορφωσιμότητας, με την έννοια ότι ένα FPGA μπορεί να επαναπρογραμματιστεί πολλές φορές μετά την κατασκευή του ή την τοποθέτησή του σε κάποια συσκευή.

Η αναδιαμορφωσιμότητα των FPGA τα καθιστά μια ευέλικτη -με πλήθος εφαρμογών- πλατφόρμα και μια βιώσιμη λύση για γρήγορη υλοποίηση και προτυποποίηση νέων συστημάτων. Ακόμα, τα τελευταία χρόνια τα FPGA έχουν κερδίσει σημαντικό ρόλο στον αναδιαμορφώσιμο υπολογισμό. Ο αναδιαμορφώσιμος υπολογισμός είναι όρος που περιγράφει μια αρχιτεκτονική υπολογιστών που συνδυάζει την ευελιξία του λογισμικού που τρέχει σε επεξεργαστές γενικού σκοπού, με την αποδοτικότητα μονάδων υψηλής επεξεργαστικής ισχύος, όπως τα FPGA. Η κύρια διαφορά με τις «παραδοσιακές» αρχιτεκτονικές έγκειται στην ικανότητα του υλικού να προσαρμόζεται κατά τη διαδικασία εκτέλεσης και να μεταβάλλει τον εαυτό του «φορτώνοντας ένα καινούργιο κύκλωμα» κάθε φορά που μια συγκεκριμένη εργασία το απαιτεί. Έτσι, αυτή η τεχνική κατασκευάζει ετερογενείς πλατφόρμες ικανές να φτάσουν υψηλότερη απόδοση με λιγότερη κατανάλωση ενέργειας και χώρου.

Επίσης, τα FPGA σε συνδυασμό με επεξεργαστές γενικού σκοπού μπορούν να υλοποιήσουν ένα σύστημα σε ψηφίδα (System on a Chip - SoC) και να μελετηθούν ως μια ενοποιημένη υλικολογισμική προσέγγιση για το σχεδιασμό συστημάτων. Για τους λόγους που αναφέρθηκαν παραπάνω και επειδή στις μέρες μας τα FPGA έχουν αυξήσει τη λογική τους χωρητικότητα, χρησιμοποιούνται σε ένα ευρύ πλήθος εφαρμογών όπως στις ψηφιακές επικοινωνίες, στην ψηφιακή επεξεργασία σήματος, στην αυτοκινητοβιομηχανία, στην ιατρική, στο διάστημα, σε αμυντικά συστήματα και άλλα.

Μερική Αναδιαμόρφωση (Partial Reconfiguration)

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

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

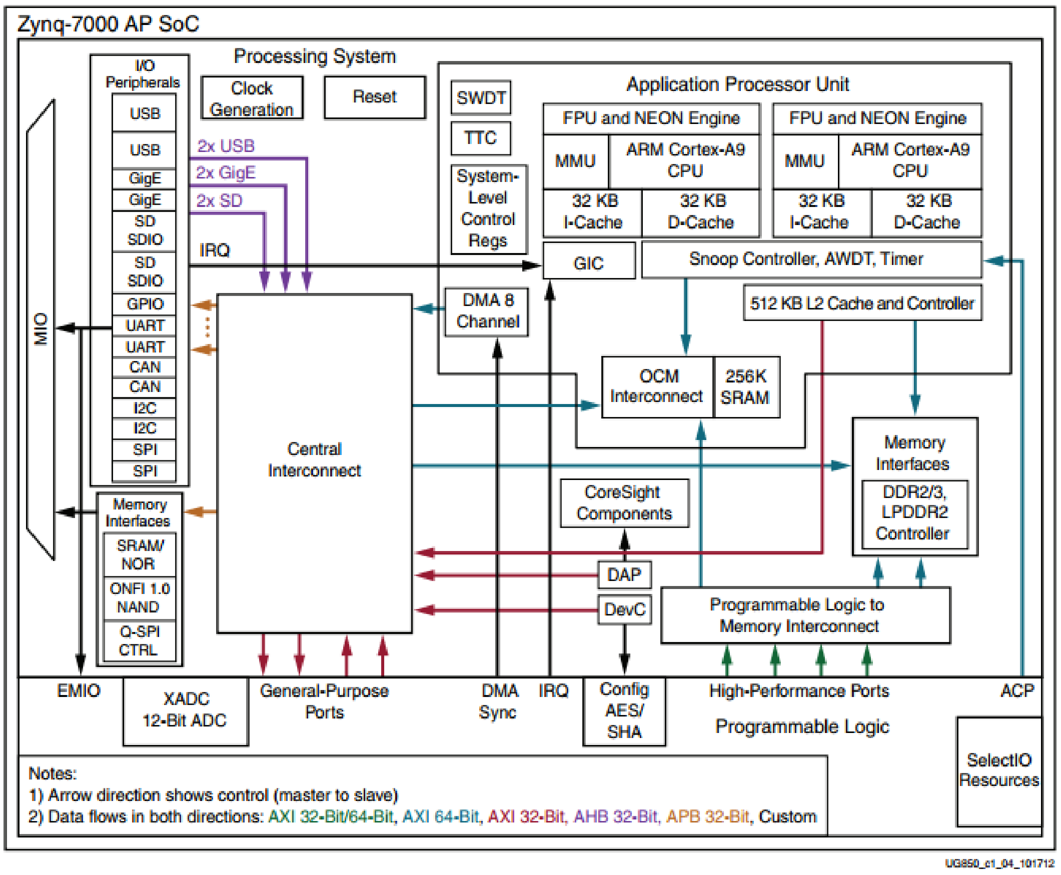
Η διαδικασία της Μερικής Αναδιαμόρφωσης απαιτεί την υλοποίηση πολλαπλών διαμορφώσεων (configurations) οι οποίες τελικά καταλήγουν σε ολικά bitstream για κάθε configuration, και μερικά bitstream για κάθε αναδιαμορφούμενο εργαλείο. Ο αριθμός των απαιτούμενων configuration ποικίλει ανάλογα με τον αριθμό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

Αναπτυξιακή Πλακέτα ZC702

Η ZC702 είναι μια χαμηλού κόστους αναπτυξιακή πλακέτα, που βασίζεται στο XC7Z020, μέλος της οικογένειας Xilinx Zynq®-7000 All Programmable SoC. Το XC7Z020 συνδυάζει ένα δύο πυρήνων ARM® Cortex-A9 επεξεργαστικό σύστημα με ένα 7-Series FPGA και στοχεύει σε ένα ευρύ πεδίο εφαρμογών. Ακόμα η αναπτυξιακή πλακέτα περιλαμβάνει πλήθος υποδοχών επέκτασης για εύκολη πρόσβαση του χρήστη και σύνδεση με άλλα περιφερειακά. Συγκεκριμένα τα πιο σημαντικά χαρακτηριστικά της πλακέτα ZC702 είναι:

- Zynq XC7Z020-1CLG484C device
- 1 GB DDR3 μνήμη (4 x 256 Mb)
- 128 Mb Quad SPI flash memory
- USB 2.0 ULPI (UTMI+ low pin interface) transceiver
- Secure Digital (SD) connector
- USB JTAG interface
- Clock sources
- USB-to-UART bridge
- I2C bus multiplexed to:
    - Si570 user clock
    - ADV7511 HDMI codec
    - M24C08 EEPROM (1 kB)
    - 1-To-16 TCA6416APWR port expander
    - RTC-8564JE real time clock
    - FMC1 LPC connector
    - FMC2 LPC connector
    - PMBUS data/clock
- Configuration options:
    - Quad SPI flash memory
    - USB JTAG configuration port (Digilent module)
    - Platform cable header JTAG configuration port
    - 20-pin PL PJTAG header
    - 20-pin PS JTAG header

Το σχηματικό του Zynq-7000 AP SoC φαίνεται στο Σχήμα 1.



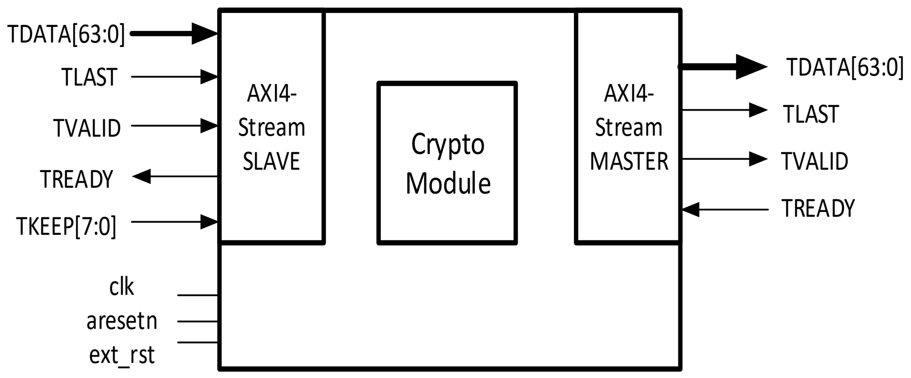
Σχήμα 1 Σχηματικό XC7Z020

Η αναπτυξιακή πλακέτα ZC702 μπορεί να χρησιμοποιηθεί σε πληθώρα εφαρμογών. Σε αυτή την εργασία χρησιμοποιήθηκε για την ανάπτυξη ενός συστήματος κρυπτογραφίας με δυνατότης Μερικής Αναδιαμόρφωσης, όπου κατά το χρόνο εκτέλεσης ένα μέρος του FPGA αναδιαμορφώνεται και διαφορετικοί κρυπτογραφικοί επιταχυντές εναλλάσσονται λαμβάνοντας υπ’ όψιν τα εισερχόμενα ρεύματα δεδομένων και άλλες απαιτήσεις, με σκοπό να μεγιστοποιηθεί η διέλευση πληροφορίας και η απόδοση της εφαρμογής μας.

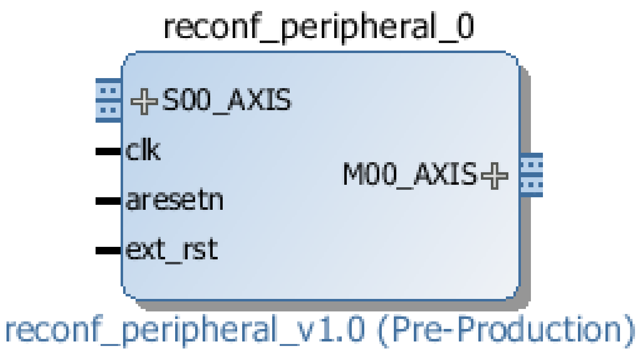
Διαδικασία Υλοποίησης του Συστήματος

Ενσωμάτωση των IPs στο υλικό

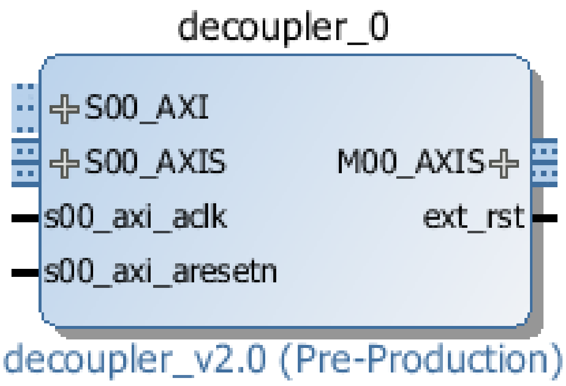
Για τους σκοπούς της παρούσας διπλωματικής επιλέχθηκαν 4 διαφορετικά IPs που υλοποιούν τους αλγορίθμους AES128, AES192, AES256 και SHA3-512. Για να ενσωματώσουμε τα IPs στην υλοποίησή μας και να επιτραπεί η επικοινωνία μεταξύ τους και του επεξεργαστικού συστήματος, σχεδιάσαμε τις κατάλληλες διεπαφές (interfaces) για κάθε ένα από αυτά. Για κάθε IP, σχεδιάστηκαν δύο διεπαφές, μία για την ανάγνωση και μία για την εγγραφή δεδομένων. Οι διεπαφές γράφτηκαν σε Verilog, σχεδιάστηκαν να είναι συμβατές με το πρωτόκολλο AXI4-Stream της ARM και υλοποιήθηκαν με τη βοήθεια του Vivado IP Packager. Επίσης, για την Μερική Αναδιαμόρφωση υλοποιήθηκε ένας Αποζεύκτης (Decoupler), που η χρήση του ήταν να απομονώνει και να εμποδίζει κάθε εξερχόμενο σήμα από τα IPs κατά τη διαδικασία της Μερικής Αναδιαμόρφωσης, καθώς και ένα βοηθητικό IP («Dummy” IP) που χρησιμοποιήθηκε για τη Σύνθεση της στατικής λογικής. Τα σχεδιαγράμματα των υλοποιημένων IP φαίνονται στα Σχήματα 2, 3 και 4.



Σχήμα 2 Crypto IPs



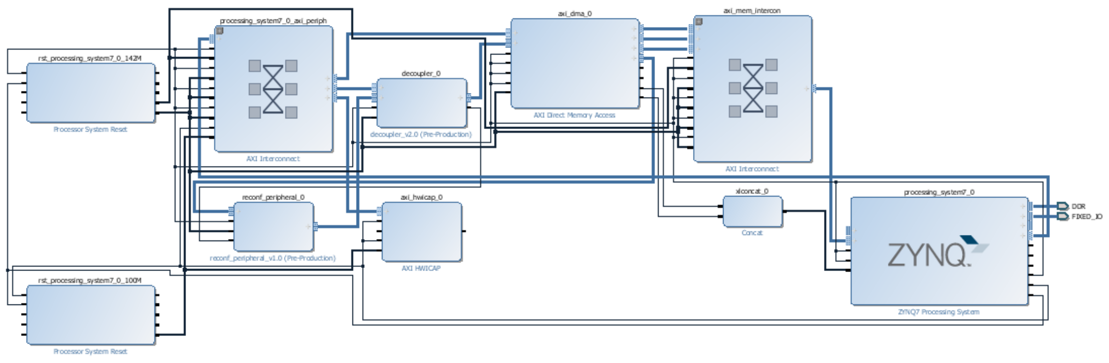
Σχήμα 3 “Dummy” IP



Σχήμα 4 Decoupler IP

Δημιουργία του Συστήματος

Σε αυτό το βήμα, και πριν τη διαδικασία της Μερικής Αναδιαμόρφωσης, για τη Σύνθεση της στατικής λογικής θα χρησιμοποιηθεί το “Dummy” IP αντί για τα κρυπτογραφικά IPs. Το υλοποιημένο σύστημα για ένα κρυπτογραφικό περιφερειακό φαίνεται στο Σχήμα 5.



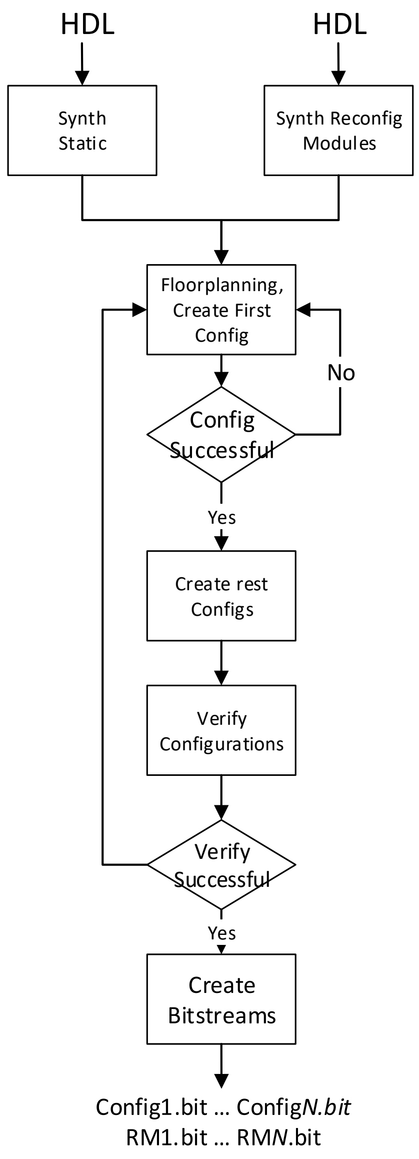
Σχήμα 5 Σύστημα για ένα κρυπτογραφικό περιφερειακό

Διαδικασία Μερικής Αναδιαμόρφωσης

Τα παρακάτω βήματα συμπυκνώνουν τη διαδικασία σχεδιασμού ενός πρότζεκτ Μερικής Αναδιαμόρφωσης στην εφαρμογή Vivado:

1) Σύνθεση της στατικής λογικής και των αναδιαμορφώσιμων εργαλείων ξεχωριστά.
2) Δημιουργία φυσικών περιορισμών (Pblocks) για τον καθορισμό των αναδιαμορφώσιμων περιοχών.
3) Καθορισμός της ιδιότητας HD.RECONFIGURABLE σε κάθε αναδιαμορφώσιμη περιοχή.
4) Υλοποίηση ενός ολόκληρου design (στατική λογική και ένα αναδιαμορφώσιμο εργαλείο σε κάθε αναδιαμορφώσιμη περιοχή)
5) Αποθήκευση ενός checkpoint για το πλήρες design.
6) Αφαίρεση των αναδιαμορφώσιμων εργαλείων από το design και αποθήκευση ενός checkpoint μόνο με στατική λογική.
7) Κλείδωμα της στατικής περιοχής.
8) Πρόσθεση καινούργιων αναδιαμορφώσιμων εργαλείων στο στατικό design και υλοποίηση του νέου configuration, αποθηκεύοντας ένα checkpoint για το πλήρες deisgn.
9) Επανάληψη του βήματος 8 για όλα τα αναδιαμορφώσιμα εργαλεία
10) Επαλήθευση συμβατότητας των configuration
11) Παραγωγή bitstream για κάθε configuration

Η διαδικασία σε διάγραμμα ροής φαίνεται στο Σχήμα 6.



Σχήμα 6 Διαδικασία Partial Reconfiguration

Ανάπτυξη Εφαρμογής Baremetal

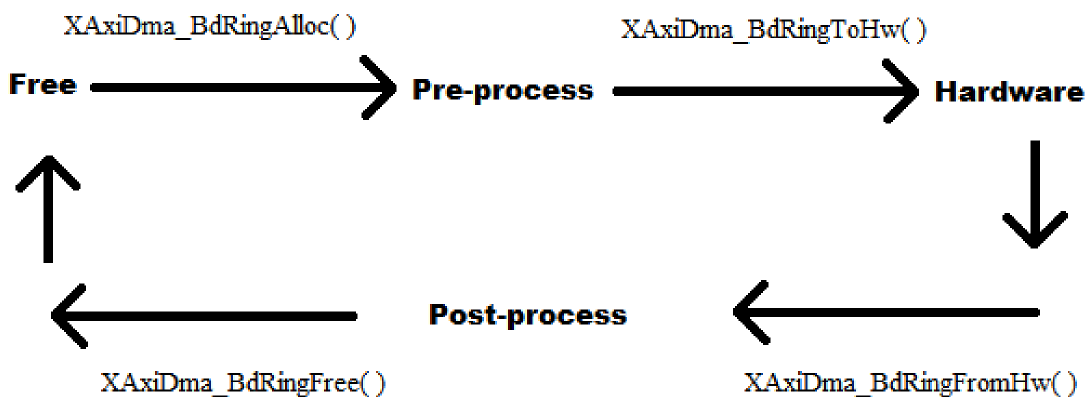
Για την αξιολόγηση του συστήματος υλοποιήθηκε μια εφαρμογή Baremetal. Η εφαρμογή επιλέγει το επιθυμητό interface του Zynq για τη Μερική Αναδιαμόρφωση (PCAP ή ICAP), αρχικοποιεί το Scatter/Gather AXI DMA και χειρίζεται όλα τα δεδομένα από και προς τα περιφερειακά.

Στο Scatter/Gather DMA η μεταφορά των δεδομένων γίνεται από τους Buffer Descriptors (BDs). Οι BDs δεσμεύονται από την εφαρμογή, όπου η εφαρμογή επιλέγει τη διεύθυνση του buffer, το μέγεθος της μεταφοράς και άλλες πληροφορίες ελέγχου. Τα BD rings μοιράζονται από την εφαρμογή χρήστη και το υλικό. Το υλικό αναμένει τους BDs ως συνδεμένες λίστες, όπου ο τελευταίος BD του ring είναι συνδεμένος με τον πρώτο.

Μέσα στο ring, υπάρχουν 4 ομάδες από BDs, όπου κάθε ομάδα αποτελείται από 0 ή περισσότερους γειτονικούς BDs:

- Free: Οι BDs που μπορούν να δεσμευτούν από την εφαρμογή.
- Pre-process: Οι BDs που έχουν δεσμευτεί και είναι υπό τον έλεγχο της εφαρμογής. Η εφαρμογή προετοιμάζει αυτούς τους BDs για την επόμενη συναλλαγή με το DMA.
- Hardware: Οι BDs που έχουν εισαχθεί στο υλικό. Αυτοί οι BDs είναι υπό τον έλεγχο του υλικού και δεν πρέπει να τροποποιούνται από την εφαρμογή.
- Post-process: Οι BDs που έχουν επεξεργαστεί από το υλικό και έχουν επιστρέψει στον έλεγχο της εφαρμογής. Η εφαρμογή μπορεί να ελέγξει την κατάσταση της μεταφοράς ή να τους βάλει στην ομάδα Free.

Το Σχήμα 7 απεικονίζει τις μεταβάσεις ενός BD κατά τη διαδικασία μιας συνεχόμενης μεταφοράς.



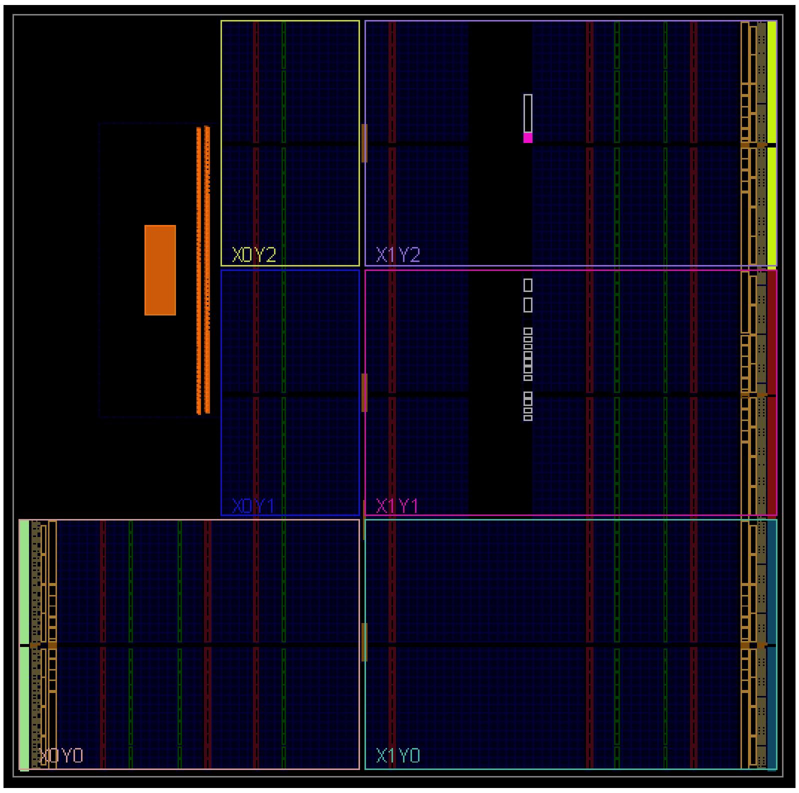
Σχήμα 7 Μεταβάσεις ενός BD κατά τη διαδικασία μια συνεχόμενης μεταφοράς

Αξιολόγηση Πειραματικής Διαδικασίας

Σε αυτή την ενότητα γίνεται αξιολόγηση της πειραματικής διαδικασίας και των διατάξεων που υλοποιήθηκαν. Αρχικά παρουσιάζονται οι προκλήσεις που αντιμετωπίστηκαν και οι συμβιβασμοί που έγιναν ώστε να γίνει δυνατή η Μερική Αναδιαμόρφωση με τα διάφορα περιφερειακά που είχαμε στην κατοχή μας και σχολιάζουμε το floorplanning που έγινε το οποίο είναι ένα κρίσιμο σημείο της Μερικής Αναδιαμόρφωσης και κάνουμε μια σύγκριση των χρόνων διαμόρφωσης για ένα μερικό bitstream σε σχέση με ένα full configuration. Έπειτα γίνεται αξιολόγηση των IPs που υλοποιήθηκαν και παρουσιάζεται η επιτάχυνση που επιτεύχθει. Τέλος, παρουσιάζεται ένα πλήρες λειτουργικό design με δυο αναδιαμορφώσιμα περιφερειακά. Το σύστημα αυτό έχει δυο εισερχόμενα ρεύματα δεδομένων που χρειάζονται κρυπτογράφηση και έχει τη δυνατότητα να αναδιαμορφώνεται με δύο διαφορετικά ή δύο ίδια IPs, ανάλογα με τα εισερχόμενα δεδομένα, με σκοπό να επιταχύνει το συνολικό χρόνο υπολογισμού.

Γενική Περιγραφή των Hardware Υλοποιήσεων

Τα αρχικά cores του αλγορίθμου AES δεν ήταν κατάλληλα για να επιτευχθεί η Μερική Αναδιαμόρφωση στη συσκευή μας. Ο αρχικός AES128 χρησιμοποιούσε 86, ο AES192 100 και ο AES256 121 Block RAM tiles. Το FPGA του Zynq (xc7z020clg484-1) περιέχει συνολικά 140 Block RAM tiles διεσπαρμένα σε όλη την έκταση της συσκευής. Αυτό σημαίνει ότι δεν χωρούσαν περισσότερα από ένα περιφερειακά στη συσκευή μας και η Μερική Αναδιαμόρφωση δεν είχε νόημα, αφού εξ' αιτίας της μεγάλης χρησιμοποίησης Block RAM σε όλη την έκταση της συσκευής, ένα μόνο περιφερειακό θα καταλάμβανε το μεγαλύτερο μέρος του συνολικού διαθέσιμου χώρου. Το σχεδιάγραμμα του Zynq φαίνεται στο Σχήμα 8. Τα Block RAM tiles είναι σημειωμένα με το καφέ χρώμα.



Σχήμα 8 Σχεδιάγραμμα του Zynq

Για τα design μας, δόθηκε εντολή στον Vivado Synthsizer να χρησιμοποιήσει Distributed RAM αντί για Block RAM, εισάγοντας το ειδικό σχόλιο (* ram_style = "distributed" *) στον πηγαίο κώδικα. Αυτό είχε σαν αποτέλεσμα αρκετά μεγαλύτερη χρησιμοποίηση από LUTs (ειδικά στον AES256), όμως έκανε τη Μερική Αναδιαμόρφωση δυνατή αφού πλέον τα cores μπορούσαν να χωρέσουν σε ένα αρκετά μικρότερο χώρο.

Στον Πίνακα 1 παρουσιάζουμε τους συνολικούς πόρους της συσκευής μαζί με την τελική χρησιμοποίηση των synthesized IPs. Το «Dummy» IP δεν παρουσιάζεται καθώς δεν είναι κανονικό IP, αφού δεν περιέχει λογική. Επίσης, όπως φαίνεται, ο AES256 χρησιμοποιεί περίπου το 32,81% των πόρων της συσκευής από μόνος του. Αυτό το γεγονός θα έχει επιρροή στο τελικό floorplanning, αφού θα είναι δύσκολο να βρεθεί μια τόσο μεγάλη συνεχόμενη περιοχή χωρίς στοιχεία που δεν μπορούν να αναδιαμορφωθούν.

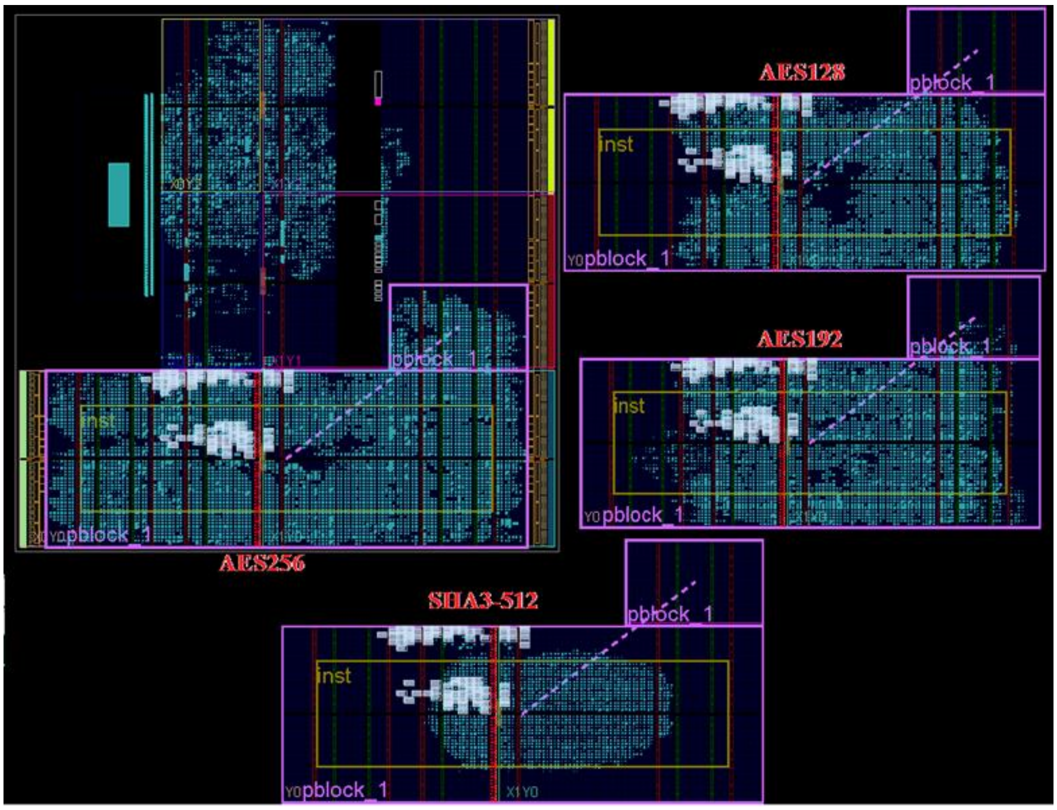
Zynq (xc7z020clg484-1)	Slice LUTs (53200)	Slige Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)
AES128	10775	6337	4064	2032
AES192	12602	8130	4448	2224
AES256	20330	10642	7744	3872
SHA3-512	6258	2361	0	0
Decoupler	220	169	00	0

Πίνακας 1 Η χρησιμοποίηση των IP cores

Επίσης, ο αρχικός AES είχε τη δυνατότητα να δέχεται ολόκληρη την είσοδο (128 bit plaintext και 128/192/256 bit cipherkey) σε ένα μόνο κύκλο ρολογιού, ενώ οι θύρες υψηλής επίδοσης του Zynq έχουν πλάτος 64-bit. Ο SHA3-512 ήταν πιο βολικός, αφού χρησιμοποιεί 64-bit είσοδο και έχει μικρότερο αποτύπωμα στη λογική. Ως αποτέλεσμα τα interfaces των 64-bit που υλοποιήθηκαν μείωσαν αρκετή από την αρχική επίδοση των AES cores, ενώ στην περίπτωση του SHA3-512 η επίδοση έμεινε σχεδόν ανέπαφη.

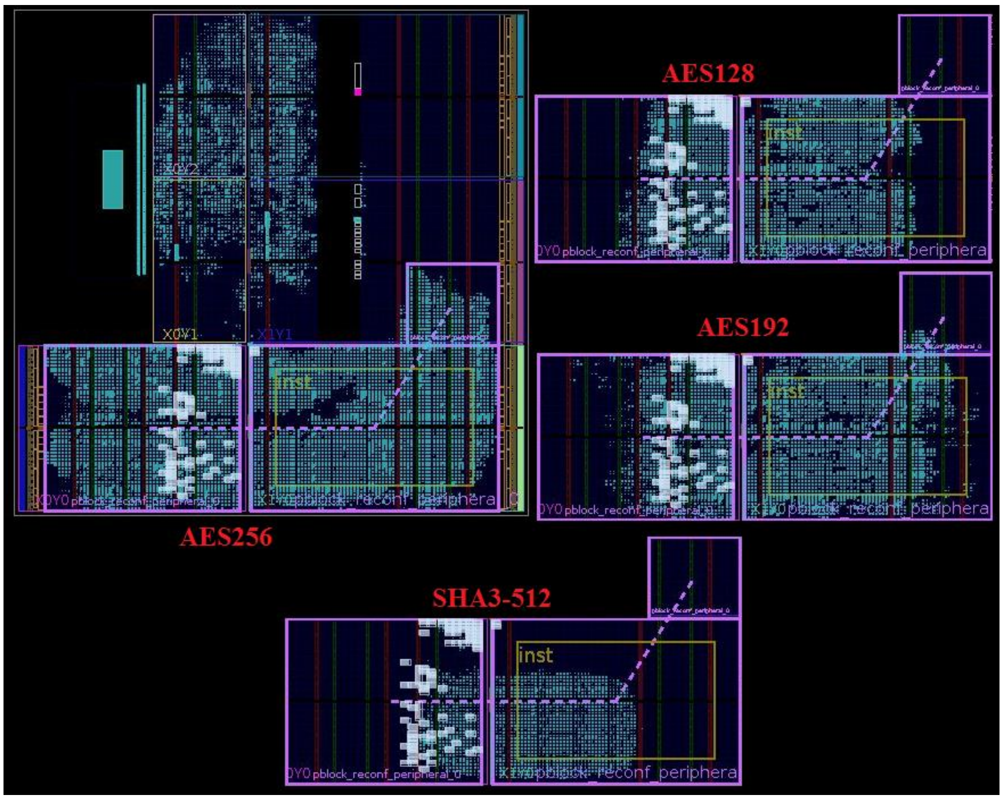
Floorplanning

Στη Μερική Αναδιαμόρφωση, κάθε αναδιαμορφώσιμη περιοχή απαιτείται να έχει ένα Pblock το οποίο καθορίζει τους φυσικούς πόρους που είναι διαθέσιμη για τα αναδιαμορφώσιμα modules. Ένα Pblock πρέπει να περιέχει μόνο έγκυρα αναδιαμορφώσιμα στοιχεία. Πολλαπλά ορθογώνια Pblock μπορεί να χρησιμοποιηθούν για τη δημιουργία μιας αναδιαμορφώσιμης περιοχής, αλλά για το καλύτερο routability, θα πρέπει να είναι συνεχόμενα. Κενά ώστε να αποφευχθούν μη αναδιαμορφώσιμοι πόροι επιτρέπονται, αλλά σε γενικές γραμμές, όσο πιο απλό είναι το γενικό σχήμα, τόσο ευκολότερο θα είναι το place and route του design. Στο δικό μας design, τα κρυπτογραφικά cores καταλάμβαναν αρκετούς από τους πόρους της συσκευής, έτσι ήταν δύσκολο να βρεθεί μια συνεχόμενη περιοχή χωρίς μη αναδιαμορφώσιμα στοιχεία. Ένα floorplanning με back-to-back violations φαίνεται στο Σχήμα 9.



Σχήμα 9 Floorplanning με back-to-back violations

Σε αυτό το floorplanning, το Pblock εκτείνεται διαμέσου ενός interconnect (κόκκινο χρώμα στο Σχήμα 9) δύο διαφορετικών περιοχών ρολογιού και οι στήλες interconnect δεν είναι αναδιαμορφώσιμες στα 7-Series FPGAs. Για το συγκεκριμένο design, τα warnings μπορούν να αγνοηθούν αφού δεν υπάρχει στατική λογική μέσα στο μη-αναδιαμορφώσιμο interconnect που να επηρεάζει το σύστημα. Επειδή τα IP χρησιμοποιούν πολλούς πόρους και ήταν δύσκολο να βρεθεί μια συνεχόμενη περιοχή χωρίς μη-αναδιαμορφώσιμα στοιχεία για το floorplanning, το Pblock μπορεί να «σπάσει» ώστε να αποφύγει το interconnect. Ένα έγκυρο floorplanning χωρίς violation φαίνεται στο Σχήμα 10.



Σχήμα 10 Floorplanning χωρίς violations

Όπως φαίνεται, μετά το implementation το Vivado εισάγει τα λεγόμενα partition pins (άσπρο χρώμα στο Σχήμα 10) μέσα στα όρια του Pblock που καθορίζουν την αναδιαμορφώσιμη περιοχή. Αυτά τα εικονικά I/O τοποθετούνται ως σημεία επικοινωνίας εκτός του Pblock και πρέπει να παραμένουν ίδια σε όλα τα αναδιαμορφώσιμα module. Τα partition pins δε χρειάζονται επιπρόσθετους πόρους όπως LUTs ή flip-flops για να υλοποιηθούν και δεν εισάγουν επιπρόσθετη καθυστέρηση στα σημεία που ενώνουν.

Χρόνοι Configuration

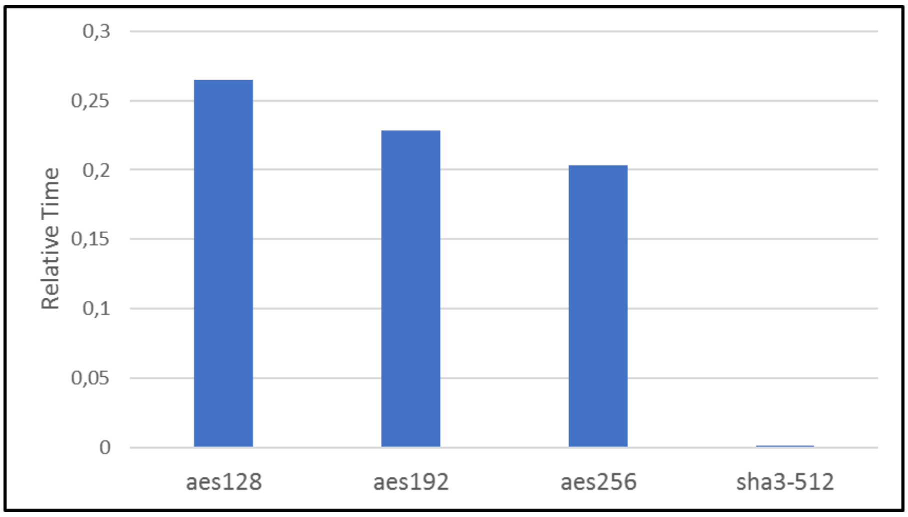
Ο χρόνος για το configuration κλιμακώνεται γραμμικά όσο το μέγεθος του bitstream αλλάζει με τον αριθμός των αναδιαμορφώσιμων frames. Επίσης εξαρτάται από την εφαρμογή ή το λειτουργικό σύστημα που χρησιμοποιείται καθώς και πιο interface καλείται να εκτελέσει την αναδιαμόρφωση. Ο Πίνακας 2 δείχνει σχετικούς χρόνους configuration για μια baremetal εφαρμογή.

	Full Bitstream	Partial Bitstream
Bitstream Size	4,045,678 bytes	1,660,388 bytes
JTAG	~4 s	~2s
PCAP	31.08 ms	12.76 ms
ICAP	-	162.72 ms

Πίνακας 2 Σύγκριση μεταξύ των διαφορετικών configuration interfaces

Αξιολόγηση των κρυπτογραφικών IPs που υλοποιήθηκαν

Σε αυτή την ενότητα, γίνεται σύγκριση μεταξύ των κρυπτογραφικών αλγορίθμων να «τρέχουν» αποκλειστικά στον επεξεργαστή ARM του Zynq και των υλοποιήσεων που φτιάχτηκαν. Στο Σχήμα 11 φαίνονται οι σχετικοί χρόνοι εκτέλεσης των επιταχυντών σε σχέση με τους αλγόριθμους να «τρέχουν» στον ARM.

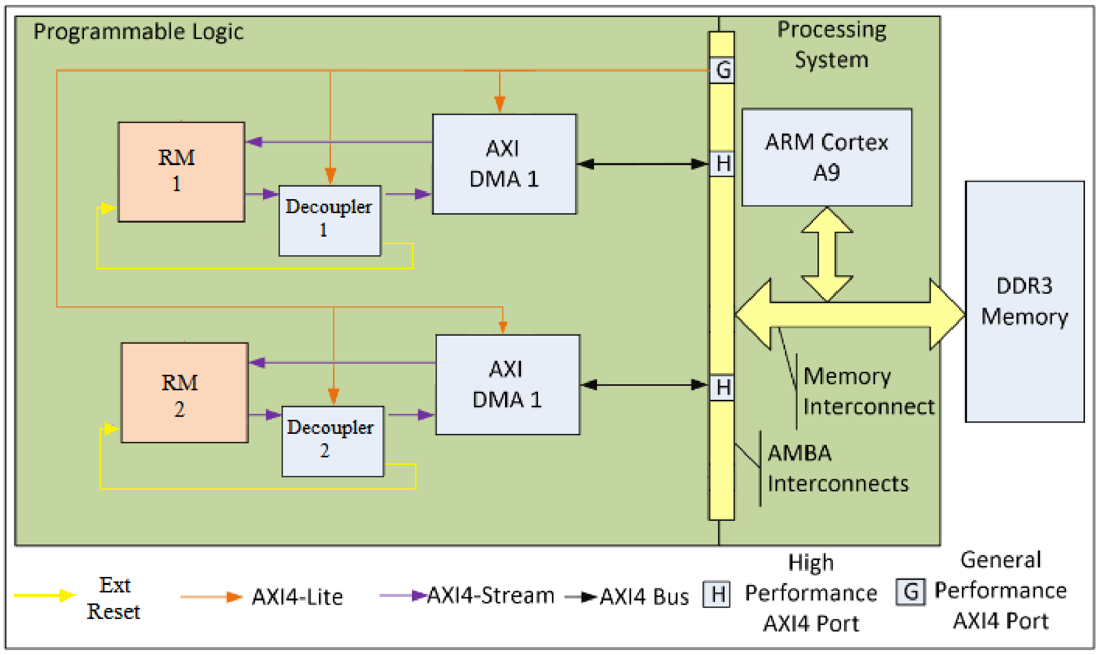


Σχήμα 11 Σχετικοί χρόνοι εκτέλεσης σε σύγκριση με τις αντίστοιχες υλοποιήσεις σε software

Οι τιμές πάρθηκαν από 100000 υπολογισμούς για κάθε αλγόριθμο AES και για τον SHA3-512 για ένα υπολογισμό μεγέθους 10000 x 64 bits. Στο διάγραμμα, μικρότερες τιμές σημαίνουν μεγαλύτερη απόδοση σε σχέση με τις υλοποιήσεις στον ARM. Για παράδειγμα, ο σχετικός χρόνος εκτέλεσης τους AES128 είναι λίγο πάνω από το 0,25 που σημαίνει ότι έχει συνολική επιτάχυνση κάτω από 4 (για την ακρίβεια 3,78).

Experimental Evaluation

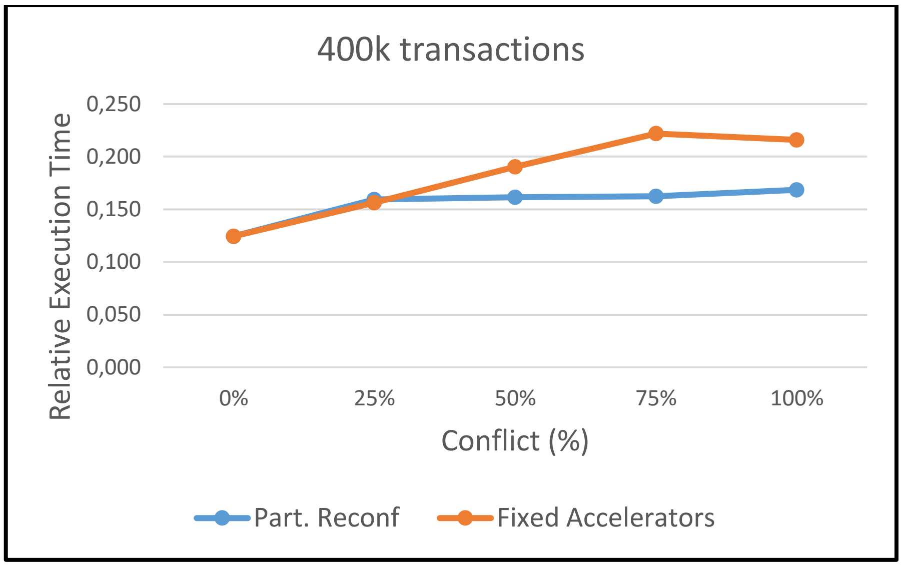
Σε αυτή την ενότητα, κάνουμε μια σύγκριση μεταξύ ενός συστήματος χωρίς δυνατότητες Μερικής Αναδιαμόρφωσης με ένα σύστημα εκμεταλλεύεται την τεχνική και την εφαρμόζει για να προσαρμοστεί στις απαιτήσεις μια εφαρμογής και να επιταχύνει το συνολικό χρόνο εκτέλεσης. Κάποιες φορές, σε συστήματα με παραπάνω από ένα περιφερειακά η εφαρμογή δε τα χρησιμοποιεί όλα και κάποια περιφερειακά μένουν άεργα, ενώ άλλα χρησιμοποιούνται καθ' όλη τη διάρκεια της εφαρμογής. Ένα σύστημα με ικανότητες Μερικής Αναδιαμόρφωσης μπορεί να εκμεταλλευτεί αυτή την κατάσταση και να αναδιαμορφώσει ένα άεργο περιφερειακό με ένα που χρειάζεται για να επιταχυνθεί η συνολική διέλευση του συστήματος. Σε αυτό το τεστ, σχεδιάσαμε ένα σύστημα δυο συστήματα, ένα με δυνατότητες Μερικής Αναδιαμόρφωσης και ένα χωρίς. (Στο εξής θα αναφερόμαστε ως PR system στο σύστημα με Μερική Αναδιαμόρφωση και no-PR system στο σύστημα χωρίς). Τα συστήματα έχουν δυο περιφερειακά το καθένα και υλοποιούν τους αλγορίθμους AES128 και AES192. Στο no-PR system οι αλγόριθμοι είναι fixed και δεν μπορούν να αλλάξουν, ενώ το PR system μπορεί να διαμορφώσει τη συσκευή είτε με δυο διαφορετικούς, είτε με δυο ίδιους αλγόριθμους όταν χρειάζεται. Για παράδειγμα, ας υποθέσουμε ότι έχουμε ένα σύστημα με δυο εισερχόμενα ρεύματα δεδομένων όπου γίνονται κρυπτογραφήσεις με τον AES128 και τον AES192 παράλληλα. Στο no-PR system αν κάποια στιγμή σταματήσουν οι υπολογισμοί του AES192, το αντίστοιχο περιφερειακό παραμένει άεργο καθ' όλη την υπόλοιπη διάρκεια του προγράμματος, ενώ ο AES128 συνεχίζει την εκτέλεση κανονικά. Στο PR system, το σύστημα μπορεί να αναδιαμορφώσει τον AES192, με σκοπό να χρησιμοποιεί δύο περιφερειακά AES128 και να μοιράσει τον υπολογιστικό φόρτο μεταξύ τους. Το σύστημα με τα δυο περιφερειακά φαίνεται στο Σχήμα 12.



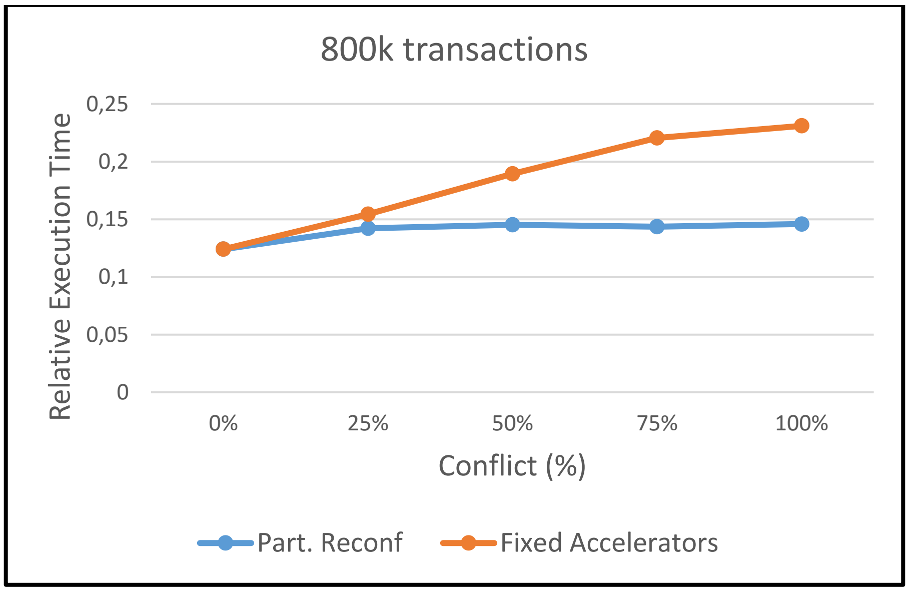
Σχήμα 12 Σύστημα με δυο κρυπτογραφικά IPs

Στα πειράματά μας υλοποιήσαμε ένα απλό software controller, ο οποίος διαβάζει τα δεδομένα από τα δυο εισερχόμενα ρεύματα, τα στέλνει στα περιφερειακά και διαβάζει πίσω τα αποτελέσματα. Επίσης, στο PR system εκτελεί την Μερική Αναδιαμόρφωση. Τα εισερχόμενα ρεύματα έχουν παραχθεί τυχαία, αλλά έχουν μελετηθεί εκ των προτέρων και ο αριθμός των συνολικών αναδιαμορφώσεων είναι από πριν γνωστός.

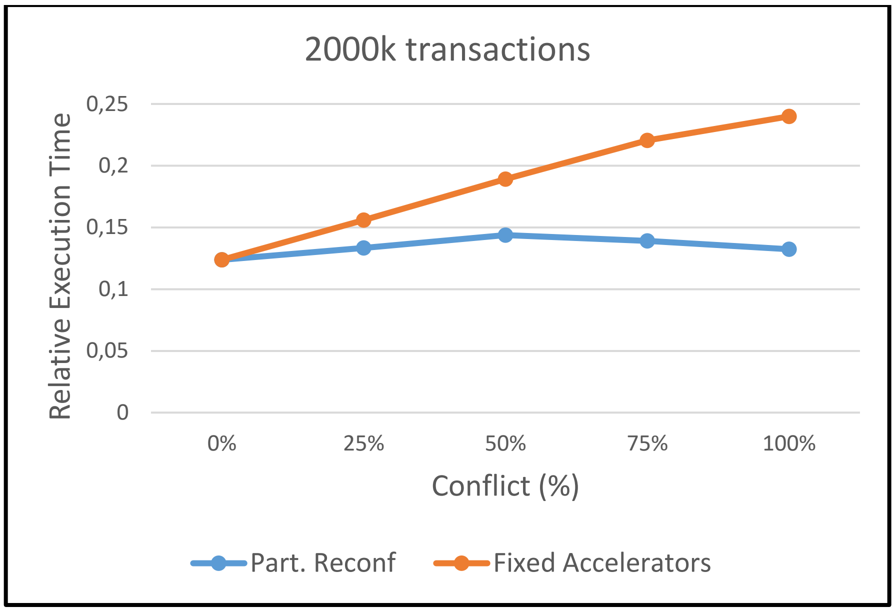
Επιπλέον, έχουμε θεωρήσει ότι τα δύο ρεύματα δεδομένων έχουν συνολικά 5 επίπεδα από Conflict (Σύγκρουση). 100% Conflict σημαίνει ότι κάθε στιγμή το σύστημα χρειάζεται να κρυπτογραφήσει δεδομένα με μόνο ένα από τους δυο αλγορίθμους, ενώ 0% Conflict σημαίνει ότι για κάθε στιγμή υπάρχουν πάντα δεδομένα και για τους δυο. Τα αποτελέσματα για 400k, 800k και 2000k όπου συνολικά γίνονται 5 αναδιαμορφώσεις φαίνονται στα σχήματα 13, 14, 15.



Σχήμα 13 Αποτελέσματα για 400k υπολογισμούς και 5 μερικές αναδιαμορφώσεις



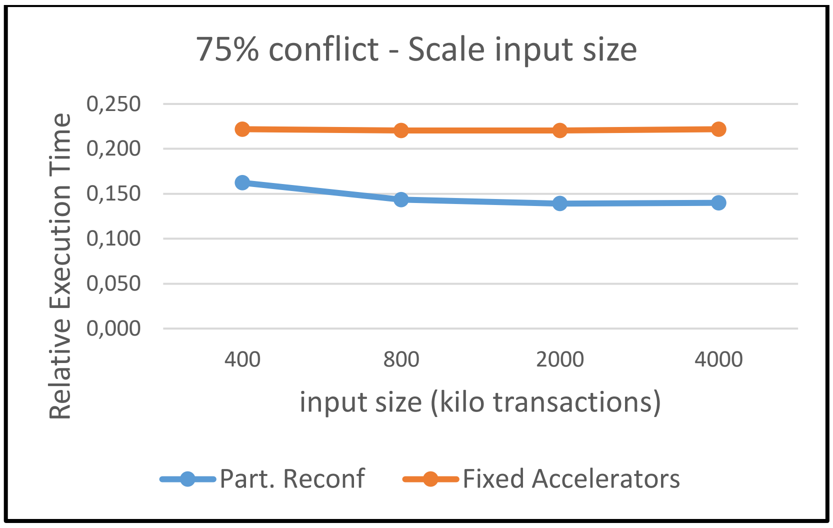
Σχήμα 14 Αποτελέσματα για 800k υπολογισμούς και 5 μερικές αναδιαμορφώσεις



Σχήμα 15 Αποτελέσματα για 2000k υπολογισμούς και 5 μερικές αναδιαμορφώσεις

Σε κάθε ένα από τα παραπάνω σχήματα, ο κάθετος άξονας αναπαριστά το σχετικό χρόνο εκτέλεσης κάθε συστήματος σε σχέση με τους υπολογισμούς όπως θα έτρεχαν στον επεξεργαστή του Zynq, ενώ ο οριζόντιος άξονας αναπαριστά τα 5 επίπεδα Conflict. Για κάθε testcase, το PR system έχει μέση επιτάχυνση 86% σε σχέση με το software και 20% σε σχέση με το no-PR system. Γενικά, και στα 3 σχήματα το PR system κλιμακώνει στο ίδιο κατώφλι επιτάχυνσης εξ' αιτίας του ότι χρησιμοποιεί και τα δυο περιφερειακά κάθε στιγμή. Στο σχήμα 13 για Conflict 25%, το PR system αποδίδει χειρότερα από το no-PR system επειδή ο συνολικός χρόνος εκτέλεσης είναι μικρός και άμεσα συγκρίσιμος με το χρόνο που χρειάζεται για να γίνει μια Μερική Αναδιαμόρφωση. Επίσης, η μικρή διακύμανση στην επιτάχυνση σε κάθε σχήμα είναι αναμενόμενη, λόγω της τυχαίας φύσης κάθε ρεύματος δεδομένων όπου μπορεί να υπάρχουν περισσότερες ή λιγότερες κρυπτογραφήσεις με ένα αλγόριθμο.

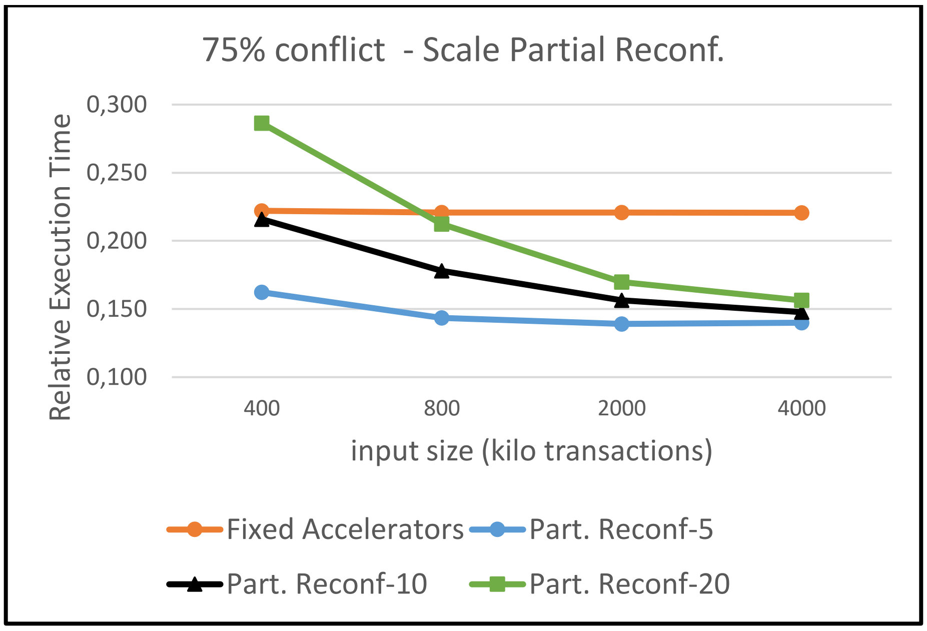
Τα αποτελέσματα για σταθερό 75% Conflict φαίνονται στο Σχήμα 16.



Σχήμα 16 Αποτελέσματα και 75% Conflict και διάφορα μεγέθη εισόδου

Στο Σχήμα 16 δείχνουμε πως κλιμακώνει η συνολική επιτάχυνση για διαφορετικά μεγέθη εισόδου και για σταθερό 75% Conflict. Όπως φαίνεται, η σχετική επιτάχυνση κλιμακώνει σε ένα κατώφλι για κάθε σύστημα όσο η συνολική είσοδος μεγαλώνει.

Για το τελευταίο πείραμα, θεωρήσαμε ένα testcase όπου γίνονται 5, 10 και 20 συνολικές αναδιαμορφώσεις. Τα αποτελέσματα φαίνονται στο Σχήμα 17.



Σχήμα 17 Αποτελέσματα για διαφορετικό αριθμό αναδιαμορφώσεων

Όπως φαίνεται στο Σχήμα 17, η συνολική επιτάχυνση κλιμακώνει ξανά στο ίδιο κατώφλι. Για 20 αναδιαμορφώσεις και 400k συνολικές κρυπτογραφήσεις ο συνολικός χρόνος εκτέλεσης είναι σχετικά μικρός και επηρεάζεται άμεσα από το χρόνο που χρειάζεται για να γίνει μια Μερική Αναδιαμόρφωση. Σε αυτή την περίπτωση, το PR system αποδίδει χειρότερα από το no-PR system. Όμως, καθώς το μέγεθος εισόδου μεγαλώνει και ο συνολικός χρόνος δεν είναι άμεσα συγκρίσιμος και δεν επηρεάζεται από το χρόνο της αναδιαμόρφωσης το σύστημα κλιμακώνει στο ίδιο κατώφλι, όπως τα υπόλοιπα συστήματα, όπου και τα δυο περιφερειακά χρησιμοποιούνται πλήρως για κάθε δεδομένη χρονική στιγμή.

This page is intentionally left blank

List of Figures

List of Tables

This page is intentionally left blank

1 Introduction

1.1 Objective Summary

Systems often rely on micro-processors for a wide variety of applications, from small common systems to large complicated systems with high performance requirements. In recent years, the continued push to gain the best computing performance possible has led to the realization of heterogeneous computing. In such systems processors co-exist with other specialized components dedicated to particular purposes. These systems' requirements demand units of high performance processing, low power consumption and small physical size. For these reasons, Field-Programmable Gate Array (FPGAs) have gained much interest, as they satisfy the above requirements and provide additional characteristics such as rapid prototyping and reconfigurability, meaning that FPGAs can be reprogrammed several times even after their manufacturing or installation on a device.

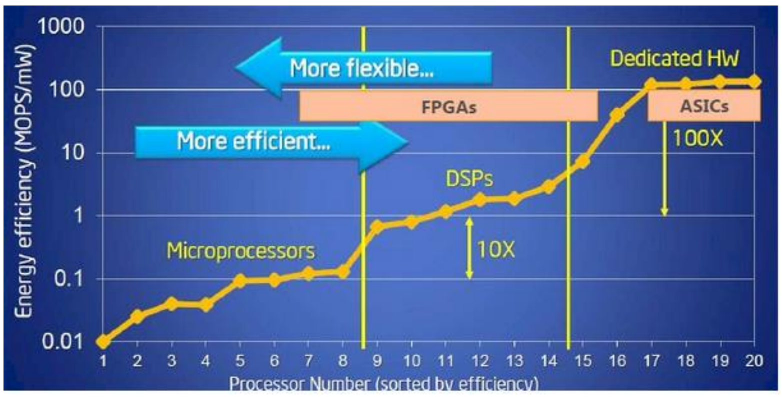


Figure 1 Processing efficiency vs flexibility

Furthermore, in recent years FPGAs have gained a significant role in reconfigurable computing. Reconfigurable computing is a term used to describe a computer architecture that combines the flexibility of software running on general purpose processors with the efficiency of high performance computing fabrics, like FPGAs. The main difference with "traditional" architectures is the ability of the hardware to adapt during runtime by altering itself and "loading a new circuit" in the computation fabric each time a new task demands it. Partial

Reconfiguration takes this advantage one step further, by allowing an FPGA to alter only a part of itself during runtime, while the rest of the logic continuous to operate normally without compromising the integrity of the computation running on those parts of the device that are not being reconfigured. This technique leads to reduction of the amount of resources required to implement a given function, with consequent reductions in cost and power consumption, provides flexibility in the algorithms/protocols available to an application and accelerates configurable computing by enabling a design to be ready to correspond to new computation requirements much faster.

The objective of this diploma thesis is the exploration of the Partial Reconfiguration (RP) technique on FPGAs and to apply the knowledge acquired to implement a cryptographic system, by hot-swapping accelerators, with respect to the incoming input streams, targeting to accelerate and increase the throughput of our system. During this thesis, we make a comparison of the different ways to apply PR on the Xilinx Zynq-7000 SoC device and we present the compromises needed to be done in order to integrate different peripherals with different architecture on a PR design. We also discuss the floorplanning and decoupling strategies that have to be applied to ensure a design’s expected operation. Finally, we make a comparison between a system that has no PR capabilities with a system that can take advantage of the technique and apply it to adapt to an application’s requirements and accelerate the total computation time needed.

1.2 Chapter Organization

In this section, we will describe the structure of this work and what each chapter is about. This thesis consists of 7 chapters which are organized as follows:

- Chapter 2 deals with cryptography. Firstly, the concepts of cryptography in general and modern cryptography are discussed. Then, AES and SHA3 algorithm specifications are presented.
- Chapter 3 gets into the theory of hardware design and FPGAs are introduced, with a brief historic retrospection of how they emerged and an architecture explanation. Then, the available CAD tools are presented along with the Hardware Description Languages (HDL) used to program FPGAs. Finally, Zynq and ZC702 Evaluation board are presented.
- Chapter 4 deals with the technical background of this thesis. In this chapter, Partial Reconfiguration is discussed along with the Zynq available interfaces to perform PR and limitations to the reconfigurable fabric. AMBA and AXI4-Stream are introduced

along with the methodology to implement an AXI4-Stream compliant interface. Finally, AXI DMA and AXI DMA in Scatter/Gather mode are presented.

- Chapter 5 is about the employed workflow for implementing a reconfigurable design on the ZC702 evaluation board. We discuss the modifications needed in order to integrate the cryptographic peripherals to the device, the static design generation, the PR workflow and the application developed to test the system.
- Chapter 6 does an evaluation of the employed work. We present details about the implemented peripherals' resource utilization, the acceleration times, the resulted floorplanning and reconfiguration times. Finally, a benchmark is conducted that tries to show the benefits of Partial Reconfiguration by taking advantage of the technique and applying it to accelerate computation times.
- Finally, in Chapter 7 we summarize our works and make proposals for future work to be made.

This page is intentionally left blank

2 Cryptography

2.1 Introduction

The term cryptography originates from the Greek words κρυπτός (kryptós) = "hidden, secret"; and γράφειν (graphein) = "writing"; and is about the practice and analysis of techniques for secure communication in the presence of third parties called adversaries. More generally, cryptography deals with methods and protocols that prevent third parties from reading private messages and is related to various aspects of information security such as data integrity, data confidentiality, authentication and non-repudiation.

Traditionally, cryptography was synonymous with encryption, the transformation of a readable text to apparent nonsense. The originator of the encrypted text (Alice) shared the decoding technique with the recipient (Bob) preventing any unwanted units (Eve) from recovering the original information. Since World War I and the creation of the first computers, the techniques used to accomplish this task have become more and more complex and cryptography in general has started to find a wider spectrum of applications including methods for data integrity checking, digital signatures and authentication amongst others.

Modern cryptography is based on mathematical theory and computer science to accomplish its goals. Cryptographic algorithms are designed around problems that are hard to be solved with the existing knowledge and, although such systems can be broken in theory, it is impractical for any adversary to do so. However, theoretical advances in computer science and mathematics and the swiftly evolution of computing technology makes imperative for these methods to be continually updated and adapted.

Modern cryptography can be divided to many fields of study. The most important ones are the symmetric-key cryptography and asymmetric (public) - key cryptography. In symmetric-key cryptography it is computationally “easy” to determine the decryption key knowing only the encryption key (where practically in most cases the keys are the same) and vice versa. In public-key cryptography the encryption key is made public while the decryption key is kept private. The sender selects an encryption/decryption key pair and others can send him messages encrypted with the public key, but only he can decrypt them using his private key. [1]

2.2 Advanced Encryption Standard (AES)

2.2.1 Introduction

The Advanced Encryption Standard (AES) is a specification for the symmetric encryption of data established by the U.S National Institute of Standards and Technology (NIST) in 2002. Originally called Rijndael (Rijndael is a play on the names of the two inventors), it was developed by Belgian cryptographers, Joan Daemen and Vincent Rijmen, who submitted their proposal at NIST during the AES selection process. The standard specifies a symmetric block cipher algorithm that can process data blocks of 128 bits, using cipher keys with length of 128, 192 and 256 bits. Rijndael was designed to handle additional block sizes and key lengths but they were not adopted in the standard.

On 2 of January 1997, NIST announced the initiation of the AES development effort and made a formal call for algorithms on 12 of September 1997. The announcement was followed by a standardization process in which fifteen competing designs were presented and evaluated before the Rijndael algorithm was selected. Finally, the AES standard was announced in the United States by the NIST as U.S FIPS PUB 197 (FIPS 197) on 26 of November, 2001.

AES became effective as a federal government standard on 26 of May 2002, is included in the ISO/IEC 18033-3 standard and is the first and only publicly accessible cipher approved by the National Security Agency (NSA) for top secret information when used in an NSA-approved cryptographic module. [2]

2.2.2 Algorithm Specification

Overview

[3] Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing Nb bytes, where Nb is the block length divided by 32. For this standard, the length of the input block, the output block and the State is 128 bits, so Nb = 4. At the start of the encryption or decryption algorithm the input is copied into the State array and operations are conducted on it. At the end, the final State value is copied to the output.

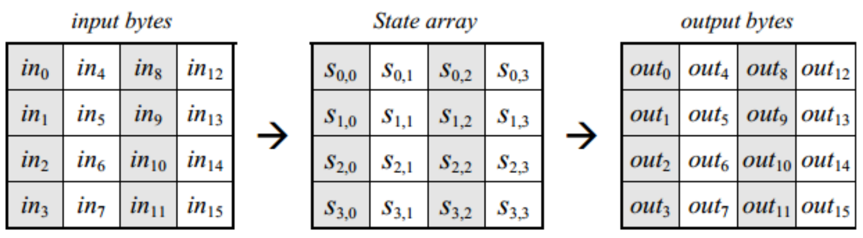

input bytes
State array
output bytes
in_0 in_4 in_8 in_{12}
in_1 in_5 in_9 in_{13}
in_2 in_6 in_{10} in_{14}
in_3 in_7 in_{11} in_{15}
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
out_0 out_4 out_8 out_{12}
out_1 out_5 out_9 out_{13}
out_2 out_6 out_{10} out_{14}
out_3 out_7 out_{11} out_{15}

Figure 2 State array input and output

The length of the Cipher Key, K, is 128, 192, or 256 bits. Just like Nb, the key length is represented by Nk = 4, 6, or 8. The number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr and the values of Nr are portrayed in Figure 3.

	Key Length *(Nk words)*	**Block Size** *(Nb words)*	**Number of Rounds** *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 3 Key-Block-Round Combinations

In the next subsections, we discuss the Rijndael substitution array S-Box, the Key Expansion routine and then we present the encryption - decryption algorithms.

S-Box

S-Box provides the non-linearity of the algorithm and serves as a look-up table.

The S-Box used by SubBytes () transformation in encryption algorithm is presented in hexadecimal form in Figure 4.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
x	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 4 Substitution values for the byte xy (in hexadecimal format)

For example, {53} is substituted by value {ed} (row: 5, column: 3).

Similarly, the S-Box used by InvSubBytes () transformation in decryption algorithm is presented in Figure 5.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
x	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 5 Inverse S-box

Key Expansion

The AES algorithm takes the Cipher Key, K, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of Nb (Nr + 1) words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted [w_i], with i in the range 0 <= i < Nb(Nr + 1).

The Key Expansion transformation is presented as pseudo-code below:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

SubWord() takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word.

RotWord() takes a word [a0,a1,a2,a3] as input, performs a cyclic permutation, and returns the word [a1,a2,a3,a0].

Rcon[i] (round constant word array) contains the values given by [x^{i-1},{00},{00},{00}], with x^{i-1} being powers of x (x is denoted as {02}) in the field GF(2^8) (note that i starts at 1, not 0).

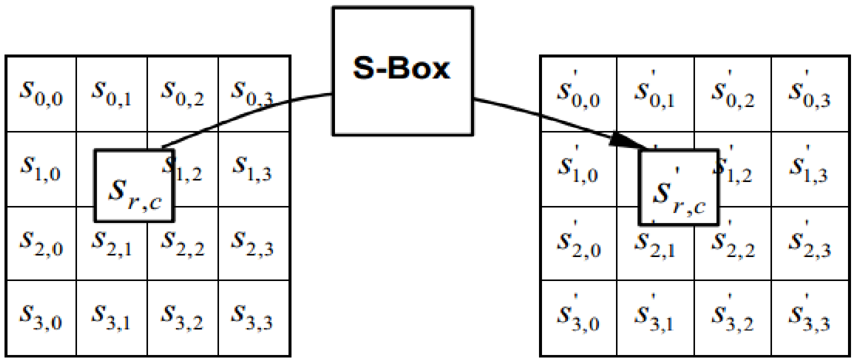
Cipher (Encryption Algorithm)

The cipher is presented in pseudo-code below. The transformations SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() are described in the following pages. The array w[] contains the key schedule derived by the Key Expansion routine.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

SubBytes () Transformation

SubBytes () Transformation substitutes each byte of the State with the corresponding value of S-Box.



S-Box
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{r,c}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{r,c}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 6 SubBytes()

ShiftRows () Transformation

In the ShiftRows() transformation, the bytes in the last 3 rows of the State are shifted through different offsets, while the first row is not shifted.

Specifically, the ShiftRows() transformation proceeds as shown in Figure 7.

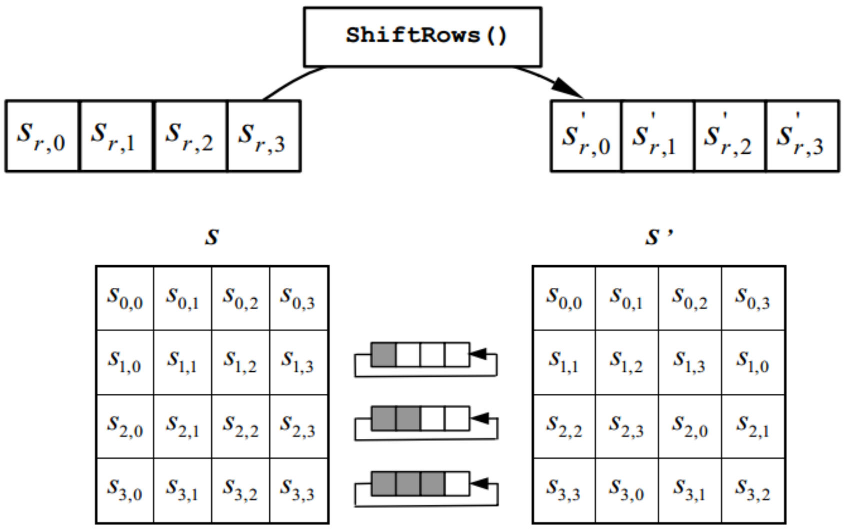

ShiftRows()
$s_{r,0}$ $s_{r,1}$ $s_{r,2}$ $s_{r,3}$
$s'_{r,0}$ $s'_{r,1}$ $s'_{r,2}$ $s'_{r,3}$
S
S'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 7 ShiftRows()

This has the effect of moving bytes to “lower” positions in the row, while the “lowest” bytes wrap around into the “top” of the row.

MixColumns () Transformation

The MixColumns() transformation operates on the State column-by-column, where the columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial a(x), given by $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$.

Thus, $s'(x) = a(x) \otimes s(x)$:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

Figure 8 illustrates the MixColumns**()** transformation.

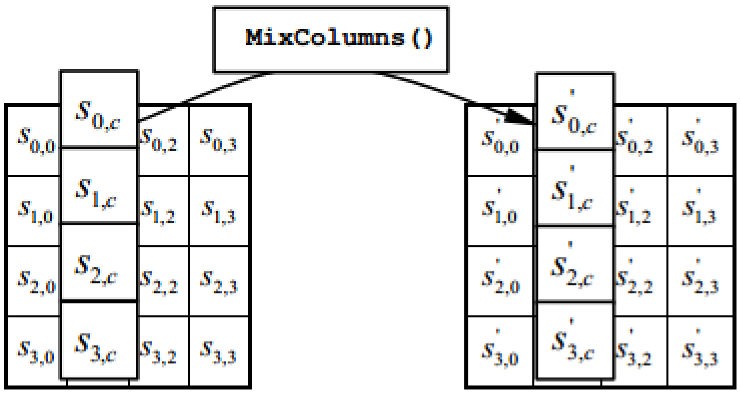


Figure 8 MixColumns()

AddRoundKey () Transformation

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of Nb words from the key schedule Those Nb words are each added into the columns of the State, such that.

$$[\, S'_{0,c}, S'_{1,c}, S'_{2,c}, S'_{3,c} \,] = [\, S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus [w_{round*Nb+c}\,]$$

where $[w_i]$ are the key schedule words and round is a value in the range $0 \leq round \leq Nr$.

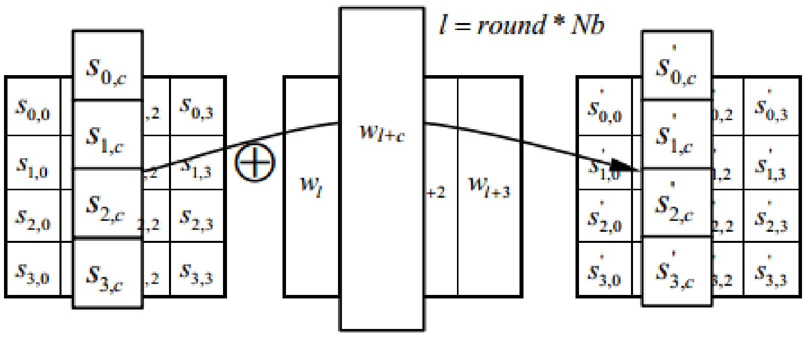


Figure 9 AddRoundKey()

Inverse Cipher (Decryption Algorithm)

The Cipher transformations can be inverted and implemented in reverse order to produce the Inverse Cipher. The Inverse Cipher is presented in pseudo-code below. The transformations used in the Inverse Cipher, InvShiftRows(), InvSubBytes(), InvMixColumns() are described in the following subsections while AddRoundKey() is its own reverse since it only involves an application of bitwise XOR and remains the same as in Cipher.

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)
      InvSubBytes(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

InvShiftRows() Transformation

InvShiftRows is the inverse of ShiftRows transformation and acts over the State array as portrayed in Figure 10.

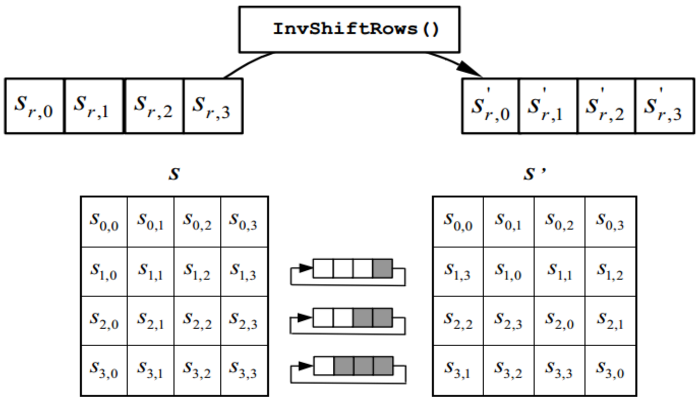

InvShiftRows()
$s_{r,0}$ $s_{r,1}$ $s_{r,2}$ $s_{r,3}$
$s'_{r,0}$ $s'_{r,1}$ $s'_{r,2}$ $s'_{r,3}$
s
s'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,3}$ $s_{1,0}$ $s_{1,1}$ $s_{1,2}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,1}$ $s_{3,2}$ $s_{3,3}$ $s_{3,0}$

Figure 10 InvShiftRows()

InvSubBytes() Transformation

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse S-Box is applied to each byte of the State.

InvMixColumns() Transformation

InvMixColumns() is the inverse of the MixColumns() transformation. Here the columns are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by $a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$

Furthermore, $s'(x) = a^{-1}(x) \otimes s(x)$. Thus:

$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$

$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$

$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$

$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$

2.3 SHA3

2.3.1 Introduction

SHA-3 is the newest member of the Secure Hash Algorithm family. The SHA-3 standard was released by NIST on 5th of August 2015 and is a subset of the cryptographic primitive family Keccak. Keccak was designed by Guido Bertoni, Joan Daemen (who also co-designed the Rijndael cipher with Vincent Rijmen), Michaël Peeters, and Gilles Van Assche and was the winner amongst 51 candidates of the NIST hash function competition.

In 2006, NIST started to organize the NIST hash function competition to create a new hash standard because of the successful attacks on MD5 and SHA-0 and theoretical attacks on SHA-1. NIST perceived the need for an alternative, dissimilar cryptographic hash, to emphasize on diversity and provide some complementary implementation and performance characteristics to those of the older members of SHA family. Thus, SHA-3 is not meant to replace SHA-2, as there not any significant attack on SHA-2 has been demonstrated, but it is there to provide an alternative with some extra characteristics than its predecessors.

By the end of 2008, Keccak was accepted as one of the 51 candidates and in July 2009, it was amongst the 14 algorithms that were selected for the second round. Keccak advanced to the last round in December 2010 and on 2nd of October 2012, it was selected as the winner of the competition. In 2014, the NIST published a draft FIPS 202 "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions" which was finally approved on 5th of August 2015 and SHA-3 became a hashing standard.

The SHA-3 family consists of the four cryptographic hash functions SHA3-224, SHA3-256, SHA3-384 and SHA3-512 and the two extendable-output functions (XOFs), SHAKE128 and SHAKE256. A XOF is a function on bit strings in which the output can be extended to any desired length. The suffixes “128” and “256” indicate the security strengths that these two functions can generally support, in contrast to the suffixes for the hash functions, which indicate the hash lengths. SHAKE128 and SHAKE256 are the first XOFs that NIST has standardized. [4], [5]

2.3.2 Algorithm Specification

In this subsection, the KECCAK-*p* permutations are specified, with two parameters:

1) b: the fixed length of the strings that are permuted, called the *width* of the permutation.
2) $n_{r:}$ the number of iterations of an internal transformation, called *round*.

The KECCAK-*p* permutation with n_r rounds and width b is denoted by KECCAK-*p*[b, n_r], for any b in {25, 50, 100, 200, 400, 800, 1600} and any positive integer n_r.

A round of a KECCAK-*p* permutation (Rnd) consists of a sequence of *5* transformations, which are called the step mappings. The permutation is performed on an array of values for b bits that is repeatedly updated, called the state, which is initially set to the input values of the permutation.

The state for the KECCAK-*p*[b, n_r] permutation contains b bits. There are also two other quantities related to b: b/25 and $\log_2$(b/25), denoted by w and l, respectively. The seven possible values for these variables that are defined for the KECCAK-p permutations are given in Figure 11.

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
ℓ	0	1	2	3	4	5	6

Figure 11 Keccak-p permutations widths and related quantities

It is convenient to represent the input and output states of the permutation as b-bit strings, and to represent the input and output states of the step mappings as 5-by-5-by-w arrays of bits. Thus, the state of permutation is represented as S = S[0] || S[1] || … || S[b-2] || S[b-1] (where || is the concatenation string) while the state is represented as **A**, where **A**[x,y,z] = S[w(5y+x)+z].

Step Mappings

The 5 step mappings that comprise a round of KECCAK-p(b,nr) are denoted by θ, ρ, π, χ and ι. In the specifications below, the input state array is denoted as **A**, while the output state array as **A′**.

Specification of θ

1. For all pairs (x, z) such that $0 \le x < 5$ and $0 \le z < w$, let

   $C[x, z]=\mathbf{A}[x, 0, z] \oplus \mathbf{A}[x, 1, z] \oplus \mathbf{A}[x, 2, z] \oplus \mathbf{A}[x, 3, z] \oplus \mathbf{A}[x, 4, z]$.

2. For all pairs (x, z) such that $0 \le x < 5$ and $0 \le z < w$ let

   $D[x, z]=C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod w]$.

3. For all triples (x, y, z) such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let

   $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus D[x, z]$.

Specification of ρ

```
1. For all $z$ such that $0 \le z < w$, let $\mathbf{A}'[0, 0, z] = \mathbf{A}[0, 0, z]$.
2. Let $(x, y) = (1, 0)$.
3. For t from 0 to 23:
      a. for all z such that $0 \le z < w$, let $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, (z-(t+1)(t+2)/2) \bmod w]$;
      b. let $(x, y) = (y, (2x+3y) \bmod 5)$.
4. Return A′.
```

Specification of π

```
1. For all triples $(x, y, z)$ such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let
      $\mathbf{A}'[x, y, z]=\mathbf{A}[(x + 3y) \bmod 5, x, z]$.
2. Return A′.
```

Specification of χ

```
1. For all triples $(x, y, z)$ such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let
      $\mathbf{A}'[x, y, z] = \mathbf{A}[x, y, z] \oplus ((\mathbf{A}[(x+1) \bmod 5, y, z] \oplus 1) \cdot \mathbf{A}[(x+2) \bmod 5, y, z])$.
2. Return A′.
```

Specification of ι

```
1. For all triples $(x, y, z)$ such that $0 \le x < 5$, $0 \le y < 5$, and $0 \le z < w$, let $\mathbf{A'}[x, y, z] = \mathbf{A}[x, y, z]$.
2. Let $RC = 0^w$.
3. For $j$ from 0 to $l$, let $RC[2j–1] = rc(j+7ir)$.
4. For all $z$ such that $0 \le z < w$, let $\mathbf{A'}[0, 0, z] = \mathbf{A'}[0, 0, z] \oplus RC[z]$.
5. Return $\mathbf{A'}$.
```

where function rc is denoted below:

```
1. If $t$ mod 255 = 0, return 1.
2. Let $R$ = 10000000.
3. For $i$ from 1 to $t$ mod 255, let:
      a. $R = 0 \| R$;
      b. $R[0] = R[0] \oplus R[8]$;
      c. $R[4] = R[4] \oplus R[8]$;
      d. $R[5] = R[5] \oplus R[8]$;
      e. $R[6] = R[6] \oplus R[8]$;
      f. $R = \mathrm{Trunc}_8[R]$.
4. Return $R[0]$.
```

KECCAK-p[b, nr]

Given a state array A and a round index i_r, the round function Rnd is the transformation that results from applying the step mappings θ, ρ, π, χ, and ι, in that order, i.e.:

$$\mathrm{Rnd}(A, ir) = \iota(\chi(\pi(\rho(\theta(A)))), i_r)$$

Specification of KECCAK-$p[b, n_r]$(S) algorithm:

```
1. Convert $S$ into a state array, $\mathbf{A}$
2. For $i_r$ from $12+2l-n_r$ to $12+2l-1$, let $\mathbf{A} = \mathrm{Rnd}(\mathbf{A}, i_r)$.
3. Convert $\mathbf{A}$ into a string $S'$ of length $b$.
4. Return $S'$.
```

Sponge Construction

The sponge construction is a framework for specifying functions on binary data with arbitrary output length. The construction employs the following three components:

- An underlying function on fixed-length strings, denoted by f,
- A parameter called the rate, denoted by r, and
- A padding rule, denoted by pad.

The sponge construction is presented in Figure 12.

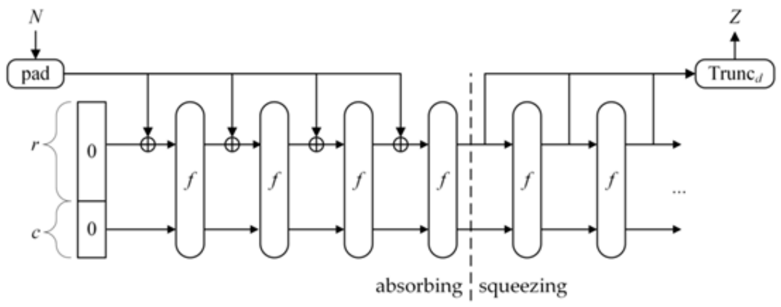


Figure 12 The sponge construction

Specification of SPONGE[f, pad, r](N, d)

1. Let P=N || pad(r, len(N)).
2. Let n=len(P)/r.
3. Let c=b–r.
4. Let P_0, … , P_{n-1} be the unique sequence of strings of length r such that P = P_0 || … || P_{n-1}.
5. Let S=0^b.
6. For i from 0 to n–1, let S=f (S ⊕ (Pi || 0^c)).
7. Let Z be the empty string.
8. Let Z=Z || Trunc$_r$(S).
9. If d≤|Z|, then return Trunc$_d$(Z); else continue.
10. Let S=f(S), and continue with Step 8.

KECCAK

Specification of pad10*1(x,m)

1. Let j = (– m – 2) mod x.
2. Return P = 1 || 0^j || 1.

Specification of KECCAK[c]

KECCAK[c] = SPONGE[KECCAK-p[1600, 24], pad10*1, 1600–c].

Thus, given an input bit string N and an output length d,

KECCAK[c] (N, d) = SPONGE[KECCAK-p[1600, 24], pad10*1, 1600–c] (N, d)

SHA-3 FUNCTION SPECIFICATIONS

Given a message M, the four SHA-3 hash functions are defined from the KECCAK[c] function by appending a two-bit suffix to M and by specifying the length of the output, as follows:

SHA3-224(M) = KECCAK[448] (M || 01, 224);

SHA3-256(M) = KECCAK[512](M || 01, 256);

SHA3-384(M) = KECCAK[768] (M || 01, 384);

SHA3-512(M) = KECCAK[1024](M || 01, 512).

Given a message M, the two SHA-3 XOFs, SHAKE128 and SHAKE256, are defined from the KECCAK[c] function specified by appending a four-bit suffix to M, for any output length d:

SHAKE128(M, d) = KECCAK[256] (M || 1111, d),

SHAKE256(M, d) = KECCAK[512] (M || 1111, d).

This page is intentionally left blank

3 *Acceleration with Field-Programmable Gate Arrays (FPGAs)*

3.1 Introduction

Modern systems often rely on micro-processors for a wide variety of applications, from small common systems to large complicated systems with high performance requirements. In these systems, processors take up the task of computing and range from general purpose processors to application specific units, like graphics processing units (GPUs) and other specialized components dedicated to particular purposes. In recent years, the requirements of several application demand units of high performance processing, low power consumption and small physical size. Such a device is a Field-Programmable Gate Array (FPGA), with the additional characteristic of reconfigurability, meaning that FPGAs can be reprogrammed several times even after their manufacturing or installation on a device.

3.2 Field-Programmable Gate Arrays

A Field Programmable Gate Array (FPGA) is a digital integrated circuit (IC) constituted of a matrix of reconfigurable logic blocks which are connected through programmable interconnects. In contrast to common processors an FPGA rewires itself to implement the desired functionality rather than running a software application and can be modified “in the field” more than once, in contrast to Application Specific Integrated Circuits (ASICs).

The reconfigurability of FPGAs makes them a flexible -with a wide range of applications-platform and a viable solution for quick implementation or prototyping of new systems. Despite a little trade-off in performance, with FPGAs engineers can implement, test or alter their designs with ease in comparison with ASICs that take up a large amount of time for prototyping and bear a higher cost. [6]

Furthermore, in recents years FPGAs have gained a significant role in reconfigurable computing. Reconfigurable computing is a term used to describe a computer architecture that combines the flexibility of software running on general purpose processors with the efficiency of high performance computing fabrics, like FPGAs. The main difference with "traditional" architectures is the ability of the hardware to adapt during runtime by altering itself and "loading a new circuit" in the computation fabric each time a new task demands it. Thus, this technique constructs heterogeneous platforms that are capable of reaching higher performance with less power and area consumption. [7]

Additionally, FPGAs in conjunction with general purpose processors can implement a System on a Chip (SoC) and be studied as a unified software-hardware platform approach for designing systems. For all the reasons described above and because nowadays FPGAs have shrunk into the deep-submicron region -a fact that has greatly increased their logic capacity- they are used in a wide range of applications including Digital Communications, Digital Signal Processing (DSP), Image Processing, Automotive, Medical, Security, Aerospace and Defense and others.

3.3 Architecture

FPGAs consist of three fundamental components Configurable Logic Blocks (CLBs), I/O blocks and programmable routing. Most FPGAs also include a number of embedded hard blocks that perform certain tasks, such as hardwired memories, multipliers and DSP (Digital Signal Processing) Blocks that are interconnected with the CLBs. An abstract design of a generic FPGA can be seen in Figure 13.

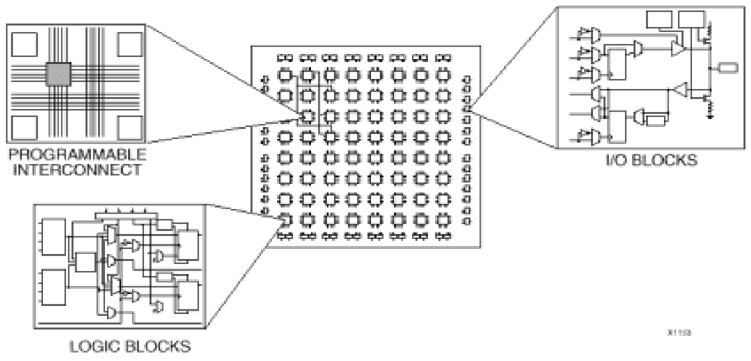


Figure 13 Typical FPGA architecture

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. A CLB contains four slices and each slice is composed by two logic cells.

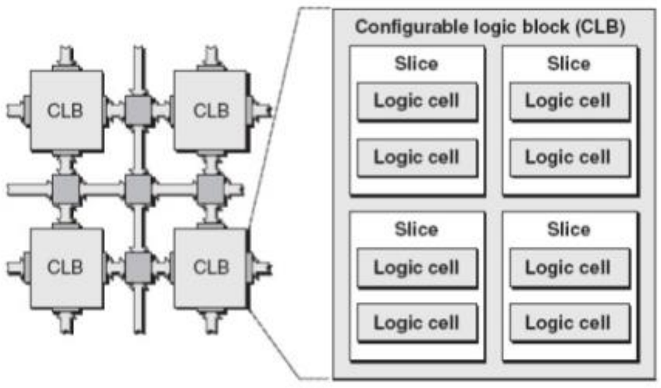


Figure 14 An array of CLBs composed by four slices and two logic cells per slice

A logic cell consists of a Look Up Table (LUT) with -usually- four inputs, a multiplexer and a flip-flop.

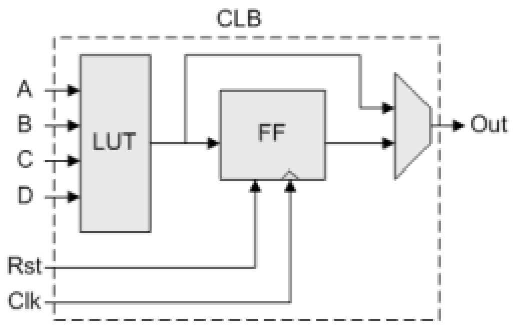


Figure 15 simplified schematic of a logic cell

Finally, the programmable high speed I/O blocks enable the FPGA to communicate with a variety of devices in the outside world, such as sensors or other peripherals. 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. [8]

In the next subsections, there is a more thorough presentation of some of the above essential components.

Configurable Logic Blocks (CLBs) and Look up Tables (LUTs)

Inside an FPGA, CLBs are organized in a two-dimensional array connected through the programmable routing channels. In general, a typical CLB consists of a number of logical cells, organized in slices. A logic cell has the aforementioned architecture with a LUT, a multiplexer and a flip-flop.

An N-input LUT is a functional unit that is able to compute any function of N-inputs. Its operation resembles that of truth table of a logic function. Given the truth table of a logic function, a LUT is responsible for generating the corresponding output by matching the input with the correct output of one of the 2^N possible outcomes of the table. Besides of basic functions, LUTs can implement an N-bit shift register or can be used as a distributed memory module of N-bits. Also, several LUTs can be combined to implement more complex logic functions.

Since the very first FPGAs, the size of a look-up table is a subject of debate. On one hand, larger LUTs would reduce the number of logic blocks and wiring delay between them by allowing more complex operations to be performed on a single logic block. Yet, larger LUTs would introduce additional delays due to requirement of larger multiplexers and would result in increasing wasted resources if the implemented functionality was of lower demands. Empirical studies have shown that the 4-LUT structure the best trade-off between area and delay for a wide range of applications.

Hard Blocks

Modern FPGAs, apart from LUTs, include additional blocks fixed into the silicon providing additional functionality, to reduce the required area and provide increased speed compared to building those functionalities from primitives. Examples of hardwired blocks include multipliers, 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.

Interconnection

As we already mentioned, logic blocks in a typical FPGA are identical and are organized in a two-dimensional, island-like architecture. This metaphor is used to describe the logic blocks as islands floating in a sea of interconnection, where the connection of the blocks through the routing resources is performed with the use of connection blocks and switch boxes as show in the figure below.

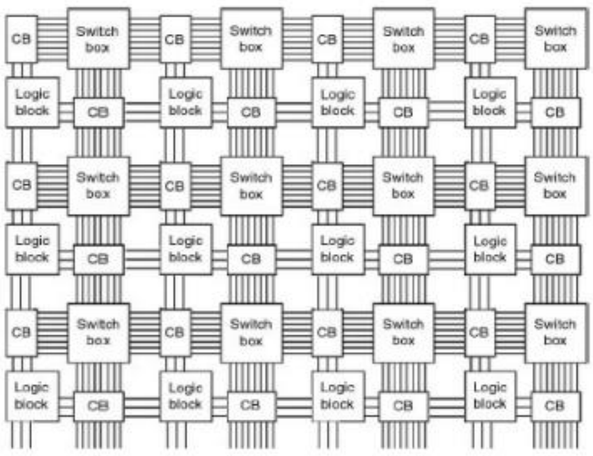


Figure 16 An island-style architecture with connect blocks and switch boxes [9]

A connection block allows -through programmable switches- logic block I/Os to be assigned to arbitrary horizontal and vertical tracks, while a 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. A key fact of this architecture is that it separates interconnection from logic allowing long-distance routing to be accomplished without consuming logic block resources.

3.4 *Programming and CAD Tools*

Implementing a circuit on a modern FPGA requires a high number of logic to be correctly configured. It is evident that this task is impossible to be performed by a human designer who would have to manually reprogram each element individually. That is why Specific Computer-Aided Design Tools (CADs) are crucial for the efficient programming of FPGAs.

Programming of FPGAs is usually done with the description of the circuit at a higher level of abstraction, typically using a Register-Transfer Level (RTL) Hardware Description Language (HDL), such as Verilog and VHDL (a more detailed description of Verilog follows in section 3.4.1, since it is the language that the crypto-IPs and their interfaces are implemented). HDLs should not be considered as software programming languages, as software programming languages are sequential in nature, while HDLs are not. They can allow both sequential and concurrent execution and include ways of describing the propagation time and signal strengths. [10]

Furthermore, in recent years, another method has gained much publicity that is named High-Level Synthesis (HLS). HLS tools provide an even greater abstraction level by describing circuits in higher level languages, such as C, and transforming the code into RTL implementations. This method resembles more to a software-development-like procedure rather than a hardware one, and provides a quick way to implement software algorithms to hardware.

After describing a circuit in RTL specification, the CAD tools are used to implement the design and produce the final bitstream, which is the binary file that comprises the necessary information to specify the state of each individual element inside the FPGA. The process of converting an RTL specification to bitstream is presented in section 3.4.2.

3.4.1 Verilog

Verilog (portmanteau of the words "verification" and "logic") is a hardware description language used to model electronic circuits. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. Verilog was created by Prabhu Goel, Phil Moorby and Chi-Lai Huang, who were working for Gateway Design Automation Inc., between late 1983 and early 1984. At 1990, Cadence Design Systems acquired Gateway Design Automation and became the owner of Verilog and Verilog-XL, an HDL simulator that would become the de facto standard for the next decade. At 1995, Cadence decided to make the language available for open standardization and transferred Verilog into the public domain under the Open Verilog International (OVI) organization. The language was submitted to IEEE and became IEEE Standard 1364-1995. There were also two other standard revisions performed, in 2001 and in 2005, while in 2009 Verilog and SystemVerilog (a superset of Verilog with object-oriented programming capabilities) were merged into IEEE Standard 1800-2009.

Verilog was built with C programming language in mind, so both share some common syntax characteristics. Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Also, its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C. However, there are some differences including using begin/end instead of curly braces {}, requiring that variables be given a definite bit-width size, instead of assuming it from the type of the variable, and some other minor ones.

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of 'wire' includes signal values (1, 0, floating, undefined) and signal strengths (strong, weak, etc.). This style allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's value is resolved by a function of the source drivers and their strengths.

Not all statements in Verilog are synthesizable. To make the simulation task easier and more efficient, the language defines “system tasks”, which can handle simple I/O and other aspects of simulation. These tasks include, simple file operations, printing debugging messages, defining simulation time etc. [10]

3.4.2 Design Flow and Tools

After describing a design with a hardware description language, like Verilog, a process, which maps the design to a specific FPGA architecture, takes place. This process is broken down to five steps, Logic Synthesis, Technology mapping, Placement, Routing and finally, Bitstream Generation.

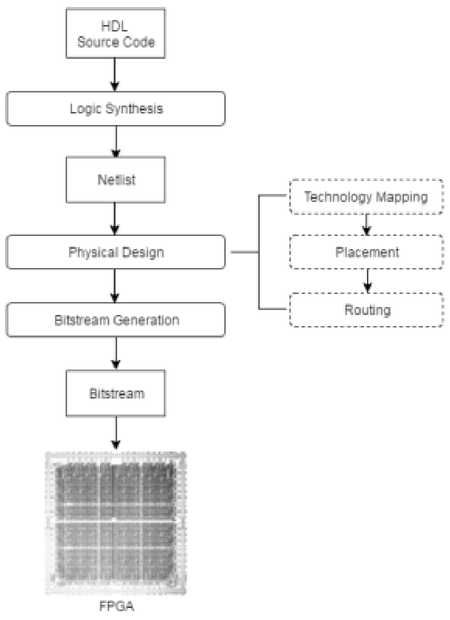


Figure 17 A typical FPGA design flow

Logic Synthesis

The first stage of synthesis converts the circuit description from an HDL file (or a schematic) into a netlist of basic gates. The next step is converting the above netlist in a netlist of FPGA logic blocks according to the desired synthesis properties (speed, area or power specifications). During this stage, several optimizations take place removing redundant logic and simplifying the design.

Technology Mapping

In this stage, several LUTs and registers are packed into one logic block respecting limitations imposed by the FPGA architecture. Also, a number of optimizations are available depending on the goals the designer has chosen. The optimization goal in this phase is to pack LUTs so that the number of logic blocks and routed signals is minimized.

Placement

In this stage, heuristic placement algorithms determine which logic block within the FPGA should implement each of the logic blocks required by the circuit. The optimization goals are to place connected logic blocks close together to minimize the required wiring (wirelength-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement).

Routing

After placement, 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 of the device is represented as a directed graph. Thus, routing a connection corresponds to finding a path in this routing-resource graph. Since most of the delay in FPGA designs is routing delay, a timing-driven optimization in the routing stage is crucial to minimize overall circuit delay.

Bitstream Generation

In this final step, after the design has been successfully placed and routed, the CAD tool creates a bitstream which can be downloaded to the FPGA and configure it accordingly. [9]

3.5 *ZC702 Evaluation Board*

The ZC702 Evaluation Board is a low-cost evaluation and development board based on the XC7Z020, member of the Xilinx Zynq®-7000 All Programmable SoC family. XC7Z020 combines a Dual-Core ARM® Cortex-A9 Processing System (PS) with a 7 Series Artix-7 FPGA of 85K Programmable Logic Cells (PL) that can target a wide range of applications. Furthermore, the board includes several expansion connectors for easy user access and connection with other peripherals. Specifically, some of the most important (and the ones of interest in this thesis) features of ZC7020 Board are described above: [11]

- Zynq XC7Z020-1CLG484C device
- 1 GB DDR3 component memory (four 256 Mb x 8 devices)
- 128 Mb Quad SPI flash memory
- USB 2.0 ULPI (UTMI+ low pin interface) transceiver
- Secure Digital (SD) connector
- USB JTAG interface using a Digilent module
- Clock sources
- USB-to-UART bridge
- I2C bus multiplexed to:
    - Si570 user clock
    - ADV7511 HDMI codec
    - M24C08 EEPROM (1 kB)
    - 1-To-16 TCA6416APWR port expander
    - RTC-8564JE real time clock
    - FMC1 LPC connector
    - FMC2 LPC connector
    - PMBUS data/clock
- Configuration options:
    - Quad SPI flash memory
    - USB JTAG configuration port (Digilent module)
    - Platform cable header JTAG configuration port
    - 20-pin PL PJTAG header
    - 20-pin PS JTAG header

Also, the board includes a plethora of LEDs -for status indication or user defined activities-, DIP switches and pushbuttons.

A system level block diagram of Zynq-7000 AP SoC is shown in Figure 18:

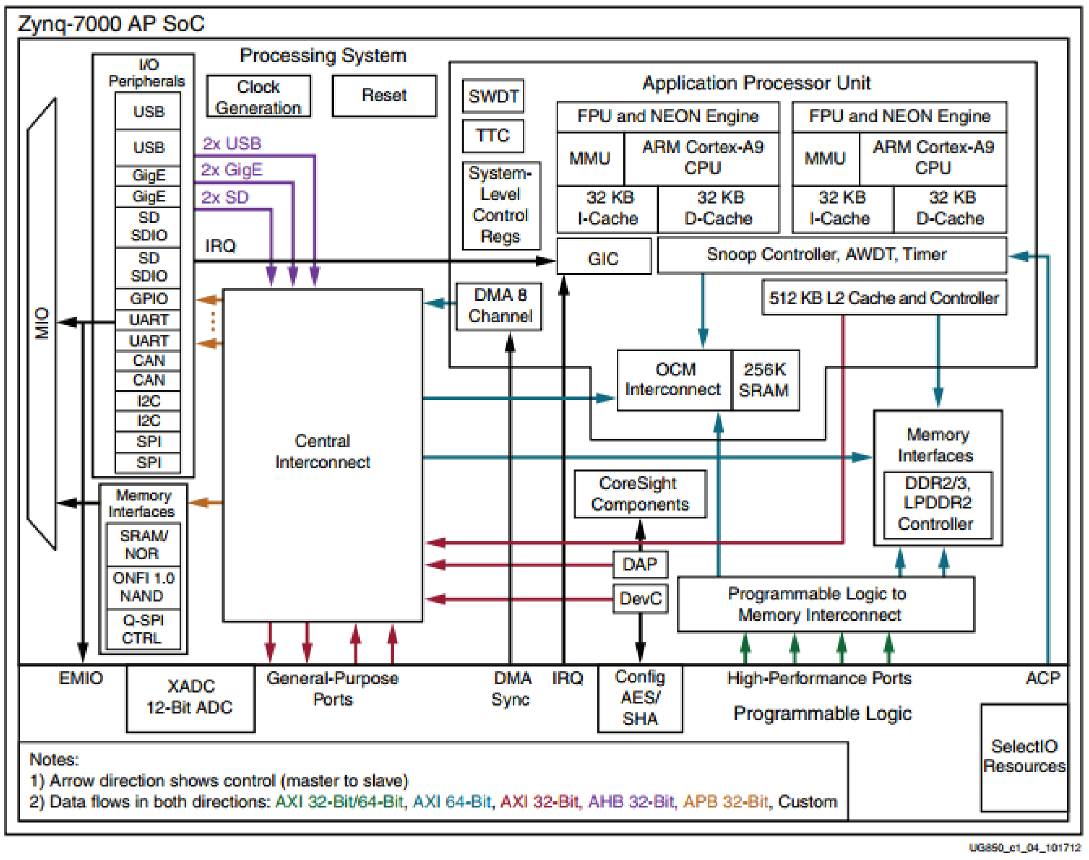


Figure 18 Zynq-7000 AP SoC

The ZC7020 evaluation board can be used for several applications. In this thesis, the board is used for designing a cryptosystem with reconfiguration capabilities, where in runtime a part of the FPGA is reconfigured and different cryptographic accelerators are hot-swapped with respect of the incoming input streams and requirements, in order to maximize the throughput and efficiency of our application. In the next chapter, before presenting our implemented system, we try to make a thorough explanation of the technical background needed and all the IPs/technologies used in this thesis.

4 Technical Background

4.1 Partial Reconfiguration

4.1.1 Overview

One major advantage of FPGAs is that they provide the flexibility of "in the field" programming and re-programming without going through re-fabrication process. Partial Reconfiguration (PR) takes this flexibility one step further, by allowing partial modification of an operating FPGA design, while the rest of the design continues to function normally. This is done by loading a partial configuration file, usually a partial BIT file. After a full BIT file configures the FPGA, partial BIT files can be downloaded to modify reconfigurable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured. Figure 19 illustrates the basic premise of Partial Reconfiguration. [12]

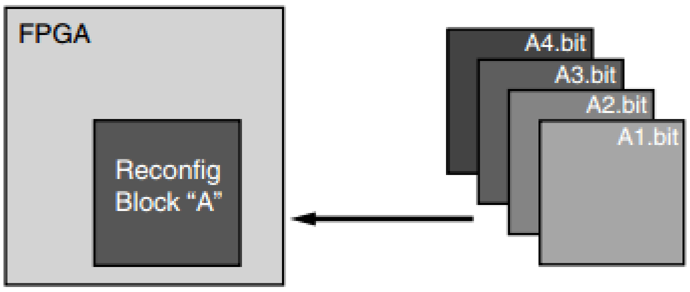


Figure 19 Basic Premise of Partial Reconfiguration

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

There are many reasons why Partial Reconfiguration can be advantageous. These include:

- Reducing the size of the FPGA required to implement a given function, with consequent reductions in cost and power consumption
- Providing flexibility in the choices of algorithms or protocols available to an application
- Enabling new techniques in design security
- Improving FPGA fault tolerance
- Accelerating configurable computing
- Enables new types of FPGA designs that would be otherwise impossible to implement

PR flow requires the implementation of multiple configurations which ultimately results in full bitstreams for each configuration, and partial bitstreams for each Reconfigurable Module. The number of configurations required varies by the number of modules that need to be implemented. However, all configurations share the same top-level, or static, placement and routing results. These static results are exported from the initial configuration, and imported by all subsequent configurations using checkpoints. Thus, the static logic is identical in every configuration and only the reconfigurable logic differs.

Not all logic is permitted to be actively reconfigured. Global logic and clocking resources must be placed in the static region to not only remain operational during reconfiguration, but to benefit from the initialization sequence that occurs at the end of a full device configuration.

Logic that can be placed in a Reconfigurable Module in Zynq-7000 AP SoC devices includes:

- All logic components that are mapped to a CLB slice in the device. This includes LUTs (look-up tables), FFs (flip-flops), SRLs (shift registers), RAMs, and ROMs.
- Block RAM and FIFO
- DSP blocks: DSP48E1
- PCIe® (PCI Express): Entered using PCIe IP

All other logic must remain in static logic and must not be placed in an RM, including:

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

4.1.2 Configuration Modes in Zynq-7000 AP SoC

Zynq-7000 AP SoC supports three modes to configure the device with a partial bitstream, JTAG, PCAP, and ICAP. JTAG is a well-known standard, used by the industry for prototyping and debugging, and is the default interface used to configure the device through Vivado Logic Analyzer. Its main purpose it to provide a way for quick testing and debugging while implementing the final design, so it not meant to be used afterwards. PCAP is the primary configuration mechanism for Zynq-7000 AP SoC residing in the Device Configuration Interface (DevC) inside the PS, while ICAP is used to manage partial reconfiguration completely within the PL (either through the PR Controller IP or through HWICAP module). The three interfaces are mutually exclusive and cannot be used simultaneously. Switching between ICAP and PCAP is possible through devc.CTRL [PCAP_PR] bit. PL configuration paths are shown in Figure 20 . Note that JTAG can be used only in non-secure mode, while PCAP and ICAP can be used both in secure and non-secure mode.

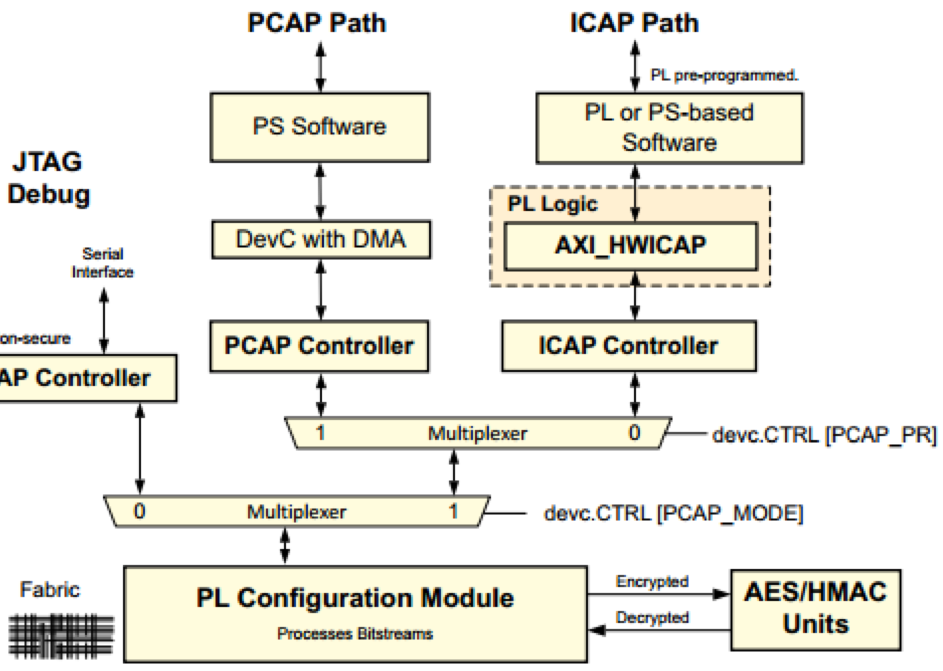


Figure 20 PL configuration paths [13]

4.2 Advanced Microcontroller Bus Architecture (AMBA)

The ARM® Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, onchip interconnect specification for the connection and management of functional blocks in SoC designs and is used to facilitate the development of multi-processor designs with large numbers of controllers and peripherals. AMBA was introduced by ARM in 1996 and since then the protocols of AMBA family have become the de facto standard for 32-bit embedded processors, because they are well documented and can be used without royalties.

Through years, the scope of AMBA has gone far beyond microcontroller buses. Today, it is widely used on a wide range of ASIC and SoC parts. In recent years, Xilinx has adopted parts of the specification, specifically Advanced Extensible Interface 4 (AXI4) (that is discussed later in this chapter). AXI4 is the default interface used in a plethora of IPs created by Xilinx, and the protocol that users are encouraged to work with in their designs.

The design principles of AMBA originate from the fact that an important aspect of a SoC is not only which components it utilizes, but also the interconnection of them. Its objectives are the facilitation of development of embedded microcontroller products -by allowing the reuse of IP cores and peripherals across diverse IC processes- and the encouragement of modular system design -by improving processor independence and allowing development of reusable peripherals and IP libraries- while minimizing the silicon infrastructure and providing high performance and low power on-chip communication. The AMBA 4 specification defines the following buses/interfaces: [14]

- 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 thesis, we focus on AXI4 protocol and specifically on AXI4-Stream.

4.2.1 AMBA 4 AXI4-Stream

The AXI4-Stream protocol 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 generates data, to a single slave, that receives data, but also to connect 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.

The following byte definitions are used in this specification: [15]

Data byte	A byte of data that contains valid information that is transmitted between the source and destination.
Position byte	A byte that indicates the relative positions of data bytes within the stream. This is a placeholder that does not contain any relevant data values that are transmitted between the source and destination.
Null byte	A byte that does not contain any data information or any information about the relative position of data bytes within the stream.

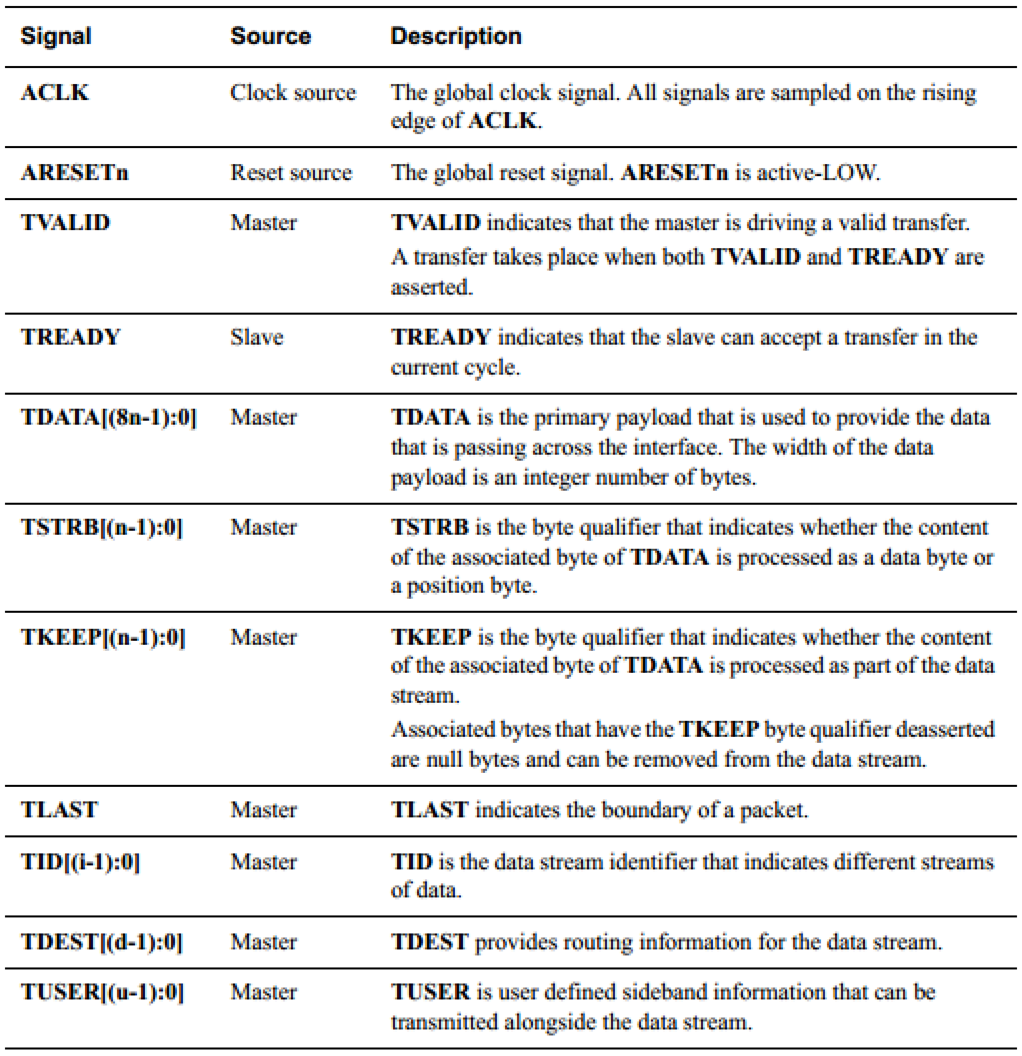
The following stream terms are used in this specification:

Transfer	A single transfer of data across an AXI4-Stream interface. A single transfer is defined by a single TVALID, TREADY handshake.
Packet	A group of bytes that are transported together across an AXI4-Stream interface. A packet is similar to an AXI4 burst. A packet may consist of a single transfer or multiple transfers. Infrastructure components can use packets to deal more efficiently with a stream in packet-sized groups.

Frame	The highest level of byte grouping in an AXI4-Stream. A frame contains an integer number of packets. A frame can be a very large number of bytes, for example an entire video frame buffer.
Data Stream	The transport of data from one source to one destination. A data stream can be: • a series of individual byte transfers • a series of byte transfers grouped together in packets.

Types of streams include byte streams, continuous aligned streams, continuous unaligned streams and sparse streams. A byte stream is the transmission of a number of data and null bytes. On each TVALID, TREADY handshake, any number of data bytes can be transferred. Null bytes have no meaning and can be inserted or removed from the stream. A continuous aligned stream is the transmission of a number of data bytes where every packet has no position or null bytes. A continuous unaligned stream is the transmission of a number of data bytes where there are no position bytes between the first data byte and the last data byte of each packet (but can have any number of contiguous position bytes at the start, at the end, or both at the start and end of a packet). A sparse stream is the transmission of a number of data bytes and position bytes. All data and position bytes must be maintained and transmitted from source to destination.

n: Data bus width in bytes.
i : **TID** width. Recommended maximum is 8-bits.
d: **TDEST** width. Recommended maximum is 4-bits.
u: **TUSER** width. Recommended number of bits is an integer multiple of the width
of the interface in bytes



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. **ARESETn** is active-LOW.
TVALID	Master	**TVALID** indicates that the master is driving a valid transfer. A transfer takes place when both **TVALID** and **TREADY** are asserted.
TREADY	Slave	**TREADY** indicates that the slave can accept a transfer in the current cycle.
TDATA[(8n-1):0]	Master	**TDATA** is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TSTRB[(n-1):0]	Master	**TSTRB** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as a data byte or a position byte.
TKEEP[(n-1):0]	Master	**TKEEP** is the 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 data stream.
TLAST	Master	**TLAST** indicates the boundary of a packet.
TID[(i-1):0]	Master	**TID** is the data stream identifier that indicates different streams of data.
TDEST[(d-1):0]	Master	**TDEST** provides routing information for the data stream.
TUSER[(u-1):0]	Master	**TUSER** is user defined sideband information that can be transmitted alongside the data stream.

Table 1 AXI4-Stream signals

The TVALID and TREADY are the handshake signals which determine when information is passed across the interface. For a transfer to occur both the TVALID and TREADY signals must be asserted. Either TVALID or TREADY can be asserted first or both can be asserted in the same ACLK cycle. To avoid a deadlock, a master is not permitted to wait until TREADY is asserted. Once TVALID is asserted it must remain asserted until the handshake occurs. On the other hand, a slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY and if a slave asserts TREADY, it is permitted to deassert TREADY before TVALID is asserted.

The following figures demonstrate examples of the handshake sequence. The arrow shows when the transfer occurs.

TVALID before TREADY handshake

In Figure 21 the master presents the data and control information and asserts the TVALID signal HIGH. Once the master has asserted TVALID, the data or control information from the master must remain unchanged until the slave drives the TREADY signal HIGH, indicating that it can accept the data and control information. In this case, transfer takes place once the slave asserts TREADY HIGH. The arrow shows when the transfer occurs.

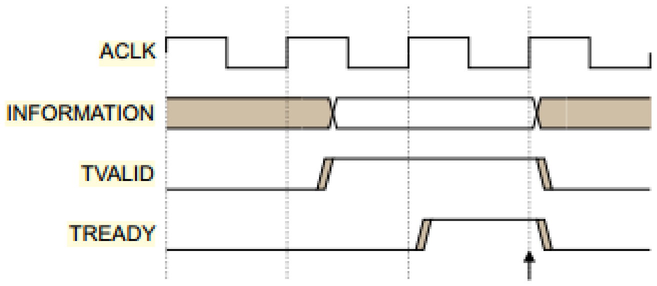


Figure 21 TVALID before TREADY handshake

TREADY before TVALID handshake

In Figure 22 the slave drives TREADY HIGH before the data and control information is valid. This indicates that the destination can accept the data and control information in a single cycle of ACLK. In this case, transfer takes place once the master asserts TVALID HIGH. The arrow shows when the transfer occurs.

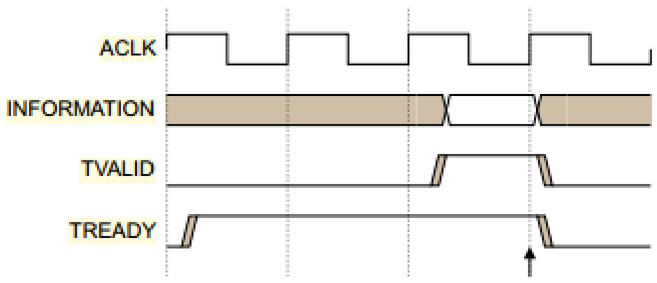


Figure 22 TREADY before TVALID handshake

TVALID with TREADY handshake

In Figure 23 the master asserts TVALID HIGH and the slave asserts TREADY HIGH in the same cycle of ACLK. In this case, transfer takes place in the same cycle as shown by the arrow.

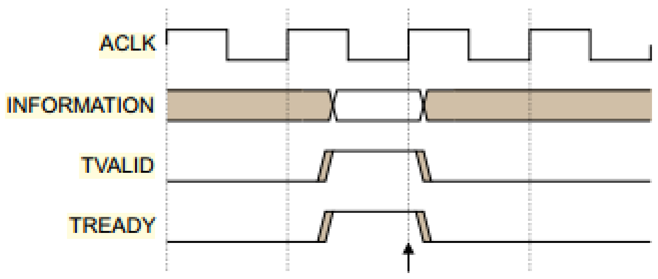


Figure 23 TVALID with TREADY handshake

For the simplest transfer to occur, TVALID, TREADY and TDATA are sufficient (However the protocol allows to omit TREADY and TDATA but it is not recommended). In this thesis, for the implementation of the interfaces of the crypto-IPs, we have used TVALID, TREADY, TDATA, TLAST and TKEEP. Because SHA3-512 operates with inputs of arbitrary length, TKEEP is used as a byte qualifier to inform the IP which bytes of the last packet of the input are valid and which are not. The remaining signals have not been used and are not discussed further. [15]

4.3 AXI Direct Memory Access (DMA)

The Xilinx® LogiCORE™ IP AXI Direct Memory Access (AXI DMA) core is a soft Xilinx IP core that provides high-bandwidth direct memory access between memory and AXI4-Stream peripherals. AXI DMA can operate in two different modes, Direct Register mode (or Simple DMA mode) and Scatter/Gather mode (SGDMA), where, in both modes, user can check for the completion of transaction either by polling the hardware or through interrupts. Simple DMA allows an application to define a single transaction between AXI DMA and the device (that means that after every transaction the processor has to define a new one), while SGDMA allows an application to define a list of transactions in memory which the hardware will process without further application intervention (thus, the processor is kept less busy and is allowed to do other tasks to keep itself and the hardware busy). It also utilizes interrupt coalescing techniques to minimize the interrupts from the hardware to the processor. Simple DMA provides a lower performance but less FPGA-resource intensive mode which may be sufficient for several applications, while SGDMA provides a faster, but more difficult to set up in user application, mode. In this thesis, Scatter/Gather mode was the choice of preference.

In AXI DMA, initialization, status, and management registers are accessed through an AXI4-Lite slave interface, while primary high-speed DMA data movement, between system memory and stream target, is done through the AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master, and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master. The MM2S channel and S2MM channel operate independently. The AXI DMA provides 4 KB address boundary protection (when configured in non-Micro DMA), automatic burst mapping, as well as the ability to queue multiple transfer requests using nearly the full bandwidth capabilities of the AXI4-Stream buses. Furthermore, the core provides byte-level data realignment allowing memory reads and writes starting at any byte offset location.

The MM2S channel supports an AXI Control stream for sending user application data to the target IP. For the S2MM channel, an AXI Status stream is provided for receiving user application data from the target IP. The optional Scatter/Gather Engine fetches and updates buffer descriptors from system memory through the AXI4 Scatter Gather Read/Write Master interface.

Figure 24 illustrates the functional composition of the core.

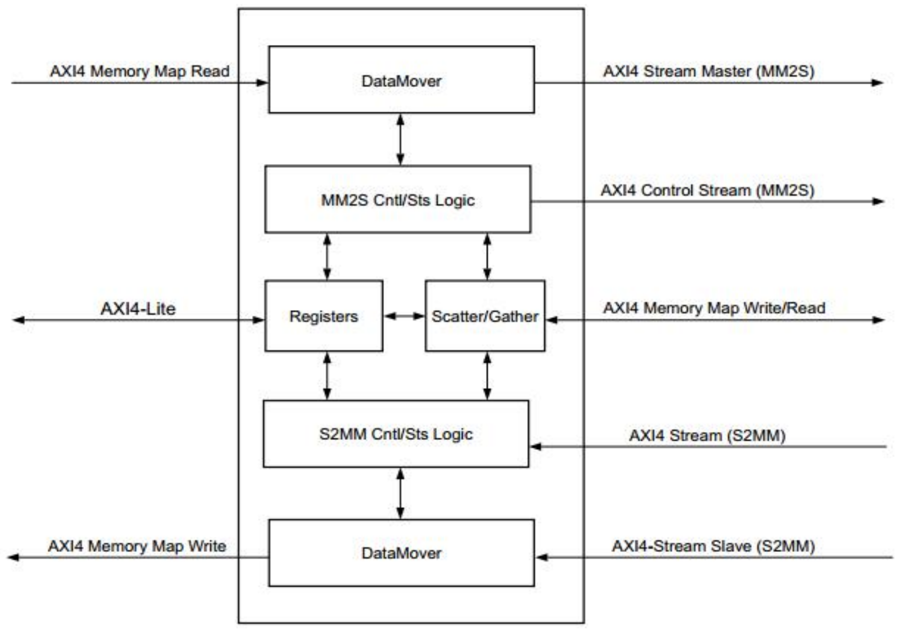


Figure 24 AXI DMA block diagram [16]

5 *Employed Workflow on ZC702 Evaluation Board*

5.1 Setting up the IPs for HW integration

5.1.1 Overview

For the purposes of this thesis 4 different IPs where selected. These cores implement the algorithms AES-128, AES-192, AES-256 and SHA3-512, are designed by Homer Hsing, can be found at [17], [18] and are distributed under Apache License, Version 2.0.

In order to integrate the above IPs in our project and enable communication between the cores and the Zynq PS, we designed appropriate interfaces for each one of them. For each core, 2 interfaces were implemented, one slave (whose purpose is to accept data) and one master (whose purpose is to send data). The interfaces were written in Verilog, designed to be compliant with the AXI4-Stream protocol and were implemented with the help of Vivado IP Packager. Also, for the Partial Reconfiguration requirements a simple Decoupler was implemented, that will be used to cut-off all outgoing signals from the crypto-IPs during reconfiguration, and a "dummy" IP, that will be used to synthesize the static logic and before employing the Partial Reconfiguration workflow.

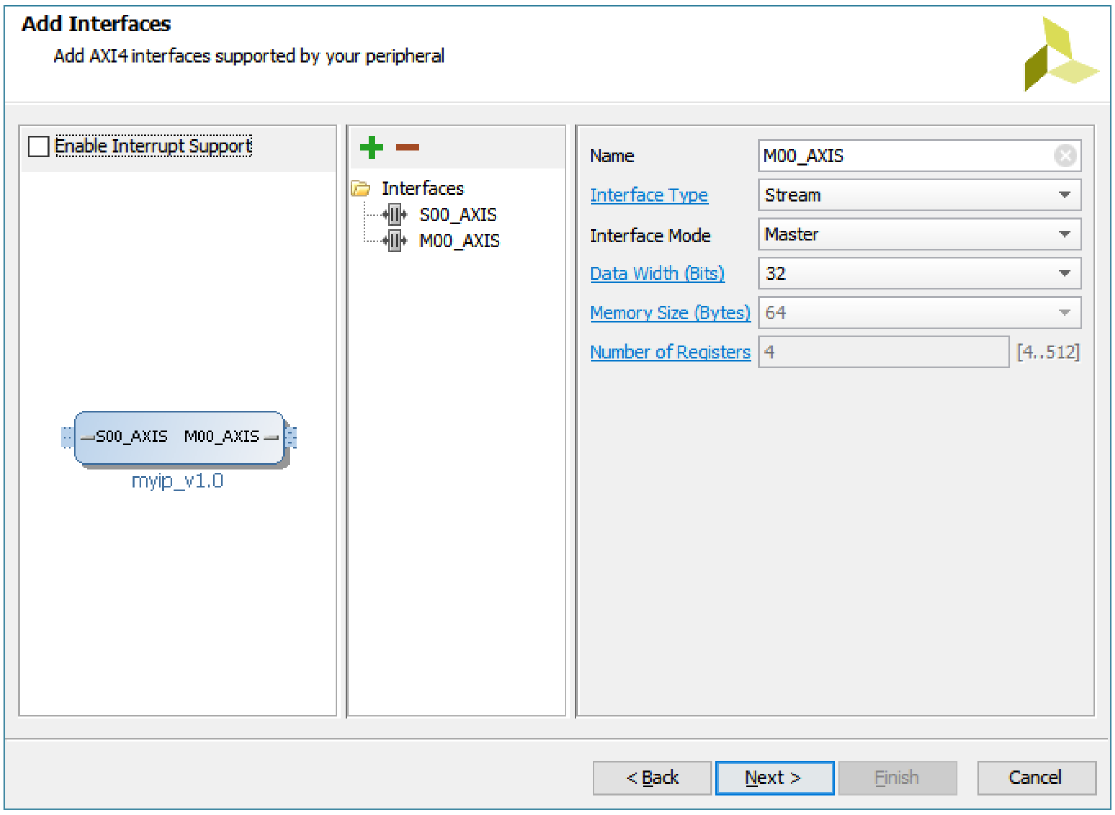
The complete procedure for implementing and packaging a custom IP in Vivado is presented in the following subsection.

5.1.2 Creating and Packaging custom IPs in Vivado IP Packager

Crypto Cores

In order to implement our custom IPs, we created a new Vivado project to serve as a parent project and repository for all the custom cores. Custom IP creation can be done inside any Vivado project, but we decided that it was a good idea to have a separate project for maintaining and testing all the cores and include that repository to all our different projects.

After selecting "Create a new AXI4 peripheral" in Vivado IP Packager, the wizard prompts to specify our core's information and define its interfaces. For our design, each peripheral has a master and a slave AXI4-Stream interface of 64-bit data width. Partial Reconfiguration demands that every Reconfigurable Module (RM) has identical signals as seen outside of the peripheral (or ground to zero any additional signals that are not been used). AES cores can absorb all the input (plaintext and key) in a single clock cycle, while SHA3-512 core accepts data by a 64-bit-wide input port, so 64 bit data width was chosen for all the interfaces. That seemed convenient as later the IPs will communicate with the Zynq PS through its High Performance ports, which are also 64-bits-wide. The IP Packager wizard has a default value of 32 for data width that cannot be changed, but this can be done later by modifying the source code.


Add Interfaces

Add AXI4 interfaces supported by your peripheral

☐ Enable Interrupt Support

Interfaces
- S00_AXIS
- M00_AXIS

Name	M00_AXIS
Interface Type	Stream
Interface Mode	Master
Data Width (Bits)	32
Memory Size (Bytes)	64
Number of Registers	4 [4..512]

Figure 25 View of Vivado IP Packager wizard

After specifying the interfaces, Vivado automatically generates three template files, one for the top file and one for each interface. The top file contains the definitions of all signals visible outside of the peripheral and instantiates the interfaces, while the interface files contain definitions of their own parameters and some indicative logic that users can modify and extend to their needs.

For our custom IPs we modified these files to only contain TDATA, TVALID, TREADY, TLAST and TKEEP signals of AXI4-Stream, as described in 4.2.1. TDATA carries the data that the modules process, TVALID and TREADY are used for implementing the AXI4-Stream handshake, while TLAST notifies when the last data of a packet is being sent. Also, because SHA3-512 accepts an input of arbitrary length, TKEEP is used to notify the module's slave of how many bytes of its last 64-bit input are actually valid.

Furthermore, we modified the top files to instantiate the cryptographic modules and added some additional logic to synchronize the cryptographic modules with their interfaces. In each interface file, a Finite State Machine -that decides the interface's behavior- was implemented, and additional logic -that ensures the correct data transfer from and to the interface- was added.

Finally, an external reset port was added, which is used to reset the IP after the Partial Reconfiguration. In Figure 26, the layout of the crypto IPs is presented. The IP contains the instantiated crypto module along with its interfaces, its clock and the reset signals.

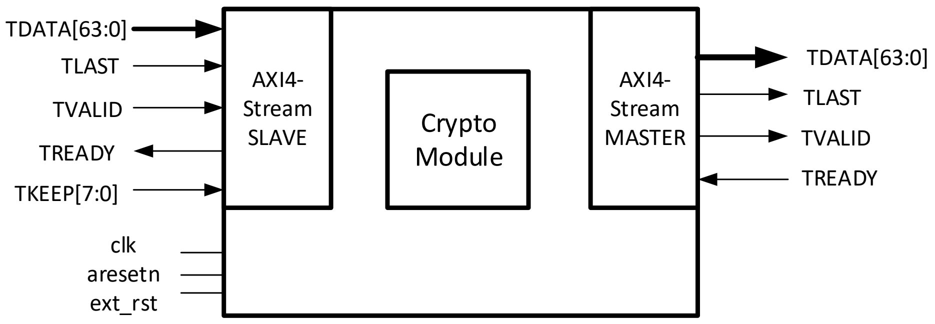


Figure 26 Crypto cores' layout

"Dummy" IP

After implementing our custom IPs, we created an additional "dummy" peripheral named "reconf_peripheral" that will be used to synthesize the static logic before the Partial Reconfiguration workflow. After synthesizing the static logic, reconf_peripheral will be used as a black box in which all the IPs will be added to implement all the different configurations. This "dummy" IP only contains a top file with definitions of the signals seen outside the peripheral and no additional logic. In Figure 27 the reconf_peripheral IP is shown, as presented inside Vivado IP Integrator.

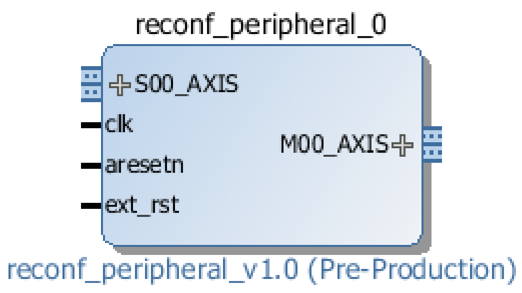


Figure 27 "Dummy" IP

Decoupler IP

Because the logic inside a Reconfigurable Partition is modified while the device is operating, the static logic connected to outputs of Reconfigurable Modules (RMs) must ignore all incoming data during the PR procedure. The reconfigurable logic is in an unknown state during reconfiguration and RMs do not output valid data until PR is complete and the RMs are reset. [12] A system designer should apply a decoupling strategy to his design by finding a way to isolate the logic until reconfiguration is complete or by simply ignoring all incoming signals from an RM if this does not affect the system's operation. In this project, we implemented a Decoupler IP. This IP under normal operation allows all signals to pass through it unaffected, while during Partial Reconfiguration it grounds all outcoming signals from the crypto IPs' master interfaces and sends a reset signal to them. The Decoupler is controlled by the Zynq PS through an AXI4-Lite interface and is shown in Figure 28.

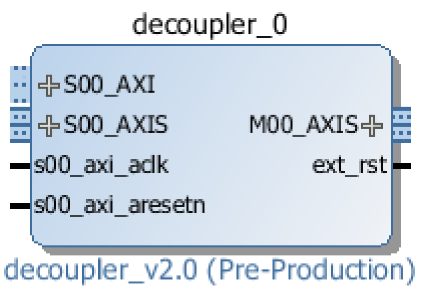


Figure 28 Decoupler IP

After the above process, Vivado packages each IP and our peripherals are ready to be used inside the Vivado IP Integrator.

Usually, to verify the correct operation of logic, the Zynq Bus Functional Model (BFM) [19] is used, which enables the functional verification of PL by mimicking the PS-PL interfaces. In this project, because Zynq BFM requires an additional license, the verification was done by developing a testbench which tried to simulate all the different scenarios the IPs could encounter. After the simulation, the cores where downloaded to Zynq along with the Integrated Logic Analyzer (ILA) core to monitor the internal signals and verify their correct operation.

5.2 *System Generation*

At this point, our custom IPs are generated and we are ready to create our system. This step is done prior to the Partial Reconfiguration workflow, so the "dummy" IP will be used for all instances instead of the cryptographic modules. In this section, we present a system generation process with one cryptographic peripheral, but this process can be generalized to include as many peripherals the FPGA's resources can handle.

The first step is to include our IP repository into this new project. After including the repository, a new block design is created and the Zynq Processing System (PS) IP is added. The IP should be re-customized to comprise all necessary ports and interfaces needed. The IRQ_2FP port is enabled and will later be used by the AXI DMA to notify the PS about reading/writing operations. Also, a single PL fabric clock is used for all our PL peripherals, except HWICAP. Any PL clock can theoretically range from 0 to 250 MHz. For our design, the first clock is set to 150 MHz, as it is the maximum value the AXI DMA can operate, while the second is set to 100 MHz which is the HWICAP's maximum one. Finally, we enable as many High Performance (HP) ports as the number of cryptographic peripherals we want to use. After all necessary changes the PS IP should look like Figure 29

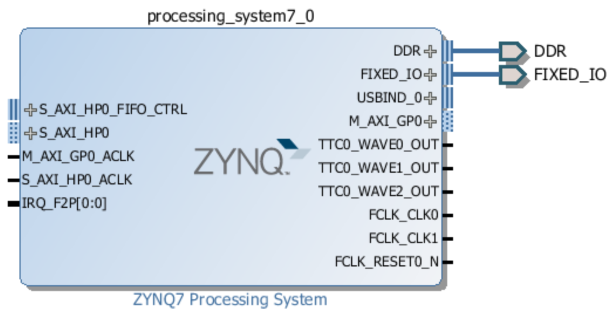
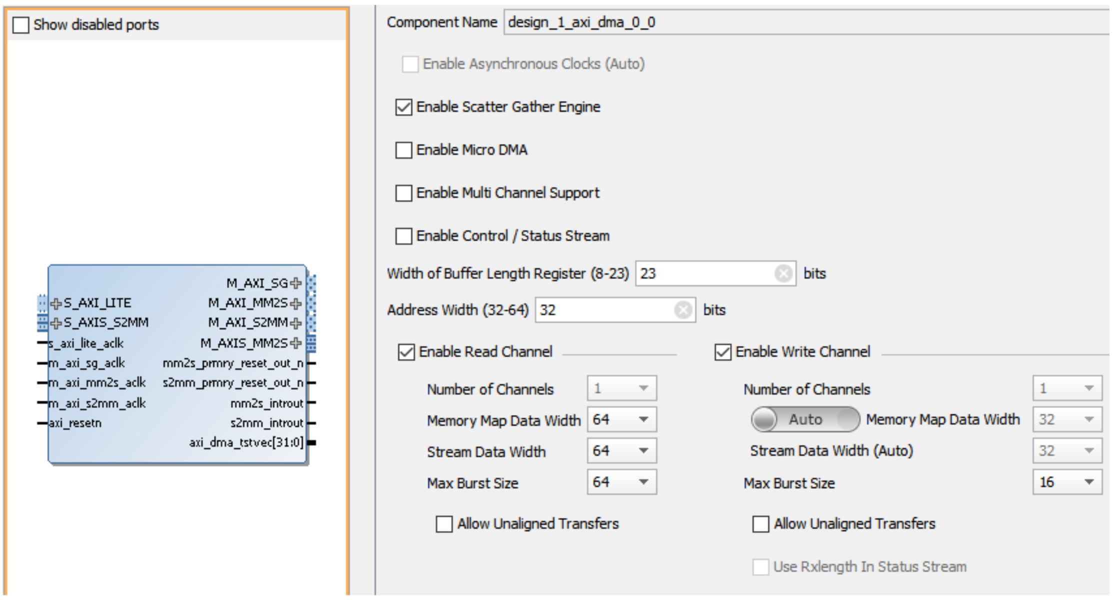


Figure 29 Zynq7 Processing System

After customizing the Zynq PS IP, an AXI DMA IP is added. This IP should also be re-customized to fit the design's needs. More analytically, we enable the Scatter Gather Engine and disable Micro DMA, Multi Channel Support and Control/Status Stream. We also set Width of Buffer Length Register to 23 bits and Stream Data Width to 64 as in out IPs, TDATA field is set to 64 bits wide. After all modifications AXI DMA IP should look like in Figure 30



Component Name design_1_axi_dma_0_0

[ ] Enable Asynchronous Clocks (Auto)
[✓] Enable Scatter Gather Engine
[ ] Enable Micro DMA
[ ] Enable Multi Channel Support
[ ] Enable Control / Status Stream

Width of Buffer Length Register (8-23) 23 bits
Address Width (32-64) 32 bits

[✓] Enable Read Channel
Number of Channels 1
Memory Map Data Width 64
Stream Data Width 64
Max Burst Size 64
[ ] Allow Unaligned Transfers

[✓] Enable Write Channel
Number of Channels 1
Auto Memory Map Data Width 32
Stream Data Width (Auto) 32
Max Burst Size 16
[ ] Allow Unaligned Transfers
[ ] Use Rxlength In Status Stream

Figure 30 View of AXI DMA IP customization wizard

After the above procedure, Vivado runs Connection Automation and automatically makes all the necessary changes to connect the AXI DMA to Zynq PS. To make the connection correctly, Vivado also adds two new peripherals, AXI Interconnect IP and Processor System Reset.

AXI Interconnect IP

AXI Interconnect IP connect one or more AXI memory-mapped Master devices to one or more AXI memory-mapped Slave devices. It has the capability of connecting 1-16 Master devices and 1-16 Slave devices. This means that an AXI Interconnect IP can be utilized to connect up to 16 interfaces. Also, in our design, this module translates the AXI3 compliant data of PS to AXI4 compliant, in which AXI DMA operates.

Processor System Reset

Processor System Reset is intended to implement a Power on Reset. This means that, this IP detects when power is applied to the device and generates a reset impulse that travels through the entire circuit, placing all peripherals to a known state.

At this point, only our custom peripheral with the Decoupler IP needs to be added to the project, as well as the Concat IP which concatenates the AXI DMA interrupt signals and drives them to the PS. Also, the HWICAP IP is added to allow the device's reconfiguration through the ICAP interface. The full implemented design is shown in Figure 31

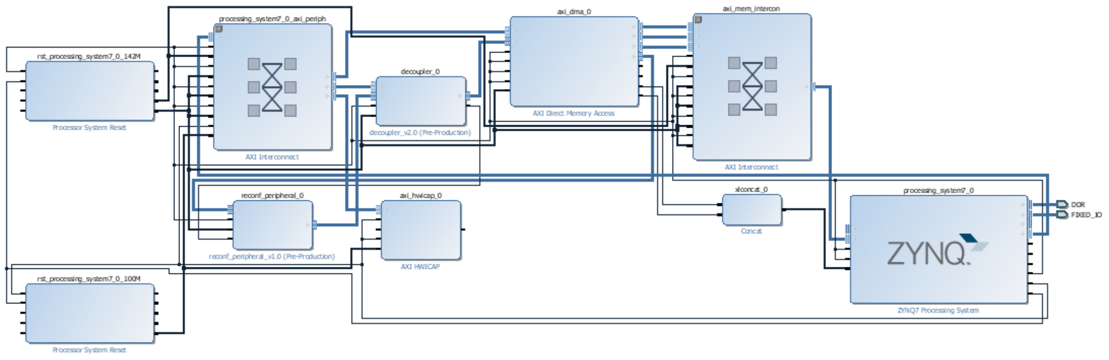


Figure 31 Desing with one reconfigurable peripheral

5.3 *Partial Reconfiguration Workflow*

The following steps summarize processing a PR design in Vivado:

1) Synthesize the static and Reconfigurable Modules separately.
2) Create physical constraints (Pblocks) to define the reconfigurable regions.
3) Set the HD.RECONFIGURABLE property on each Reconfigurable Partition.
4) Implement a complete design (static and one Reconfigurable Module per Reconfigurable Partition) in context.
5) Save a design checkpoint for the full routed design.
6) Remove Reconfigurable Modules from this design and save a static-only design checkpoint.
7) Lock the static placement and routing.
8) Add new Reconfigurable Modules to the static design and implement this new configuration, saving a checkpoint for the full routed design.
9) Repeat Step 8 until all Reconfigurable Modules are implemented.
10) Run a verification utility (pr_verify) on all configurations.
11) Create bitstreams for each configuration. [12]

The process is presented as flowchart in Figure 32.

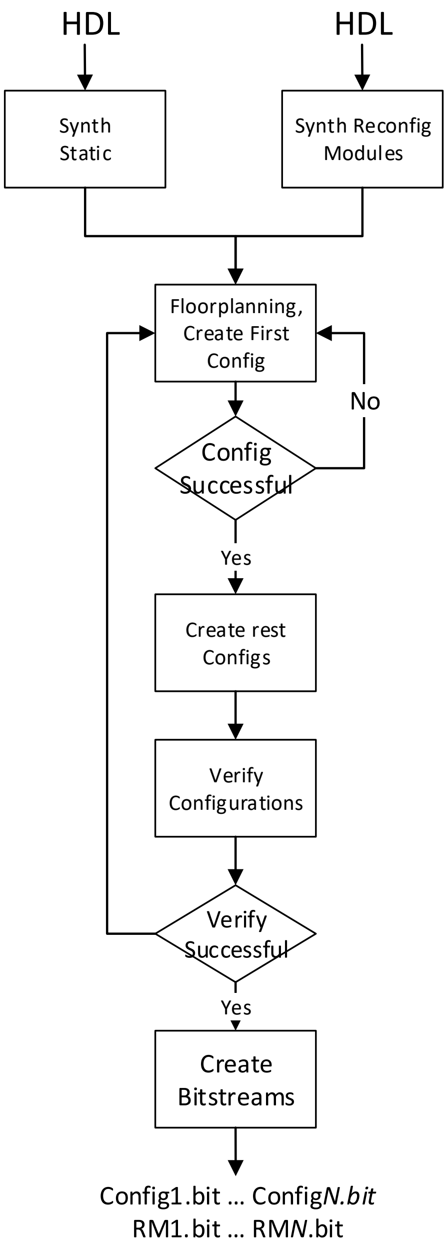


Figure 32 Partial Reconfiguration Workflow

Partial Reconfiguration can be performed to replace part of an IP's logic or even multiple peripherals. For example, consider that there is an IP that is connected to the PS via an AXI interface and implements a very basic function, like an adder. With Partial Reconfiguration, it is possible to reconfigure only the part of the IP that is responsible to implement the adder's operation while the interface of the IP can remain in static logic. This has the advantage of faster reconfiguration times, but requires that the functions implemented have similar architecture and similar interaction with the interfaces. In this thesis, we study the Partial Reconfiguration of an entire peripheral as seen in Figure 33.

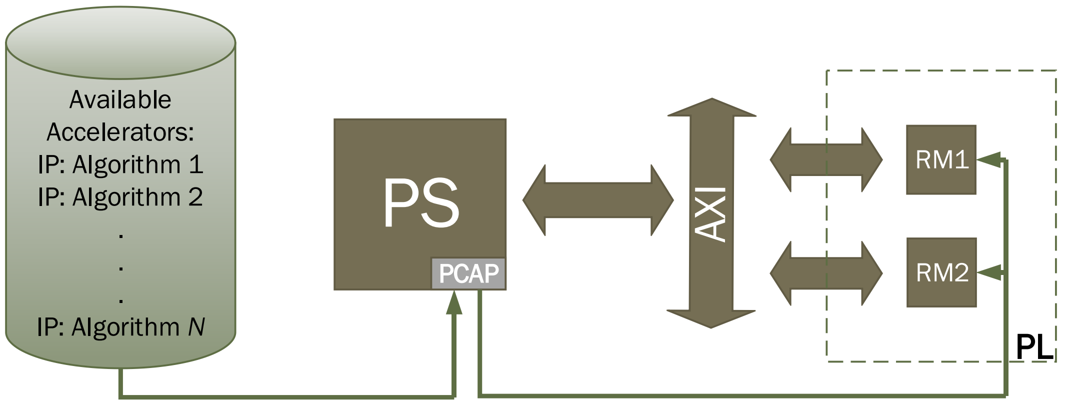


Figure 33 Partial Reconfiguration of entire peripherals

In Figure 33 a system with two Reconfigurable Modules is presented and in this system, there are *N* algorithms available. Whenever the application needs to perform a different operation with a different algorithm, the whole peripheral RM1 or RM2 is hot-plugged and the new algorithm is inserted.

Prior to Vivado 2016.3, PR flow was not supported by the graphic interface of the suite, so everything had to be done in no-project mode by running commands or Tcl scripts. In Vivado 2016.3, PR flow in project mode was introduced, where users have the ability to create PR projects using Vivado's interface guidance. This thesis was implemented in Vivado 2015.4, so PR flow in project mode was not feasible. For our needs, we implemented 8 individual Tcl scripts that automate the whole procedure and can be easily modified to create any future projects. These scripts can be run inside Vivado, where the user -at the end of each script's execution- can set parameters not known in advance, like setting the debug for ILA core or floorplanning the design. For the scripts to operate normally, the user must follow a specific hierarchy for his sources, synthesis/implementation checkpoints and bitstreams, as the Vivado does not create them automatically and they have to be set in advance. They can also be used as generic scripts to guide and implement any design.

In this section, we discuss the operation of the scripts implemented and how they were applied in our design.

1-static_design_creation_with_dummy_ip.tcl

This script generates and synthesizes the system described in Section 5.2. The Design CheckPoint (DCP) produced will be used in scripts 3 and 4 to implement all the configurations.

"Dummy" IP will be used as a black box where all crypto IPs will be loaded prior to implementation.

2-synthesize_reconfig_modules.tcl

This script synthesizes all crypto IPs.

3-create_first_configuration_and_place_static_logic.tcl

This script loads the DCP from script 1 and a DCP from script 2, creates the floorplanning and implements the design. For the floorplanning, the most resource intensive RM is used. If the configuration fails, the user must create a different floorplanning and run the script again. If the configuration is successful, the RM is extracted and static logic is locked and used in script 4 to implement all the other configurations.

4-create_rest_configurations.tcl

This script creates the rest configurations.

5-verify_configurations

This script verifies that all the configurations are compatible with the first one.

6-bitstreams_generation.tcl

This script generates the bitstreams. For every configuration, a full bitstream and a partial one, for each RM, is generated.

7-compressed_bit_generation.tcl

This scripts functions as script 6 but generates compressed bitstreams. Compressed bitstreams can be used to configure the device more quickly.

5.4 Baremetal Application Development

To test the design, a baremetal application was developed. The application selects the desirable interface for Partial Reconfiguration (PCAP or ICAP), initiates the AXI DMA Scatter/Gather engine and handles all the data transferred to/from the peripherals.

The essential function calls to initialize the interfaces and perform Partial Reconfiguration are presented in Table 2 and Table 3.

PCAP	
Function	Operation
XDcfg_CfgInitialize()	Initializes the Device Configuration (DevC) driver
XDcfg_EnablePCAP()	Enables PCAP
XDcfg_SetControlRegister()	Sets the contents of the Control Register of the DevC. To enable Partial Reconfiguration with PCAP we set the XDCFG_CTRL_PCAP_PR_MASK
XDcfg_Transfer()	Starts the bitstream transfer.

Table 2 Functions to utilize PCAP

ICAP	
Function	Operation
XDcfg_CfgInitialize()	Initializes the Device Configuration (DevC) driver.
XDcfg_ClearControlRegister()	Clears the specified bit positions of the Control Register. It is used to clear XDCFG_CTRL_PCAP_PR_MASK and disable PCAP.
XHwIcap_CfgInitialize()	Initializes the HWICAP instance.
XHwIcap_DeviceWrite	Starts the bitstream transfer.

Table 3 Functions to utilize ICAP

In Scatter/Gather DMA the object used to describe a transfer is referred to as a Buffer Descriptor (BD). BDs are allocated in the user application, where the application sets the buffer address, transfer length and control information for the transfer. BD rings are shared by the user application and the hardware. The hardware expects BDs to be setup as a linked list where the last BD in the ring is linked to the first one. The DMA walks through the list by following the next pointer field of a completed BD and stops when the just completed BD is the same as the BD specified in the Tail Ptr register in the hardware.

Within a ring, there are four groups of BDs, where each group consists of 0 or more adjacent BDs :

- Free: The BDs that can be allocated by the application
- Pre-process: The BDs that have been allocated and are under application control. The application modifies these BDs through the driver API to prepare them for DMA transactions
- Hardware: The BDs that have been enqueued to hardware. These BDs are under hardware control (in a state of awaiting processing, in process, or already processed) and should not be modified by the user application.
- Post-process: The BDs that have been processed by the hardware and have returned to the application control. The application can check the transfer status or put them into the Free group.

Figure 34 illustrates the transitions of BDs during a continuous transfer.

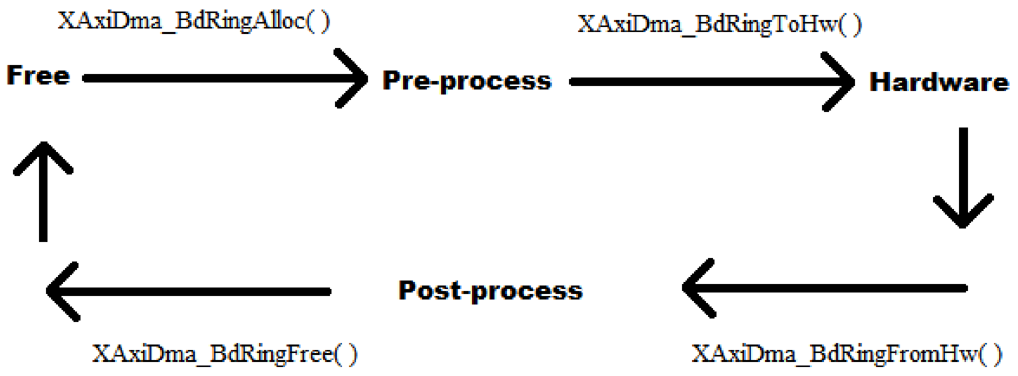


Figure 34 Transitions of BDs during a continuous transfer

Our application creates two BD rings, a TX ring that is used to send data to the SGDMA and an RX ring to receive data from the SGDMA. The BDs of the TX ring are freed and can be allocated to send data to the DMA, while the BDs of the RX ring are allocated and sent to hardware, so that data can be received at any time. The application also sets a coalescing count for each BD ring, which is a packet threshold counter that defines when interrupts will fire. Once the application wants to transmit data to the hardware, it allocates the necessary BDs and sends them to hardware. Once the transmit is done, the TX interrupt handler gets the processed BDs from hardware and sets them under application control, while the RX interrupt handler gets the processed BDs and sends them back to the hardware once again.

The essential function calls to set up the SGDMA and perform transactions are presented in the Table 4.

Function	Operation
XAxiDma_BdRingCntCalc()	Determines how many BDs will fit within a given memory space.
XAxiDma_BdRingCreate()	Creates a BD ring.
XAxiDma_BdRingSetCoalesce()	Sets the coalescing count.
XAxiDma_BdRingAlloc()	Reserve locations in a BD ring
XAxiDma_BdRingToHw()	Enqueue a set of BDs to hardware. The BDs must have previously been allocated by XAxiDma_BdRingAlloc().
XAxiDma_BdRingFromHw()	Returns a set of BDs that have been processed by hardware.

Table 4 SGDMA API functions

This page is intentionally left blank

6 *Evaluation of Workflow*

6.1 Overview

In this chapter, we make an evaluation of our workflow and implementations. First of all, we present the challenges our designs faced and the compromises that were made in order to make Partial Reconfiguration feasible with the different peripherals we had. Then we discuss the floorplanning that was made, which is a crucial part of the PR workflow, and make a comparison for the configuration time need for a partial bitstream in contrast with a full configuration. Then, we make an evaluation of the IPs implemented and the acceleration that was achieved. Finally, we present a fully functional design with two reconfigurable peripherals. This system has two input streams of data that need to be encrypted and has the ability to reconfigure itself with two different or two identical crypto IPs, depending on the input data, in order to accelerate the total computation time.

6.2 General Description of Hardware Implementations

The original AES cores were not suitable to implement Partial Reconfiguration in our device. The original AES128 utilized 86, AES192 100 and AES256 121 Block RAM tiles. Zynq's FPGA (xc7z020clg484-1) has a total of 140 Block RAM tiles spread all over across the device. This means that no more than one peripheral could fit in our device and Partial Reconfiguration would be pointless, because as a result of the high Block RAM utilization spreading across the device, a single peripheral would occupy the entirety of the device's space. Zynq's layout is presented in Figure 35. Block RAM tiles are highlighted as the columns with the brown color.

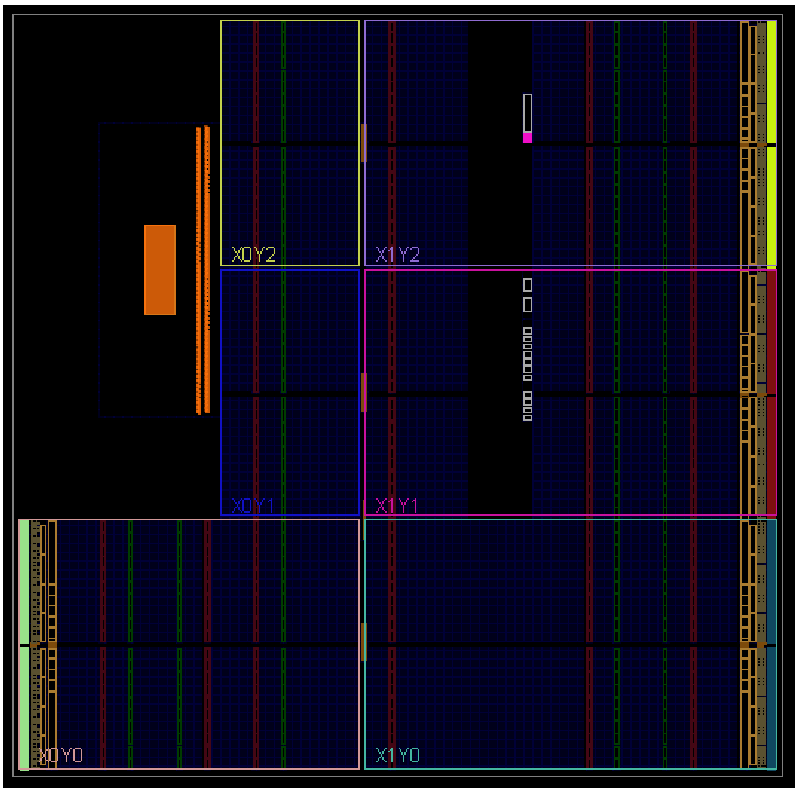


Figure 35 Zynq's (xc7z020clg484-1) layout

For our designs, we ordered Vivado Synthesizer to use Distributed RAM instead of Block RAM, by inserting the "special" comment (* ram_style = "distributed" *) in the source code. This resulted in a much higher LUT utilization (especially in AES256), but it made Partial Reconfiguration feasible because the cores could fit in a smaller space.

In Table 5 we present the device's total resources along with the final utilization of the synthesized IPs. The "dummy" IP (reconf_peripheral) is not presented as it is not a real IP and contains no logic. Also, as it can be seen, AES256 utilizes ~38.21% of the device's total resources by itself. That will have an impact on the final floorplanning, because it will be difficult to find such a big contiguous area without elements that cannot be reconfigured.

Zynq (xc7z020clg484-1)	Slice LUTs (53200)	Slige Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)
AES128	10775	6337	4064	2032
AES192	12602	8130	4448	2224
AES256	20330	10642	7744	3872
SHA3-512	6258	2361	0	0
Decoupler	220	169	0	0

Table 5 IP cores' utilization

Also, the original AES cores had the ability to be fed with their entire input (128 bit plaintext and 128/192/256 bit cipherkey) in a single clock cycle, while Zynq's High Performance ports are 64-bit wide. SHA3-512 core was more convenient as it uses a 64-bit input and has a smaller footprint in logic. As a result, the 64-bit wide interfaces implemented, limited AES cores' original performance, while in SHA3-512 case the performance remained intact.

6.3 Floorplanning

In Partial Reconfiguration, each Reconfigurable Partition is required to have a Pblock to define the physical resources available for the Reconfigurable Modules. A Pblock must contain only valid reconfigurable element types, as described in 4.1.1. Multiple Pblock rectangles may be used to create the Reconfigurable Partition region, but for the greatest routability, they should be contiguous. Gaps to account for non-reconfigurable resources are permitted, but in general, the simpler the overall shape, the easier the design will be to place and route. In our design, the cryptographic cores take up much of the device's resources, so it was difficult to find a contiguous place without non-reconfigurable regions. A floorplanning with back-to-back violations is presented in Figure 36.

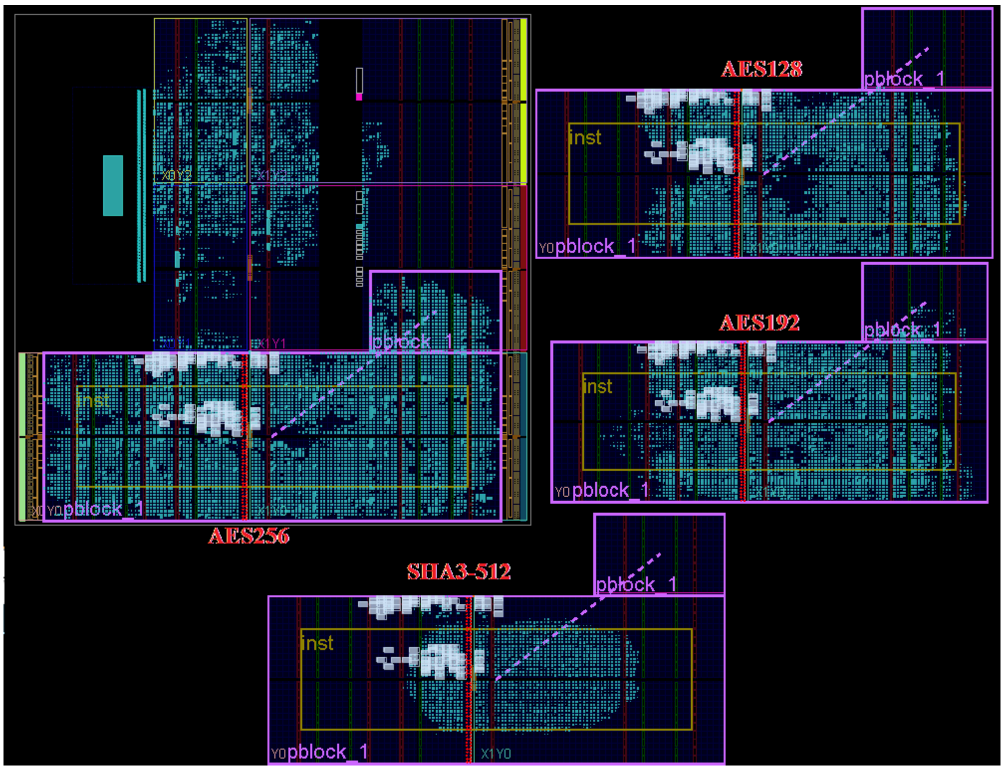


Figure 36 Floorplanning with back-to-back violation

In this floorplanning, the Pblock spans through an interconnect (red color in Figure 34) of two different clock regions and interconnect columns are not reconfigurable on 7-Series devices. In this case, a prohibit constraint has been automatically inserted by Vivado which forbids placement in this area. For this particular project, warnings can be avoided as there is no static logic inside the non-reconfigurable interconnect affecting the design. Because our IPs are of big utilization and it was difficult to find a contiguous area without non-reconfigurable elements for floorplanning, the Pblock can be split to avoid the interconnect. A valid floorplanning with no violations is shown in Figure 37.

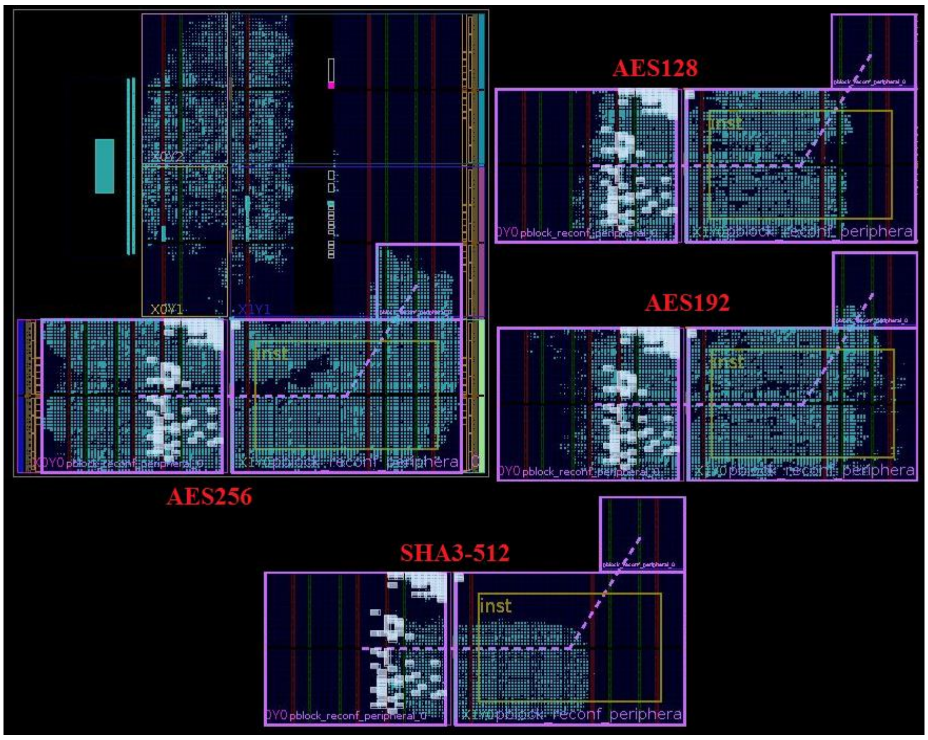


Figure 37 Floorplanning with no violations

As can be seen, after implementation, Vivado inserts partition pins within the Pblock ranges define for the Reconfigurable Partition. These virtual I/O are established within interconnect tiles as the anchor points that remain consistent from one module to the next. No physical resources such as LUTs or flip-flops are required to establish these anchor points, and no additional delay is incurred at these points.

6.4 Configuration Time

The configuration time scales linearly as the bitstream size grows with the number of reconfigurable frames, with small variances depending on the location and the contents of the frames. It also depends on which software application or Operating System (OS) is used and which interface is utilized to perform the reconfiguration. Table 6 shows configuration times for a design running the baremetal application described in 5.4 with one reconfigurable peripheral where all the cryptographic algorithms (AES128, AES192, AES256 and SHA3-512) are utilized. In PCAP's and ICAP's case the bitstreams where stored in the device's DDR as binary files, while JTAG loaded them from an external source through Vivado's Hardware Manager.

	Full Bitstream	Partial Bitstream
Bitstream Size	4,045,678 bytes	1,660,388 bytes
JTAG	~4 s	~2s
PCAP	31.08 ms	12.76 ms
ICAP	-	162.72 ms

Table 6 Comparison between configuration interfaces

Note that for uncompressed bitstreams, each RM in the design has an identical partial bitstream size, despite of the resources it utilizes inside the Reconfigurable Partition. The difference in reconfiguration times between the three interfaces was expected. JTAG's primary function is as a serial interface for prototyping and debugging which is used to load a bitstream from an external source, so it is not expected to achieve fast reconfiguration times. PCAP achieves the fastest time, as expected, due to the DMA engine of AXI-PCAP bridge inside the Device Configuration Interface (DevC), while the overhead introduced when the bitstream data are supplied from the memory to ICAP has an impact in ICAP's performance. Note that usually when ICAP is used, the bitstream is kept in BRAM inside the FPGA. In such cases ICAP shows a much better performance than in Table 6.

6.5 Evaluation of the Cryptographic IPs implemented

In this section, an overall comparison between the cryptographic algorithms running exclusively on Zynq's ARM processor and our accelerated designs is made. For AES software implementation, the source code was provided by ARM's mbed TLS [20] open security library, while SHA3-512 was provided by a contribution to the OpenPGP project [21]. Mbed TLS provides AES implementations for a various number of block cipher operation modes. In this test the original AES block encryption function was used without any operation mode. In Figure 38 there are presented relative execution times for the accelerators in comparison with the algorithms running exclusively on ARM.

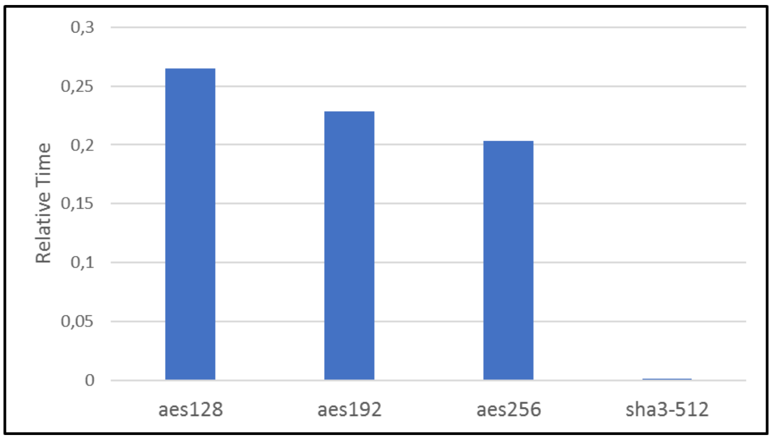


Figure 38 Relative execution times in comparison with the respective software implementations

These values were taken by performing 100000 transactions for each AES algorithm and for SHA3 by performing a single transaction of 10000 x 64 bits. In this diagram, smaller values mean greater performance in comparison with the algorithms' software implementations running on Zynq's ARM. For example, the relative execution time for AES128 is a little above 0.25 meaning that it has an overall acceleration below 4 (actually, for AES128 the speedup is x 3.78).

As it can be seen, AES algorithms do not have a great overall acceleration. This, is a result of the system's limitations (for multiple transactions an overhead of the communication time between the peripherals and the memory is introduced) and the non-optimized implemented interfaces which greatly affected the original cores' performance. The original AES cores where pipelined, meaning that the IPs could be fed with their input (plaintext and cipherkey) in every new cycle and provide a new result (ciphertext) in every new cycle. The interfaces implemented feed the core with a single encryption's input, wait for the result to be ready and as soon as the Master opens to write the output back, the Slave starts accepting data. This was done in order to quickly integrate the IPs in our system and continue with the rest and most important designing. On the other hand, SHA3-512 has much faster execution times than its software implementation because the interfaces implemented where much more suitable with the core's architecture and because SHA3-512 does not do multiple I/O operations, since for a very large message it only outputs a 512-bit wide hash.

6.6 *Experimental Evaluation*

In this section, we make a comparison between a system that has no Partial Reconfiguration capabilities with a system that can take advantage of the technique and apply it to adapt to an application's requirements and accelerate the total computation time needed. Sometimes, in systems with more than one peripherals, the application does not utilize all the peripherals, or some peripherals remain idle for an amount of time while others are utilized from the beginning until the end of an application. A system with PR capabilities can take advantage of this situation and reconfigure an idle peripheral with one that is needed, to accelerate the overall system's throughput. In this benchmark, we have designed two systems, one with PR capabilities and one with no PR capabilities. The systems have two peripherals each, for the cryptographic algorithms AES128 and AES192. The system with PR capabilities can configure the device with the different algorithms or with two instances of the same algorithm when needed, to distribute the computation load between them and thus, minimize the total computation time, while in the system with no PR capabilities the two different IPs are fixed and cannot be reconfigured. For example, suppose that we have a system that handles two data streams, one with AES128 and one with AES192 computations, in parallel. In the system with no PR capabilities, if the second stream is completed, AES192 peripheral remains idle for the rest of the application while AES128 continuous its execution normally. In the design with PR capabilities, the system can reconfigure AES192, in order to utilize two AES128 peripherals simultaneously and distribute the data between them. The system with the two peripherals implemented is presented in Figure 39.

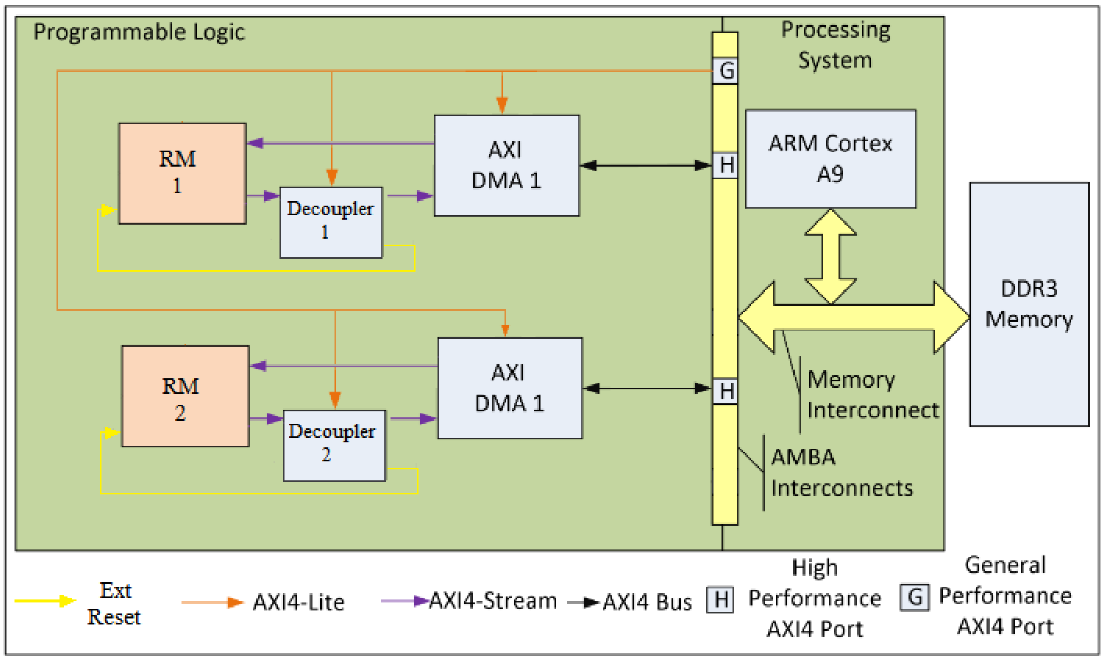


Figure 39 System with 2 crypto peripherals

For our experiments, we have implemented a simple software controller, described in Section 5.4. This controller reads the data from the 2 inputs streams, feeds the data to the accelerators and brings the results back by implementing the Scatter/Gather functionality. It also performs the reconfiguration of the peripherals in the system with PR capabilities. The input streams are randomly generated, but have been studied in advance, and the number of total reconfigurations needed is known prior to the tests' execution.

Furthermore, we have considered that the two data streams have 5 levels of Conflict. 100% Conflict means that in any given time the system needs to encrypt data with only one of the algorithms, while 0% Conflict means that there are always operations to be performed with both of the algorithms available. The results for 3 datasets of 400k, 800k and 2000k total computations where a total of 5 partial reconfigurations need to be performed are presented in Figures 40, 41, 42.

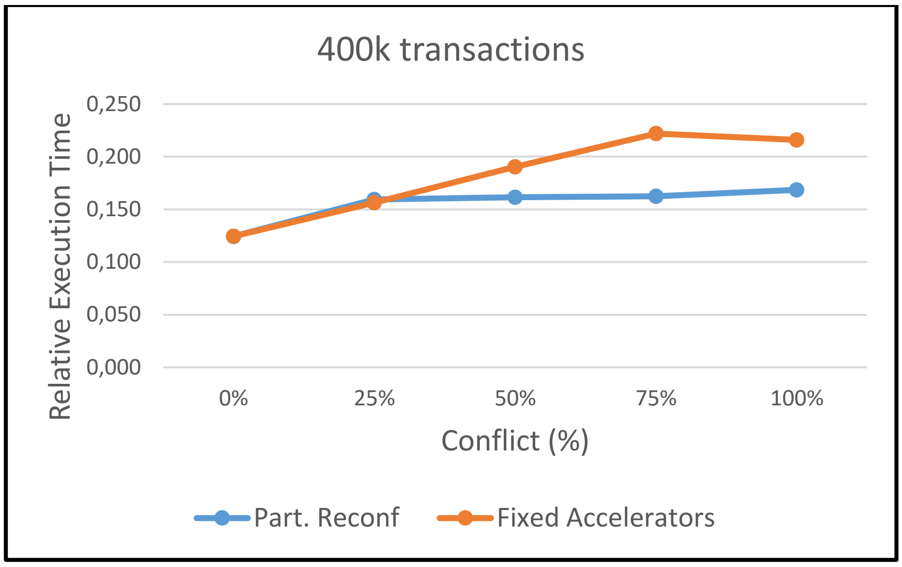


Figure 40 Experimental results of 400k total computations and 5 partial reconfigurations

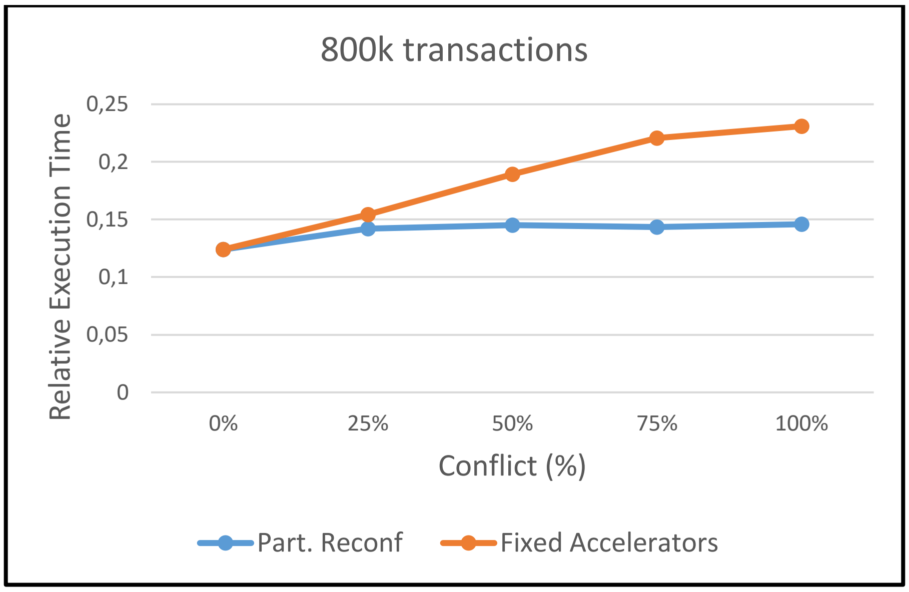


Figure 41 Experimental results of 800k total computations and 5 partial reconfigurations

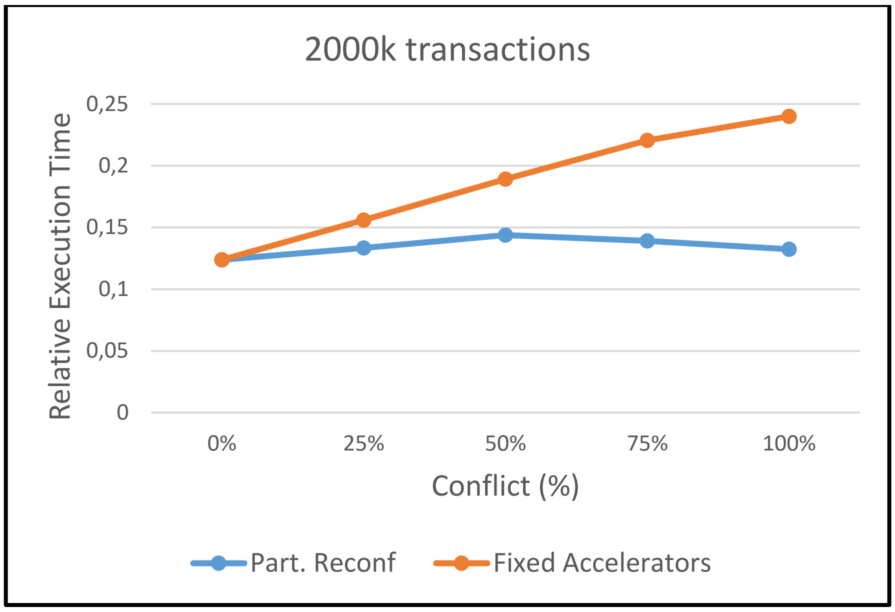


Figure 42 Experimental results of 2000k total computations and 5 partial reconfigurations

In each of the Figures above, the vertical axis represents each of the two systems' relative execution time in comparison with the tests running on Zynq's processor, while the horizontal axis represents the 5 levels of Conflict. For every testcase, the system with PR capabilities has an average acceleration of 86% compared to the software and 20% compared to the system with no PR capabilities. Overall, in the three Figures the PR system scales to the same acceleration threshold due to utilizing both of the peripherals at any given time. In Figure 40 for Conflict 25%, the PR system performs worse than the no-PR system because the total computation time is relatively small and is directly affected by the time needed for a partial reconfiguration to be performed. Also, the little fluctuation on the acceleration on each graph is expected, due to the random nature of the data input streams where there may be more or less total transactions for a specific algorithm.

The results of a fixed 75% Conflict are presented in Figure 43

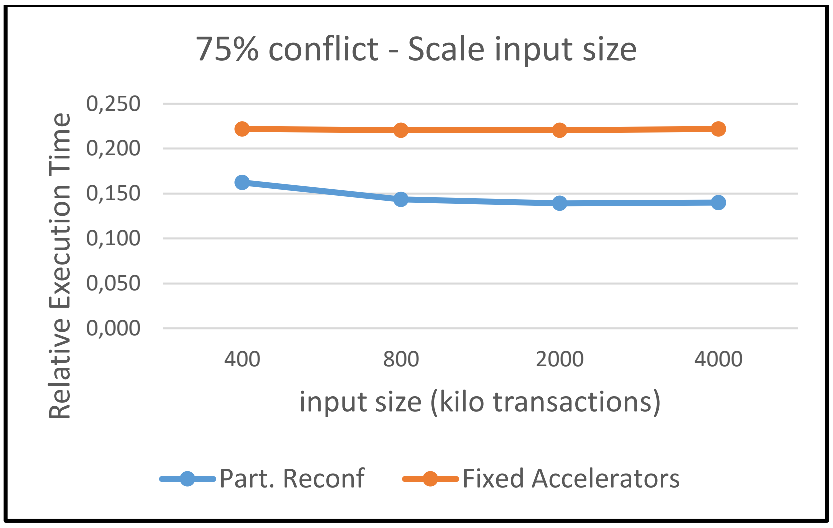


Figure 43 Experimental result of 75% Conflict and various input sizes

In Figure 43 we show how the total acceleration scales for different input sizes and a fixed 75% Conflict. Like before, the vertical axis represents the total acceleration of the two systems compared to the ARM processor. As we can see, the relative accelerations scale to a threshold for each system as the total input size gets bigger.

For our last experiment, we have considered a testcase where a total of 5, 10 or 20 partial reconfigurations are done. Just as before, the input data streams are randomly generated but the total number of the reconfigurations performed is known prior to each of the tests' execution. The results are shown in Figure 44.

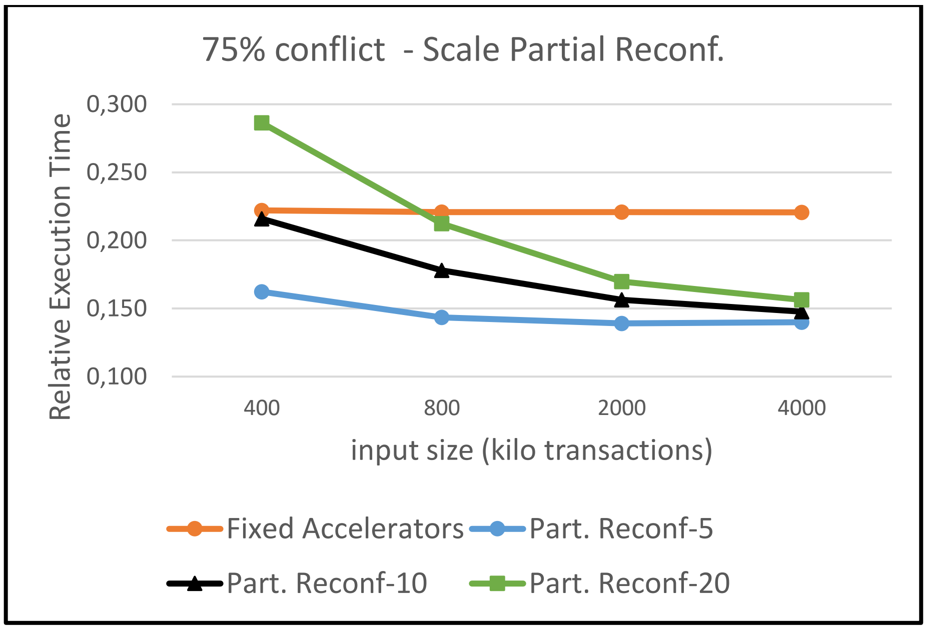


Figure 44 Experimental results for a different number of total reconfigurations

As it can be seen in Figure 44, the total acceleration scales once again to the same threshold. For 20 reconfigurations and 400k total transaction the total execution time relatively small and directly affected by the time needed to perform a partial reconfiguration. In this, case the PR system performs worse than the no-PR system. However, as the input size grows the total computation time is not affected by the reconfiguration time and the system scales to the same threshold as the other systems, where both of the peripherals are fully utilized at any given time.

This page is intentionally left blank

7 Conclusion

7.1 Summary

At this point, the work for this diploma thesis has reached the end. This work tried to explore the Partial Reconfiguration technology on FPGAs and apply the knowledge acquired to implement a cryptographic system on an Zynq SoC device. Firstly, we made all the modification needed to utilize the cryptographic peripherals in the SoC, and designed the appropriate interfaces to allow communication between them and the Zynq's processor. To implement Partial Reconfiguration, we constructed two IPs, a "dummy" peripheral and a Decoupler. The "dummy" IP, that contained no logic but only definitions of the reconfigurable IPs as seen outside, was necessary to synthesize the static design and was used as a black-box in which the cryptographic peripherals were loaded prior to implementation of the various configurations. The Decoupler was utilized to isolate outcoming signals from the Reconfigurable Partitions to the rest of the system during Partial Reconfiguration and reset the Reconfigurable Modules to a known initial state after reconfiguring was done. The cryptographic peripherals communicated with Zynq's Processing System through an AXI DMA in Scatter/Gather mode. The SGDMA resulted in a high-speed communication between the peripherals and the PS and reduced the total number of interrupts from the DMA to the ARM and thus, it allowed the processor to handle the peripherals more efficiently. Then, we made an evaluation of our work and presented details about the implemented system and the acceleration we achieved. Finally, we showed Partial Reconfiguration's advantages by utilizing it in a system with two peripherals. Whenever a peripheral completed its work for the rest of the application, while the other continued to operate, the idle peripheral was reconfigured with an instance of the working one and the computation load was distributed equally to the two IPs, thus reducing the total computation time.

7.2 *Final Thoughts and Future Work*

Heterogeneous platforms combining general purpose processing units and programmable logic have gained an increased interest as the next generation of FPGA-based hybrid devices. In these systems, general purpose processing units provide the flexibility and a way to control and distribute the computational load, while FPGAs have the role of implementing custom, application specific accelerators. During the development of this diploma thesis, we had in mind the greater picture of the technologies used. While Zynq-7000 is a low-end device that introduces a lot of limitations to the final project, the acquired knowledge can be applied to several future platforms and similar architectures. Moreover, Partial Reconfiguration can be useful to a variety of devices, from embedded low-end platforms with limited resources and capabilities to high performance systems, that need to accelerate critical parts of an application and must have a quick response whenever the application's requirements differentiate.

A nice idea for future work would be to implement a more sophisticated software controller for the system presented in 6.6, for it to be able to operate in a real-world scenario. The controller could decide if it would be advantageous to perform a reconfiguration by observing the data streams and knowing the amount of time needed to perform a partial reconfiguration. We also could study how the Partial Reconfiguration procedure affects the device's power consumption.

This thesis's results could be applied to any platform with Partial Reconfiguration or Cryptography requirements. One application that combines both, is Software-defined Radios (SDRs). SDR is a communication system where components that are traditionally implemented in hardware (e.g mixers, filters, amplifiers, modulators/demodulators, etc.) are instead implemented by means of software. This gives the ability to a system to change transmission protocols and a single device can be used in a wide variety of communication schemes. In recent years, FPGAs have been integrated in SDRs to accelerate these tasks. Our thesis results could be used in such devices to take advantage of Partial Reconfiguration to change transmission protocols and be able to respond in different communication schemes quickly. It could also be used to apply encryption/decryption schemes whenever desirable.

References

[1] “Cryptography”, https://en.wikipedia.org/wiki/Cryptography

[2] “Advanced Encryption Standard”, https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[3] National Institute of Standards and Technology, “Announcing the ADVANCED ENCRYPTION STANDARD (AES) (FIPS PUB 197)”, 2001

[4] “SHA-3”, https://en.wikipedia.org/wiki/SHA-3

[5] National Institute of Standards and Technology, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (FIPS PUB 202)”, 2015

[6] “Field-Programmable Gate Array”, https://en.wikipedia.org/wiki/Field-programmable_gate_array

[7] “Reconfigurable computing”, https://en.wikipedia.org/wiki/Reconfigurable_computing

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

[9] 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

[10] “Verilog”, https://en.wikipedia.org/wiki/Verilog

[11] Xilinx, “ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC (UG850)”, September 2015

[12] Xilinx, “Partial Reconfiguration (UG909)”, June 2016

[13] Xilinx, “Zynq-7000 All Programmable SoC, Technical Reference Manual (UG585)”, September 2016

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

[15] ARM, “AMBA® 4 AXI4-Stream Protocol, Specification”, 2010

[16] Xilinx, “AXI DMA v7.1 (PG021)”, October 2016

[17] Hsing Homer, “AES :: Overview”, http://opencores.org/project,tiny_aes

[18] Hsing Homer, “SHA3 (KECCAK) :: Overview”, http://opencores.org/project,sha3

[19] “Zynq Bus Functional Model (BFM)”, https://www.xilinx.com/products/intellectual-property/zynq_bfm.html

[20] “mbed TLS”, https://tls.mbed.org/

[21] “A single-file C implementation of SHA-3 with Init/Update/Finalize API”, https://github.com/brainhub/SHA3IUF