ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

ΤΜΗΜΑ ΦΥΣΙΚΗΣ & ΠΛΗΡΟΦΟΡΙΚΗΣ, ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
Δ.Π.Μ.Σ. «ΜΕΤΑΠΤΥΧΙΑΚΑ ΔΙΠΛΩΜΑΤΑ ΕΙΔΙΚΕΥΣΗΣ ΣΤΗΝ ΗΛΕΚΤΡΟΝΙΚΗ ΡΑΔΙΟΗΛΕΚΤΡΟΛΟΓΙΑ (Ρ/Η) ΚΑΙ ΣΤΟΝ ΗΛΕΚΤΡΟΝΙΚΟ ΑΥΤΟΜΑΤΙΣΜΟ (Η/Α)»

Sobel Edge Detector SoC Design on Zynq 7000 AP SoC

Πατσή Μαρία Ελένη

ΑΜ: 7110132400205

Εργασία στο πλαίσιο του μαθήματος:
«Προηγμένη Σχεδίαση Ψηφιακών Συστημάτων»

Υπεύθυνοι Καθηγητές: Βασιλόπουλος Διονύσης, Κρανίτης Νεκτάριος

Αθήνα, 2025

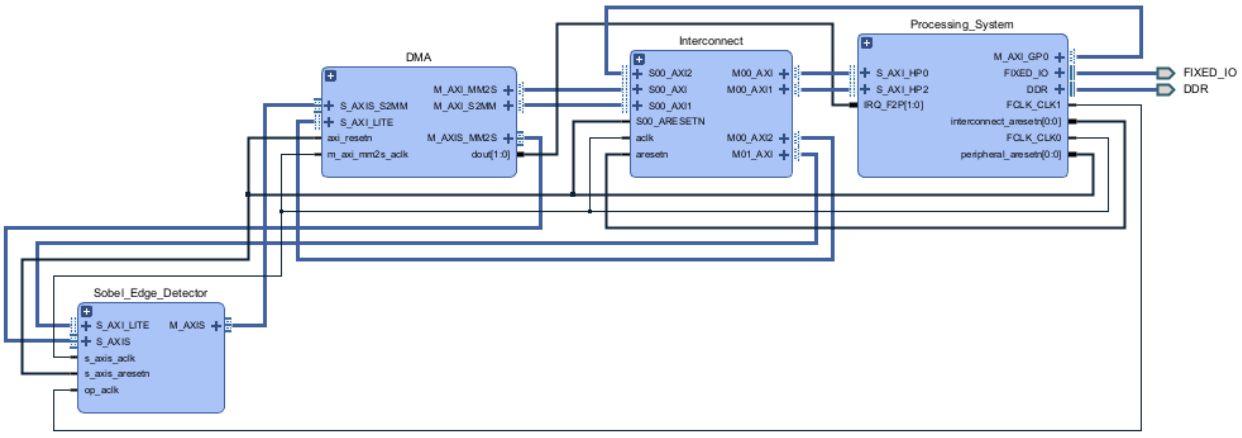
Περιεχόμενα

Επισκόπηση

Η παρούσα τεχνική αναφορά παρουσιάζει τη σχεδίαση, υλοποίηση και αξιολόγηση ενός ολοκληρωμένου συστήματος σε τσιπ (System-on-Chip – SoC) για την επιτάχυνση του αλγορίθμου ανίχνευσης ακμών Sobel, βασισμένου στην πλατφόρμα Xilinx Zynq-7000 AP SoC. Στόχος της εργασίας είναι η μελέτη και αξιοποίηση των πλεονεκτημάτων της συνδυασμένης αρχιτεκτονικής Processing System (PS) και Programmable Logic (PL) του Zynq, ώστε να επιτευχθεί σημαντική βελτίωση των επιδόσεων σε σχέση με την αμιγώς software υλοποίηση του αλγορίθμου.

Αρχικά, ο αλγόριθμος ανίχνευσης ακμών Sobel υλοποιήθηκε σε γλώσσα C και εκτελέστηκε στους πυρήνες ARM Cortex-A9 του επεξεργαστικού συστήματος του Zynq, σε περιβάλλον PetaLinux. Η εκτέλεση του προγράμματος επέτρεψε τη συλλογή στατιστικών επίδοσης (όπως χρόνος εκτέλεσης και throughput), τα οποία χρησιμοποιήθηκαν ως σημείο αναφοράς για την αξιολόγηση της επιτάχυνσης που επιτυγχάνεται με την υλοποίηση σε υλικό. Στη συνέχεια, σχεδιάστηκε σε VHDL ένα αυτόνομο Sobel Edge Detector IP Core, το οποίο επιταχύνει τον υπολογισμό του αλγορίθμου στο PL του Zynq. Το IP Core διαθέτει AXI4-Lite διεπαφή για έλεγχο και πρόσβαση σε εσωτερικούς μετρητές απόδοσης, καθώς και AXI4-Stream διεπαφές για είσοδο και έξοδο δεδομένων, επιτρέποντας πλήρη συμβατότητα με το σύστημα DMA και το υπόλοιπο SoC.

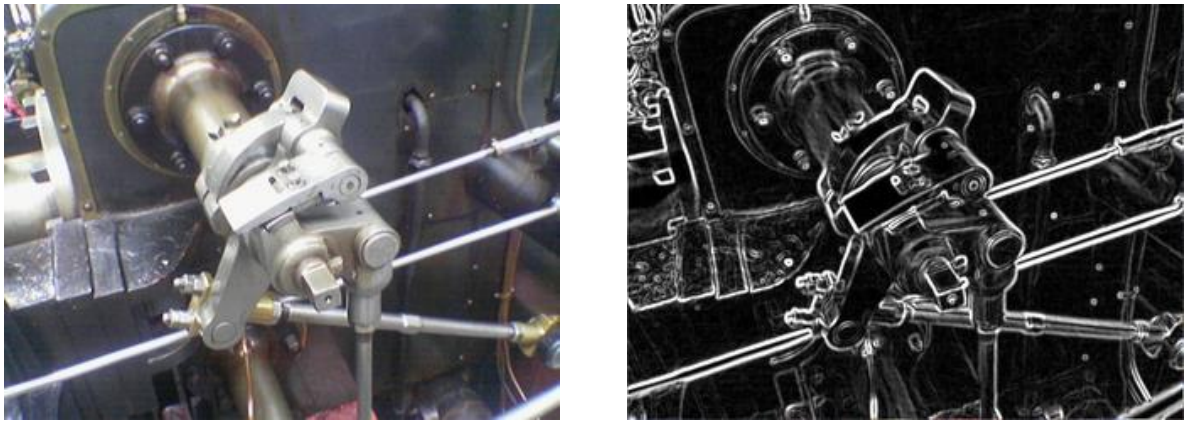
Ο σχεδιασμός υλοποιήθηκε με στόχο ρυθμό επεξεργασίας έως 200MSamples/sec σε ρολόι 200MHz, αξιοποιώντας τεχνικές διοχέτευσης (pipelining) και παράλληλης επεξεργασίας για μέγιστη απόδοση. Μετά την προσομοίωση και επαλήθευση του IP Core με testbench σε VHDL, η μονάδα ενσωματώθηκε σε πλήρες Sobel Edge Detector SoC. Η αρχιτεκτονική του συστήματος περιλαμβάνει τη διασύνδεση του IP Core με τον AXI DMA, μέσω του οποίου μεταφέρονται τα δεδομένα εικόνας από και προς τη DDR3 DRAM του Zynq. Η επικοινωνία και ο έλεγχος της μονάδας πραγματοποιούνται μέσω προγράμματος χρήστη σε C, που εκτελείται στο PetaLinux και παρέχει μετρήσεις απόδοσης και σύγκριση αποτελεσμάτων με τη software εκδοχή του αλγορίθμου. Τέλος, συγκρίθηκαν οι δύο υλοποιήσεις, η καθαρά software και η επιταχυνόμενη στο υλικό, ως προς τον χρόνο εκτέλεσης, το throughput και την κατανάλωση πόρων υλικού. Τα αποτελέσματα απέδειξαν σημαντική επιτάχυνση της εκτέλεσης μέσω της υλοποίησης του Sobel αλγορίθμου στο FPGA fabric, επιβεβαιώνοντας τα πλεονεκτήματα της αρχιτεκτονικής hardware/software co-design για εφαρμογές επεξεργασίας εικόνας.



Εικόνα 1 - Διάγραμμα του Sobel Edge Detector SoC Design.

Αλγόριθμος Ανίχνευσης Ακμών Sobel

Η ανίχνευση ακμών αποτελεί μια από τις πιο θεμελιώδεις και σημαντικές λειτουργίες στην επεξεργασία εικόνας, με εφαρμογές που εκτείνονται από το computer vision και την τεχνητή νοημοσύνη (AI) έως συστήματα ασφαλείας και παρακολούθησης. Ο στόχος της διαδικασίας είναι ο εντοπισμός περιοχών μιας εικόνας στις οποίες παρατηρείται απότομη μεταβολή της έντασης φωτεινότητας. Οι περιοχές αυτές αντιστοιχούν συνήθως στα όρια των αντικειμένων και επομένως η ανίχνευση ακμών επιτρέπει τον εντοπισμό και τη διάκριση των δομών μιας σκηνής. Για τις ανάγκες της παρούσας εργασίας, χρησιμοποιούνται μονοχρωματικές εικόνες ανάλυσης 512×512 pixels με 8-bit ανά pixel, όπου οι τιμές κυμαίνονται από 0 (μαύρο) έως 255 (λευκό). Η ανίχνευση ακμών πραγματοποιείται μέσω του αλγορίθμου Sobel ή αλλιώς Sobel – Feldman, ενός από τους πιο γνωστούς και απλούς αλγόριθμους για τον υπολογισμό παραγώγων της έντασης της εικόνας.



Εικόνα 2 - Παράδειγμα εφαρμογής του αλγορίθμου ανίχνευσης ακμών Sobel.

Θεωρητικό υπόβαθρο

Ο αλγόριθμος Sobel βασίζεται στον υπολογισμό των παραγώγων της έντασης της εικόνας κατά οριζόντια και κάθετη κατεύθυνση. Για κάθε pixel, υπολογίζονται δύο τιμές, η οριζόντια συνιστώσα της παραγώγου G_x και η κατακόρυφη συνιστώσα της παραγώγου G_y. Αυτές οι παράγωγοι υπολογίζονται μέσω μιας διαδικασίας συνέλιξης (convolution), όπου για κάθε pixel λαμβάνεται υπόψη μια γειτονική περιοχή 3×3 pixels και εφαρμόζονται μάσκες συνέλιξης (convolution masks/kernels) με συγκεκριμένους συντελεστές βαρών. Οι τυπικές μάσκες του αλγορίθμου Sobel έχουν τη μορφή:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Για κάθε pixel της αρχικής εικόνας $O(row, col)$, οι μάσκες τοποθετούνται κεντραρισμένες πάνω του και υπολογίζονται οι τιμές:

$$D_x = \sum_{i=-1}^{1} \sum_{j=-1}^{1} O(row+1, col+j) \cdot G_x(i,j)$$

$$D_y = \sum_{i=-1}^{1} \sum_{j=-1}^{1} O(row+1, col+j) \cdot G_y(i,j)$$

Οι δύο μερικές παράγωγοι που προκύπτουν συνδυάζονται για να προκύψει το μέτρο του διανύσματος κλίσης (gradient magnitude), το οποίο καθορίζει την ένταση της ακμής μέσω της Ευκλείδειας μετρικής:

$$|D| = \sqrt{D_x^2 + D_y^2}$$

Ωστόσο, για απλούστερη και αποδοτικότερη υλοποίηση χρησιμοποιείται συχνά η κατά προσέγγιση μορφή της Manhattan μετρικής, όπου:

$$|D| \cong |D_x| + |D_y|$$

Η προσέγγιση αυτή διατηρεί τις ίδιες περιοχές μέγιστων και ελαχίστων τιμών (δηλαδή τις ίδιες ακμές), ενώ αποφεύγει τους υπολογιστικά δαπανηρούς τετραγωνισμούς και ρίζες. Αυτό που όμως δεν αναφέρθηκε είναι πως τα pixels που βρίσκονται στις άκρες της εικόνας δεν διαθέτουν πλήρες σύνολο γειτονικών τιμών (3×3 περιοχή). Για τον χειρισμό αυτών των περιπτώσεων εφαρμόζεται είτε απλή μηδενική επέκταση (zero padding), όπου τα ελλείποντα pixels λαμβάνουν τιμή 0, ή γίνεται αντιγραφή των πλησιέστερων pixels (replication padding), όπου τα όρια "αντιγράφουν" τις τιμές των κοντινών pixels. Με αυτό τον τρόπο, εξασφαλίζεται ότι όλες οι περιοχές της εικόνας μπορούν να υποβληθούν σε συνέλιξη. Στην αντίθετη περίπτωση, μπορούν να μην συνεισφέρουν στον αλγόριθμο χωρίς να προκληθεί κάποιο πρόβλημα. Το μόνο που αλλάζει είναι οι τελικές διαστάσεις της εικόνας, και θα έχει 2 λιγότερες γραμμές και 2 λιγότερες στήλες.

Ψευδοκώδικας του Αλγορίθμου Sobel

Η διαδικασία υπολογισμού του Sobel αλγορίθμου περιγράφεται συνοπτικά στον ακόλουθο ψευδοκώδικα, όπου τα $O(row, col)$ αντιπροσωπεύουν τα pixels της αρχικής εικόνας, ενώ τα $D(row, col)$ τα pixels της εξόδου (εικόνας παραγώγου). Οι τιμές των $Gx(i, j)$ και $Gy(i, j)$ αντιστοιχούν στα βάρη των μασκών συνέλιξης.

```
for row from 1 to image_height loop
    for col from 1 to image_width loop
        Dx := 0
        Dy := 0
        for i from -1 to +1 loop
            for j from -1 to +1 loop
                Dx := Dx + O(row + i, col + j) * Gx(i, j)
                Dy := Dy + O(row + i, col + j) * Gy(i, j)
            end loop
        end loop
        D(row, col) := abs(Dx) + abs(Dy)
    end loop
end loop
```

Αλγόριθμος 1 – Ψευδοκώδικας αλγορίθμου ανίχνευσης ακμών Sobel.

Υλοποίηση του Sobel Edge Detector σε Software

Κατά το αρχικό στάδιο της ανάπτυξης υλοποιήθηκε εφαρμογή ανίχνευσης ακμών με χρήση του αλγορίθμου Sobel, προγραμματισμένη στη γλώσσα C και εκτελούμενη σε περιβάλλον Linux. Η δομή του λογισμικού βασίζεται σε ένα εκτελέσιμο πρόγραμμα το οποίο δέχεται, μέσω της γραμμής εντολών, δύο ορίσματα: το αρχείο εισόδου και το αρχείο εξόδου τύπου **.raw**. Το αρχείο εισόδου περιέχει τα δεδομένα φωτεινότητας των pixel μιας εικόνας σε grayscale μορφή, ενώ στο αρχείο εξόδου αποθηκεύονται τα αποτελέσματα της επεξεργασίας. Η κύρια ροή του προγράμματος υλοποιείται στο αρχείο **main.c**, το οποίο περιλαμβάνει κλήσεις συναρτήσεων από τις βιβλιοθήκες **sobel.h**, **timer.h**, **util.h** και **sobel_constants.h**. Αρχικά, πραγματοποιείται έλεγχος εγκυρότητας των ορισμάτων που δίνονται κατά την εκτέλεση, και σε περίπτωση ελλιπών παραμέτρων εμφανίζεται κατάλληλο

μήνυμα. Στη συνέχεια, δεσμεύεται δυναμική μνήμη για τρεις δισδιάστατους πίνακες τύπου **uint8_t** (unsigned integers of 8-bit width), έναν για την εικόνα εισόδου, έναν για την έξοδο του φίλτρου Sobel με τη μετρική Manhattan, και έναν για τη μετρική Euclidean.

Ακολουθεί η φόρτωση της εικόνας εισόδου μέσω της συνάρτησης **load_raw_image()**, ενώ η διάρκεια της διαδικασίας μετριέται με τη χρήση βοηθητικών συναρτήσεων χρονισμού από τη βιβλιοθήκη **timer.h**. Έπειτα, εφαρμόζεται ο αλγόριθμος Sobel με δύο διαφορετικές μετρικές. Πρώτα εκτελείται η συνάρτηση **sobel_manhattan()**, η οποία υπολογίζει την ένταση των ακμών με βάση το άθροισμα των απόλυτων τιμών των παραγώγων κατά τις διευθύνσεις x και y. Στη συνέχεια, εκτελείται η συνάρτηση **sobel_euclidean()**, όπου η ένταση των ακμών υπολογίζεται με τη ρίζα του αθροίσματος των τετραγώνων των ίδιων παραγώγων. Ο χρόνος εκτέλεσης κάθε αλγορίθμου μετριέται και αποθηκεύεται, προκειμένου να πραγματοποιηθεί σύγκριση επιδόσεων.

Μετά την ολοκλήρωση της επεξεργασίας, τα αποτελέσματα αποθηκεύονται σε αρχεία τύπου **.raw**. Η έξοδος της μετρικής Manhattan αποθηκεύεται στο αρχείο που δίνεται από τον χρήστη ως δεύτερο όρισμα, μέσω της συνάρτησης **save_raw_image()**, ενώ η έξοδος της μετρικής Euclidean αποθηκεύεται προαιρετικά σε ξεχωριστό αρχείο. Τέλος, εκτυπώνεται συνοπτικός πίνακας επιδόσεων που περιλαμβάνει τους χρόνους φόρτωσης, επεξεργασίας και αποθήκευσης, καθώς και τον υπολογισμό του speedup factor μεταξύ των δύο μετρικών. Μετά την ολοκλήρωση της εκτέλεσης, η δεσμευμένη μνήμη αποδεσμεύεται και το πρόγραμμα τερματίζει επιτυχώς, εμφανίζοντας σχετικό μήνυμα ολοκλήρωσης.

Sobel Edge Detection Αλγόριθμος σε C

Η υλοποίηση του Sobel filter σε γλώσσα C, όπως και αναφέρθηκε προηγουμένως, έγινε με δύο τρόπους, **sobel_manhattan()** και **sobel_euclidean()**, οι οποίες εφαρμόζουν τον ίδιο αλγόριθμο συνέλιξης αλλά διαφέρουν στον τρόπο υπολογισμού του μεγέθους της κλίσης καθώς χρησιμοποιούν διαφορετική μετρική απόστασης, την Manhattan και την Ευκλείδεια αντίστοιχα. Για κάθε pixel της εικόνας, υπολογίζονται οι τιμές G_x και G_y με βάση τα γειτονικά pixel, και στη συνέχεια προκύπτει η τελική τιμή έντασης της ακμής είτε με την αντίστοιχη μετρική. Για να αντιμετωπιστεί το πρόβλημα των οριακών pixel, χρησιμοποιείται βοηθητική συνάρτηση **get_padded_pixel()**, η οποία εφαρμόζει “mirror padding”, δηλαδή αντιγράφει τις τιμές των κοντινών pixel στα άκρα της εικόνας, ώστε να μη χαθεί πληροφορία στις άκρες. Η τελική τιμή κάθε pixel κανονικοποιείται στο εύρος 0–255, ώστε η έξοδος να είναι έγκυρη grayscale εικόνα 8-bit. Στην Εικόνα 3 μπορούμε να δούμε την υλοποίηση του αλγορίθμου για την Manhattan μετρική.

Επιδόσεις και Αποτελέσματα Προγράμματος σε Windows

Ενδεικτικά, για την επεξεργασία της εικόνας **lena_512_512.raw** (που μπορούμε να δούμε και στο αριστερό μέρος στην Εικόνα 4) η μετρική Manhattan παρουσίασε ταχύτερη εκτέλεση σε σχέση με τη Euclidean, με speedup factor ίσο με ***1.22x,*** γεγονός που υποδεικνύει ότι η υπολογιστική πολυπλοκότητα της Manhattan μετρικής είναι μικρότερη, λόγω της απουσίας τετραγωνισμών και υπολογισμού τετραγωνικής ρίζας. Παράλληλα, η οπτική ποιότητα των αποτελεσμάτων παραμένει συγκρίσιμη μεταξύ των δύο μεθόδων, καθιστώντας τη Manhattan εκδοχή μια αποδοτικότερη εναλλακτική για περιβάλλοντα όπου προτεραιότητα έχει η ταχύτητα επεξεργασίας. Για την εκτέλεση στους πυρήνες ARM Cortex-A9 του επεξεργαστικού συστήματος του Zynq, σε περιβάλλον PetaLinux, θα αναφερθούμε στην σελίδα 19.

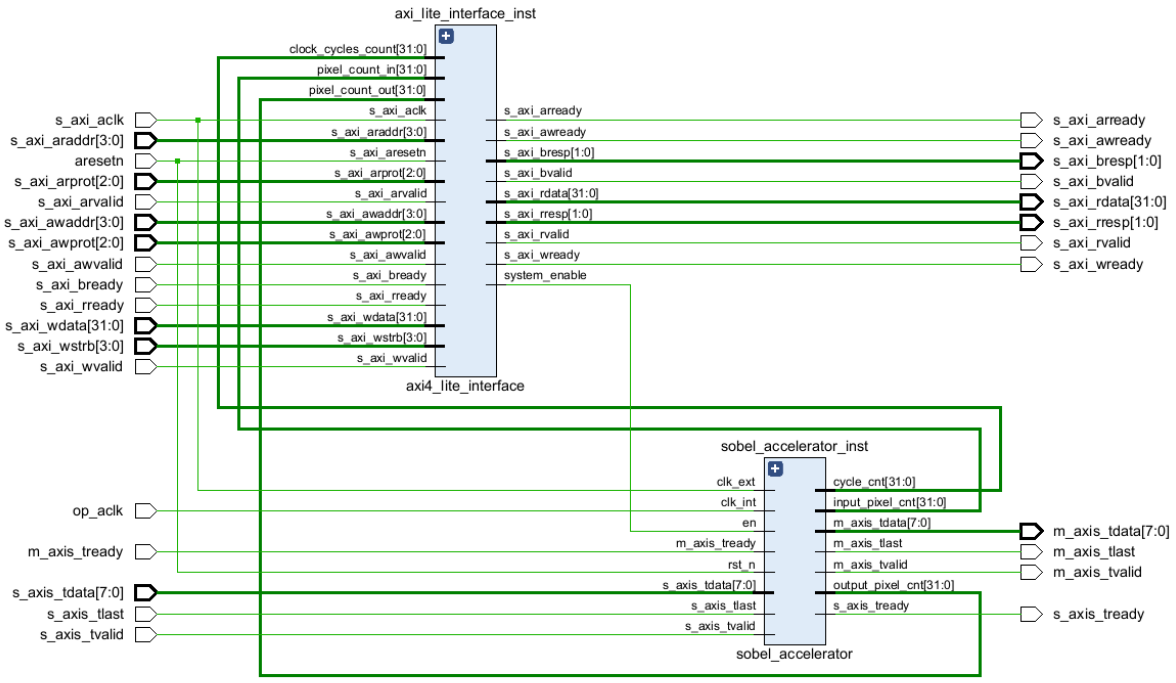
```c
// --- Sobel Kernels ---
static const int Gx[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
static const int Gy[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};

// --- Sobel Processing ---
static int get_padded_pixel(uint8_t input[ROW][COLUMN], int row, int col) {
    // Handle border by mirroring
    int r = row < 0 ? 0 : (row >= ROW ? ROW - 1 : row);
    int c = col < 0 ? 0 : (col >= COLUMN ? COLUMN - 1 : col);
    return input[r][c];
}

void sobel_manhattan(uint8_t input[ROW][COLUMN], uint8_t output[ROW][COLUMN]) {
    for (int r = 0; r < ROW; r++) {
        for (int c = 0; c < COLUMN; c++) {
            int sx = 0, sy = 0;

            // Apply Sobel kernel
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    int pixel = get_padded_pixel(input, r + i, c + j);
                    sx += pixel * Gx[i + 1][j + 1];
                    sy += pixel * Gy[i + 1][j + 1];
                }
            }

            // Manhattan magnitude and clamp to 0-255
            int magnitude = abs(sx) + abs(sy);
            output[r][c] = magnitude > 255 ? 255 : magnitude;
        }
    }
}
```

Εικόνα 3 - Ο Sobel Edge Detection αλγόριθμος σε C, με αντιγραφή των pixel στα edges και με χρήση της Manhattan μετρικής.

Εικόνα 4 – Στα αριστερά βλέπουμε την αρχική εικόνα Lena. Στο κέντρο μπορούμε να δούμε το αποτέλεσμα του Software Sobel με την Ευκλείδεια μετρική, ενώ στα δεξιά μπορούμε να δούμε το ίδιο αποτέλεσμα για την μετρική Manhattan.

Μετά την επιτυχή ανάπτυξη και αξιολόγηση του αλγορίθμου Sobel σε γλώσσα C, η επόμενη φάση αφορούσε τη μεταφορά και επιτάχυνσή του σε υλικό (hardware), με στόχο τη βελτίωση των επιδόσεων και την αξιοποίηση της παραλληλίας που προσφέρουν τα FPGA. Η υλοποίηση πραγματοποιήθηκε σε Zynq-7000 AP SoC, μια πλατφόρμα που συνδυάζει διπύρηνο ARM Cortex-A9 επεξεργαστή με προγραμματιζόμενη λογική (Programmable Logic – PL), επιτρέποντας την ομαλή συνεργασία λογισμικού και υλικού. Ο επιταχυντής Sobel υλοποιήθηκε ως IP core με χρήση VHDL, ενώ η σύνθεση και υλοποίηση του υλικού (synthesis και implementation) γίνεται με τον synthesizer VIVADO (v2022.2), που είναι συμβατός με όλα τα προϊόντα Xilinx όπως είναι και το Zedboard. Η επικοινωνία μεταξύ επεξεργαστή και επιταχυντή βασίζεται στα πρωτόκολλα AXI4-Lite για έλεγχο και AXI4-Stream για μεταφορά δεδομένων.

Sobel IP



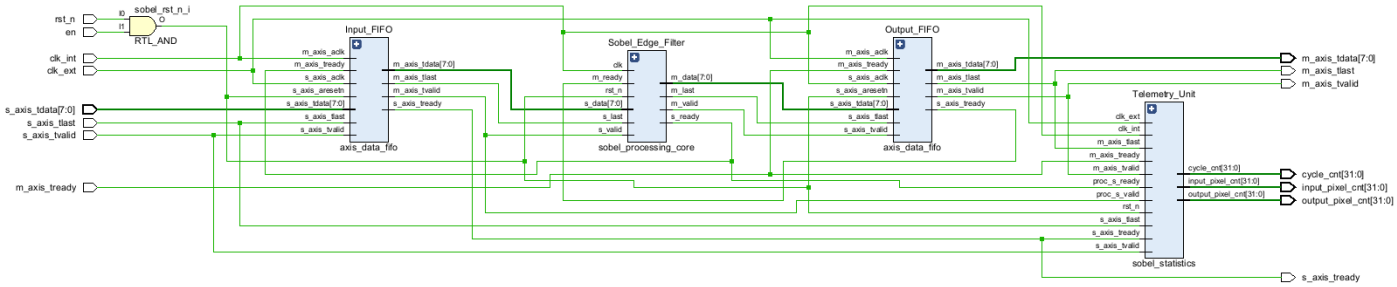
Εικόνα 5 - Η διεπαφή του sobel accelerator και του υπόλοιπου συστήματος.

Η μονάδα **axi_lite_interface.vhd** λειτουργεί ως ενδιάμεσος μεταξύ του ARM επεξεργαστή και του επιταχυντή, επιτρέποντας την επικοινωνία και τον έλεγχο της λειτουργίας του. Μέσω αυτής της διεπαφής, ο επεξεργαστής μπορεί να ξεκινήσει ή να σταματήσει την επεξεργασία μέσω του register 0x00, να παρακολουθεί την τρέχουσα κατάσταση λειτουργίας του συστήματος μέσω του register 0x04, καθώς και να λαμβάνει δεδομένα τηλεμετρίας από τη μονάδα **sobel_statistics.vhd**, όπως τον αριθμό των pixel που έχουν επεξεργαστεί, τον συνολικό χρόνο εκτέλεσης και τον αριθμό των κύκλων ρολογιού που απαιτήθηκαν.

Η ενσωμάτωση του IP core πραγματοποιείται στο Vivado Block Design, όπου ο επιταχυντής συνδέεται με τον AXI Interconnect και τον AXI DMA. Με αυτόν τον τρόπο, η ροή λειτουργίας ξεκινά όταν ο επεξεργαστής

αποστείλει την εντολή εκκίνησης μέσω του AXI4-Lite, έπειτα ο DMA μεταφέρει τα δεδομένα της εικόνας προς τον επιταχυντή μέσω του AXI4-Stream, ο οποίος εκτελεί τον αλγόριθμο Sobel σε πραγματικό χρόνο. Τέλος, τα αποτελέσματα επιστρέφουν μέσω του ίδιου διαύλου στη μνήμη DDR, έτοιμα για απεικόνιση ή περαιτέρω ανάλυση. Η αρχιτεκτονική αυτή εξασφαλίζει υψηλή αξιοπιστία, ενώ παραμένει πλήρως συμβατή με τις υποδομές του Zynq-7000 AP SoC, επιτρέποντας άμεση συνεργασία μεταξύ του λογισμικού και της προγραμματιζόμενης λογικής.

Τα pixel της εικόνας εισέρχονται στον επιταχυντή ως ροή, ένα pixel ανά κύκλο, και η σχεδίασή της αρχιτεκτονικής πρέπει να είναι τέτοια ώστε και η ροή εξόδου να είναι ίδια, να παράγεται δηλαδή ένα pixel εξόδου ανά κύκλο. Η είσοδος της εικόνας μεταφέρεται μέσω AXI DMA προς τον επιταχυντή, όπου γίνεται η επεξεργασία. Στη συνέχεια, τα αποτελέσματα αποθηκεύονται ξανά στη μνήμη μέσω του ίδιου μηχανισμού. Για την εξασφάλιση ομαλής επικοινωνίας μεταξύ διαφορετικών χρονισμών, χρησιμοποιούνται dual-clock FIFOs, ενώ η επεξεργασία είναι πλήρως pipelined όπου κάθε στάδιο επεξεργάζεται διαφορετικό pixel σε κάθε κύκλο ρολογιού. Μάλιστα ο Sobel Edge Detector IP θα λειτουργεί με ρολόι διπλάσιας συχνότητας από αυτή που λειτουργεί το υπόλοιπο σύστημα. Συγκεκριμένα το σύστημα θα λειτουργεί με ρολόι συχνότητας 100 MHz ενώ ο Sobel Edge Detector IP με συχνότητα 200 MHz. Μέσω της AXI4-Lite διεπαφής, ο ARM επεξεργαστής μπορεί να ενεργοποιεί ή να απενεργοποιεί το IP, να παρακολουθεί την πρόοδο της επεξεργασίας και να διαβάζει στατιστικά από τους καταχωρητές ελέγχου.



Εικόνα 6 – Η δομή του sobel_accelerator.vhd. Αποτελείται από την Input FIFO, το Sobel Edge Filter, την Output FIFO και το Telemetry Unit.

Όπως μπορούμε να δούμε και στην Εικόνα 6, ο επιταχυντής Sobel Edge Accelerator αποτελείται από τέσσερα κύρια components. Τις δύο FIFO, το Sobel Edge Filter και το Telemetry Unit. Το τελευταίο δεν παίζει κάποιο ρόλο στην επεξεργασία δεδομένων αλλά υπάρχει μόνο για τον υπολογισμό στατιστικών της επεξεργασίας. Όλοι οι υπολογισμοί πραγματοποιούνται στην μονάδα Sobel Edge Filter, όπου γίνεται η ανάλυση των κλίσεων και η εξαγωγή των ακμών, ενώ οι επιμέρους υπομονάδες της θα αναλυθούν παρακάτω αφού αναλύσουμε την λειτουργία των πρωτοκόλλων που αξιοποιεί το σύστημα.

Πρωτόκολλα Επικοινωνίας AXI4-Lite και AXI4-Stream

Όπως αναφέρθηκε και παραπάνω, η επικοινωνία του επιταχυντή με το υπόλοιπο σύστημα βασίζεται σε δύο βασικά πρωτόκολλα της οικογένειας AXI4 (Advanced eXtensible Interface 4), τα οποία προσφέρουν υψηλή ευελιξία, ταχύτητα και συμβατότητα με το υπόλοιπο οικοσύστημα του Zynq SoC. Τα δύο αυτά πρωτόκολλα χρησιμοποιούνται παράλληλα αλλά για διαφορετικούς σκοπούς: το AXI4-Lite για επικοινωνία ελέγχου και ρυθμίσεων, και το AXI4-Stream για τη μεταφορά των δεδομένων υψηλού ρυθμού (π.χ. pixel stream από ή προς τον επιταχυντή).

Πρωτόκολλο AXI4-Lite

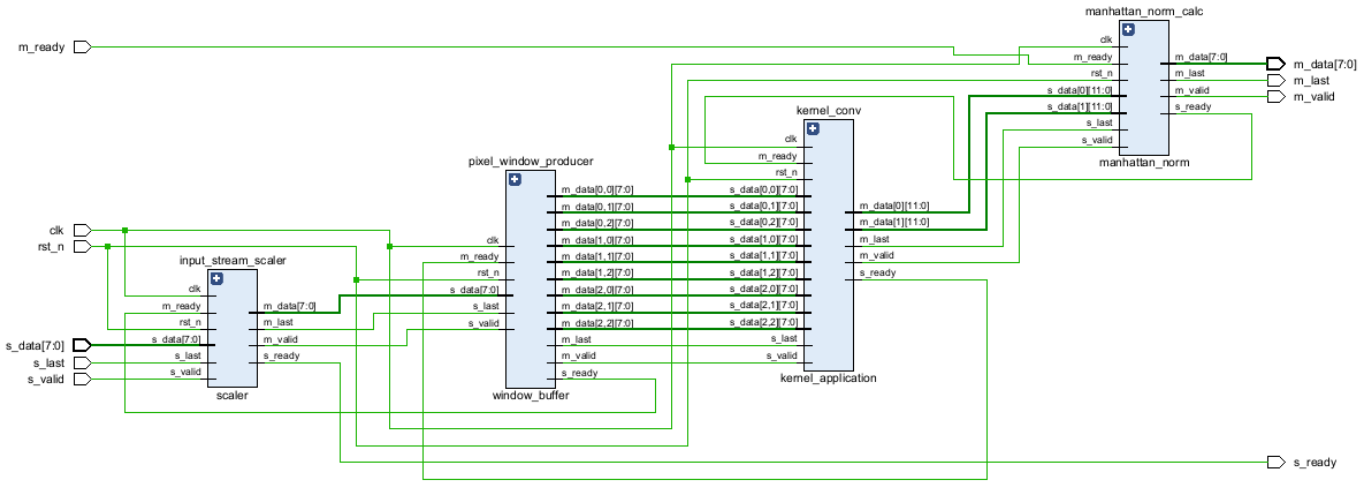
Χρησιμοποιείται για τον έλεγχο και τη ρύθμιση του IP μέσω register-based επικοινωνίας. Ο επεξεργαστής (ARM CPU) μπορεί να γράφει τιμές εντολών (start, reset κ.λπ.) ή να διαβάζει τιμές κατάστασης, counters και flags. Η επικοινωνία βασίζεται στα σήματα AWADDR, WDATA, ARADDR, RDATA και στους δείκτες εγκυρότητας VALID/READY, διασφαλίζοντας αξιόπιστη ανταλλαγή δεδομένων.

Πρωτόκολλο AXI4-Stream

Χρησιμοποιείται για συνεχή ροή δεδομένων, όπως είναι τα δεδομένα εικόνας ή βίντεο, χωρίς την χρήση διευθύνσεων αλλά μέσω σηματοδότησης. Το κάθε δείγμα, στην περίπτωσή μας pixel, μεταφέρεται μέσω του TDATA[7:0], ενώ τα σήματα TVALID, TREADY και TLAST ελέγχουν τη ροή και το τέλος του frame. Για τον master, το TVALID είναι σήμα εξόδου που συνοδεύει κάθε δεδομένο εξόδου, ενώ για τον slave είναι σήμα εισόδου που υποδεικνύει ότι τα εισερχόμενα δεδομένα είναι έγκυρα. Αντίστροφα, το TREADY παράγεται από τον slave και σηματοδοτεί ότι είναι έτοιμος να δεχτεί νέα δεδομένα. Ο master προωθεί δεδομένα μόνο όταν TVALID = 1 και TREADY = 1, εξασφαλίζοντας handshaking χωρίς απώλειες. Κάθε AXI IP block διαθέτει τουλάχιστον ένα slave και ένα master interface, ώστε να δέχεται και να προωθεί δεδομένα αντίστοιχα. Στο παρόν design, όλα τα components που θα δημιουργηθούν θα αποτελούν AXI blocks με αντίστοιχα interfaces, διευκολύνοντας τη διασύνδεση μέσω του AXI interconnect. Ο συνδυασμός των δύο πρωτοκόλλων επιτρέπει την ανεξάρτητη λειτουργία ελέγχου και μεταφοράς δεδομένων. Επιπλέον, η αρχιτεκτονική αυτή είναι πλήρως συμβατή με το AXI οικοσύστημα του Zynq, επιτρέποντας την άμεση σύνδεση με DMA, ARM CPU και άλλα IP cores.

Κεντρική Μονάδα Επεξεργασίας Sobel

Η κεντρική μονάδα επεξεργασίας Sobel, Sobel Edge Filter που αντιστοιχεί στο **sobel_processing_core.vhd** αποτελείται από επιμέρους μονάδες που συνεργάζονται για να υλοποιήσουν πλήρως τη λειτουργία του φίλτρου Sobel. Κάθε μονάδα αναλαμβάνει συγκεκριμένο κομμάτι της επεξεργασίας, από την προετοιμασία των pixel μέχρι τον υπολογισμό των παραγώγων και τη δημιουργία του τελικού μέτρου κλίσης, διατηρώντας παράλληλη ροή δεδομένων και αποδοτική χρήση των πόρων του FPGA.



Εικόνα 7 - Η δομή του sobel processing core. Αποτελείται από το input stream scaler, το pixel window producer, το kernel conv και το manhattan norm calc.

Αρχικά, η μονάδα input stream scaler που αντιστοιχεί στο **scaler.vhd** είναι υπεύθυνη για την προεπεξεργασία των δεδομένων εισόδου, εξασφαλίζοντας ότι τα pixel της εικόνας βρίσκονται στο κατάλληλο εύρος

τιμών ώστε όλο και λιγότερα να κάνουν overflow στην συνέχεια. Διαιρώντας τις αρχικές τιμές των pixel στην αρχή του pipeline έχει ως αποτέλεσμα η εικόνα που θα προκύψει να είναι πιο σκοτεινή. Εφόσον όμως δεν κάναμε κάτι αντίστοιχο και στην software έκδοση του αλγόριθμου, είναι πιο συνετό να μην διαιρέσουμε τα δεδομένα εισόδου, καθώς αν διαιρούσαμε θα είχαμε διαφορετικό οπτικό αποτέλεσμα. Παρ'ολ'αυτά διατηρούμε την μονάδα σε περίπτωση που χρειάζεται να εφαρμόσουμε κάποιο scaling, αλλά στην πραγματικότητα απλά προωθεί τα δεδομένα στο επόμενο βήμα επεξεργασίας.

Στη συνέχεια, η μονάδα pixel window producer που αντιστοιχεί στο **window_buffer.vhd** υλοποιεί έναν ολισθαίνοντα buffer που συγκεντρώνει τα pixel μιας εικόνας ώστε να σχηματίζει ένα παράθυρο 3×3 γύρω από κάθε τρέχον pixel. Κάθε νέο pixel προστίθεται στην αρχή του buffer, ενώ τα υπάρχοντα μετακινούνται, δημιουργώντας μια συνεχή ροή δεδομένων. Όταν έχουν ληφθεί τουλάχιστον δύο πλήρεις σειρές και τρία pixel της τρέχουσας σειράς, σχηματίζεται το 3×3 παράθυρο, το οποίο εξάγεται προς επεξεργασία. Οι μετρητές σειρών και στηλών παρακολουθούν τη θέση του pixel, ενώ σημαίες valid και last ενημερώνουν για την εγκυρότητα και το τέλος της εικόνας. Με αυτόν τον τρόπο διατηρείται συνεχής ροή δεδομένων, καθώς η πρόσβαση στα γειτονικά pixels πραγματοποιείται εσωτερικά χωρίς επιπλέον καθυστερήσεις στη μνήμη.

Η μονάδα kernel conv που αντιστοιχεί στο **kernel_application.vhd** που ακολουθεί εφαρμόζει τις δύο μάσκες Sobel, G_x και G_y σε κάθε παράθυρο 3×3 pixel που παρέχει ο pixel_window_producer. Ο υπολογισμός των οριζόντιων και κατακόρυφων βαθμίδων έντασης γίνεται μέσω πράξεων αθροισμάτων και πολλαπλασιασμών σε signed arithmetic, με τα αποτελέσματα να καταλήγουν σε έναν καθορισμένο αριθμό bits για κάθε gradient. Μάλιστα μέσω 3 processes γίνεται ένα εσωτερικό pipeline με 3 stages. Αυτό έγινε διότι στην αρχική του μορφή με ένα stage υπήρχαν timing violations κατά το implementation, ενώ με δύο stages το WNS ήταν οριακά στο μηδέν.

Η τελική επεξεργασία λαμβάνει χώρα στη μονάδα manhattan norm calc που αντιστοιχεί στο **manhattan_norm.vhd**, όπου υπολογίζεται το μέτρο της κλίσης βάσει της Manhattan μετρικής. Αφού έρθουν ως είσοδο τα αποτελέσματα της προηγούμενης υπομονάδας, υπολογίζεται η μετρική. Το αποτέλεσμα περιορίζεται στο εύρος 0–255, ώστε να αντιστοιχεί σε έγκυρη τιμή φωτεινότητας στο pixel εξόδου. Η συγκεκριμένη μετρική χρησιμοποιείται αντί της Ευκλείδειας, καθώς αποφεύγει τις πολύπλοκες πράξεις τετραγωνισμού και υπολογισμού ριζών, διατηρώντας ωστόσο τις ίδιες περιοχές μέγιστης έντασης (δηλαδή τις ίδιες ακμές).

Η οργάνωση του sobel processing core αξιοποιεί παραλληλία επιπέδου pixel (pixel-level parallelism), επιτρέποντας την ταυτόχρονη επεξεργασία πολλών pixels ανά κύκλο. Ολόκληρη η αρχιτεκτονική λειτουργεί σε περιβάλλον pipelined επεξεργασίας, με στόχο τη μέγιστη απόδοση, τη σταθερότητα και την αποδοτική εκμετάλλευση των πόρων του FPGA. Χάρη στη δομή αυτή, ο Sobel Edge Detector Accelerator επιτυγχάνει υψηλή ταχύτητα επεξεργασίας, καθιστώντας τον κατάλληλο για ενσωμάτωση σε συστήματα πραγματικού χρόνου που βασίζονται στο Zynq-7000 AP SoC.

Αξιολόγηση Σχεδίασης

Η παρούσα ενότητα εστιάζει στην αξιολόγηση και επαλήθευση της σχεδίασης του Sobel Edge Detection IP Core, τόσο σε επίπεδο προσομοίωσης όσο και σε πλήρη ενσωμάτωση στο SoC. Η λειτουργικότητα του επιταχυντή επαληθεύτηκε αρχικά μέσω VHDL testbench, συγκρίνοντας τα αποτελέσματα με το software, ενώ στη συνέχεια πραγματοποιήθηκε η ενσωμάτωσή του στο Sobel Edge Detector SoC χρησιμοποιώντας το Xilinx Vivado, συνδέοντας το με τον επεξεργαστή και τον DMA. Το testbench προσομοιώνει πλήρως τη ροή δεδομένων εισόδου και εξόδου, καταγράφοντας τα αποτελέσματα για τον έλεγχο της ορθότητας της επεξεργασίας εικόνας και της λειτουργίας των μετρητών του κυκλώματος. Παράλληλα, μετρήθηκαν οι χρόνοι εκτέλεσης και οι πόροι που χρησιμοποιούνται κατά την υλοποίηση στο FPGA, ενώ η λειτουργία του συστήματος δοκιμάστηκε σε περιβάλλον PetaLinux, αξιολογώντας την επίδοση του ολοκληρωμένου SoC και το throughput του IP Core. Τα αποτελέσματα επιβεβαιώνουν την ορθότητα

της αρχιτεκτονικής, τη σωστή διαχείριση των clock domains και την επιτυχή επιτάχυνση της επεξεργασίας εικόνας σε πραγματικό χρόνο.

Επαλήθευση Λειτουργίας IP

Η επαλήθευση επικεντρώθηκε στην επιβεβαίωση της σωστής λειτουργίας του Sobel processing core σε επίπεδο IP πριν την ενσωμάτωση με τις FIFO, το AXI4 interface και στην συνέχεια στο SoC. Κατά την προσομοίωση ελέγχθηκαν η ορθή μεταγωγή δεδομένων μέσω AXI-Stream, η σωστή λειτουργία σημάτων ελέγχου (valid/ready/last) και η συμπεριφορά των εσωτερικών στοιχείων υπό κανονική ροή δεδομένων. Η δοκιμή διασφάλισε επίσης τη χρονική συμμόρφωση του pipeline και την ορθότητα της ακολουθίας επεξεργασίας των pixel όπως επιβάλλει ο αλγόριθμος Sobel.

Πρόγραμμα δοκιμής (testbench)

Σκοπός του testbench είναι η πλήρης προσομοίωση της λειτουργίας του Sobel Edge Filter, με ανάγνωση των pixel εισόδου από αρχείο, τροφοδοσία τους στο DUT και καταγραφή των εξόδων σε αρχείο ώστε να είναι δυνατή η ανακατασκευή της επεξεργασμένης εικόνας. Για την ανάγνωση/εγγραφή αρχείων χρησιμοποιείται η βιβλιοθήκη TEXTIO και την STD_LOGIC_TEXTIO, διότι το testbench διαβάζει τις τιμές των pixel εισόδου από ένα αρχείο .txt, όπου κάθε γραμμή αντιστοιχεί στην τιμή ενός pixel της αρχικής εικόνας (σε μορφή ακέραιου). Αντίστοιχα, οι τιμές εξόδου που παράγει η μονάδα καταγράφονται σε ένα δεύτερο αρχείο .txt.

Η προσομοίωση οργανώνεται σε διάφορες διεργασίες (processes), καθεμία με ξεχωριστό ρόλο:

1. **Παραγωγή ρολογιού**

   Το testbench παράγει ένα κύριο σήμα ρολογιού με περίοδο 5 ns (200 MHz). Ο παλμός δημιουργείται από διεργασία που αντιστρέφει την τιμή του σήματος κάθε CLK_PERIOD/2 μέχρι να τερματιστεί η προσομοίωση.

2. **Επαναφορά (reset)**

   Υπάρχει διεργασία επαναφοράς που κρατά το low active reset σε '0' για μερικούς κύκλους (reset interval) και στη συνέχεια το απελευθερώνει σε '1' πριν ξεκινήσει η κανονική λειτουργία του DUT. Η διαδικασία εξασφαλίζει ότι οι καταστάσεις ξεκινούν από γνωστή τιμή.

3. **Διεργασία εισαγωγής δεδομένων (stimulus_process)**

   Η διεργασία διαβάζει σειριακά τις τιμές των pixel από το αρχείο εισόδου (ένα pixel ανά γραμμή, σε μορφή ακεραίου), τις μετατρέπει σε std_logic_vector και τις στέλνει στο DUT μέσω των σημάτων AXI-Stream: s_data, s_valid, s_ready, s_last. Κάθε pixel αποστέλλεται μόνο όταν το DUT δηλώνει ετοιμότητα (s_ready = '1') και οι μεταβιβάσεις συγχρονίζονται στην ανερχόμενη ακμή του ρολογιού. Όταν ολοκληρωθεί η αποστολή όλων των pixel, η διεργασία κλείνει το αρχείο και εκτυπώνει τον αριθμό των pixel που στάλθηκαν.

4. **Διεργασία λήψης εξόδου (output_process)**

   Η διεργασία παρακολουθεί το σήμα εξόδου m_valid και, όταν είναι ενεργό, διαβάζει το m_data και το γράφει σειριακά στο αρχείο εξόδου. Το m_ready κρατιέται ενεργό ώστε το testbench να είναι πάντα έτοιμο να δεχθεί δεδομένα. Η διεργασία τερματίζεται μετά τη λήψη του συνολικού πλήθους εξόδου (OUTPUT_PIXELS) και κλείνει το αρχείο.

5. **Έλεγχος ολοκλήρωσης προσομοίωσης**

   Ο έλεγχος ολοκλήρωσης βασίζεται σε σημαία sim_ended που ενεργοποιείται όταν η διεργασία εξόδου έχει λάβει όλα τα pixel. Μετά την ολοκλήρωση, το testbench καταγράφει συνοπτικά στατιστικά: πλήθος pixel εισόδου και εξόδου, συνολικός χρόνος προσομοίωσης και επιδόσεις.

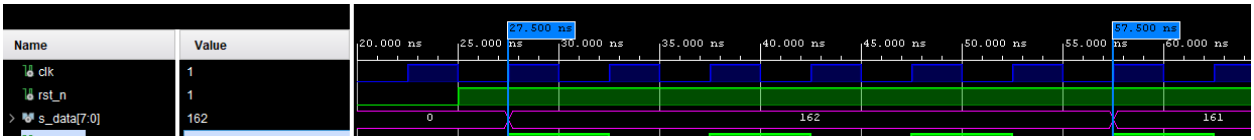
6. Παρακολούθηση / Debug (debug_process)

Μία από τη διεργασία καταγράφει περιοδικά στατιστικά (αριθμό pixel που στάλθηκαν/λήφθηκαν ανά χρονικό διάστημα) και στο τέλος υπολογίζει την απόδοση σε δείγμα/κύκλο (samples/cycle). Αυτή η διεργασία βοηθά στην επαλήθευση της ορθής συμπεριφοράς των handshakes και στην εκτίμηση throughput.

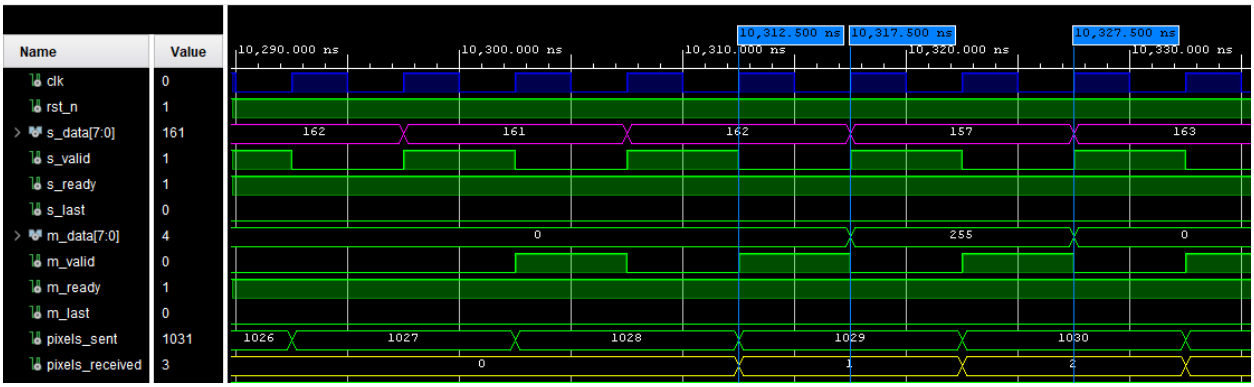
Αποτελέσματα testbench

Με το testbench λαμβάνουμε την αναμενόμενη εικόνα έξοδο και την συγκρίνουμε με τις υπόλοιπες στην Εικόνα 12 και βλέπουμε ότι είναι πρακτικά πανομοιότυπες. Επίσης συγκρίνουμε τις κατανομές της φωτεινότητας των pixels με την software υλοποίηση σε C στην Εικόνα 13. Για είσοδο 512×512 το αναμενόμενο πλήθος pixel εξόδου είναι 510·510 = 260100 pixel, το οποίο επαληθεύτηκε από το testbench.

Το pipeline του Sobel Edge Filter έχει 6 stages και επειδή το δεύτερο stage του pixel window producer απαιτεί 2 γραμμές και 3 pixel δεδομένα εισόδου (1028 pixel) ώστε να παράξει το πρώτο pixel εξόδου, το συνολικό latency προκύπτει να είναι 1028 + 5 = 1032 κύκλους. Μετρήθηκε throughput περίπου 0.996 samples/cycle (Δεν είναι στην μονάδα λόγω του αρχικού latency που απαιτείται). Επίσης να αναφερθεί ότι θα είχαμε ακόμα παραπάνω latency αν κάναμε το testbench συμπεριλαμβάνοντας και τις FIFO. Οι FIFO έχουν υλοποιηθεί μέσω του IP Catalog της Xilinx και, επειδή λειτουργούν με διαφορετικούς χρονισμούς, απαιτούνται ορισμένοι κύκλοι για τον συγχρονισμό μεταξύ των clock domains. Σύμφωνα με το IP Catalog του Vivado, η καθυστέρηση για κάθε FIFO αντιστοιχεί σε συνολικά 5 κύκλους (1 στον WR_CLK και 4 στον RD_CLK), οπότε θα είχαμε ακόμη 10 κύκλους latency, πάνω από τους 1032, σύνολο 1042 κύκλους.



Εικόνα 8 – Μέρος του Waveform από το Behavioral Simulation του Sobel Processing Core. Με το που το active low reset ανεβαίνει, μετράμε το συνολικό latency των 6 κύκλων που απαιτείται για το πρώτο pixel που εισέρχεται στον sobel processing core. Από εκεί και πέρα απαιτούνται δύο κύκλοι των 10 ns για να εισέλθει το κάθε νέο pixel.



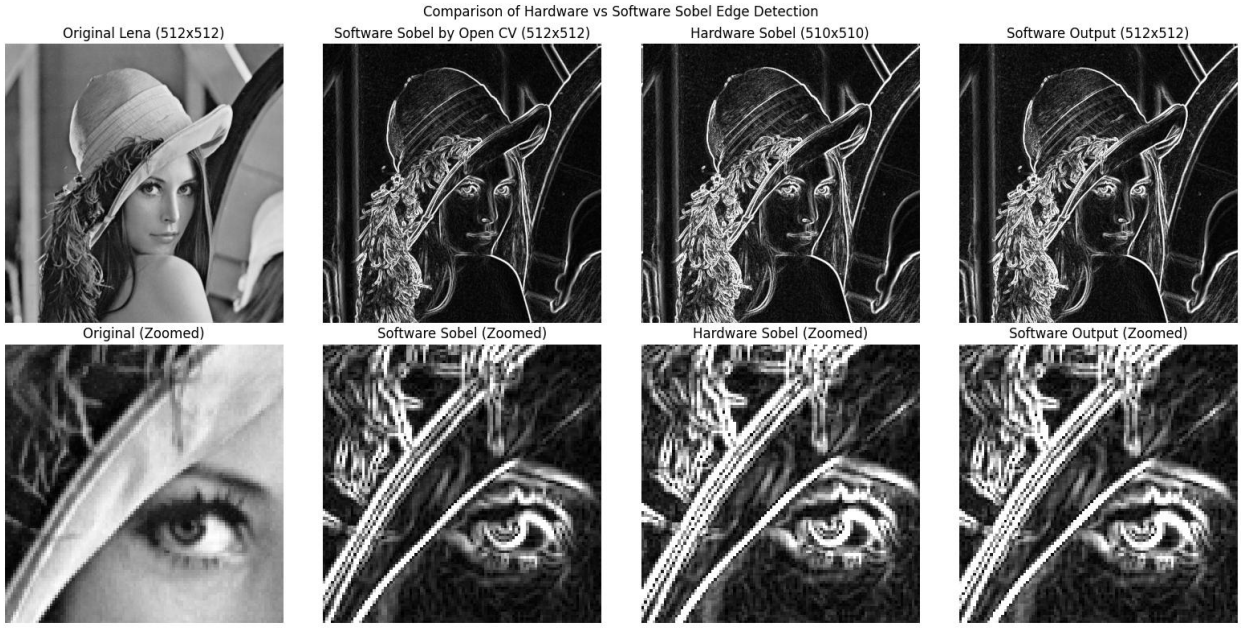
*Εικόνα 9 - Μέρος του Waveform από το Behavioral Simulation του Sobel Processing Core. Βλέπουμε πως το πρώτο pixel εξόδου βγαίνει όταν τα pixel εισόδου είναι 1028, δηλαδή 512*2 + 3 για να γεμίσει η fifo και άλλο ένα sample extra,*

```
-------------------------------------------------------------------------
-- Stimulus Process: Read from file and send to DUT
-------------------------------------------------------------------------
stimulus_process : process
    variable file_line  : line;
    variable pixel_val  : integer;
    variable line_count : integer := 0;
    variable timeout_start : time;
begin
    -- Wait for reset to complete
    wait until rst_n = '1';
    wait until rising_edge(clk);

    report "Starting stimulus process at time " & time'image(now);
    start_time <= now;

    -- Open input file
    file_open(input_f, INPUT_FILE, read_mode);

    -- Send all pixels
    while not endfile(input_f) and line_count < TOTAL_PIXELS loop
        readline(input_f, file_line);
        read(file_line, pixel_val);

        -- Simple handshake without procedure
        s_data <= std_logic_vector(to_unsigned(pixel_val, pixel_width));
        s_valid <= '1';
        if line_count = TOTAL_PIXELS - 1 then
            s_last <= '1';
        else
            s_last <= '0';
        end if;

        -- Wait for ready with timeout
        timeout_start := now;
        while s_ready = '0' loop
            wait until rising_edge(clk);
            if (now - timeout_start) > SEND_TIMEOUT then
                report "Send timeout at pixel " & integer'image(line_count) & " at time " & time'image(now) severity warning;
                exit;
            end if;
        end loop;

        -- Complete transaction
        wait until rising_edge(clk);
        s_valid <= '0';
        s_last <= '0';

        pixels_sent <= line_count + 1;
        line_count := line_count + 1;

        -- Small delay between pixels to simulate realistic data rate
        wait for CLK_PERIOD;
    end loop;

    file_close(input_f);

    if line_count = TOTAL_PIXELS then
        report "Successfully sent all " & integer'image(TOTAL_PIXELS) & " input pixels";
    else
        -- report "Sent " & integer'image(line_count) & " out of " & integer'image(TOTAL_PIXELS) & " pixels";
    end if;

    wait;
end process stimulus_process;
```

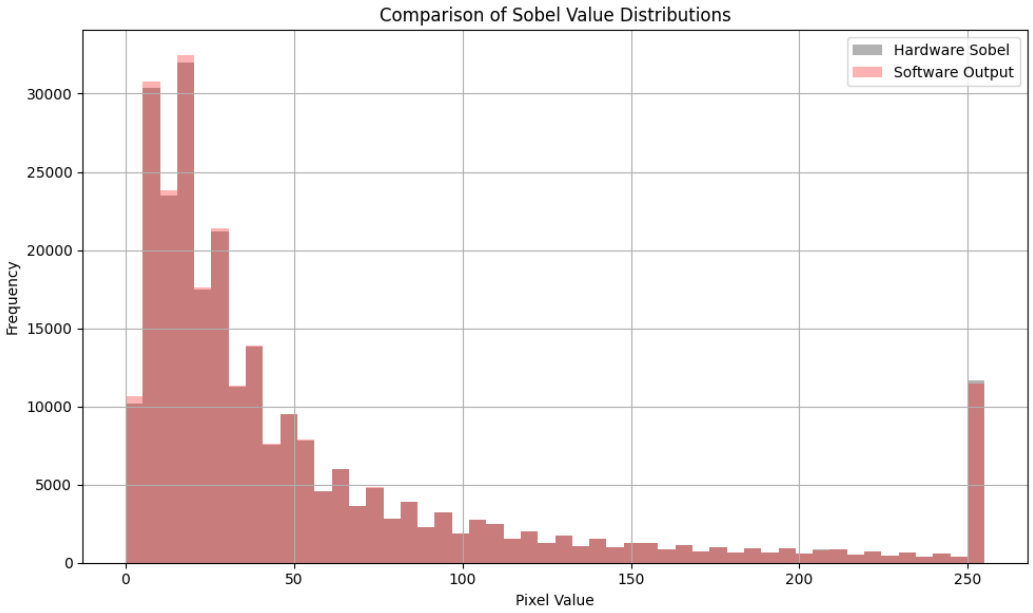
Εικόνα 10 – Η διεργασία εισαγωγής δεδομένων εισόδου – Input Stimulus Process.

```
-- ------------------------------------------------------------------------
-- Output Process: Receive from DUT and write to file
-- ------------------------------------------------------------------------
output_process : process
    variable file_line  : line;
    variable pixel_val  : integer;
    variable timeout_start : time;
    variable pixel_count : integer := 0;
begin
    -- Wait for reset to complete
    wait until rst_n = '1';
    wait until rising_edge(clk);

    report "Starting output process at time " & time'image(now);

    -- Open output file
    file_open(output_f, OUTPUT_FILE, write_mode);

    -- Receive all output pixels
    while pixel_count < OUTPUT_PIXELS loop
        -- Simple handshake without procedure
        m_ready <= '1';

        -- Wait for valid with timeout
        timeout_start := now;
        while m_valid = '0' loop
            wait until rising_edge(clk);
            if (now - timeout_start) > RECEIVE_TIMEOUT then
                report "Receive timeout at pixel " & integer'image(pixel_count) & " at time " & time'image(now) severity warning;
                m_ready <= '0';
                exit;
            end if;
        end loop;

        -- Capture data
        pixel_val := to_integer(unsigned(m_data));

        -- Write to file
        write(file_line, pixel_val);
        writeline(output_f, file_line);

        -- Check for last signal
        if m_last = '1' then
            report "Received m_last signal at pixel " & integer'image(pixel_count);
        end if;

        -- Complete transaction
        wait until rising_edge(clk);
        m_ready <= '0';

        pixels_received <= pixel_count + 1;
        pixel_count := pixel_count + 1;

        if pixel_count >= OUTPUT_PIXELS then
            exit;
        end if;
    end loop;

    file_close(output_f);
    end_time <= now;

    if pixel_count = OUTPUT_PIXELS then
        report "Successfully received all " & integer'image(OUTPUT_PIXELS) & " output pixels";
    else
        report "Received " & integer'image(pixel_count) & " out of " & integer'image(OUTPUT_PIXELS) & " pixels";
    end if;

    -- End simulation
    wait for CLK_PERIOD * 10;
    sim_ended <= true;
    wait;
end process output_process;
```

Εικόνα 11 – Η διεργασία ανάγνωσης δεδομένων εξόδου – Output Capture Process.



Εικόνα 12 - Σύγκριση των διαφορετικών υλοποιήσεων του αλγορίθμου ανίχνευσης ακμών Sobel. Πάνω φαίνονται ολόκληρες οι εικόνες ενώ στο κάτω μέρος βλέπουμε zoomed ένα μέρος της κάθε εικόνας. Από αριστερά προς τα δεξιά έχουμε, την αρχική εικόνα, Sobel μέσω της βιβλιοθήκης Open CV της python, Sobel που παράχθηκε από το testbench, και την εικόνα που παράχθηκε από το πρόγραμμα που υλοποιήθηκε σε C.

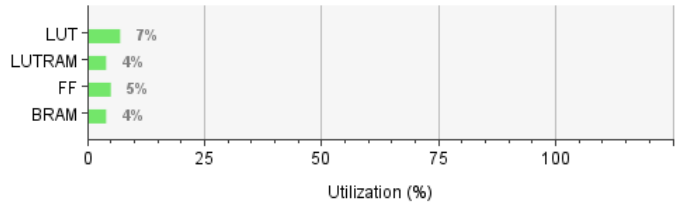


Εικόνα 13 - Σύγκριση των κατανομών της φωτεινότητας των pixel των εικόνων του testbench (Hardware Sobel) και της υλοποίησης σε C (Software Output). Βλέπουμε πως είναι σχεδόν πανομοιότυπες.

Αποτελέσματα implementation της αρχιτεκτονικής από το εργαλείο σύνθεσης

Αφού τα testbench επιβεβαίωσαν ότι η λειτουργία του συστήματος ήταν ορθή, πραγματοποιήθηκε η σύνθεση και η υλοποίηση (synthesis & implementation) της συνολικής αρχιτεκτονικής μέσω του εργαλείου VIVADO. Σκοπός αυτής της διαδικασίας ήταν η δημιουργία του bitstream, το οποίο θα ενσωματωθεί στο build του λειτουργικού PetaLinux και θα φορτωθεί στην πλακέτα Zedboard μέσω SD κάρτας, προκειμένου να προγραμματιστεί το programmable logic (PL) του FPGA. Κατά τη φάση του implementation, το VIVADO παράγει επίσης αναφορές (reports) σχετικά με τους πόρους του PL fabric που χρησιμοποιήθηκαν, καθώς και πληροφορίες για τυχόν χρονικές παραβιάσεις με στόχους συχνοτήτων 100 MHz και 200 MHz. Τέλος, για το λειτουργικό σύστημα PetaLinux, δημιουργήθηκαν τα αντίστοιχα binaries με στόχο την επεξεργασία εικόνων ανάλυσης 512x512. Στην επόμενη εικόνα μπορούμε να δούμε το utilization report.

Resource	Utilization	Available	Utilization %
LUT	3576	53200	6.72
LUTRAM	678	17400	3.90
FF	5183	106400	4.87
BRAM	6	140	4.29



Εικόνα 14 - Utilization Summary του Sobel SoC.

Hierarchy

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	3576	5183	1615	2898	678	6	130	2
design_1_i (design_1)	3576	5183	1615	2898	678	6	0	2
DMA (DMA_imp_1JYPGAM)	1359	1973	618	1242	117	5	0	0
Interconnect (Interconnect_imp_19FMNWV)	1574	1613	611	1274	300	0	0	0
Processing_System (Processing_System_imp_18RI0ON)	17	29	11	16	1	0	0	2
Sobel_Edge_Detector (Sobel_Edge_Detector_imp_39XXY9)	627	1568	415	367	260	1	0	0
Sobel_Edge_Detector_0 (design_1_Sobel_Edge_Detector_0_1)	627	1568	415	367	260	1	0	0
U0 (Sobel_Edge_Detector)	627	1568	415	367	260	1	0	0
axi_lite_interface_inst (axi4_lite_interface)	14	74	26	14	0	0	0	0
sobel_accelerator_inst (sobel_accelerator)	613	1494	398	353	260	1	0	0
Input_FIFO (fifo_generator_0__xdcDup__1)	79	213	59	77	2	0.5	0	0
Output_FIFO (fifo_generator_0)	78	213	54	76	2	0.5	0	0
Sobel_Edge_Filter (sobel_processing_core)	419	876	240	163	256	0	0	0
div4_scaler (scaler)	3	10	6	3	0	0	0	0
kernel_convolution (kernel_application)	75	140	45	75	0	0	0	0
magnitude_calculation (manhattan_norm)	54	45	18	54	0	0	0	0
window_producer_buffer (window_buffer)	287	681	186	31	256	0	0	0

Εικόνα 15 - Το utilization report του Sobel SoC.

Στην Εικόνα 15 βλέπουμε τους υπολογιστικούς πόρους που αξιοποιήθηκαν από το Sobel Edge Detector και τα επιμέρους submodules του. Τα Look-Up Tables (LUTs) χρησιμοποιούνται τόσο για την υλοποίηση της

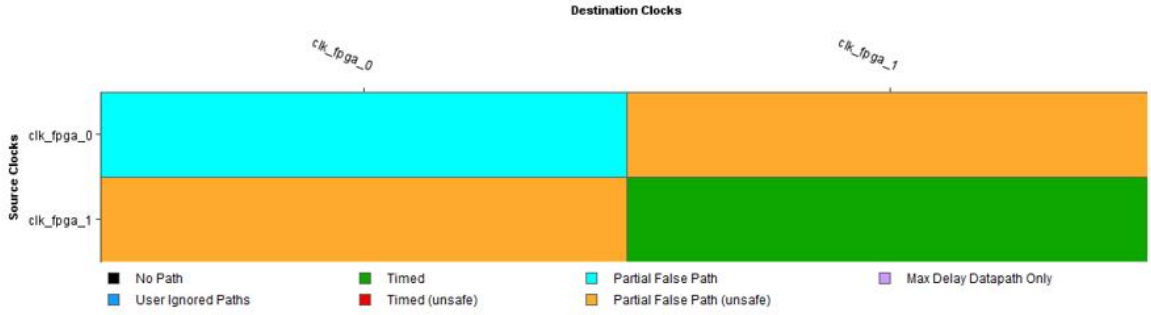
ακολουθιακής λογικής, όσο και ως μνήμη (LUT as Memory). Η Block RAM χρησιμοποιείται μόνο από τις FIFO δομές, οι οποίες έχουν δημιουργηθεί μέσω του IP Catalog της Xilinx, όπως είναι και αναμενόμενο. Στην Εικόνα 16 βλέπουμε πως δεν υπάρχουν παραβιάσεις χρονισμού (timing violations), ωστόσο το WNS είναι λίγο μικρό για το ρολόι των 5ns. Στην Εικόνα 17 βλέπουμε το clock interaction report, όπου και μπορούμε να δούμε το WNS για διαφορετικό source-destination clock.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,298 ns	Worst Hold Slack (WHS):	0,036 ns	Worst Pulse Width Slack (WPWS):	1,520 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	17124	Total Number of Endpoints:	17084	Total Number of Endpoints:	6184

All user specified timing constraints are met.

Εικόνα 16 - Το timing summary του Sobel SoC.



Id	Source Clock	Destination Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Path Req (WNS)	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total Endpoints (THS)	Path Req (WHS)	Clock Pair Classification	Inter-Clock Constraints
1	clk_fpga_1	clk_fpga_1	rise - rise	0.616	0.000	0	2666	5.000	rise - rise	0.038	0.000	0	2666	0.000	Clean	Timed
2	clk_fpga_1	clk_fpga_0	rise - rise	2.001	0.000	0	54	5.000	rise - rise	0.170	0.000	0	54	0.000	No Common Clock	Partial False Path (unsafe)
3	clk_fpga_0	clk_fpga_1	rise - rise	0.298	0.000	0	916	5.000	rise - rise	0.393	0.000	0	916	0.000	No Common Clock	Partial False Path (unsafe)
4	clk_fpga_0	clk_fpga_0	rise - rise	1.671	0.000	0	13597	10.000	rise - rise	0.036	0.000	0	13597	0.000	Clean	Partial False Path

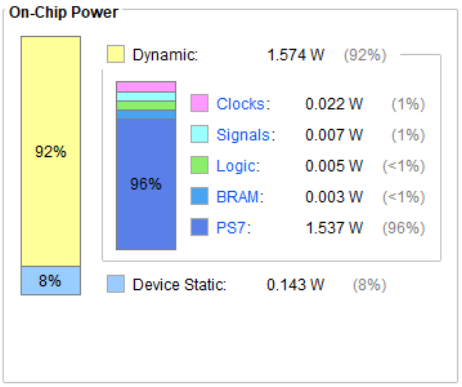
Εικόνα 17 - Clock Interaction Report για το Sobel SoC.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.717 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**44,8°C**
Thermal Margin:	40,2°C (3,3 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



Εικόνα 18 - Power Analysis του Sobel SoC.

Δοκιμή σε περιβάλλον PetaLinux

Το PetaLinux αποτελεί ένα ενσωματωμένο (embedded) λειτουργικό σύστημα, ειδικά προσαρμοσμένο για χρήση σε SoC πλατφόρμες που περιλαμβάνουν FPGA. Στην περίπτωση του Zedboard, το λειτουργικό εκτελείται στον επεξεργαστή ARM Cortex-A9 του συστήματος. Για να εγκατασταθεί το PetaLinux στην αναπτυξιακή πλακέτα, πρέπει πρώτα να δημιουργηθούν τα απαραίτητα αρχεία εκκίνησης (bootloader) και το file system του λειτουργικού, μαζί με το bitstream που προγραμματίζει το programmable logic (PL) της πλακέτας. Παράλληλα, διαμορφώνονται και τα περιφερειακά στοιχεία του συστήματος, όπως ο AXI Interconnect και ο DMA Controller.

Η παραγωγή των απαραίτητων system binaries πραγματοποιείται μέσω scripts που δόθηκαν στο πλαίσιο της εργασίας. Για το σκοπό αυτό, δημιουργείται το bitstream του Block Design μέσα από το VIVADO, και έπειτα γίνεται export του αρχείου .xsa, το οποίο περιλαμβάνει το bitstream και το configuration του hardware. Στη συνέχεια, με τα build tools του PetaLinux παράγονται τα εκτελέσιμα του λειτουργικού χρησιμοποιώντας το αρχείο .xsa, και τα αρχεία αυτά αντιγράφονται σε μία SD κάρτα με την κατάλληλη μορφοποίηση, η οποία τοποθετείται στο Zedboard. Πριν από αυτό το στάδιο, γίνεται μεταγλώττιση του εκτελέσιμου αρχείου του προγράμματος Sobel, χρησιμοποιώντας compiler κατάλληλο για επεξεργαστές ARM Cortex-A9, από τα αντίστοιχα αρχεία C. Το παραγόμενο εκτελέσιμο στη συνέχεια τοποθετείται σε συγκεκριμένο φάκελο μέσα στο file system του PetaLinux που υπάρχει στην SD κάρτα. Αφού η SD κάρτα ετοιμαστεί με όλα τα απαιτούμενα αρχεία, τοποθετείται στο slot του Zedboard.

(αναμένονται αποτελέσματα από το εργαστήριο όπου θα γίνει η επαλήθευση της λειτουργίας του SoC)

Βιβλιογραφία

Sobel, Irwin. (2014). An Isotropic 3x3 Image Gradient Operator. Presentation at Stanford A.I. Project 1968.

Wikipedia contributors, 'Sobel operator', *Wikipedia, The Free Encyclopedia,* 22 September 2025, 03:20 UTC, <https://en.wikipedia.org/w/index.php?title=Sobel_operator&oldid=1312694173> [accessed October 2025]