**Σχεδιασμός Ενσωματωμένων Συστημάτων**
**9ο εξάμηνο ΣΗΜΜΥ**
**4η εργαστηριακή άσκηση**

**Ομάδα 6:**
**Δούκας Θωμάς Α.Μ.: 03116081**
**Ψαρράς Ιωάννης Α.Μ.:03116033**

Στην παρούσα εργαστηριακή άσκηση θα μελετήσουμε τον προγραμματισμό σε FPGA με High Level Synthesis. Συγκεκριμένα, θα προσπαθήσουμε να επιταχύνουμε την εκτέλεση μιας εφαρμογής που τρέχει στο hardware. Για τις ανάγκες της άσκησης θα χρησιμοποιηθεί το Xilinx Zybo FPGA στο οποίο θα εκτελεστεί optimized C κώδικας. Η εφαρμογή αφορά την ανακατασκευή ημιτελών εικόνων από χειρόγραφα στοιχεία μέσω νευρωνικών δικτύων GANs (Generative Adversarial Networks).

## Άσκηση 1

Στο πρώτο ερώτημα θα επικεντρωθούμε στην ανάπτυξη της εφαρμογής στο embedded Zybo FPGA αλλά και στην επιτάχυνση της. Από τον κώδικα που παρέχεται στην εργαστηριακή άσκηση καλούμαστε να πραγματοποιήσουμε επιτάχυνση της συνάρτησης *forward_propagation* που βρίσκεται στο αρχείο *network.cpp*. Με τη βοήθεια του εργαλείου *SDSoC 2016.4* θα πραγματοποιήσουμε τις κατάλληλες αλλαγές, θα κάνουμε εκτίμηση των αποτελεσμάτων της εκτέλεσης και τελικά θα παράξουμε τα απαραίτητα αρχεία ώστε να είναι δυνατή η εκτέλεση της εφαρμογής στη πλακέτα Zybo.

### 1) Estimate Unoptimized Performance.

Αρχικά θα ξεκινήσουμε λαμβάνοντας μετρήσεις σχετικά με τους απαιτούμενους κύκλους εκτέλεσης της unoptimized έκδοσης για την συνάρτηση *forward_propagation*. Θέτοντας την παραπάνω ως συνάρτηση HW στο εργαλείο SDSoC 2016.4 πραγματοποιούμε estimate performance. Το εργαλείο παράγει προβλέψεις σχετικά με τους κύκλους που απαιτούνται αλλά και τα resourses που θα χρησιμοποιηθούν από την πλακέτα Zybo για την εκτέλεση της συνάρτησης στο hardware. Τα αποτελέσματα εμφανίζονται στη συνέχεια.

**Details**

**Performance estimates for 'forward_propagation in main.cp ...**

| | |
|---|---|
| HW accelerated (Estimated cycles) | 683780 |

**Resource utilization estimates for HW functions**

| Resource | Used | Total | % Utilization |
|---|---|---|---|
| DSP | 3 | 80 | 3,75 |
| BRAM | 16 | 60 | 26,67 |
| LUT | 1760 | 17600 | 10 |
| FF | 892 | 35200 | 2,53 |

## 2) Execute on Zybo

Ακολούθως, για τον δοθέντα κώδικα, δημιουργούμε την sd boot card η οποία περιλαμβάνει το εκτελέσιμο αρχείο .elf, το bitstream και το λειτουργικό petalinux του συστήματος και χρησιμοποιείται προκειμένου να εκκινεί το σύστημα. Μεταφέροντας, με τη βοήθεια των εντολών scp και zscp, τα παραγόμενα αρχείο από το τοπικό μηχάνημα στο portal και ύστερα στο Zybo μπορούμε να πραγματοποιήσουμε reboot ώστε να φορτωθεί το νέο σύστημα. Αναφέρουμε ότι στο φάκελο με τα απαραίτητα αρχεία συμπεριλαμβάνεται και το αρχείο data.txt. Στη συνέχεια χρησιμοποιούμε την εντολή ./Lab4.elf στο φάκελο */mnt/* προς εκτέλεση της εφαρμογής.

```
root@Avnet-Digilent-ZedBoard-2016_3:/mnt# ./Lab4.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 682947
Software cycles : 1476020
Speed-Up        : 2.16125
Saving results to output.txt...
```

Με βάση τα παραπάνω αποτελέσματα παρατηρούμε ότι η για την εκτέλεση του software απαιτούνται 1.476.020 κύκλοι. Αντίστοιχα, για την περίπτωση του hardware φαίνεται να απαιτούνται σημαντικά λιγότεροι κύκλοι, με την τιμή τους να είναι **682.947**.

Ιδιαίτερη σημασία έχει το γεγονός πως **τα παραγόμενα αποτελέσματα από το Zybo προσεγγίζουν σημαντικά εκείνα που παρουσιάστηκαν στο ερώτημα 1 της άσκησης.** Όπως φαίνεται και στη προηγούμενη υποενότητα η πρόβλεψη απαιτούμενων κύκλων για το hardware εμφανίζει 683.780 κύκλους, ενώ ταυτόχρονα τα πραγματικά αποτελέσματα παρουσιάζουν πως χρειάστηκαν 682.947 κύκλοι για την επιτυχή εκτέλεση της εφαρμογής. Το γεγονός αυτό μας προϊδεάζει πως το resource estimation είναι μια αρκετά αξιόπιστη διαδικασία και μπορούμε να βασιστούμε στα αποτελέσματα της χωρίς να απαιτείται η συνεχής δοκιμή των αποτελεσμάτων στη πλακέτα Zybo.

Επιπρόσθετα, οφείλουμε να σχολιάσουμε το speedup που επιφέρει στην εκτέλεση των υπολογισμών που μας αφορούν η χρήση του διαθέσιμου hardware έναντι της software εκτέλεσης στον ARM. Πιο συγκεκριμένα, από τους υπολογισμούς που πραγματοποιεί η συνάρτηση *main,* η οποία καλεί τον generator τόσο για hardware όσο και για software και μετρά τις αντίστοιχες επιδόσεις, παρατηρούμε πως το speedup υπολογίζεται 2,16125. Όπως φαίνεται και από τους απαιτούμενος κύκλους, η παραπάνω μετρική επιβεβαιώνει πως οι υπολογισμοί πραγματοποιούνται με **διπλάσια ταχύτητα** στην περίπτωση του hardware.

### 3) Design Space Exploration

Στη συνέχεια θα πραγματοποιήσουμε design space exploration με σκοπό τον προσδιορισμό των κατάλληλων optimizations τα οποία θα επιταχύνουν τον αλγόριθμο. Με τη βοήθεια του estimation καταγράφουμε την εκτίμηση των κύκλων που χρειάζεται κάθε νέα έκδοση της συνάρτησης *forward_propagation* ενώ ταυτόχρονα ενημερωνόμαστε για τους πόρους που απαιτείται να δεσμευτούν. Όπως είδαμε και προηγουμένως, η πρόβλεψη που παράγεται είναι αρκετά ικανοποιητική και μπορούμε να την εμπιστευτούμε καθώς προσεγγίζει ικανοποιητικά τα πραγματικά αποτελέσματα της εκτέλεσης στην πλακέτα.

Για την επιτάχυνση της εφαρμογής στηριζόμαστε στο γεγονός πως η συνάρτηση που μας απασχολεί αποτελείται από 4 διαφορετικές δομές επανάληψης (for loops). Επιπλέον μελέτη της συνάρτησης *forward_propagation* μας δείχνει πως υπάρχει εξάρτηση δεδομένων ανάμεσα στις δομές επαναλήψεων και άρα κάθε στάδιο οφείλει να εκτελείται μετά το προηγούμενο του ώστε να διασφαλιστεί η λήψη των σωστών αποτελεσμάτων. Για το λόγο αυτό επικεντρωθήκαμε στην χρήση των HLS pipeline και HLS unroll.

Συγκεκριμένα, επιδιώξαμε να μειώσουμε του κύκλους που απαιτούνται για την εκτέλεση κάθε σταδίου ξεχωριστά τοποθετώντας τα αντίστοιχα pragmas στα κατάλληλα σημεία του κώδικα. Σημαντική παρατήρηση για την παραπάνω διαδικασία είναι το γεγονός πως για τις για κάθε μία από τις δομές *for* οι εντολές του *iteration* επιδρούν πάνω σε συγκεκριμένες θέσεις μνήμης και δεν επηρεάζονται από προηγούμενες ή επόμενες επαναλήψεις.

Ως εκ τούτου επιλέγουμε την οδηγία **pragma HLS unroll** με στόχο να “ξεδιπλώσουμε” τις δομές επαναλήψεων. Με βάση την λειτουργία της παραπάνω εντολής, προσδιορίζοντας την τιμή του unroll **factor** καθορίζουμε το πλήθος των αντιγράφων του σώματος που δημιουργούνται στο σχέδιο RTL. Τα αντίγραφα αυτά βοηθούν σημαντικά στην επιτάχυνση της εφαρμογής καθώς δίνουν την δυνατότητα παράλληλης εκτέλεσης των υπολογισμών. Για την άσκηση επιλέξαμε να αφήσουμε απροσδιόριστη την παραπάνω τιμή , γεγονός που υποδηλώνει το ολικό unroll της δομής επανάληψης. Ωστόσο η πρακτική αυτή αποδείχθηκε λανθασμένη καθώς οι διαθέσιμοι πόροι της πλακέτας δεν ήταν αρκετοί. Σημαντικά μείωση του unroll factor για την περίπτωση του read_input (392 επαναλήψεις) επιτρέπει την χρήση των διαθέσιμων πόρων, χωρίς να απαιτούνται περισσότεροι.

Στη συνέχεια αξιοποιώντας την οδηγία **pragma HLS pipeline** έχουμε την δυνατότητα να μειώσουμε ακόμα περισσότερο τους κύκλους εκτέλεσης στο hardware εκτελώντας παράλληλα τις εντολές του σώματος των δομών επαναλήψεων. Για την εντολή pragma αρκεί να προσδιορίσουμε την τιμή του initiation interval (***II***) ώστε να ορίσουμε σε πόσους κύκλους είναι δυνατή η εκκίνησης του επόμενου loop. Αφήνοντας κενή την συγκεκριμένη παράμετρο ορίζουμε ως επιθυμητή την default τιμή 1, η οποία είναι και η ιδανική. Ωστόσο, όπως θα δούμε παρακάτω, η δεν επιλέγεται πάντα η συγκεκριμένη τιμή.

Ακολούθως παρατίθεται η τελική έκδοση του κώδικα:

```cpp
void forward_propagation(float *x, float *y)
{
    quantized_type xbuf[N1];
    l_quantized_type layer_1_out[M1];
    l_quantized_type layer_2_out[M2];

    //limit resources to max DSP number of Zybo - do not change
    #pragma HLS ALLOCATION instances=mul limit=80 operation

    read_input:
    for (int i=0; i<N1; i++)
    {
        #pragma HLS unroll factor=2
        #pragma HLS pipeline
        xbuf[i] = x[i];
    }

    // Layer 1
    layer_1:
    for(int i=0; i<N1; i++)
    {
        #pragma HLS pipeline
        for(int j=0; j<M1; j++)
        {
            #pragma HLS unroll
            l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
            quantized_type term = xbuf[i] * W1[i][j];
            layer_1_out[j] = last + term;
        }
    }
    layer_1_act:
    for(int i=0; i<M1; i++)
    {
        #pragma HLS unroll
        layer_1_out[i] = ReLU(layer_1_out[i]);
    }

    // Layer 2
    layer_2:
    for(int i=0; i<M2; i++)
    {
        #pragma HLS pipeline
        l_quantized_type result = 0;
        for(int j=0; j<N2; j++)
        {
            #pragma HLS unroll
            l_quantized_type term = layer_1_out[j] * W2[j][i];
            result += term;
        }
        layer_2_out[i] = ReLU(result);
    }

    // Layer 3
    layer_3:
    for(int i=0; i<M3; i++)
    {
        #pragma HLS pipeline
        l_quantized_type result = 0;
        for(int j=0; j<N3; j++)
        {
            #pragma HLS unroll
            l_quantized_type term = layer_2_out[j] * W3[j][i];
            result += term;
        }
        y[i] = tanh(result).to_float();
    }
}
```

Η παραπάνω στρατηγική εμφανίζει τα εξής αποτελέσματα κατά την διαδικασία του estimation.

**Details**

**Performance estimates for 'forward_propagation in main.cp ...**

| | |
|---|---|
| HW accelerated (Estimated cycles) | 11992 |

**Resource utilization estimates for HW functions**

| Resource | Used | Total | % Utilization |
|---|---|---|---|
| DSP | 80 | 80 | 100 |
| BRAM | 41 | 60 | 68,33 |
| LUT | 6696 | 17600 | 38,05 |
| FF | 5911 | 35200 | 16,79 |

Αρχικά παρατηρούμε ότι οι εκτιμώμενοι κύκλοι έχουν μειωθεί σημαντικά σε σχέση με την αντίστοιχη unoptimized εκδοχή. Από τους αρχικούς 683.780 καταφέρουμε να πραγματοποιούμε την εκτέλεση σε **11.992 κύκλους.** Ωστόσο, η συγκεκριμένη υλοποίηση δεν επιτυγχάνει τα αποτελέσματα χωρίς κάποιο επιπλέον κόστος. Παρατηρούμε πως για να επιταχυνθεί η εκτέλεση χρησιμοποιούνται σημαντικά περισσότεροι πόροι σε σχέση με την αρχική έκδοση. Πιο συγκεκριμένα, αναφέρουμε πως παρατηρούμε αύξηση των χρησιμοποιούμενων πόρων κατά:

- 77 DSP
- 25 BRAM
- 4936 LUT
- 5019 FF

Συνεχίζουμε παράγοντας την sd_card όπως και στο προηγούμενο ερώτημα ώστε να εκτελέσουμε την εφαρμογή στο Zybo. Τα αποτελέσματα είναι:

```
root@Avnet-Digilent-ZedBoard-2016_3:/mnt# ./Lab4.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 12214
Software cycles : 1478343
Speed-Up        : 121.037
Saving results to output.txt...
root@Avnet-Digilent-ZedBoard-2016_3:/mnt#
```

Παρατηρούμε και πάλι πως τα αποτελέσματα που προκύπτουν κατά την εκτέλεση στο Zybo προσεγγίζουν την αντίστοιχη εκτίμηση. Οι πραγματικοί κύκλοι είναι **12.214** έναντι 682.947 της umoptimized εκδοχής και η μείωση των απαιτούμενων κύκλων αγγίζει ποσοστό **98,2%**.

Εξίσου σημαντική είναι η σύγκριση με την εκτέλεση σε software. Από την τιμή του speedup παρατηρούμε ότι με την χρήση των HLS pragmas καταφέραμε να παράξουμε εφαρμογή η οποία εκτελείται στο hadrware 121 φορές πιο γρήγορα από την αντίστοιχη εκτέλεση στο software. Συγκεκριμένα:
**Speed-Up : 121.037**

Τονίζουμε και πάλι πως η επιτάχυνση αυτή έρχεται με το κόστος τη χρήση περισσότερου υλικού-πόρων.

### 4) HLS Report

Η HLS Report που παράγεται εμφανίζει τα παρακάτω αποτελέσματα σχετικά με το Latency:

⊟ Loop

| | Latency | | | Initiation Interval | | | |
|---|---|---|---|---|---|---|---|
| Loop Name | min | max | Iteration Latency | achieved | target | Trip Count | Pipelined |
| - read_input | 395 | 395 | 6 | 2 | 1 | 196 | yes |
| - layer_1 | 393 | 393 | 3 | 1 | 1 | 392 | yes |
| - layer_2 | 52 | 52 | 4 | 1 | 1 | 50 | yes |
| - layer_3 | 400 | 400 | 10 | 1 | 1 | 392 | yes |

Παρατηρούμε ότι το **Layer 3** είναι εκείνο που έχει το μεγαλύτερο latency με την τιμή του να είναι **400**. Από τα παραπάνω κατανοούμε πως το Layer 3 χρειάζεται περισσότερους κύκλους από τα υπόλοιπα μέρη της συνάρτησης ώστε να εκτελεστούν όλες οι επαναλήψεις της δομής loop. Το γεγονός αυτό είναι, μάλλον, αναμενόμενο καθώς παρατηρώντας τον αρχικό κώδικα βλέπουμε ότι υπάρχει 1 εμφωλευμένο loop στο συγκεκριμένο layer και πραγματοποιούνται συνολικά 396*50 επαναλήψεις κατά την σειριακή εκτέλεση.

Από την δεξιά στήλη παρατηρούμε ότι όλες οι δομές επαναλήψεων χαρακτηρίζονται ως pipelined από το εργαλείο SDSoC 2016.4 και άρα το σχέδιο που υλοποιήσαμε **είναι fully pipelined**.

Προκειμένου να καταγράψουμε τα είδη των μαθηματικών εκφράσεων που χρησιμοποιούνται από το σχέδιο μας παρατηρούμε τα αποτελέσματα που εμφανίζονται στην καρτέλα Resourse profile του HLS Report. Ο πίνακας με τα δεδομένα εμφανίζεται στην επόμενη σελίδα.

| | BRAM | DSP | FF | LUT | Bits P0 | Bits P1 | Bits P2 | Banks/Depth | Words | W*Bits*Banks |
|---|---|---|---|---|---|---|---|---|---|---|
| forward_propagation | 82 | 80 | 5911 | 6612 | | | | | | |
| I/O Ports(2) | | | | | 64 | | | | | |
| Instances(2) | 0 | 0 | 268 | 677 | | | | | | |
| Memories(113) | 82 | | 305 | 245 | 1032 | | | 113 | 34326 | 311394 |
| Expressions(475) | 0 | 80 | 0 | 4568 | 4664 | 4684 | 1255 | | | |
| - | 0 | 0 | 0 | 156 | 32 | 156 | 0 | | | |
| * | 0 | 80 | 0 | 0 | 1090 | 980 | 0 | | | |
| + | 0 | 0 | 0 | 2458 | 2924 | 2941 | 0 | | | |
| and | 0 | 0 | 0 | 8 | 8 | 8 | 0 | | | |
| ashr | 0 | 0 | 0 | 322 | 108 | 108 | 0 | | | |
| icmp | 0 | 0 | 0 | 122 | 323 | 93 | 0 | | | |
| or | 0 | 0 | 0 | 23 | 19 | 11 | 0 | | | |
| select | 0 | 0 | 0 | 1276 | 80 | 300 | 1255 | | | |
| shl | 0 | 0 | 0 | 176 | 64 | 64 | 0 | | | |
| xor | 0 | 0 | 0 | 27 | 16 | 23 | 0 | | | |
| Registers(446) | | | 5338 | | 6109 | | | | | |
| Channels(0) | 0 | | 0 | 0 | 0 | | | 0 | 0 | 0 |
| Multiplexers(75) | 0 | | 0 | 1122 | 985 | | | 0 | | |

Παρατηρούμε ότι χρησιμοποιούνται όλα τα είδη πράξεων στο σχέδιο που έχει υλοποιηθεί. Ωστόσο τα περισσότερα από αυτά χρησιμοποιούν Look up tables για την εκτέλεση τους. Εξαίρεση αποτελεί **η έκφραση του πολλαπλασιασμού**, η οποία είναι η μοναδική που χρησιμοποιεί DSP. Όπως φαίνεται και από τις εκτιμήσεις στο προηγούμενο ερώτημα, οι εκφράσεις πολλαπλασιασμού αξιοποιούν και τα 80 διαθέσιμα DSPs. Το γεγονός αυτό είναι φυσικά αναμενόμενο καθώς τα DSP Blocks χρησιμοποιούνται για αποδοτικότερη υλοποίηση των πράξεων πολλαπλασιασμού.

## Άσκηση 2

Στην δεύτερη άσκηση, καλούμαστε να αξιολογήσουμε την ποιότητα ανακατασκευής των εικόνων που παράγουν τόσο το SW όσο και το HW.

### Α)

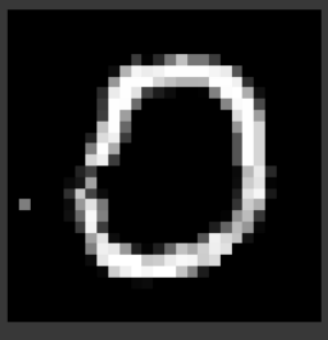
Αρχικά, λαμβάνουμε το αρχείο output.txt που προκύπτει από το εκτελέσιμο. Στη συνέχεια τρέχουμε το plot_output.ipynb για τους αριθμούς με *idx* ***10, 11*** *και* ***12*** και λαμβάνουμε τις combined εικόνες για τις παραπάνω εισόδους. Προσέχουμε να χρησιμοποιούμε τα κατάλληλα αρχεία εξόδου που παρήχθησαν από το Zybo για καθεμία από τις εκτελέσεις. Τα αποτελέσματα που λάβαμε από τις τρεις αυτές εκτελέσεις παρατίθενται ακολούθως:

*(Σημειώνεται ότι ο πάνω χαρακτήρας αφορά την εκτέλεση για software και ο κάτω την εκτέλεση για hardware)*

- ***idx = 10***

```
plt.figure()
plt.imshow(output_hw.reshape((28,28)).astype(np.float32), cmap = 'gray')
plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```



- ***idx = 11***

```
[17] plt.imshow(output_sw.reshape((28,28)).astype(np.float32), cmap = 'gray')
     plt.axis('off')

     plt.figure()
     plt.imshow(output_hw.reshape((28,28)).astype(np.float32), cmap = 'gray')
     plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```

- ***idx = 12***

```
plt.figure()
plt.imshow(output_hw.reshape((28,28)).astype(np.float32), cmap = 'gray')
plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```

**Β)**

Στο δεύτερο ερώτημα καλούμαστε να διαμορφώσουμε κατάλληλα τα αρχεία της εφαρμογής, για να αλλάξουμε την δεκαδική ακρίβεια των datatypes. Συγκεκριμένα για τις περιπτώσεις 4 και 10 bit πραγματοποιήσαμε αλλαγές στο αρχείο *network.h*, αρχικοποιώντας τις μεταβλητές BITS και BITS_EXP ως εξής:

*#define BITS 4*
*#define BITS_EXP 64*

και

*#define BITS 10*
*#define BITS_EXP 4096*

Επιπλέον θα πρέπει να αντικαταστήσουμε το αρχείο tanh.h με τα αρχεία εκείνα που περιέχουν τις κατάλληλες pre-computed τιμές την tanh. Στις περιπτώσεις που μας απασχολούν χρησιμοποιούμε τα αρχεία *tanh_4.h* αλλά και *tanh_10.h*.

Έπειτα από την πραγματοποίηση των αλλαγών, πρέπει να επανεκτελέσουμε την εφαρμογή ώστε παραχθούν δυο νέα *output.txt* αρχεία, ένα για ακρίβεια 4-bit και ένα για 10-bit. Τέλος, καλούμαστε να παρουσιάσουμε τις combined εικόνες που προκύπτουν για τα ίδια ***idx*** με το προηγούμενο ερώτημα.

Τα αποτελέσματα εκτέλεσης του δεύτερου ερωτήματος είναι τα παρακάτω:

- ***idx = 10***

**<u>4-bit accuracy</u>:**


**<u>10-bit accuracy</u>:**

- ***idx = 11***

**4-bit accuracy**:

```
plt.figure()
plt.imshow(output_hw.reshape((28,28)).astype(np.float32), cmap = 'gray')
plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```

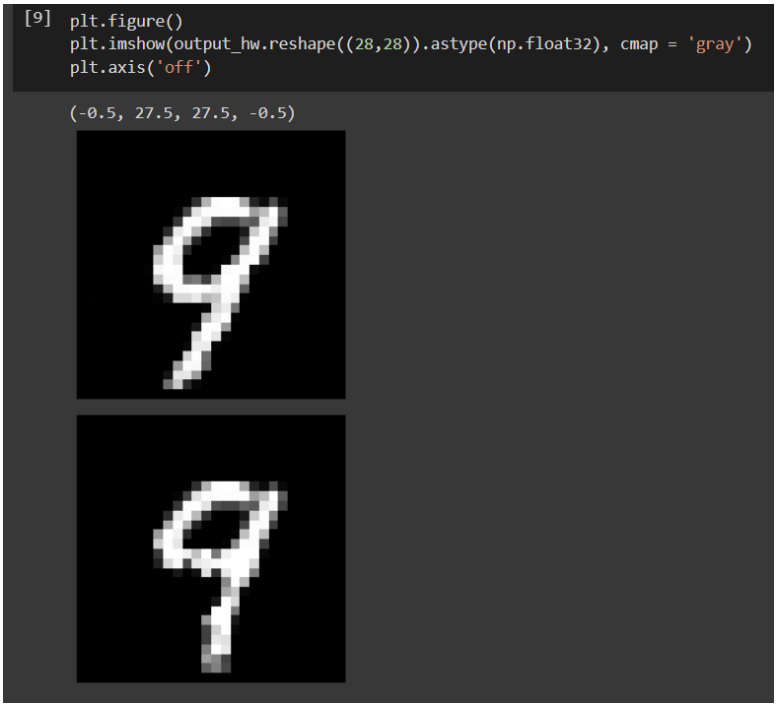
**10-bit accuracy**:

```
plt.figure()
plt.imshow(output_hw.reshape((28,28)).astype(np.float32), cmap = 'gray')
plt.axis('off')
```

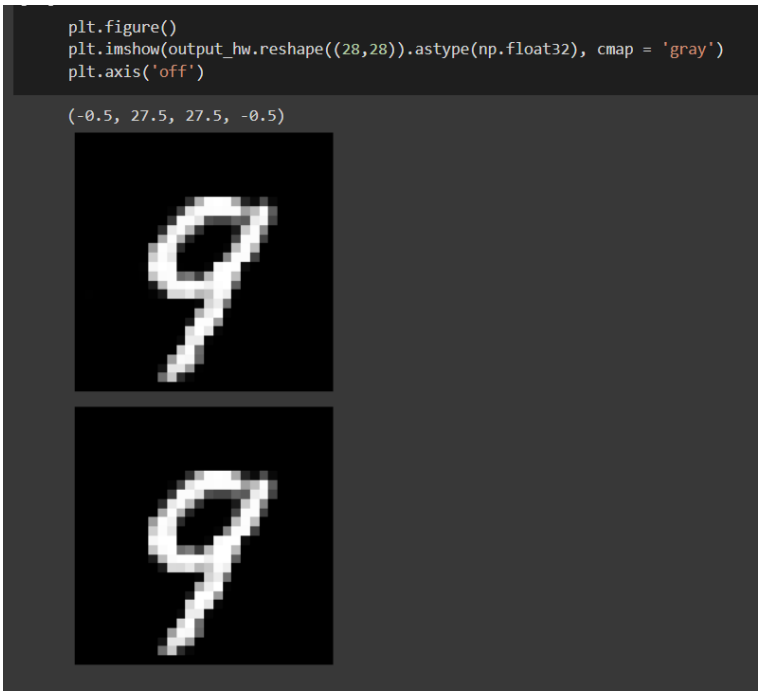
```
(-0.5, 27.5, 27.5, -0.5)
```

- ***idx = 12***

**4-bit accuracy**:




**10-bit accuracy**:




Μετά από εξέταση και σύγκριση των παραπάνω εικόνων παρατηρούμε ότι η μεταβολή της δεκαδικής ακρίβειας, έχει εμφανείς επιπτώσεις στην ποιότητα των αποτελεσμάτων του Hardware. Πιο συγκεκριμένα, βλέπουμε πως για μικρή δεκαδική ακρίβεια (4-bit), ταυτόχρονα επιτυγχάνεται μικρή ακρίβεια και στην ανακατασκευή των εικόνων, με χαρακτηριστικά, την

εμφάνιση κενών σημείων, αλλά και τον λανθασμένο προσανατολισμό τμημάτων των αριθμών. Αντίθετα, με τη χρήση μεγάλης δεκαδικής ακρίβειας (10-bit), η ποιότητα ανακατασκευής βελτιώνεται αισθητά, χωρίς να φαίνονται ιδιαίτερα σφάλματα στην απεικόνιση των εικόνων. Το γεγονός αυτό είναι αναμενόμενο καθώς η μεγαλύτερη δεκαδική ακρίβεια στα δεδομένα εισόδου οδηγεί σε μεγαλύτερη ακρίβεια των παραγόμενων αποτελεσμάτων.

**Γ)**

Στο τρίτο ερώτημα της άσκησης, συγκρίνουμε τις 3 παραπάνω περιπτώσεις δεκαδικής ακρίβειας (4-bit, 8-bit και 10-bit) με βάση τις μετρικές **Max Pixel Error** και **Peak Signal-to-Noise ratio**.

Για το ψηφίο με idx = 10 έχουμε:

- **4-bit accuracy**

```
[26] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  255
     Peak Signal-to-Noise Ratio:  14.051663945384881
```

- **8-bit accuracy**

```
[13] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  16
     Peak Signal-to-Noise Ratio:  42.6822168370888
```

- **10-bit accuracy**

```
[36] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  5
     Peak Signal-to-Noise Ratio:  54.08543347142643
```

Βλέπουμε πως η ακρίβεια 10-bit παρουσιάζει τις ευνοϊκότερες τιμές, καθώς έχει το μικρότερο Maximum Pixel Error και τον μεγαλύτερο λόγο Σήματος - Θορύβου. Αντίθετα, οι χειρότερες τιμές ανήκουν στην περίπτωση του 4-bit accuracy, τόσο στην μετρική του MPE όσο και σε αυτή του Peak Signal-to-Noise Ratio.

Για το ψηφίο με idx = 11:

- **4-bit accuracy**

```
[19] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  249
     Peak Signal-to-Noise Ratio:  14.634988266184102
```

- **8-bit accuracy**

```
print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

Max pixel error:  17
Peak Signal-to-Noise Ratio:  42.56993337396983
```

- **10-bit accuracy**

```
print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

Max pixel error:  5
Peak Signal-to-Noise Ratio:  52.556099880280584
```

Ακριβώς όπως και στην προηγούμενη περίπτωση, η ακρίβεια 10-bit πετυχαίνει το καλύτερο αποτέλεσμα με βάση τις επιλεγμένες μετρικές.

Για το ψηφίο με idx = 12:

- **8-bit accuracy**

```
[27] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  13
     Peak Signal-to-Noise Ratio:  47.065287020211215
```

- **4-bit accuracy**

```
[12] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  255
     Peak Signal-to-Noise Ratio:  13.525831164368576
```

- **10-bit accuracy**

```
[51] print ("Max pixel error: ", max_error(img_data_sw, img_data_hw))
     print ("Peak Signal-to-Noise Ratio: ", psnr(img_data_sw, img_data_hw))

     Max pixel error:  4
     Peak Signal-to-Noise Ratio:  53.76982650203158
```

Όπως και στις δύο προηγούμενες περιπτώσεις, έτσι και τώρα, η ακρίβεια 10-bit έχει, αναμενόμενα, τα καλύτερα αποτελέσματα, ενώ η ακρίβεια 4-bit τα χειρότερα.

Όσον αφορά την σύγκριση μεταξύ των δύο μετρικών που χρησιμοποιούνται, **προτιμότερη κρίνεται η μετρική Peak Signal-to-Noise Ratio.** Όπως παρατηρούμε εξετάζοντας τον κώδικα, η Max Pixel Error, μας υποδεικνύει ποια είναι η μέγιστη απόκλιση μεταξύ των pixel ανάμεσα στις περιπτώσεις hardware και software. Έτσι, το αποτέλεσμα που παίρνουμε αφορά ένα συγκεκριμένο pixel, στο οποίο εντοπίζεται η μέγιστη διαφορά ανάμεσα στις δύο εικόνες. Παρόλο που μπορούν να παραχθούν ορισμένα συμπεράσματα, η μετρική αυτή αδυνατεί να μας δώσει μια ολοκληρωμένη συνολική εικόνα για την ποιότητα της ανακατασκευής, καθώς δεν γνωρίζουμε την απόκλιση των τιμών για τα υπόλοιπα pixels. Για παράδειγμα, αν λάβουμε Max Pixel Error = 16 και χρησιμοποιώντας μόνο την συγκεκριμένη μετρική για την σύγκριση hardware και software, δεν μπορούμε να υποθέσουμε με σιγουριά ότι οι υπόλοιπες διαφορές τιμών των pixel να κυμαίνονται σε περιοχή αριθμών κοντά στο 16. Είναι εξίσου πιθανό, να κυμαίνονται και σε περιοχή αριθμών κοντά στο 0 γεγονός που θα φέρει σημαντική διαφορά στην τελική απεικόνιση. Με βάση τα παραπάνω, σε σύγκριση των 2 τεχνικών μέτρησης, απορρίπτουμε εκείνη του Max Error Pixel.

Σύμφωνα με ότι παρατηρήσαμε από τη διερεύνησή μας, όσο μεγαλύτερη είναι η δεκαδική ακρίβεια, τόσο καλύτερη και αξιόπιστη η ανακατασκευή των εικόνων. Τόσο στην περίπτωση των εικόνων που εμφανίζονται κατά την εκτέλεση του notebook όσο και κατα στην διαδικασία σύγκρισης των μετρικών Max Pixel Error και Peak Signal-to-Noise Ratio είναι εμφανές ότι για μεγαλύτερες τιμές δεκαδικής ακρίβειας επιτυγχάνονται και καλύτερα αποτελέσματα. Αξιοσημείωτο είναι πως αν και οπτικά δεν παρατηρείται σημαντική διαφορά ανάμεσα στις περιπτώσεις των 8 και 10 bit οι τιμές των παραπάνω καθιστούν την δεύτερη καλύτερη. Επομένως θεωρούμε πως η **ιδανική ακρίβεια bit είναι η 10-bit accuracy.**