

# **ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

**ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ  
ΥΠΟΛΟΓΙΣΤΩΝ**

**Σχεδιασμός Ενσωματωμένων Συστημάτων**

**4<sup>η</sup> Σειρά Ασκήσεων**

**Χειμερινό Εξάμηνο, Ακαδ. Έτος: 2023-2024**



**ΟΝΟΜΑΤΕΠΩΝΥΜΟ: ΔΙΑΜΑΝΤΙΔΗΣ ΘΕΟΧΑΡΗΣ**

**ΟΝΟΜΑΤΕΠΩΝΥΜΟ: ΙΩΑΝΝΟΥ ΚΩΝΣΤΑΝΤΙΝΟΣ**

**ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ: 03119002**

**ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ: 03119840**

**ΕΞΑΜΗΝΟ: 9**

## Άσκηση 1. Performance and resources measurement

Σκοπός της παρούσας άσκησης είναι να επιταχύνουμε τον αλγόριθμο που υλοποιεί το νευρωνικό ,που ουσιαστικά πρόκειται για πολλαπλά layers (for loops) που πραγματοποιούν «βαριά» υπολογιστικά πράξεις-πολλαπλασιασμούς. Αυτό θα το καταφέρουμε αρχικά θέτοντας αυτό το κομμάτι να υλοποιηθεί στο hardware μέσω του fpga και στην συνέχεια θα βελτιστοποιήσουμε αυτή την συνάρτηση αυτή μέσα με directives στην HLS.

- A) Αρχικά θέτουμε την συνάρτηση στο hardware και τρέχουμε estimation για να δούμε σε πόσους κύκλους εκτέλεσης θα ολοκληρωθεί η κλήση της συνάρτησης.

### 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

Παραπάνω βλέπουμε ότι έχουμε 683.780 κύκλους εκτέλεσης, αλλά χρησιμοποιούμε ελάχιστους πόρους ουσιαστικά μόνο BRAMS για να αποθηκεύσουμε τα δεδομένα μας. Μόνο από αυτό υποψιαζόμαστε ότι σίγουρά μπορούμε να μειώσουμε σημαντικά τους κύκλους εκτέλεσης στην συνέχεια με βελτιστοποιήσεις.

#### Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- read_input	1568	1568	4	-	-	392	no
- layer_1	36456	36456	93	-	-	392	no
+ layer_1.1	90	90	3	-	-	30	no
- layer_1_act	60	60	2	-	-	30	no
- layer_2	4600	4600	92	-	-	50	no
+ layer_2.1	90	90	3	-	-	30	no
- layer_3	61936	61936	158	-	-	392	no
+ layer_3.1	150	150	3	-	-	50	no

#### Utilization Estimates

##### Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	888
FIFO	-	-	-	-
Instance	-	0	268	677
Memory	33	-	66	21
Multiplexer	-	-	-	174
Register	-	-	558	-
Total	33	3	892	1760
Available	120	80	35200	17600
Utilization (%)	27	3	2	10

Από το παραπάνω report παρατηρούμε ότι και τα 3 layers έχουν μεγάλο latency αλλά κυρίως τα layers 1 και 3 οπότε σίγουρα πρέπει να τα αρχίσουμε την βελτιστοποίηση με αυτά τα for loops.

Β) Στην συνέχεια τρέχουμε τα αποτελέσματα μας στο zybo για να δούμε τα πραγματικά αποτελέσματα (κύκλους εκτέλεσης) καθώς το speedup συγκριτικά με την συνάρτηση να τρέχει στο software.

```
Hardware cycles : 682914
Software cycles : 1475732
Speed-Up       : 2.16093
Saving results to output.txt...
```

Παραπάνω βλέπουμε ότι στην πράξη έχουμε ελάχιστα λιγότερους κύκλους από το estimation και το speedup στον κώδικα που μας δόθηκε είναι 2. Συνεπώς από τώρα έχουμε πλεονεκτήματα με χρήση του fpga, ακόμη όμως θα μπορούσαμε να πούμε ότι το trade-off τείνει στο ότι έχουμε μικρή βελτίωση για μια τέτοια εφαρμογή. Αφού τρέξουμε στο fpga αποθηκεύουμε τα δεδομένα εξόδου output.txt για να τα χρησιμοποιήσουμε στην συνέχεια στην άσκηση 2.

Γ) Στο συγκεκριμένο ερώτημα θα κάνουμε design space exploration για να βελτιστοποιήσουμε την επίδοση του προγράμματος που τρέχει στο HW.

Συγκεκριμένα θα δοκιμάσουμε στοχευμένα HLS pragmas και διάφορες τιμές για αυτά (πχ διαφορετικά factors, II) και θα διαπιστώσουμε αρχικά με estimation κατά πόσο έχουμε μείωση στους κύκλους εκτέλεσης.

Η πρώτη σκέψη είναι να ενεργοποιήσουμε το pipeline (με II = 1) για όλα τα loops σε όλα τα layers τόσο στα εξωτερικά όσο και στα εσωτερικά. Αυτό έχει ως αποτέλεσμα το παρακάτω πίνακα :

#### Details

##### Performance estimates for 'forward\_propagation in main.cp ...

HW accelerated (Estimated cycles)	93801
-----------------------------------	-------

##### Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68,33
LUT	6016	17600	34,18
FF	6757	35200	19,2

Παρατηρούμε μείωση στους 93 χιλιάδες κύκλους εκτέλεσης περίπου το  $\sim 1/6$  από τον χρόνο που χρειαζόταν πριν το pipeline ,το οποίο οφείλεται στο γεγονός ότι εκτελούνται πλέον πολλές εντολές (instruction) ταυτόχρονα χωρίς να περιμένουν να ολοκληρωθεί πλήρως η προηγούμενη εντολή ( γλιτώνουμε τα stalls).Επιπλέον τα εσωτερικά for loops είναι ήδη full unroll καθώς η εντολή pipeline κάνει unroll τα από κάτω loops αυτόματα χωρίς loop unroll.Φυσικά οι πόροι αυξάνονται σημαντικά καθώς χρησιμοποιούμε όσο το περισσότερα DSP είναι δυνατόν καθώς έχουμε βαριές πράξεις ( προσθέσεις , πολλαπλασιασμούς) καθώς και Luts για να υλοποιήσουμε την λογική μας αλλά και τα BRAMS αυξάνονται αφού πλέον με το pipeline αποθηκεύουμε περισσότερα δεδομένα στους ίδιους κύκλους.

Στην συνέχεια δοκιμάσουμε να υλοποιήσουμε loop unrolling με factors 8 στα εσωτερικά loops που δεν γίνονται unroll από το pipeline.Τα αποτελέσματα φαίνονται στην συνέχεια:

#### Details

Performance estimates for 'forward_propagation in main.cp ...	
HW accelerated (Estimated cycles)	23913

Resource utilization estimates for HW functions			
Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	297	60	495
LUT	41662	17600	236,72
FF	16886	35200	47,97

Βλέπουμε ότι όντως έχουμε μείωση στους κύκλους εκτέλεσης αλλά πλέον χρησιμοποιούμε υπερβολικά πολλούς πόρους σε σημείο που δεν αρκεί το συγκεκριμένο frga zybo. Οπότε το unroll είναι σχεδόν απαγορευτικό σε αυτό το σημείο.

Τελικά καταλήγουμε με το να έχουμε καλύτερη επίδοση με το να αφήσουμε pipeline στα εξωτερικά loops και array partitions όπως φαίνονται στον παρακάτω πίνακα estimation.

#### Details

Performance estimates for 'forward_propagation in main.cp ...	
HW accelerated (Estimated cyc	12031

Resource utilization estimates for HW functions			
Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	40	60	66.67
LUT	6354	17600	36.1
FF	10330	35200	29.35

Οι κύκλοι εκτέλεσης μειώνονται εξαιρετικά πολύ στους μόλις 12 χιλιάδες ενώ εξακολουθούμε να χρησιμοποιούμε το 100% των DSPs και σημαντικό κομμάτι των BRAMS.

```
void forward_propagation(float *x, float *y)
{
    // #pragma HLS dataflow
    quantized_type xbuf[N1];
    l_quantized_type layer_1_out[M1];
    l_quantized_type layer_2_out[M2];

    #pragma HLS array_partition variable=xbuf complete
    #pragma HLS array_partition variable=layer_1_out complete //cyclic factor=16//complete
    #pragma HLS array_partition variable=layer_2_out complete //cyclic factor=32//complete

    // #pragma HLS array_reshape variable=xbuf complete
    // #pragma HLS array_reshape variable=layer_1_out complete
    // #pragma HLS array_reshape variable=layer_2_out complete

    // limit resources to max DSP number of Zybo - do not change
    #pragma HLS ALLOCATION instances=mul limit=80 operation
    // #pragma HLS ALLOCATION instances=layer_3 limit=20

    read_input:
    for (int i=0; i<N1; i++)
    {
        #pragma HLS pipeline II=1
        xbuf[i] = x[i];
    }

    // Layer 1
    layer_1:
    for(int i=0; i<N1; i++)
    {
        #pragma HLS pipeline II=1
        for(int j=0; j<M1; j++)
        {
            // #pragma HLS pipeline II=1

            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 pipeline II=1
        layer_1_out[i] = ReLU(layer_1_out[i]);
    }

    // Layer 2
    layer_2:
    for(int i=0; i<M2; i++)
    {
#pragma HLS pipeline II=1
        l_quantized_type result = 0;
        for(int j=0; j<N2; j++)
        {
//#pragma HLS pipeline II=1
            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 II=1

        l_quantized_type result = 0;
        for(int j=0; j<N3; j++)
        {
//#pragma HLS unroll factor=8
//#pragma HLS pipeline II=1
            l_quantized_type term = layer_2_out[j] * W3[j][i];
            result += term;
        }
        y[i] = tanh(result).to_float();
    }
}

```

Ουσιαστικά έχουμε pipeline σε όλα τα εξωτερικά loops και complete array partition σε όλους τους βασικούς μονοδιάστους πίνακες που χρησιμοποιούμε. Αξίζει να σημειώσουμε ότι τα εσωτερικά loop unrolls δεν βελτιώνουν τους χρόνους εκτέλεσης λόγω του ανώτερο pipeline.

Υλοποίηση	Κύκλοι Εκτέλεσης	Πόροι (Συνολικά 400)
No optimized	683780	40
Only pipeline	93801	220
Pipeline-loop unroll	23913	878
Pipeline-complete arr partition	12031	232
Pipeline- arr partition W	12044	395

Δοκιμάσαμε να κάνουμε και μερικό array partition στους πίνακες w1,w2,w3 αλλά όπως είδαμε καταφέραμε απλώς να αυξήσουμε την μνήμη που χρησιμοποιούμε χωρίς να καταφέρουμε κάποια βελτίωση στους κύκλους εκτέλεσης. Αυτό μπορεί να εξηγηθεί καθώς πλέον το κομμάτι που κρατάει το σημαντικό κομμάτι των κύκλων εκτέλεσης είναι οι πράξεις και όχι οι προσβάσεις στην μνήμη. (Συνεπώς το array partition των w1,w2,w3 είναι ζημιογόνο και “άχρηστο” στη συγκεκριμένη υλοποίηση)

## Details

### Performance estimates for 'forward\_propagation in main.cp ...

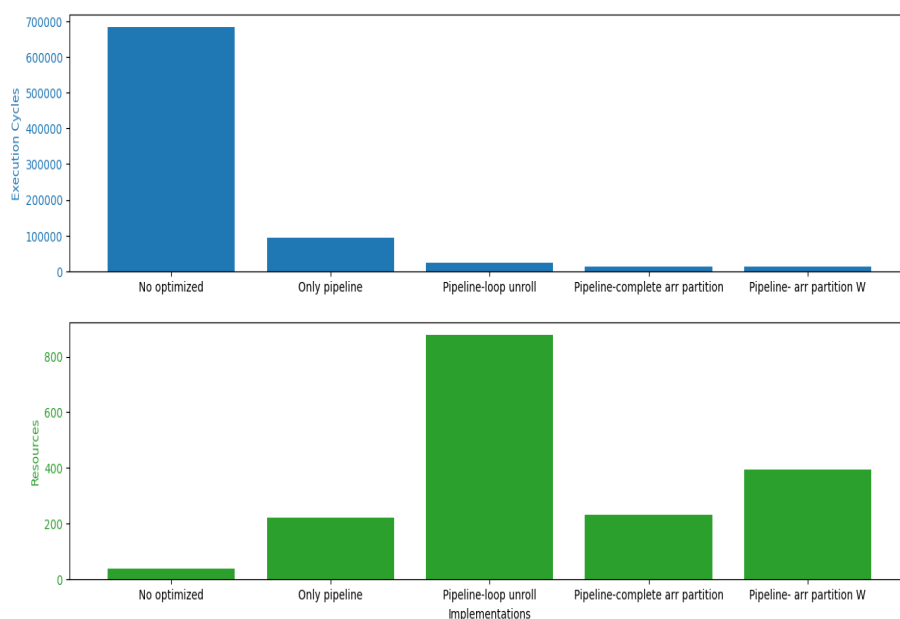
HW accelerated (Estimated cycles)	12044
-----------------------------------	-------

### Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	112	60	186,67
LUT	12293	17600	69,85
FF	14026	35200	39,85

Σημειώνουμε ότι δοκιμάσαμε επιπλέον HLS Dataflow ώστε να γίνονται σε ταυτόχρονη ροή τα επιμέρους layers(loops) αλλά έχουμε εξαρτήσεις και επιπλέον δεν αρκούν και οι πόροι του zybo.

Execution Cycles and Resources for Different Implementations



Συνεπώς μένουμε από την αναζήτηση που κάναμε ότι έχουμε ως βέλτιστους χρόνους εκτέλεσης **12 χιλιάδες κύκλους** εκτέλεσης με χρήση pipeline και array partition στους πίνακες layer\_1,layer\_2,xbuffer.

Δ)Στην τελική optimized υλοποίηση θα αναλύσουμε το HLS report.

Όπως φαίνεται παρακάτω στον πίνακα μεγαλύτερο latency έχουν τα loops που υλοποιούν τα **read\_input,layer\_1 και layer\_3**.Στο read\_input αποθηκεύουμε τα δεδομένα που έστειλε το SW σε μια BLOCK ram στο HW ,στα layer\_1 και layer\_3 κάνουμε επεξεργασία των δεδομένων. Επίσης παρατηρούμε ότι όλα τα loops είναι πλήρως pipelined συνεπώς η η σχεδίαση έχει πραγματοποιηθεί επιτυχώς.

**Current Module : forward propagation**

#### Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- read_input	394	394	4	1	1	392	yes
- layer_1	393	393	3	1	1	392	yes
- layer_1_act	30	30	1	1	1	30	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Στην συνέχεια θα εξετάσουμε τα είδη των μαθηματικών εκφράσεων καθώς και τα DSP που καταναλώνει η κάθε έκφρασή.

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
forward_propagation	81	80	10330	6272						
> I/O Ports(2)					64					
> Instances(4)	0	0	268	983						
> Memories(111)	81		273	232	1006			111	33884	306674
> Expressions(414)	0	80	0	3567	4401	4323	692			
> -	0	0	0	78	16	78	0			
> *	0	80	0	0	1112	999	0			
> +	0	0	0	2429	2916	2900	0			
> and	0	0	0	4	4	4	0			
> ashr	0	0	0	161	54	54	0			
> icmp	0	0	0	77	205	71	0			
> or	0	0	0	6	6	6	0			
> select	0	0	0	703	43	162	692			
> shl	0	0	0	88	32	32	0			
> xor	0	0	0	21	13	17	0			
> Registers(851)			9789		10359					
> Channels(0)	0		0	0	0		0		0	0
> Multiplexers(104)	0		0	1490	1485		0			

Αρχικά βλέπουμε ξεκάθαρα ότι από όλες τις λειτουργίες που πραγματοποιεί η συνάρτηση στο HW τους περισσότερους πόρους όσον αφορά τα LUTs τους καταναλώνουν οι μαθηματικές πράξεις και προφανώς όλα τα DSPs.Ενώ η μνήμη χρησιμοποιεί τα Brams και τα περισσότερα FF χρησιμοποιούνται από τους Registers.Όπως φαίνεται από το report οι **πολλαπλασιασμοί χρειάζονται τα περισσότερα DSPs** , ενώ η πρόσθεση και αφαίρεση γίνεται μέσω LUTs



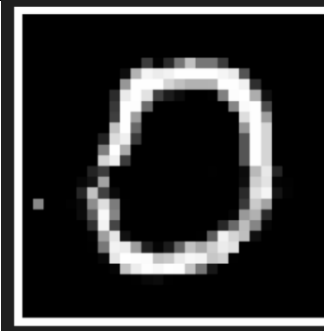
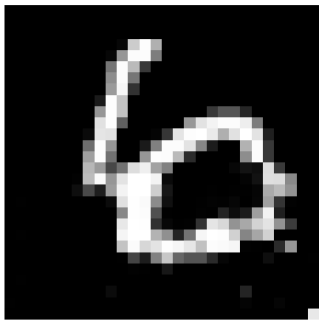
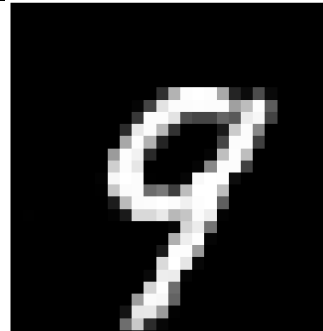
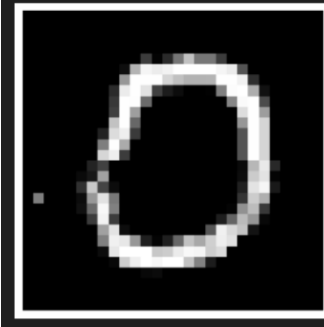
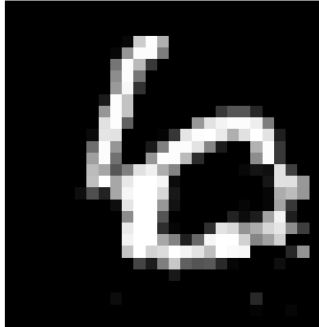
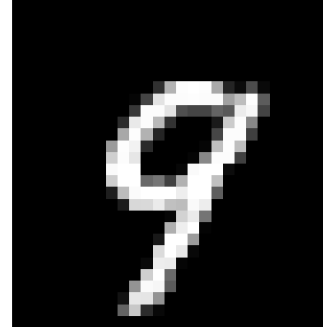
## Άσκηση 2

A) Σε αυτήν την άσκηση καλούμαστε να μετρήσουμε την ποιότητα ανακατασκευής των εικόνων κάνοντας συνδυασμό των μισών εικόνων που μας δίνονται ως είσοδο στο data.txt και των μισών εικόνων που παίρνουμε σαν αποτέλεσμα στο output.txt

A) Αρχικά θα επιχειρήσουμε να ανακατασκευάσουμε τις εικόνες με τρία διαφορετικά indexes (10,11,12) τα οποία αντιστοιχούν στους αριθμούς (0,6,9). Για αυτήν την ανακατασκευή θα χρησιμοποιηθεί ο αρχικός κώδικας χωρίς τις βελτιστοποιήσεις καθώς δεν μας ενδιαφέρει η ταχύτητα αλλά η ποιότητα των εικόνων.

Σε αυτό το ερώτημα θα χρησιμοποιήσουμε 8 bits για την αναπαράσταση της δεκαδικής ακρίβειας και άρα  $BITS\_EXP=2^{10}$ . Δηλαδή αναμένουμε να έχουμε ικανοποιητική ακρίβεια και να μην αντιλαμβανόμαστε την διαφορά με το ανθρώπινο μάτι ανάμεσα στην SW και στην HW υλοποίηση. Τα αποτελέσματα φαίνονται συγκεντρωτικά στον παρακάτω πίνακα

Δεκαδική ακρίβεια τιμών  $\tanh=8$  bits

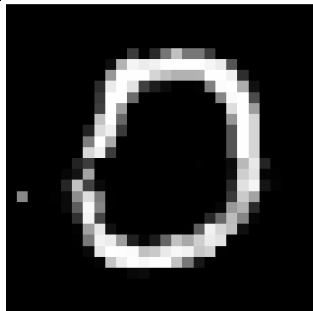
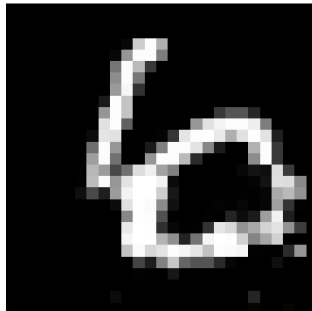
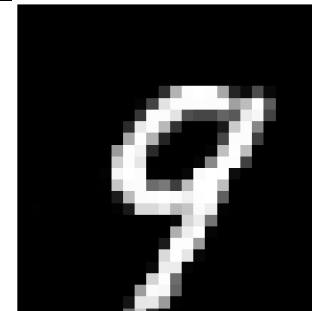
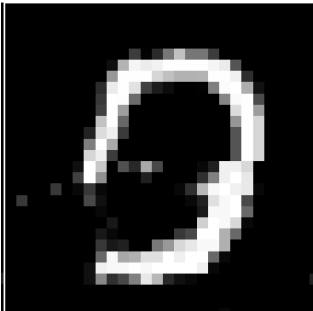

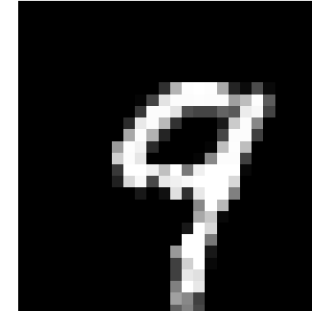
	Idx=10	Idx=11	Idx=12
SW υλοποίηση			
HW υλοποίηση			

Όπως παρατηρούμε από τα παραπάνω αποτελέσματα έχουμε ικανοποιητική ακρίβεια και άρα δεν γίνεται εύκολα αντιληπτή η διαφορά ανάμεσα στο HW και στο SW. Αυτό σημαίνει ότι τα 8 bits αρκούν για να αναπαραστήσουν με ικανοποιητική ακρίβεια την υπόλοιπη μισή εικόνα.

B) Σε αυτό το ερώτημα καλούμαστε να αλλάξουμε τον αριθμό των bits που χρησιμοποιούνται για την αναπαράσταση του δεκαδικού μέρους tanh. Οι τιμές των bits που θα δοκιμάσουμε είναι 4,6,10 bits. Σε όλες αυτές τις περιπτώσεις θα χρησιμοποιήσουμε τον αρχικό κώδικα και όχι τον βελτιστοποιημένο για τους λόγους ότι δεν μας ενδιαφέρει η ταχύτητα αλλά μόνο το τελικό αποτέλεσμα και επίσης όταν χρησιμοποιούμε μεγάλο πλήθος bits (πχ 10) δεν επαρκούν οι πόροι του FPGA.

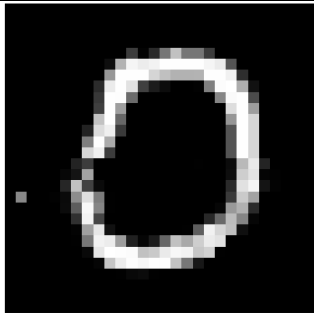
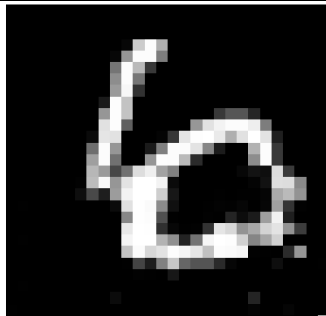
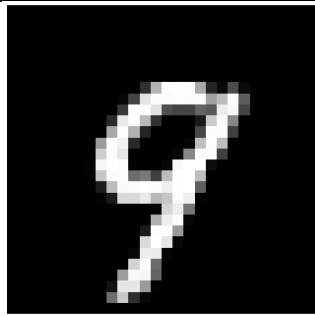
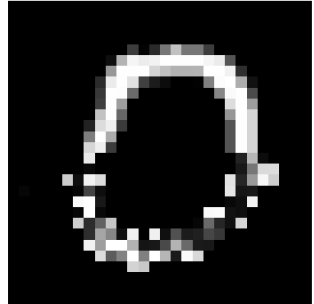
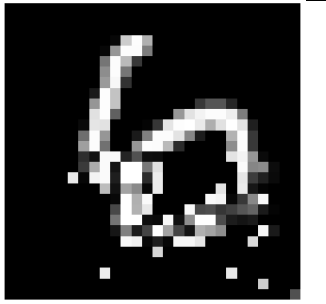
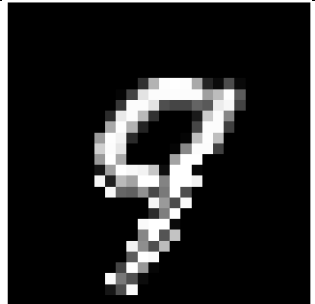
Παρουσιάζουμε τα παρακάτω αποτελέσματα συγκριτικά σε μορφή πίνακα.

Δεκαδική ακρίβεια τιμών tanh=4 bits

	Idx=10	Idx=11	Idx=12
SW υλοποίηση			
HW υλοποίηση			

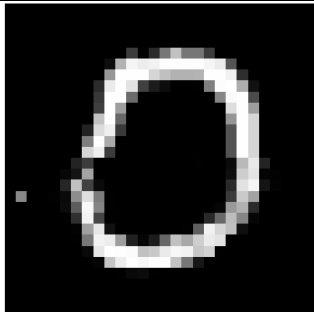
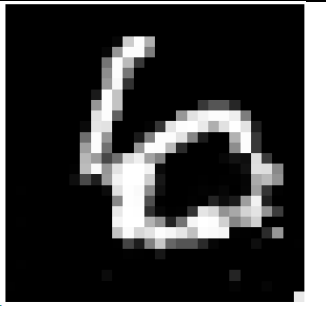
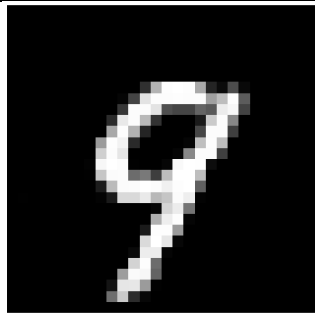
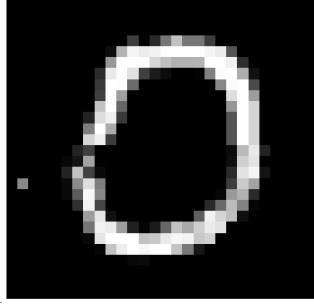
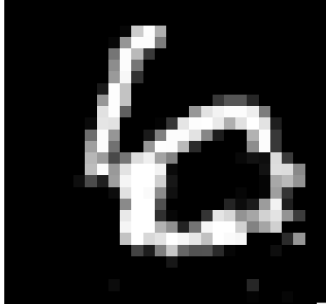
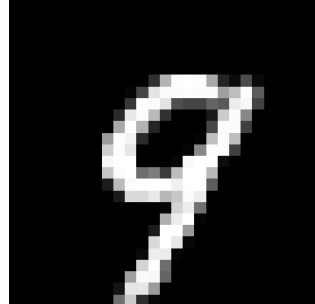
Παρατηρούμε λοιπόν ότι όταν χρησιμοποιούμε λιγότερα bits από 8 γίνεται undersampling και επομένως αυξάνεται το σφάλμα με το οποίο συμπληρώνεται το υπόλοιπο μισό από τα νούμερα. Μεγάλες διαφορές παρατηρούμε και με το μάτι στον αριθμό 0 και στον αριθμό 6.

Δεκαδική ακρίβεια τιμών tanh=6 bits

	Idx=10	Idx=11	Idx=12
SW υλοποίηση			
HW υλοποίηση			

Σε αυτήν την περίπτωση καθώς αυξάνουμε τα bits βλέπουμε ότι πάλι δεν πετυχαίνουμε την ακρίβεια που είχαμ αρχικά με τα 8 bits καθώς υπάρχουν πάλι σφάλματα τα οποία φαίνονται, δηλαδή γίνεται πάλι undersampling, αλλά τα αποτελέσματα έχουν βελτιωθεί σε σχέση με τα 4 bits που χρησιμοποιήσαμε στην προηγούμενη περίπτωση.

Δεκαδική ακρίβεια τιμών tanh=10 bits

	Idx=10	Idx=11	Idx=12
SW υλοποίηση			
HW υλοποίηση			

Όπως βλέπουμε έχουμε πετύχει σχεδόν επακριβώς τους αριθμούς και μάλιστα με μικρές διαφορές από όταν χρησιμοποιήσαμε 8 bits δηλαδή έχει γίνει oversampling.

Γ) Σε αυτό το ζητούμενο θα εξετάσουμε τα παραπάνω αποτελέσματα και θα κάνουμε bar plot τόσο για το pixel error σε όλες τις περιπτώσεις όσο και για το psnr.

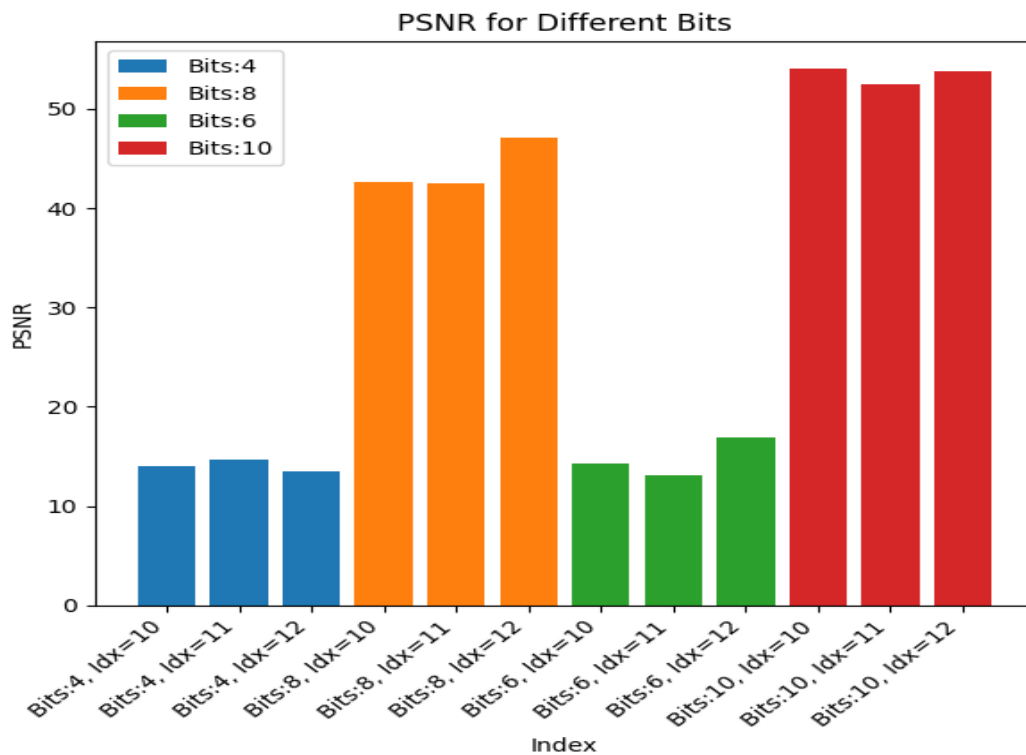
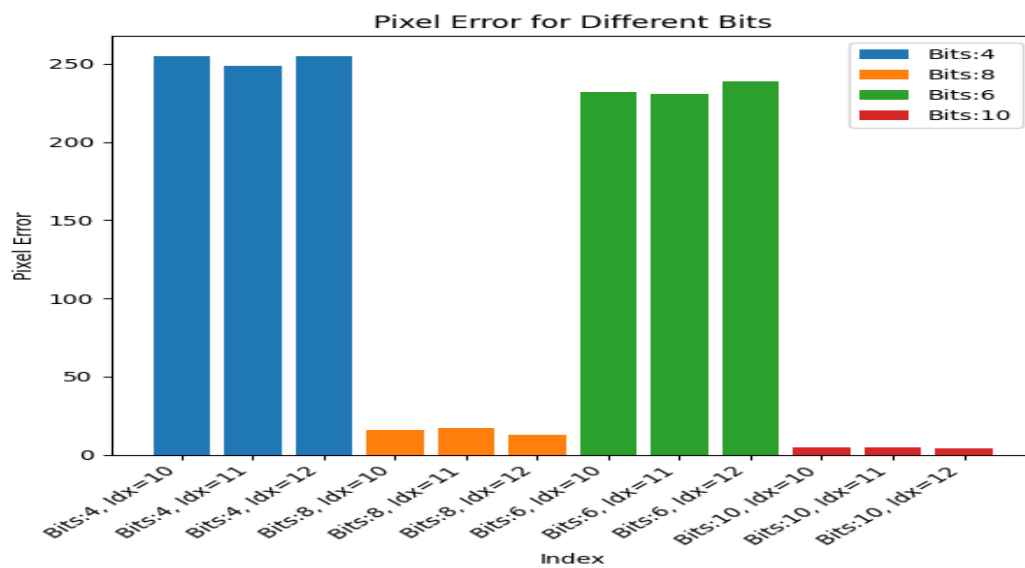
Στην συνέχεια έχουμε τις τιμές για όλες τις περιπτώσεις:

<b>bits: 4</b>	Idx =10	Idx=11	Idx=12
MAX Pixel Err	255	249	255
psnr	14.051	14.63	13.52

<b>bits:8</b>	Idx =10	Idx=11	Idx=12
MAX Pixel Err	16	17	13
psnr	42.68	42.56	47.06

<b>bits:6</b>	Idx =10	Idx=11	Idx=12
MAX Pixel Err	232	231	239
psnr	14.32	13.16	16.88

<b>bits:10</b>	Idx =10	Idx=11	Idx=12
MAX Pixel Err	5	5	4
Psnr	54.08	52.55	53.76



### Παρατηρήσεις:

Αρχικά φαίνεται ότι όσο αυξάνουμε το πλήθος των bits που χρησιμοποιούμε τόσο μειώνεται το pixel error και αντίστοιχα αυξάνει το psnr. Επιπλέον παρατηρούμε ότι για τα διαφορά bits (Dtype) έχουμε καλύτερα αποτελέσματα για διαφορετικά idx. Προφανώς έχουμε το trade-off ότι όσο καλύτερα αποτελέσματα έχουμε τόσο περισσότερους πόρους χρησιμοποιούμε διότι αυξάνουν τα bits που χρησιμοποιούμε.