ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός Ενσωματωμένων Συστημάτων
9ο Εξαμηνο ΗΜΜΥ

Συνεργάτες: Γεώργιος-Ταξιάρχης Γιαννιός, Α.Μ.: 03116156
Μπέτζελος Χρήστος, Α.Μ.: 03116067

4η Εργαστηριακή Άσκηση:
ΕΡΓΑΣΙΑ ΓΙΑ HIGH LEVEL SYNTHESIS ΣΕ FPGA

Ο **σκοπός της άσκησης** είναι να γίνει μελέτη γύρω από τον προγραμματισμό FPGA με High Level Synthesis (HLS). Πρόκειται για optimization και επιτάχυνση C κώδικα για να τρέξει τελικώς στο hardware του Xilinx Zybo FPGA. Η εφαρμογή που θα μελετηθεί για επιτάχυνση θα σχετίζεται με νευρωνικά δίκτυα και ειδικότερα τα Generative Adversarial Networks (GANs). Συγκεκριμένα, η εφαρμογή του GAN που δίνεται έτοιμη αφορά την ανακατασκευή ημιτελών εικόνων από χειρόγραφα ψηφία. Ο τελικός στόχος είναι να επιταχυνθεί ο αλγόριθμος αυτός σε σχέση με την SW υλοποίηση αλλά και να μετρηθεί η ποιότητα ανακατασκευής των εικόνων.

Η **εφαρμογή** αφορά την ανακατασκευή εικόνων από το MNIST dataset (28x28 χειρόγραφοι grayscale αριθμοί). Συγκεκριμένα, έχουμε το generator νευρωνικό μοντέλο από το GAN όπου μας δίνεται η υλοποίηση του σε C μαζί με τα trained βάρη. Η εφαρμογή δέχεται ως input το πάνω μισό των αριθμών και καλεί το νευρωνικό για να κάνει generate/predict το υπόλοιπο μισό της εικόνας. Μετά την εκτέλεση θα φανεί ο μέσος χρόνος εκτέλεσης ανά εικόνα σε κύκλους για SW και HW και το speed-up. Σκοπός είναι να μεγιστοποιηθεί το speed-up με HLS optimizations στην HW function. Το output.txt που παράγει η εφαρμογή το επεξεργαζόμαστε έπειτα μέσω Jupyter για να εμφανίσουμε τους αριθμούς.

Τα **Source files** της Εφαρμογής είναι τα παρακάτω:

- main.cpp: Τρέχει σε ARM CPU. Καλεί το μοντέλο του Generator σε SW and HW και μετράει την επίδοση.
- network.cpp: Είναι ο core κώδικας της εφαρμογής δηλαδή του Generator. Αυτός θα πρέπει να γίνει optimized με HLS για να τρέξει στο HW (μόνο η συνάρτηση *forward_propagation* θελει optimizations).
- weight_definitions.h: Περιέχει τα trained βάρη του νευρωνικού του generator.
- tanh.h: Αποθηκεύει τις pre-computed τιμές της μαθηματικής συνάρτησης Tanh για το HW.
- network.h: Περιέχει διάφορα ορίσματα του Generator (datatypes, loop limits, etc.).
- data.txt: Περιέχει το input dataset (το πάνω μισό των εικόνων). Απαιτείται να βρίσκεται στο φάκελο του εκτελέσιμου όταν τρέχει στον ARM αλλά και στο plot_output.ipynb.

Μέσω του εργαλείου **SDSoC** αναπτύσσουμε την εφαρμογή για το embedded Zybo FPGA. Συγκεκριμένα το SDSoC παρέχει ένα Eclipse τύπου περιβάλλον όπου γράφουμε C/C++ κώδικα και αναπτύσσουμε έναν accelerator που τρέχει σε FPGA με χρήση HLS optimizations. Επίσης θα μπορούμε να κάνουμε performance και resource estimation, να φτιάξουμε το bitstream του υλικού αλλά και τελικώς την sd boot card όπου θα “bootάρει” το σύστημα (τα παραγόμενα αρχεία μεσα στην sd_card περιλαμβάνουν το εκτελέσιμο (.elf), το bitstream και το λειτουργικό petalinux του συστήματος).

Αφού φτιάξουμε ένα **Xilinx SDx project** σύμφωνα με το πάνω configuration (zybo, linux, debug mode, 100Mhz) και εισάγουμε τα source files που μας δίνονται, μαρκάρουμε την συνάρτηση που θέλουμε να υλοποιηθεί στο HW. Στη συγκεκριμένη περίπτωση είναι η *forward_propagation* η οποία στο τέλος θα έχει όλα τα HLS optimizations που θα πρέπει να προσθέσουμε. Τέλος, κάνουμε build το project κάθε φορά για να υλοποιηθεί αυτό που επιθυμούμε.

Άσκηση 1. Performance and resources measurement

Η συνάρτηση **forward_propagation** είναι ο κώδικας C που τρέχει το νευρωνικό, συγκεκριμένα ο generator. Είναι διάφορα layers που τρέχουν το ένα μετά το άλλο (dense->relu->dense->relu->dense->tanh) με σκοπό να παράξουν το κάτω μισό της εικόνας. Ουσιαστικά πρόκειται για πολ/σμούς matrix-vector το οποίο είναι μια διαδικασία με πολλές πράξεις, αργή για CPU.

Α) Αρχικά δοκιμάσαμε να την θέσουμε ως HW function και κάναμε estimate performance χωρίς να πραγματοποιηθεί κάποιο optimization. Καταγράψαμε το report που δείχνει τα estimated resources και cycles της hardware function.

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

Β) Δοκιμάσαμε στη συνέχεια να φτιάξουμε την sd_card με το bitstream και να τα περάσουμε στην sd του zybo. (Αγνοήσαμε πιθανά warnings κατά το compilation (unused labels, κτλ)). Αφού περάσαμε και το data.txt, κάναμε reboot και τρέξαμε την εφαρμογή στο board. Παρατηρήσαμε, όπως φαίνεται και από το παρακάτω screenshot, ότι συμφωνεί με το estimation.Το speed-up σε σχέση με την SW εκτέλεση στον ARM είναι 2.16241.

```
root@Avnet-Digilent-ZedBoard-2016_3:/mnt# ./embedded.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 683079
Software cycles : 1477097
Speed-Up        : 2.16241
Saving results to output.txt...
```

Γ) Σε αυτό το σημείο πραγματοποιήθηκε design space exploration ώστε να βρεθούν τα optimizations και να επιταχυνθεί σημαντικά ο αλγόριθμος. Δοκιμάσαμε διάφορα HLS pragmas και είδαμε τι βγάζει το estimation.

Pipeline:
Το Loop pipelining είναι η πιο σημαντική βελτιστοποίηση στο HLS. Επιτρέπει μια νέα επανάληψη να ξεκινήσει την επεξεργασία πριν ολοκληρωθεί η προηγούμενη επανάληψη. Στο παράδειγμά μας, παρατηρήσαμε ότι δεν έχουμε εσωτερικά dependencies, οπότε ορίσαμε pipelining σε όλους τους βρόγχους.

```
read_input:
for (int i=0; i<N1; i++)
{
#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++)
     { ...
```

```
layer_1_act:
for(int i=0; i<M1; i++)
{
#pragma HLS pipeline
       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++)
       { ...
```

```
// 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++)
        { ...
```

Unroll Factor:

Προσθέσαμε το #pragma unroll στους εσωτερικούς διπλούς βρόχους ώστε να επιτευχθεί ταυτόχρονος υπολογισμός των επιμέρους γινομένων που βρίσκονται στην ίδια γραμμή (*Layer1-W1*) ή στήλη (*Layer2-W2, Layer3-W3*). Στην εντολή Unroll, επιλέχθηκε παράγοντας ίσος με 30 στο layer 1, 2 και 50 στο layer 3. Οι τιμές αυτές επιλέχθηκαν σύμφωνα με τα συνολικά iterations των εσωτερικών βρόγχων.

```
// Layer 1
layer_1:
for(int i=0; i<N1; i++)
{
#pragma HLS pipeline
        for(int j=0; j<M1; j++)
        {
#pragma HLS unroll factor=30
                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 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 factor=30
                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 factor=50
                l_quantized_type term = layer_2_out[j] * W3[j][i];
                result += term;
        }
        y[i] = tanh(result).to_float();
}
```

Details

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

HW accelerated (Estimated cycles)	86183

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68.33
LUT	6037	17600	34.3
FF	6754	35200	19.19

Array Partition:

Επειδή η μνήμη έχει περιορισμένο αριθμό read ports και write ports, χωρίσαμε τον αρχικό πίνακα σε μικρότερους. Έτσι, αυξάνεται αποτελεσματικά ο αριθμός load/store ports και άρα βελτιώνεται και το memory bandwidth. Πειραματιστήκαμε με διάφορες τιμές για τα types και το factor στα pragmas array_partition και καταλήξαμε στις παρακάτω. Οι τελικές τιμές που προκύπτουν είναι στην ουσία διαιρέτες του αρχικού μεγέθους του πίνακα. Για τους δισδιάστατους, επιλέξαμε να κάνουμε partition την διάσταση που διατρέχεται από το εσωτερικό διπλό loop.

```
#pragma HLS array_partition variable=layer_1_out block factor=15 dim 1
#pragma HLS array_partition variable=layer_2_out block factor=25 dim 1
#pragma HLS array_partition variable=W1 block factor=15 dim 2
#pragma HLS array_partition variable=W2 block factor=15 dim 1
#pragma HLS array_partition variable=W3 block factor=25 dim 1
```

Controlling resources:
Για τον περιορισμό του πλήθους των συναρτήσεων που θα υλοποιήσει στο hardware ο compiler, γίνεται χρήση της εντολής pragma HLS allocation. Παρατηρούμε ότι στο πρώτο επίπεδο έχουμε 30 νευρώνες. Σε κάθε νευρώνα καλείται η συνάρτηση ReLU, οπότε συνολικά καλείται 30 φορές. Επίσης, η συνάρτηση Tanh καλείται 1 φορά στο τελευταίο επίπεδο. Βάσει αυτών οριζουμε τα παρακάτω όρια:

```
#pragma HLS allocation instances=tanh limit=1 function
#pragma HLS allocation instances=ReLU limit=30 function
```

Details

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

HW accelerated (Estimated cycles)	12038

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68.33
LUT	7246	17600	41.17
FF	6524	35200	18.53

Παράξαμε την sd με το bitstream και τρέξαμε την εφαρμογή στο board. Καταγράψαμε ξανά κύκλους και speed-up από το board. Παρατηρήσαμε πάλι, όπως φαίνεται και από το παρακάτω screenshot, ότι συμφωνεί με το estimation.

```
root@Avnet-Digilent-ZedBoard-2016_3:/mnt# ./embedded.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 12256
Software cycles : 1478691
Speed-Up        : 120.65
Saving results to output.txt...
```

Συγκρίνοντας αυτή την υλοποίηση (optimized), με την αρχική (unoptimized), παρατηρούμε ότι το **Speed-Up** από 2.16 έχει φτάσει το **120.65**, δηλαδή πάνω από 55 φορές ταχύτερο συγκριτικά με το αρχικό.

Δ) Επίσης είδαμε για την optimized υλοποίηση μας στο SDSoC το HLS report που παράγεται. Καταγράψαμε τα latency details για κάθε loop (Latency -> Detail -> Loop).

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	395	395	5	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

Παρατηρούμε ότι το layer με το πιο μεγάλο latency είναι το layer_3. Επίσης το design μας είναι fully pipelined.

Αφού κατευθυνθήκαμε στην καρτέλα Resource profile του HLS report, καταγράψαμε τα είδη των μαθηματικών εκφράσεων (expressions) του design.

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Ban
forward_propagation	**82**	**80**	**6524**	**7174**						
I/O Ports(2)					**64**					
Instances(5)	**1**	**0**	**272**	**891**						
Memories(111)	**81**		**273**	**232**	**1007**			**111**	**33252**	**301378**
Expressions(459)	**0**	**80**	**0**	**4549**	**3820**	**5290**	**2001**			
-	**0**	**0**	**0**	**78**	**16**	**78**	**0**			
*	**0**	**80**	**0**	**0**	**965**	**1132**	**0**			
+	**0**	**0**	**0**	**2125**	**2439**	**2417**	**0**			
and	**0**	**0**	**0**	**5**	**5**	**5**	**0**			
ashr	**0**	**0**	**0**	**161**	**54**	**54**	**0**			
icmp	**0**	**0**	**0**	**68**	**180**	**59**	**0**			
or	**0**	**0**	**0**	**7**	**7**	**7**	**0**			
select	**0**	**0**	**0**	**2011**	**119**	**1500**	**2001**			
shl	**0**	**0**	**0**	**88**	**32**	**32**	**0**			
xor	**0**	**0**	**0**	**6**	**3**	**6**	**0**			
Registers(439)			**5979**		**5979**					
Channels(0)	**0**		**0**	**0**	**0**			**0**	**0**	**0**
Multiplexers(105)	**0**		**0**	**1502**	**1497**			**0**		

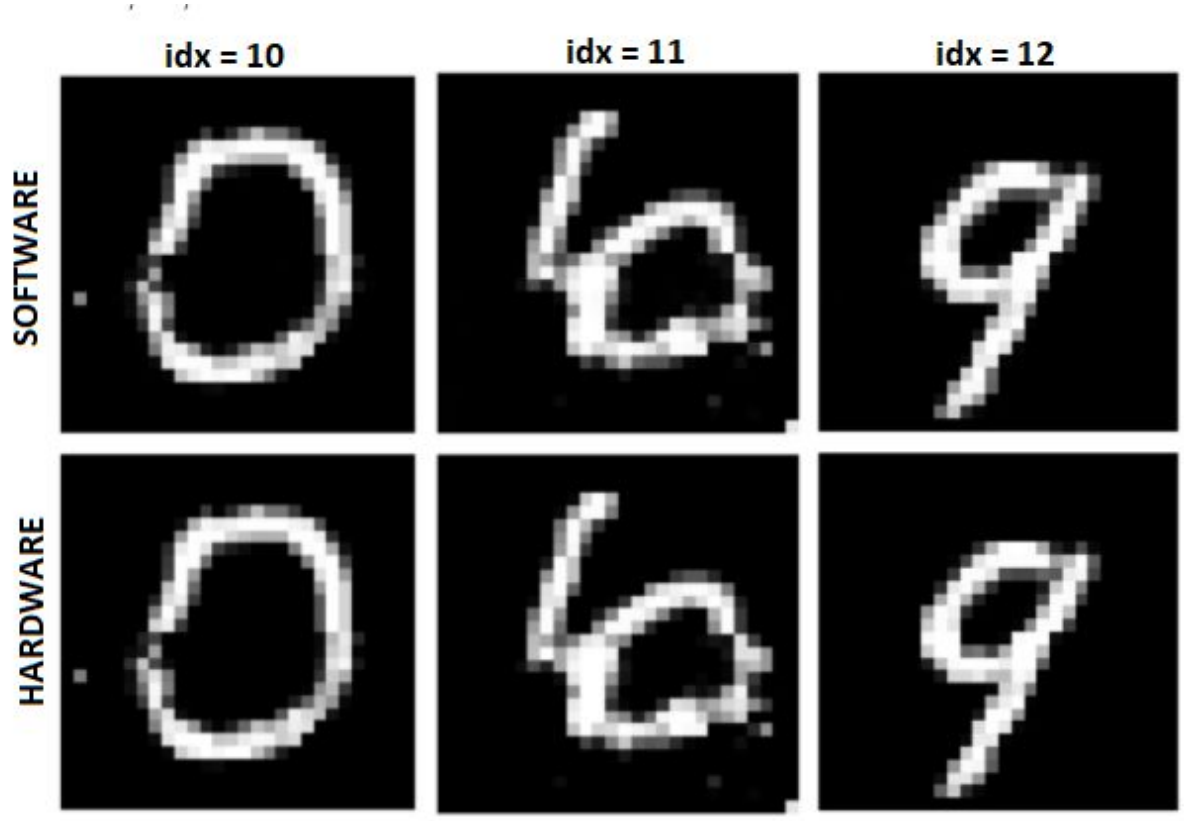
Η έκφραση που χρειάζεται τα πιο πολλά DSP είναι ο πολλαπλασιασμός.

Άσκηση 2. Quality measurement

Σε αυτό το σημείο κληθήκαμε να μετρήσουμε τη ποιότητα ανακατασκευής των εικόνων μέσω του jupyter notebook που μας δόθηκε. Μέσω αυτού κάναμε combine τις μισές εικόνες που δόθηκαν σαν input στο data.txt με τις μισές εικόνες από το output.txt που παρήγαγε το SW αλλά και το HW.

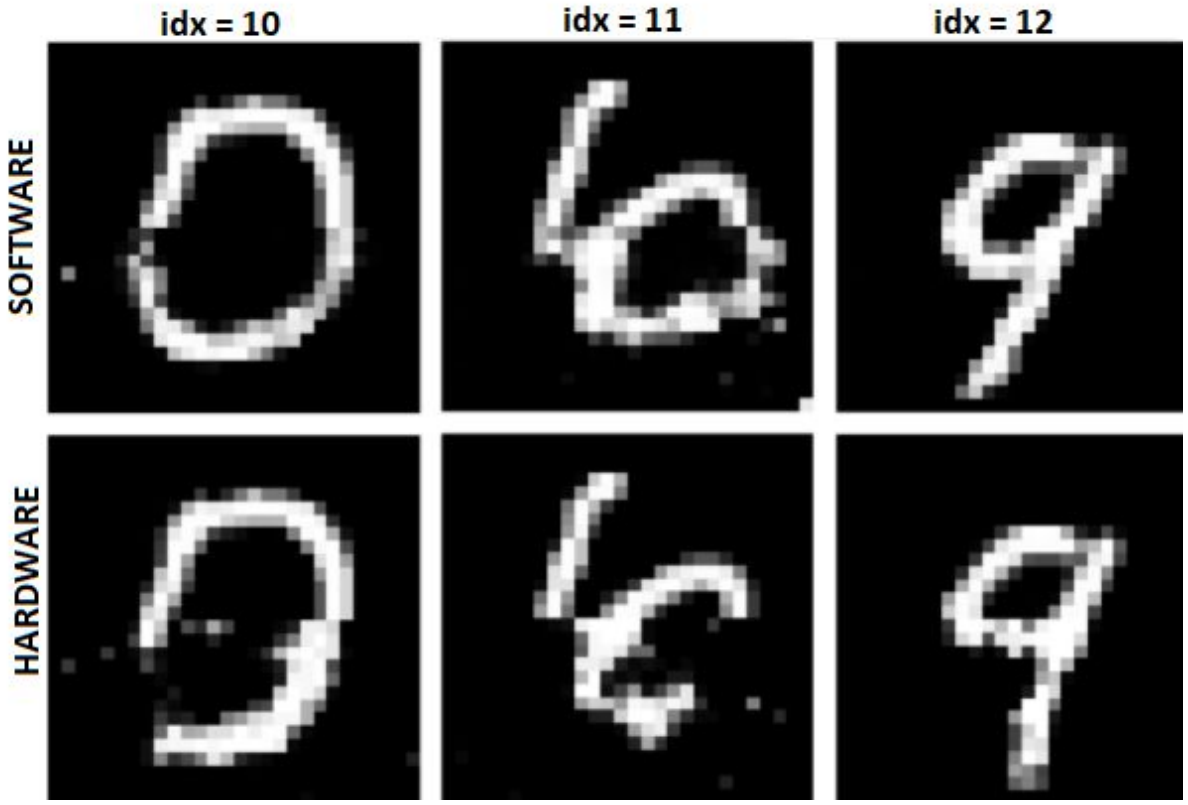
Α) Τα source files που μας δόθηκαν για το SDSoC τρέχουν τον αλγόριθμο του generator επιτυχώς. Μεταφέραμε το output.txt που παρήγαγε το εκτελέσιμο από το board στο pc μας (δεν απαιτείται να είναι optimized το design). Τρέξαμε το *plot_output.ipynb* από google collab online. Εμφανίσαμε τις combined εικόνες από SW και HW για διάφορα νούμερα, συγκεκριμένα για idx: 10, 11, 12.

Δεκαδική Ακρίβεια: 8 bits

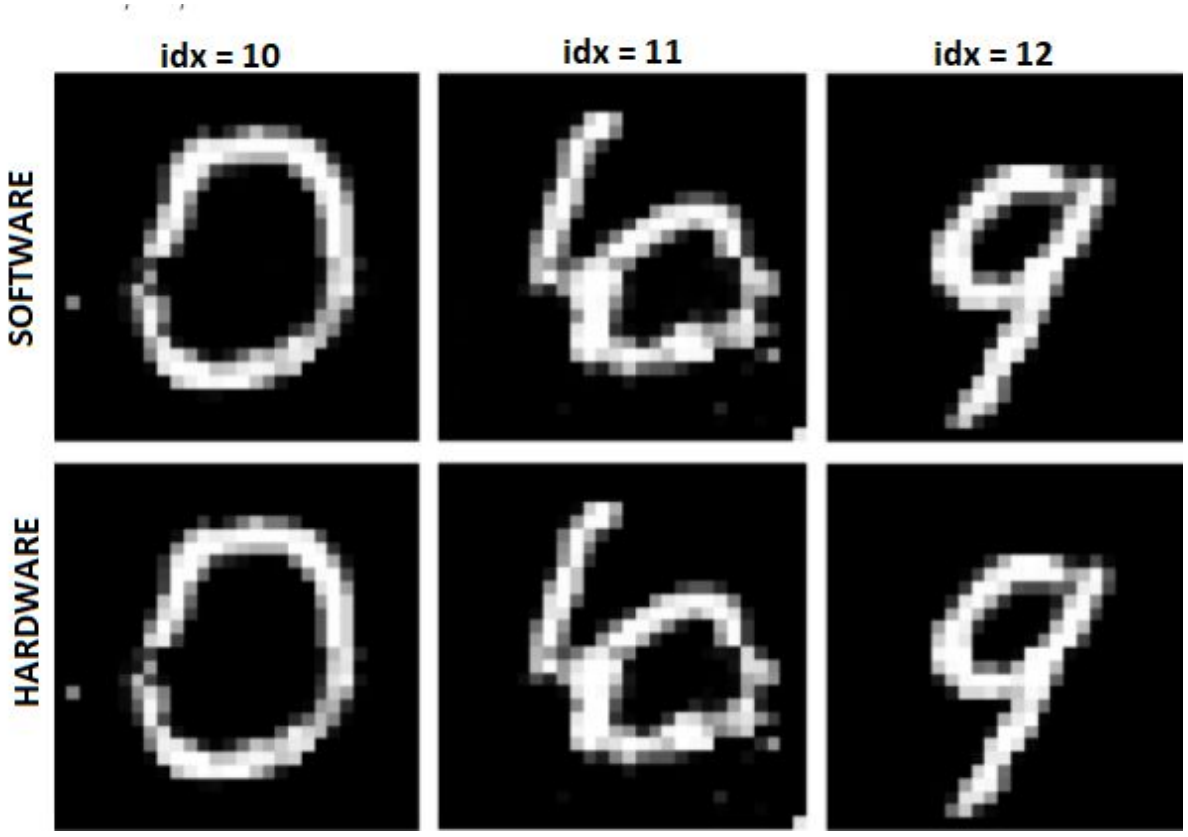


Β) Η υλοποίηση που μας δόθηκε χρησιμοποιεί datatypes που έχουν 8-bit δεκαδική ακρίβεια. Για να την αλλάξουμε αρχικά αλλάξαμε τα ορίσματα BITS και BITS_EXP στο αρχείο *network.h* όπου $BITS\_EXP = 2^{(BITS+2)}$. Έπειτα αλλάξαμε τις pre-computed τιμές της Tanh στο αρχείο *tanh.h*. Μας δόθηκαν οι υπολογισμένες τιμές για διάφορα bits στο φάκελο tanh. Δοκιμάσαμε να φτιάξουμε νέα designs με 4 και 10 bit και να τα τρέξουμε. Εμφανίσαμε τις combined εικόνες που προκύπτουν από το output πάλι για idx: 10,11,12.

Δεκαδική Ακρίβεια: 4 bits



Δεκαδική Ακρίβεια: 10 bits



Παρατηρούμε ότι όσο αυξάνεται ο αριθμός των bits δεκαδικής ακρίβειας, τόσο βελτιώνεται η ποιότητα ανακατασκευής των εικόνων. Αυτό συμβαίνει διότι με περισσότερα bits έχουμε περισσότερες pre-computed τιμές της Tanh και άρα το σφάλμα κβάντισης είναι μικρότερο.

Γ) Στο jupyter notebook μετριέται η ποιότητα ανακατασκευής των εικόνων του HW σε σχέση με το SW. Για τα διάφορα bits που δοκιμάσαμε (4,8,10) εμφανίζουμε max pixel error και Peak Signal-to-Noise Ratio (psnr).

	4 bits			**8 bits**			**10 bits**		
	idx10	idx11	idx12	idx10	idx11	idx12	idx10	idx11	idx12
mpe	255	249	255	16	17	13	5	5	4
psnr	14.05	14.63	13.52	42.68	42.56	47.06	54.08	52.55	53.76

Η τεχνική που προτιμάμε είναι το PSNR για δύο λόγους:

- Αν δύο εικόνες διαφέρουν μόνο σε ένα pixel και σε όλα τα υπόλοιπα είναι ίδιες, τότε το Max Pixel Error θα πάρει τη μέγιστη τιμή (255) παρόλο που οι εικόνες σχεδόν ταυτίζονται. Αντίθετα, το PSNR λαμβάνει υπόψη όλα τα pixel, αφού υπολογίζεται η **μέση τιμή** και έτσι επιτυγχάνει μια συνολική εκτίμηση της ποιότητας ανακατασκευής.
- Όπως φαίνεται και από την τρίτη περίπτωση (10 bits), μπορεί για κάποιες εικόνες το Max Pixel Error να είναι το ίδιο (**ακέραια** τιμή) και να μην μπορεί να εξαχθεί κάποιο συμπέρασμα. Ωστόσο, κάτι τέτοιο δεν θα συμβεί στο PSNR λόγω της **δεκαδικής** αναπαράστασης που λαμβάνει.

Όπως φαίνεται από το παραπάνω πινακάκι, η ακρίβεια bit (από 4 έως 10) που είναι ιδανική είναι η 10, καθώς επιτυγχάνει μικρότερο MPE και μεγαλύτερο PSNR. Αυτό προφανώς έγινε αντιληπτό και στις εικόνες στο ερώτημα Β, όπου είχαμε σαφώς αρκετά καλύτερη ποιότητα ανακατασκευής. Το αρνητικό με την αύξηση των bits δεκαδικής ακρίβειας είναι ότι αυξάνονται οι απαιτήσεις σε μνήμη, αφού μεγαλώνει το μέγεθος του πίνακα *tanh_vals* που περιέχει τις pre-computed τιμές της Tanh.