Digit Classification using DNNWEAVER2

10-MARCH-2020

Simulation OF LENET DIGIT CLASSIFIER in DNNWEAVER2

SHWETA UDGIRI

SUMANISH SARKAR

JAYAKUMAR SUNDARAM

2020

Contents

[1. Introduction 2](#_Toc34670522)

[2. LeNet Architecture 2](#_Toc34670523)

[3. MNIST Dataset for Training 2](#_Toc34670524)

[4. LeNet Model Training using caffe 3](#_Toc34670525)

[5. Simulation Data Generation 3](#_Toc34670526)

[5.1 Generate input image data: 3](#_Toc34670527)

[5.2 Generate weight Data 4](#_Toc34670528)

[6. DNNWeaver2 Compiler 5](#_Toc34670529)

[6.1 DNNweaver2 LeNet graph 5](#_Toc34670530)

[6.2 Generate DNNWeaver2 instruction set 6](#_Toc34670531)

[6.3 Simulate LeNet in DNNWeaver2 6](#_Toc34670532)

[7. DDR Image /Weight size Calculations for Simulation 8](#_Toc34670533)

[8. Expected Output Calculations (in Qmn data format) 11](#_Toc34670534)

[9. Simulation Data Validation 12](#_Toc34670535)

[Appendix -1 13](#_Toc34670536)

[QMN format 13](#_Toc34670537)

[Convert a fractional number to Q9.7 format 13](#_Toc34670538)

[Multiplication of 2 Qmn numbers (pos x pos): 13](#_Toc34670539)

[Multiplication of 2 Qmn numbers (neg x pos): 14](#_Toc34670540)

[Appendix – 2 15](#_Toc34670541)

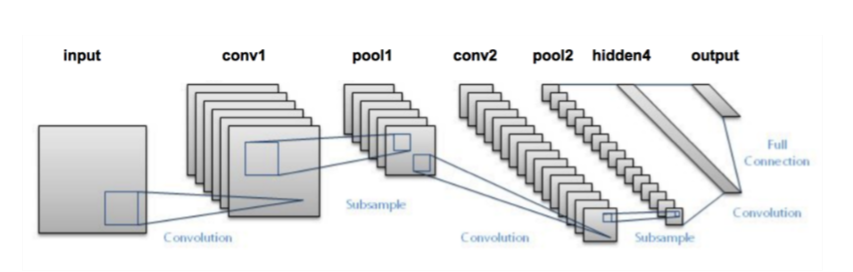
[Required Tool Chains 15](#_Toc34670542)

# Introduction

This document describes the caffe training of LeNet model , data generation process of images and weights for open source DNNWeaver2 CNN accelerator and Digits classification simulation in DNNWeaver2 CNN accelerator.

# LeNet Architecture

The LeNet architecture consists of two sets of convolutions and pooling layers, two sets of fully connected layers and finally a SoftMax classifier.



**Figure 1: LeNet Architecture**

As shown in Figure1, the two convolution layers are conv1, conv2, sub-sampling or poolinglayers are pool1, pool2. The Kernel size for each convolutions layers is 5x5 with stride 1 and maxpool kernel size is 2x2 with stride 2. There is a hidden fully connected layer (FC1), followed by the output layer (FC2).

Conv1 layer generates 20 output feature maps and Con2 layer generates 50 output channels. The number of neurons in FC1 layer is 500 and the output FC2 layers contains 10 (20-50-500-10). The input is a grey scale image of size 28x28.

LeNet architecture is widely used for Handwritten Digits Classifications.

The LeNet prototxt model is available in the below path

>**/FPGACNNProject/dnnWeaver2/dataGeneration/prototxt/lenet/lenet.prototxt**

# MNIST Dataset for Training

The LeNet model is trained using the MNIST Digits dataset. The dataset contains 6000 handwritten digits 0-9 of size 28x28 for training and another 3000 data for testing.

Sample MNIST dataset look like below,



**Figure 2: Sample MNIST Data**

The MNIST dataset is available at

**dnnWeaver2/dataGeneration/dataset/mnist**

# LeNet Model Training using caffe

This section describes the steps to train the LeNet model in Caffe framework.

Requirements:

Caffe tools framework

Python 3.6

The prototxt used for caffe framework training is available at **dnnWeaver2/dataGeneration/prototxt/lenet/ lenet\_train\_test.prototxt.**

Source the Caffe environment setup script **dnnWeaver2/dataGeneration/env/rccaffe**

Run the below script to train the model using MNIST dataset.

dnnWeaver2/dataGeneration/trainmodel/train\_lenet.py

The successful training generates the weights and storein. caffemodel format at

**dnnWeaver2/dataGeneration/caffemodelzoo/lenet/ snapshot\_20ch\_50ch\_500ch\_10ch\_lenet\_iter\_10000.caffemodel**

# Simulation Data Generation

The trained weight values for each layer of LeNet and the input image to be tested at DNNWeaver2 simulation environment needs to be pre-loaded to the DDR model in the testbench.

Below scripts are reading the image /. caffemodel files and generates the image and weights in the format required by DNNWeaver2 Systolic Array block and the DDR model.

Note: All data in Qmn format. Refer Appendix-1 for Qmn format and operations.

## 5.1 Generate input image data:

Use the following script to generate input image data,

**dnnWeaver2/dataGeneration/genscripts/image\_input\_prep/run.sh**

To generate different image data,select the test digit in run.sh

. /genImageData.py -i ../../dataset/mnist/test\_data/7\_img.bmp -o img\_7.hex >& 7.log

This script will generate following files:

1. img\_7\_qmn\_2d.dec : input pixel data of digit 7 is visualized in 2D decimal format.
2. img\_7\_qmn\_2d.hex: input pixel data of digit 7 is visualized in 2D hex format.
3. img\_7.hex: input pixel data used in DW2 simulations of digit 7 is visualized in 1D hex format.

Copy img\_7.hex to simulation data folder dnnWeaver2/dnnweaver2\_lenet/simulation/data/inputimage.txt

## 5.2 Generate weight Data

To generate weights in the required format for DW2 use the following script,

/dnnWeaver2/dataGeneration/genscripts/weight\_prep/run.sh

Command:

./genWtData.py-w ../../caffemodelzoo/lenet/snapshot\_20ch\_50ch\_500ch\_10ch\_lenet\_iter\_10000.caffemodel -p ../../prototxt/lenet/lenet.prototxt -model lenet >& gen\_weights.log

This script generates following files :

Layer Name: conv1 , Shape: (20, 1, 5, 5)

Generated weights in systolic array format in

data/lenet\_20\_50\_500\_10/conv1.float

data/lenet\_20\_50\_500\_10/conv1.hex

Layer Name: conv2 , Shape: (50, 20, 5, 5)

Generated weights in systolic array format in

data/lenet\_20\_50\_500\_10/conv2.float

data/lenet\_20\_50\_500\_10/conv2.hex

Layer Name: fc1 , Shape: (500, 1250)

Generated weights in systolic array format in

data/lenet\_20\_50\_500\_10/fc1.flt

data/lenet\_20\_50\_500\_10/fc1.hex

Layer Name: fc2 , Shape: (10, 500)

Generated weights in systolic array format in

data/lenet\_20\_50\_500\_10/fc2.flt

data/lenet\_20\_50\_500\_10/fc2.hex

\*.hex files are for the use to simulate DNNWeaver2. \*.flt files are generated to manually verify the format, if required.

copy **conv1.hex** to dnnweaver2\_lenet/simulation/data/**whex\_conv0.txt**

copy **conv2.hex** to dnnweaver2\_lenet/simulation/data/**whex\_conv1.txt**

copy **fc1.hex** to dnnweaver2\_lenet/simulation/data/**whex\_fc0.txt**

copy **fc2.hex** to dnnweaver2\_lenet/simulation/data/**whex\_fc1.txt**

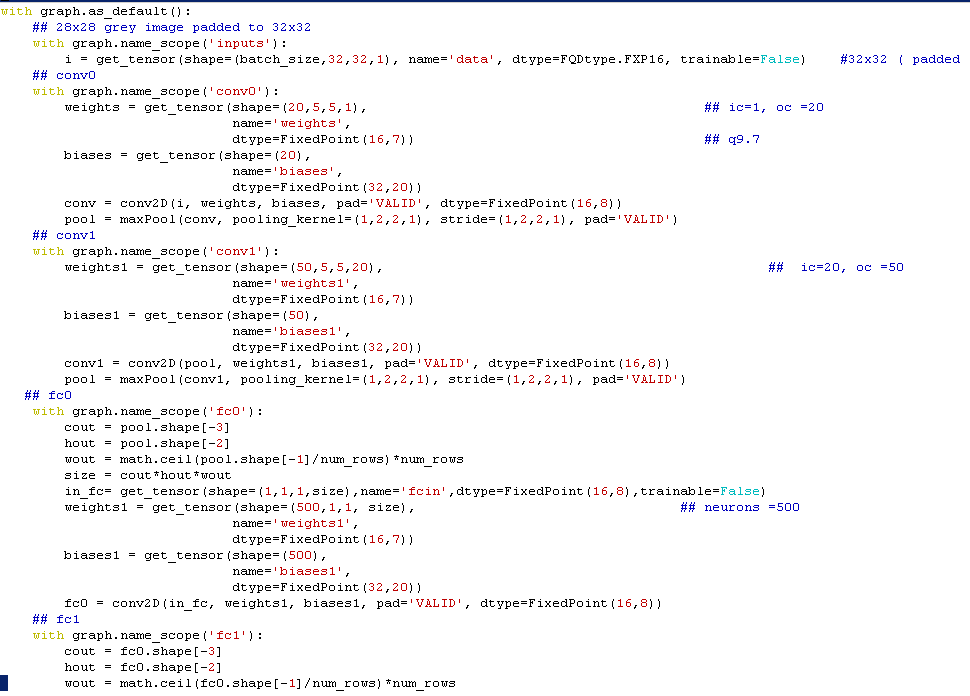
# DNNWeaver2 Compiler

## 6.1 DNNweaver2 LeNet graph

DNNweaver2 accepts the network model in tensorflow graph model. The Lenet graph is specified in the below file and input to the DNNWeaver2 compiler.

Path: dnnWeaver2/dnnweaver2\_lenet/compiler/lenet\_20\_50\_500\_100\_sys4x4.py

LeNet graph :



## 6.2 Generate DNNWeaver2 instruction set

cd dnnWeaver2/dnnweaver2\_lenet

Step1: Set Tensor Flow paths & env variables

$> source evn/.rctensorflow

cdnserv2> source env/.rctensorflow

(tfpy36cpuenv) cdnserv2>

Step2 : Generate Instructions

1) cd compiler

2) lenet model is specified in "lenet\_20\_50\_500\_100\_sys4x4.py" file

3) run compiler -> ./run.sh

This Generates the instructions for the specified network.

4) Outputs:

- instruction.bin (in hex format)

- compiler.log (contains logs and ddr address for each layer )

## 6.3 Simulate LeNet in DNNWeaver2

1. copy instruction.bin & compiler.log from "compiler" folder to simulation folder
2. cd simulation
3. In rtl/rtl/dnnweaver2\_controller.v ensure that (ARRAY\_M,ARRAY\_N)= (num\_rows,num\_cols) as in graph.

lenet\_20\_50\_500\_100\_sys4x4.py:

num\_rows = 4

num\_cols = 4

.v:

module dnnweaver2\_controller #(

parameter integer NUM\_TAGS = 2,

parameter integer ADDR\_WIDTH = 42,

parameter integer ARRAY\_N = 4,

parameter integer ARRAY\_M = 4,

1. Open tb/axi\_master\_tb\_driver.v

- Open compiler.log & search for string "Addr --"

- For each Layer there will be entry for Addr (Data,Weight,Bias & output). For eg.

INFO:Graph Compiler:MS : In Layer :fc1/Convolution

INFO:Graph Compiler:MS : Addr -- Data : 0xf000 Weights : 0x10000

INFO:Graph Compiler:MS : Addr -- Bias : 0x11000 Outputs : 0x12000

- line 202 in "tb/axi\_master\_tb\_driver.v" file update these address locations for all layers into the paramaters. (from run.log/compiler.log file )

1. Now execute simulation ./sim.sh

This runs the iverilog simulaion with the below entries.

> iverilog -c filelist.tb -g2012 -o compiler.out -Dsimulation -I . -s tb\_dnnw2\_ctrl

1. Output :

- sim.log

- tb\_dnnw2\_ctrl.vcd ( vcd sump)

1. Open gtkwave. The fc\_layer.gtkw contains the basic signal of interest.

$>gtkwave tb\_dnnw2\_ctrl.vcd fc\_layer.gtkw &

1. Validation

Here, input image digit is 7 which havehighest value neuron in FC2 layer output. The classified output details are available in sim.log

FC2 Final Output decimal\_format hex\_format

ddr\_ram[ 0]= 177 00000000000000b1

ddr\_ram[ 1]= 64936 000000000000fda8

ddr\_ram[ 2]= 130 0000000000000082

ddr\_ram[ 3]= 579 0000000000000243

ddr\_ram[ 4]= 64033 000000000000fa21

ddr\_ram[ 5]= 329 0000000000000149

ddr\_ram[ 6]= 64180 000000000000fab4

ddr\_ram[ 7]= 1995 00000000000007cb(Classified Digit)

ddr\_ram[ 8]= 65292 000000000000ff0c

ddr\_ram[ 9]= 647 0000000000000287

ddr\_ram[ 10]= 0 0000000000000000

ddr\_ram[ 11]= 0 0000000000000000

# DDR Image /Weight size Calculations for Simulation

The testbench Loads the initial input image and weight data to the input DDR memory. To perform this operation few parameters needs to be modified in the testbench based on the network and layer values outputs.

|  |  |  |
| --- | --- | --- |
| Layer Name / Parameter | Size | Remarks |
| Systolic Array Size | 4x4 |  |
| ARRAY\_N | 4 | 4 IC parallel channels |
| ARRAY\_M | 4 | 4 OC parallel Channels |
| Input Image | 28x28 | Input Image 32x32(after padding) |
| Kernel and Stride | 5x5, 1 |  |
| Conv1 | 1 (ic), 20 (oc), | Output Image Size = 28x28.(32-5+1) |
| MaxPool1 | 2x2 ,2 | Output Image Size =14x14 (28/2) |
| Conv2 | 20 (ic), 50 (oc) | Input Image size = 14x14  Output Image size = 10x10 (14-5+1) |
| MaxPool2 | 2x2 ,2 | Output Image = 5x5 (10/2) |
| FC1 | 50 (ic), 500 (oc) | 500 Neurons |
| FC2 | 500 (ic) , 10 (oc) | 10 Neurons |

The systolic array is 4x4, so 4 input channels can be applied simultaneously.So, the input image size to be loaded to DDR is “input image size” x ARRAY\_N.

**Input image size: 4096 locations**

Padded digit input image size => 32 x 32.

DDR Image size = Input Image Size \* ARRAY\_N, => 32\*32\*4 = 4096

**Conv1 Weight size: 2000 locations.**

To compute the total DDR weight size, we need to consider the systolic array size, input feature maps, output feature map and kernel size.

For Conv1, the number of input channel is 1 and output channels is 20.

The Weight size for one cycle of systolic array size 4x4 (Kx \* ky \* ARRAY\_N \* ARRAY\_M). = (5 \* 5 \* 4 \* 4) = 400.

To compute 20 Output Feature maps, the no of cycles or iterations in systolic array is Ceil (20 / ARRAY\_M) = 5.

To use 1 input feature map the no of cycles or iterations in systolic array is ceil (IC/ARRAY\_N) = ceil (1 / 4 ) = 1 ;

So, the total weight size is (1\*5\*400) = 2000 locations

Maxpool Image Size is 14x14 and stored in DDR. This is padded and feed as input for the next layer.

**Conv2 input image size:**

Padded image, input image + kernel size – 1: 14+5-1=18.

DDR image size 18\*18\*4 = 1296, where 4 is ARRAY\_N size.

**Conv2 weight size:**

Each systolic array takes 400 (5\*5\*4\*4) weight size, whereas 5\*5 is kernel size and 4\*4 is systolic ARRAY\_M x ARRAY\_N.

20 Input channels = ceil (20 / ARRAY\_N) = ceil (20/ 4) = 5

50 output channels = ceil (50/ARRAY\_M) = ceil (50/4) =13

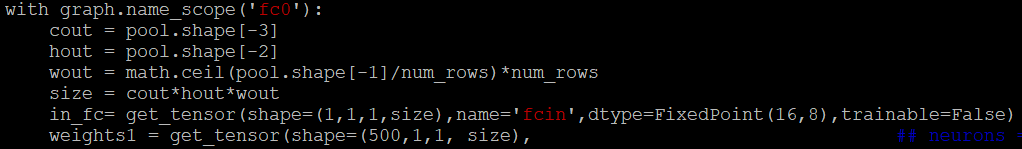
To compute 20 IC's and 50 OC’s, it takes (5\*13\*400) = 26000 weight size

After Maxpool2, image size will become 5x5 which is input to FC1 layer

**FC layers:**

2D Output of Conv+Pool is converted to single 1D array. This array is the input to all the neuron in the FC layer and another input is the weight matrix. Each element of the input 1D array can be considered as one input channel for the systolic array. So, each element is now a 1x1 image. The corresponding weight kernel size is also 1x1. For each node, we have N input channels and N weight kernels of size 1x1.

**FC1:**



No. of IC = 50 (Output of conv2, before flattening)

No. of OC = 500;

Systolic array size = 4x4;

Input image: 5x5 (Before flattening)

Kernel: 5x5 ( before flattening)

Stride: 1

IMGSIZE = 1300 (Input to FC1, after flattening)

WGHTSIZE = 650000

**FC1 Image size (IMGSIZE)**

To compute 50 IC’s, we need 13 (50/4 + 1) sets of systolic array. Each IC has 25 elements.

So image size is 13\*25\*4 =1300

fc1.jpg

**FC1 weight size (WGHTSIZE)**

Each set of systolic array has 4 IC and 4 OC. To compute 50 IC’s, we need 13 sets of systolic arrays. To compute 500 OC’s, we need 125 sets of systolic arrays.

Each systolic array takes 400 (5\*5\*4\*4) weight sizes. Here 5\*5 is kernel size and 4\*4 are systolic ARRAY\_M x ARRAY\_N.

To compute 50 IC's and 500 OC’s, it takes (13\*125\*400) = 650000 weight size

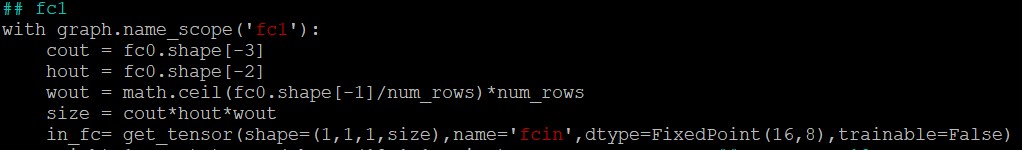
**Compiler:**

NumOutputChannels (after padding) = ceil( NumOutputChannels/M)\*M = ceil(1/4)\*4 = 4

size = cout\*hout\*wout

in\_fc = get\_tensor(shape=(1,1,1,size),name='fcin',dtype=FixedPoint(16,8),trainable=False)

**2.FC2:**

****

No. of IC = 500;

No. of OC = 10;

Systolic array size = 4x4;

Kernel: 1x1

IMGSIZE = 500

WGHTSIZE = 6000

**To compute image size(IMGSIZE):**

Each element of input 1D array is considered as 1 input channel. Here it has 500 IC's ,so image size is 500.

fc2log.jpg

**To compute weight size(WGHTSIZE):**

Each set of systolic array has 4 IC and 4 OC. To compute 500 IC’s, we need 125 sets of systolic arrays. To compute 10 OC’s, we need 3 sets of systolic array.

Each systolic array takes 16 (1\*1\*4\*4) weight sizes. Here 1\*1 is kernel size and 4\*4 is systolic ARRAY\_M x ARRAY\_N.

To compute 500 IC's and 10 OC’s, it takes (125\*3\*16) = 6000 weight size

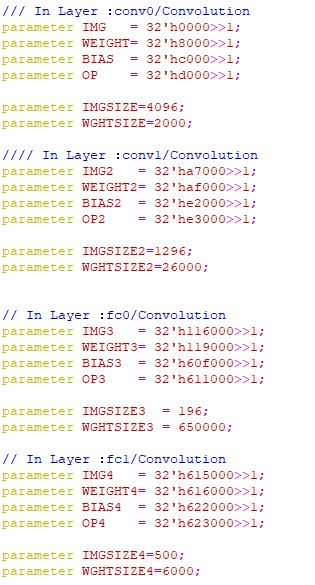


Figure 3: axi\_master\_tb\_driver.v file entry

Specify the computed Image and weights and each layer address in

/dnnWeaver2/dnnweaver2\_lenet/simulation/tb/axi\_master\_tb\_driver.v

# Expected Output Calculations (in Qmn data format)

Use the below script to generate the expected convolution/maxpool/FC layer outputs in qmn format.

Path:/svn/FPGACNNProject/dnnWeaver2/dataGeneration/genscripts/lenet\_layerresults\_compute

Command: ./run.sh

This script generates the \*.log file, includes layer wise computations which is used to verify the DDR output.

To compute results for different images and different lenet prototxt configurations, load the \*.bmp image in the run.sh.

./genResults.py -i ../../dataset/mnist/test\_data/7\_img.bmp -ptxt ../../prototxt/lenet/lenet.prototxt-w ../../caffemodelzoo/lenet/snapshot\_20ch\_50ch\_500ch\_10ch\_lenet\_iter\_10000.caffemodel>& img7\_layerresult\_qmndec.log

# Simulation Data Validation

The simulated data in DNNWeaver2 is validated against the layer wise computed data as generated in section 8. However, the entire computation in the simulation and expected data calculation is in QMN format.

So, the QMN format data needs to be converted back to floating value and compared.

**Software Data:**

The below pictures show the Qmn value to floating value conversion required for the software layers data.

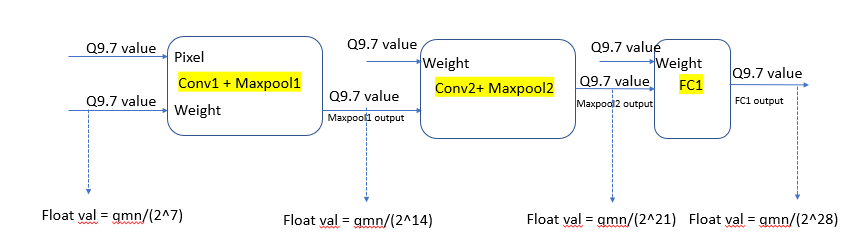


Figure 4 : Q9.7 to floating conversion Process for Software Data

For example, in “img7\_layerresult\_qmndec.log”, the expected FC2 output is specified in qmn format as below

[ 47421684473 -161626568891 36044273723 155717563951 -403610143488 88412703520 -364351692637 535969735536 -64476793451 172600868425]

To get the equivalent floating value divide the corresponding output (qmn) number by 2^35. i.e 535969735536 / 2^35 = 15.5987

**Simulation Data:**

In compiler graph we can specify the Qmn value in “dtype=FixedPoint(16,7))” . The results stored in ddr memory is 16-bit data sliced [22:7] from a 64-bit accumulated data.

So, output of each layer stored in DDR is always divide by 2^7.

In sim.log, FC2 output of 7th neuron (digit 7) is ‘1995’ in Qmn format. The floating value is 1995/ 2^7 =15.5859.

The data value in simulation 15.5859 is close to the software computed expected data 15.5987. The accuracy loss in comparison is accepted.

In this way the simulation data is manually compared with expected data across the layers to validate the functional correctness of dnnweaver2.

# Appendix -1

## QMN format

**Qmn** is a binary fixed-point format where the number of fractional bits and the number of integer bits is specified. Q format is often used in hardware that does not have a floating-point multiplier unit.

Example Qmn formats:

Q16 is the format where 16 fractional bits are represented in 16bit data size

Q1.15 represents 1 integer and 15 fractional bits in the 16bit data size.

Q8.8 represents 8 integer and 8 fractional bits in the 16bit data size.

Q9.7 represents 9 integer and 7 fractional bits in the 16bit data size.

## Convert a fractional number to Q9.7 format

Decimal fraction: 0.44722712

* 0.44722712 \* 2^7 (where 7 is the fractional bit portion)
* 57.2450
* Ignore fractions
* (57)d
* Convert 57 to Hex
* *(0039)h*

Below are few sample conversion

Num: 0.44722712: Q9.7: (57)d , 0039h

Num: 0.38406804: Q9.7: (49)d , 0031h

Num: 0.21279903: Q9.7: (27)d , 001Bh

Num: 0.005641659: Q9.7: (0)d , 0000h

Num: -0.20501006: Q9.7: (-26)d , Do 2’s complement TC: 1111111111100110 , FFE6h

Num: 0.01629672: Q9.7: (2)d , 0002h

Num: 0.15990347: Q9.7: 20 , 0014

## Multiplication of 2 Qmn numbers (pos x pos):

Num: 0.15990347: Q9.7: (20)d , 0014h

Num: 0.23478635: Q9.7: (30)d , 001Eh

* 0014h x 001Eh

(0258)h

Convert Qmn to Fraction

* (0258)h / 2^14 ( 14 = 2 times the fractional bit n )

(600)d / 2^14

* 0.03662109375

Verification:

(0.15990347) d \* (0.23478635) d = 0.03754315207

* Minor Accuracy Loss

RTL approach :

In rtl design, for multiplication of two no's (16 bitsx16 bits =32 bits) output is assigned to 16 bits, considering the portion of out bits starting from 7th bit till 22nd bits of 0285h.

0014h x 001Eh = 0004h ([22:7] bits of 0258)

0258 = 0000 0000 0|000 0000 0000 0010 0|101 1000

out = 0000 0000 0000 0100 = (0004)h = (4)d

* (4)d / 2^7 (as output is 16 bit, so divide by 2^7)
* 0.03125

## Multiplication of 2 Qmn numbers (neg x pos):

Num: -0.3230421 : Q9.7: (-41)d , Do 2’s complement TC: 1111111111010111 , FFD7

Num: 0.25686562 : Q9.7: (32)d , 0020h

* (FFD7) h \* (0020) h
* 1F FAE0
* Overflow the 16 bits ;
* Ignore the overflow data and consider FAE0 only
* Overflow indicates negative number
* Do two’s complement of FAE0
* (010100100000)2
* (1312)d
* Convert Qmn results to Fraction
* (1312)d / 2^14
* 0.080078125

Verification :

-0.3230421 \* 0.25686562 =>-0.0829784093

rtl approach :

For neg\*pos no's , take 2's compliment of output which is again 16 bit and divide the resultant decimal number by 2^7.

FFD7h x 0020h = fff5

fff5 = 1111 1111 1111 0101 (MSB bit is 1 , ouput is -ve)

2's comp = 0000 0000 0000 1011 = (-11)d

* (11)d / 2^7
* 0.0859375

# Appendix – 2

## Required Tool Chains

1. Caffe framework tools (for training and data gen scripts)
2. Python 3.6
3. iverilog & gtkwave ( for simulation and waveform viewing)