Assignment Report

Implementation of Convolutional Layer in Deep Neural Network

1. Introduction

A deep neural network (DNN) is defined to be an artificial neural network with multiple hidden layers between the input and output layers. DNNs can model complex non-linear relationships. Using many layers enables breaking down complex problems into smaller simpler ones. The network starts to build up a complete inference about the complex problem by gaining partial knowledge through these multiple layers and aggregate them together at the end to provide an accurate classification.

DNN layers can be classified to main three categories depending on their functionality. One of them is:

Convolutional Layers: These types of layers are used for extracting desired features from input data. It applies a convolution process using set of filters on input data to detect important features related to the application.

For every window slide the following computation occurs as shown in Figure 1:

1. Element-wise multiplication between filter elements and input data.
2. Summation of all multiplication results into the output pixel.
3. The convolution filter is applied as a 3-D kernel on multiple input features as shown in figure 2. The final result will be the average of all summed outputs of all input features used in convolution.
4. For every output data, a sigmoid or maxval(0, value) function is applied.

The number of computed output features in convolutional layers equals the number of filters applied on input data.

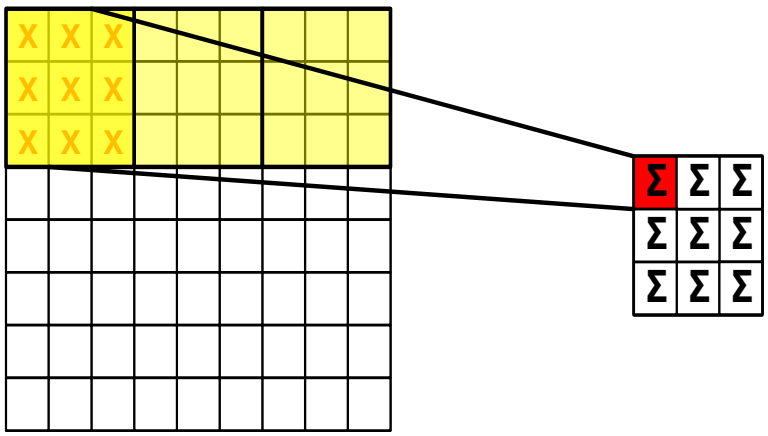


Figure 1: Computations done in convolutional Layer

To understand the number of computations required for one convolutional layer, Let us imagine a number of input features (Ni) with height and width dimensions (hi) as shown in figure 2 and using a filter with size (K x K).

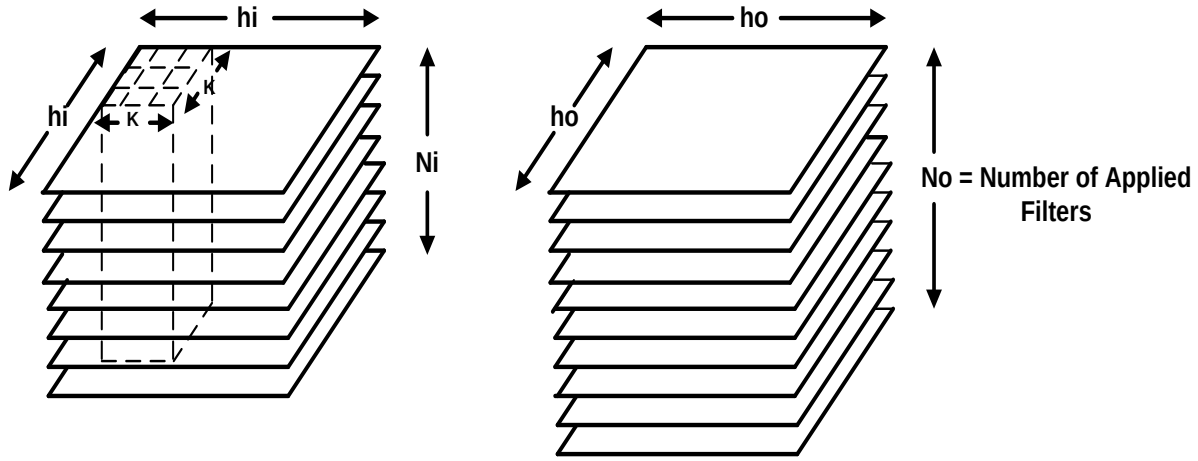


Figure 2: (Left) input features of a convolutional layer and (right) resulting output features after applying convolution

The resulting output image will have size (ho x ho) where ho is computed using equation (1):

$$ho = \frac{hi - k + 2 * padding}{Stride} + 1 \quad (1)$$

Where padding is the number of columns/rows inserted to input to adjust it for convolution computation. Stride determines the number of columns/rows the filter window shifts every window slide. The output of pooling layer adheres to the same equation too.

So if we have an output feature with number of output data = ho^2, then the number of computations for every output data are:

- **Multiplications:** $\boldsymbol{Ni * k^2}$
- **Additions:** $\boldsymbol{\left(Ni * (k^2 - 1)\right) + (Ni - 1)}$
- **Divisions: 1**
- **MaxVal/Sigmoid: 1**

Finally if we have number of used filters (No), then the final equations for the number of computations can be derived as the following:

- **Multiplications:** $\boldsymbol{(Ni * k^2) * ho^2 * No}$
- **Additions:** $\boldsymbol{\left(\left(Ni * (k^2 - 1)\right) + (Ni - 1)\right) * ho^2 * No}$
- **Divisions: 1 *** $\boldsymbol{ho^2 * No}$
- **MaxVal/Sigmoid: 1 *** $\boldsymbol{ho^2 * No}$

Please note that if we are handling an image with RGB channels these equations will be multiplied by three.

We use the convolutional neural network (CNN) proposed in [1] to understand how intensive the computations executed in convolutional layer are. The network architecture is shown in Figure 3

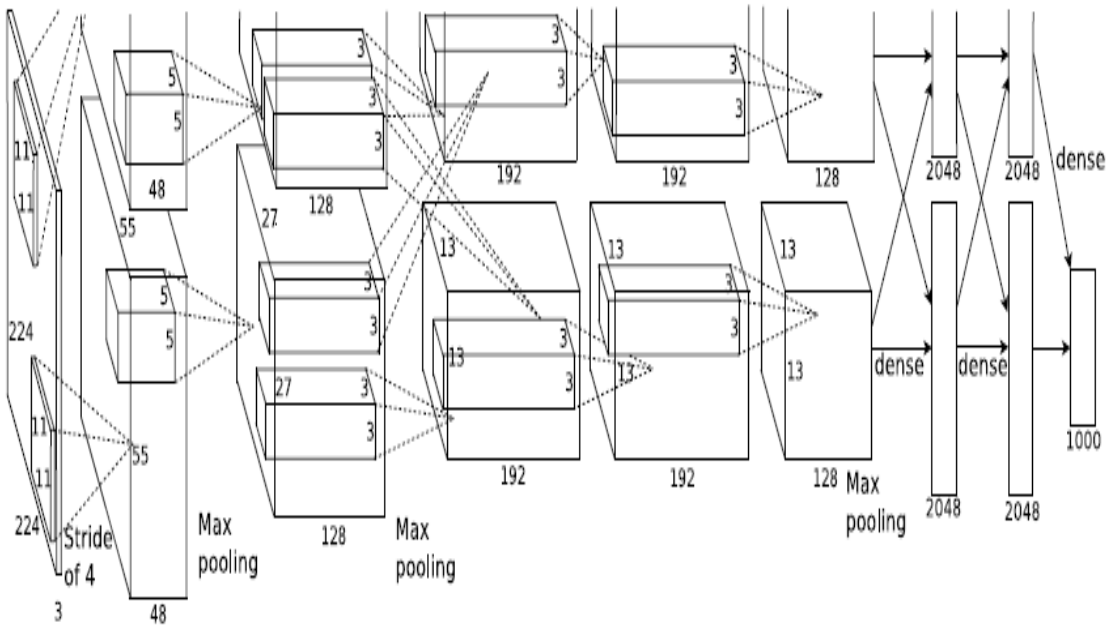


Figure 3: The architecture of DNN proposed in ImageNet Classification with Deep Convolutional Neural Networks paper [1].

Table 1 gives an insight of the number of computations needed for **one** input instance for convolutional layers in the DNN.

Table 1: Computations needed for convolutional layers of DN introduced in [1].

Layer	**Layer Parameters**	Number of computations for **one** input set
Conv1	**224 X 224 input image with 3 channels, 96 filters 11 x 11 x 1 with 3 channels**	Mult: ≈ 1.581 Billion Add: ≈ 1.567 Billion Division: ≈ 13.066 Million Transfer: ≈ 13.066 Million
Conv2	**96 55 X 55 input images with 3 channels, 256 filters 5 x 5 x 48 with 3 channels**	Mult: ≈ 2.304 Billion Add: ≈ 2.302 Billion Division: ≈ 1.92 Million Transfer: ≈ 1.92 Million

Conv3	**256 27X 27 input images with 3 channels, 384 filters 3 x 3 x 256 with 3 channels**	Mult: ≈ 1.658 Billion Add: ≈ 1.658 Billion Division: ≈ 720000 Transfer: ≈ 720000
Conv4	**384 13 X 13 input images with 3 channels, 384 filters 3 x 3 x 192 with 3 channels**	Mult: ≈ 336.42 Million Add: ≈ 336.22 Million Division: ≈ 194688 Transfer: ≈ 194688
Conv5	**384 13 X 13 input images with 3 channels, 384 filters 3 x 3 x 192 with 3 channels**	Mult: ≈ 336.42 Million Add: ≈ 336.22 Million Division: ≈ 194688 Transfer: ≈ 194688
Total		Mult: ≈ 6.215 Billion Add: ≈ 6.199 Billion Division: ≈ 16.094 Million Transfer: ≈ 16.094 Million

This shows the importance of building hardware accelerators to support computations needed in convolutional layers.

2. Module Design

In this assignment, I build a simple convolutional layer function. It will be more developed through the project to be more generic and provides better performance depending on the overall system and memory architecture.

The module has two inputs and one output as shown in Figure 4. Data is the input image, Filter is the input feature extractor to be convoluted on Data image and Out is the resultant output.

For now the size of all three attributes are defined statically. It may be kept this way in case we use small replicated modules to handler larger inputs or It may be an input to the module.

Also the stride by which the filter slides horizontally and vertically is statically defined. There other control and handshaking signals that are autogenerated by vivado HLs.

All ports are s_axilite because this module is going to be placed in a system as a slave with microblaze soft processor controlling it.

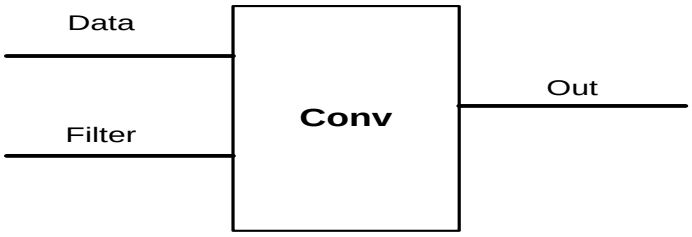


Figure 4: Conv module Diagram

```
void conv(float data[NUM_DATA_ROWS*NUM_DATA_COLS], float
filter[NUM_MASK_ROWS*NUM_MASK_COLS], float out[NUM_OUT_ROWS*NUM_OUT_COLS]) {

 // The size of the new output image
 int conv_cols = ((NUM_DATA_COLS - NUM_MASK_COLS )/ (STRIDE)) + 1;
 int conv_rows = ((NUM_DATA_ROWS - NUM_MASK_ROWS )/ (STRIDE)) + 1;
 // index used to point to the first element of the convoluted window inside input image
 int window_start_idx = 0;
 //output index at which results are stored
 int out_idx = 0;
 //data index of input pixel to be multiplied by the filter element
 int data_idx = 0;
 //filter index of element to be multiplied by input data element
 int filter_idx = 0;
 //temp to store accumulation result
 float temp = 0;

OUTER_DATA_LOOP:for (int row = 0; row < conv_rows; ++row) {
   INNER_DATA_LOOP:for (int col = 0; col < conv_cols; ++col) {
      //compute start index for every output pixel
           window_start_idx = (row * NUM_DATA_COLS * STRIDE) + (col * STRIDE);
      //compute output index at which final result will be stored
           out_idx = (row * conv_cols) + col;

      // initialize result storage to 0
           out[out_idx] = 0;
           temp = 0;
      // loop for filter computation
      OUTER_MASK_LOOP:for (int filter_row = 0; filter_row < NUM_MASK_ROWS; ++filter_row) {
          INNER_MASK_LOOP:for (int filter_col = 0; filter_col < NUM_MASK_COLS; ++filter_col) {
             // determine data and filter indices of elements to be multiplied for this iteration
            data_idx = window_start_idx + (NUM_DATA_COLS* filter_row) + filter_col;
            filter_idx = (filter_row * NUM_MASK_COLS) + filter_col;
             //Accumulate result
            temp = temp + (data[data_idx] * filter[filter_idx]);
          }
      }
      //store the final result at the apropriate element of output
      out[out_idx] = temp;
   }
}
}
```

3. Test Bench design

Test bench design is simple:

- I read an input data from a text file and provide a filter with all its elements = 1.
- The output should be the summation of all elements inside the size of filter window.
- Output result is compared to an expected output array and test passes if all comparisons are correct.

```
include <iostream>
#include <stdio.h>
#include <cstdlib>
#include <cstring>
#include <sstream>
#include <fstream>
#include "conv.h"

using namespace std;


char filename[256]="D:\\Xilinx\\Projects\\Assignment\\testdata.txt";

int main(int argc, char **argv)
{

    float red_data[NUM_DATA_ROWS*NUM_DATA_COLS];
    float blue_data[NUM_DATA_ROWS*NUM_DATA_COLS];
    float green_data[NUM_DATA_ROWS*NUM_DATA_COLS];

    float filter[NUM_MASK_ROWS*NUM_MASK_COLS];

    for(int i = 0 ; i < NUM_MASK_ROWS*NUM_MASK_COLS; i++)
    {
       filter[i] = 1;
    }


       result_t Expected_red_data[4] = {10,18,42,0};
      result_t Expected_blue_data[4] = {10,18,42,0};
      result_t Expected_green_data[4] = {10,18,42,0};

      result_t sw_red_data[4] = {0,0,0,0};
       result_t sw_blue_data[4]= {0,0,0,0};;
      result_t sw_green_data[4]= {0,0,0,0};;

    ifstream source;                    // build a read-Stream

    source.open(filename, ios_base::in);  // open data

    if (!source)  {                     // if it does not work
        cout << "Can't open file:"<<filename<<endl;
        exit(EXIT_FAILURE);
    }

   int count=0;
```

```
  READ_FILE:for(std::string line; std::getline(source, line); )
     {
        std::istringstream in(line);
        float rednum, bluenum, greennum;
        in >> rednum >> bluenum >> greennum ;
        red_data[count]=rednum;
        blue_data[count]=bluenum;
        green_data[count]=greennum;
        cout << red_data[count] << endl;
        count++;
    }

   int err_cnt = 0;

   conv(&red_data[0], &filter[0], &sw_red_data[0]);
   conv(&blue_data[0], &filter[0], &sw_blue_data[0]);
   conv(&green_data[0], &filter[0], &sw_green_data[0]);



   // Print result matrix
   cout << "{" << endl;
   for (int i = 0; i < 4; i++) {
          cout << sw_red_data[i] << endl;
           cout << sw_blue_data[i] << endl;
              cout << sw_green_data[i] << endl;;;
         if ((Expected_red_data[i] != sw_red_data[i]) ||(Expected_blue_data[i] !=
sw_blue_data[i]) ||(Expected_green_data[i] != sw_green_data[i]) ) {
             err_cnt++;
          }
   }
   cout << "}" << endl;
   if (err_cnt)
      cout << "ERROR: " << err_cnt << " mismatches detected!" << endl;
   else
      cout << "Test passed." << endl;
   return err_cnt;
}
```

4. Architectural Analysis

The following table shows a comparison among all architectural alternatives chosen to implement my design.

Table 2: Archtiectural design alternatives

Directives	Estimated Clock	Latency Min	Latency Max	Interval Min	Interval Max	BRAM_18K	DSP48E	FF	LUT
No Directive	**8.09**	**189**	**189**	**190**	**190**	**0**	**5**	**471**	**845**
Outer Loop Pipeline	**8.09**	**39**	**39**	**40**	**40**	**0**	**5**	**633**	**914**
Function DataFlow	**8.09**	**189**	**189**	**190**	**190**	**0**	**5**	**475**	**846**
Partition Arrays complete	**7.92**	**173**	**173**	**174**	**174**	**0**	**5**	**471**	**1100**
Partition + Inner loop Pipeline	**8.72**	**32**	**32**	**33**	**33**	**0**	**10**	**1116**	**2357**
Partition + outer loop Pipeline	**7.26**	**28**	**28**	**29**	**29**	**0**	**20**	**1869**	**3915**
partition block + pipeline	**7.26**	**32**	**32**	**33**	**33**	**0**	**10**	**1612**	**2060**
Outer loop PipeLine + Function DataFlow	**8.09**	**40**	**40**	**41**	**41**	**0**	**5**	**973**	**960**

- It is clear that the outer loop can be pipelined (read data, multiply, accumulate).
- Also data array can be partitioned so it can be used in parallel. Partitioning will give great performance especially if filter windows do not overlap. For every output element it will have its independent set of partitioned data and work on it separately.
- Complete partition will yield huge increase in area. Block partitioning will decrease the number of FF and LUT's required, yet preserving the latency and interval results near complete partitioning configuration.
- Adding Dataflow should have increased function performance but strangely it did not. I think because using only one function so its effect was not apparent.

- For the best performance configuration I would use the one with array partition and outer loop pipeline. But it is not area efficient.
- The best tradeoff between area and performance would be the configuration with pipelining only then pipeline + Dataflow.

5. RTL Export to HDL

Table 3: Comparison between my design Vs. RTL export using HDL

Directives	Estimated Clock	BRAM_18K	DSP48E	FF	LUT
Outer loop PipeLine + Function DataFlow	**8.09**	**0**	**5**	**973**	**960**
HDL export	**7.889**	**0**	**5**	**952**	**855**

6. RTL Co-Simulation

Co-simulation passed correctly using Verilog.

Cosimulation Report for 'conv'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	40	40	40	41	41	41
SystemC	NA	NA	NA	NA	NA	NA	NA

C simulation passed as shown in figure below.

```
{
10
10
10
18
18
18
42
42
42
0
0
0
}
Test passed.
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

7. Module Packaging and Microblaze System integration

I packaged my module and Microblaze into a simple system over NEXYS 4. Microblaze can control my module using an axilite adapter where my module is in slave mode. I added another GPIO output connected to led to use it in testing that microblaze is working correctly.

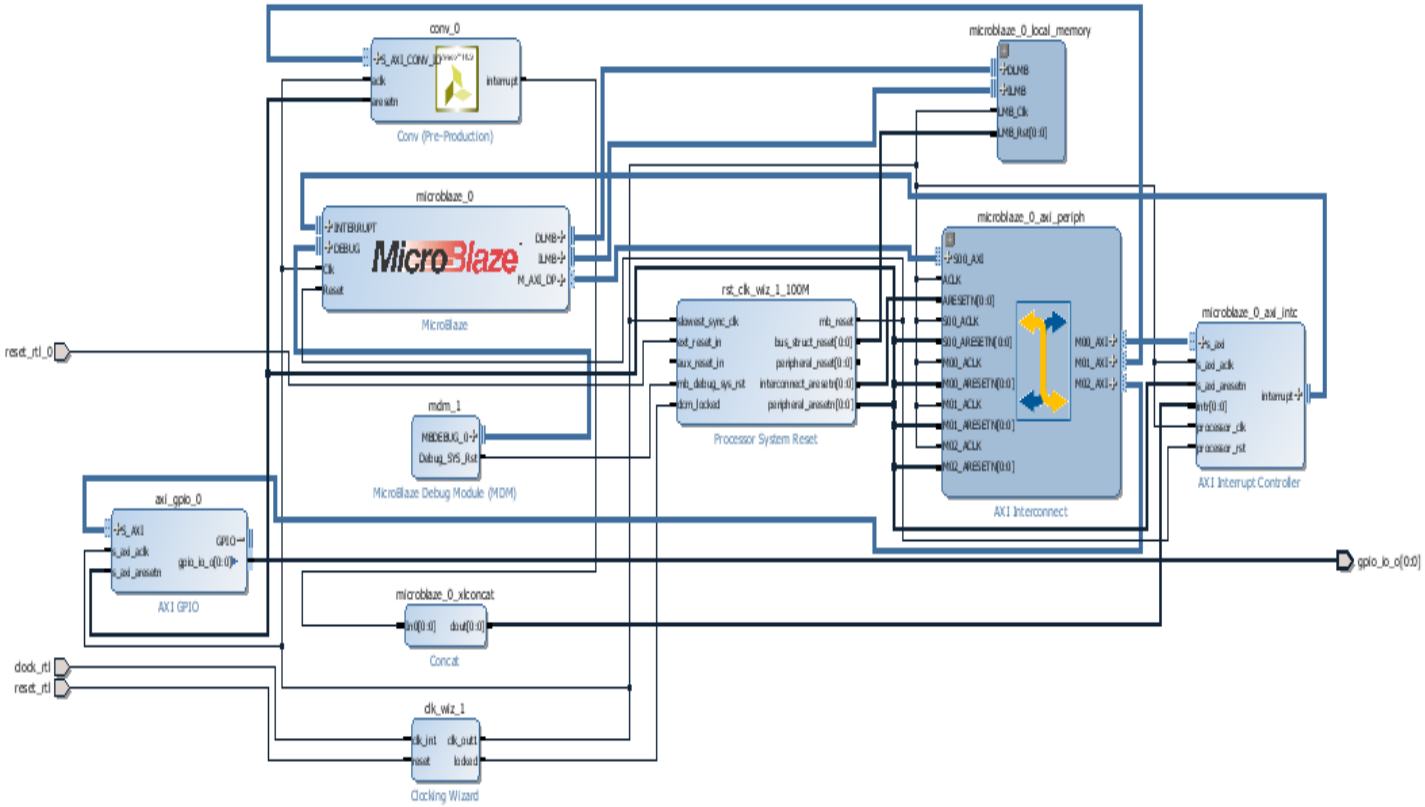


Figure 5: System Block diagram

System synthesis, implementation and bit-stream generation was done successfully as shown in Figure 6

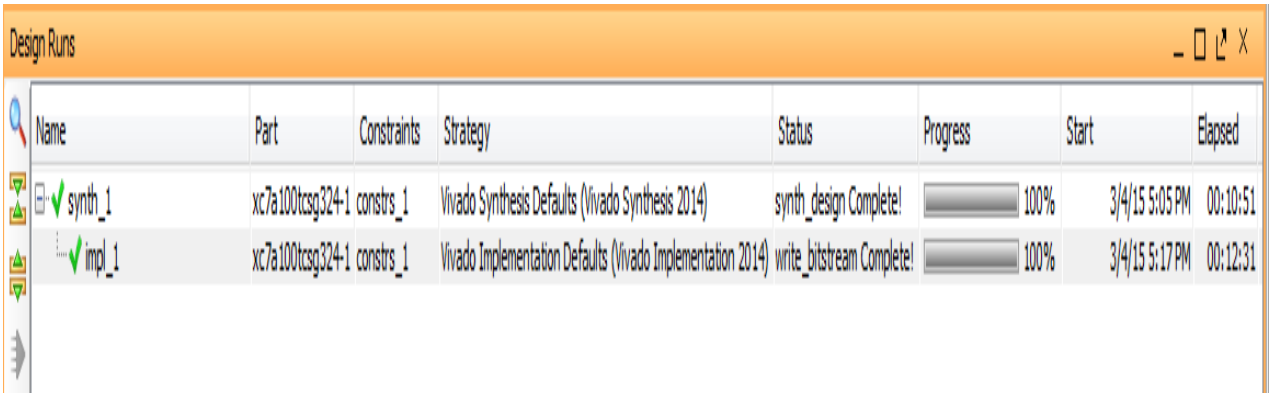
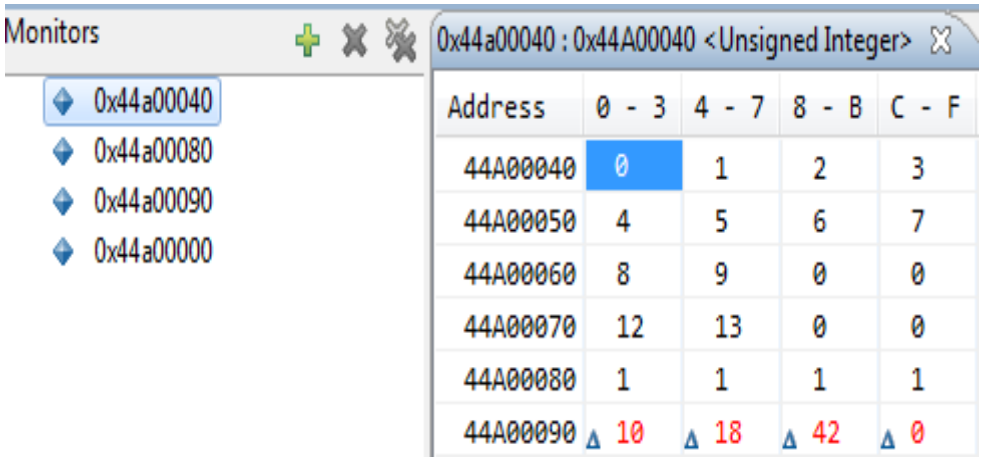


Figure 6: Design synthesis, implementation and bit-stream generation is completed successfully

I created a test program on SDK that do the following:

- Writes values to Data and Filter input ports then starts the module. I check the memory locations of Data, Filter, Out to check things are implemented correctly.



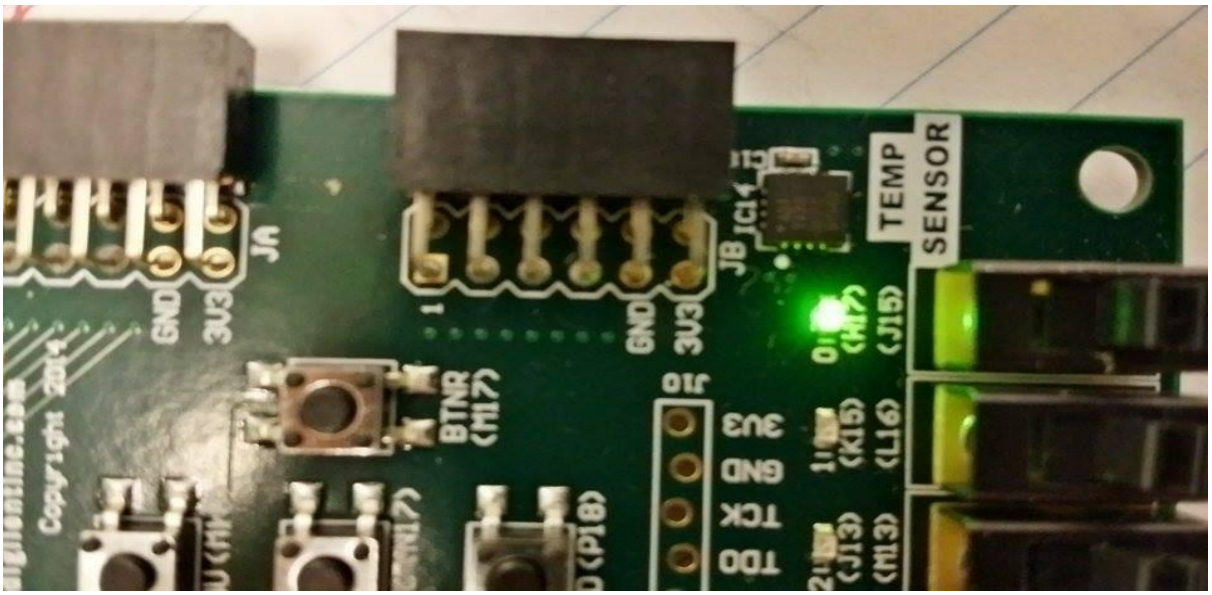
Address	0 - 3	4 - 7	8 - B	C - F
44A00040	0	1	2	3
44A00050	4	5	6	7
44A00060	8	9	0	0
44A00070	12	13	0	0
44A00080	1	1	1	1
44A00090	Δ 10	Δ 18	Δ 42	Δ 0

Data is : 0 1 2 3 / 4 5 6 7 / 8 9 0 0 / 12 13 0 0

Filter is : 1 1 1 1

Out is: 10 18 42 0

- It switches on a led that is configured inside the hardware system, H17. I check led is switched on.




```
#include <stdio.h>
#include "xparameters.h"
#include "xconv.h"
//#include "platform.h"

volatile unsigned int * led = (unsigned int *)0x40000000;
volatile  unsigned int *ConvModule  = ( unsigned int *)0x44a00000;
int x = 0;
```

```
int main()
{
/* Write data input port*/
/*My module object is Xconv_0*/
       XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 0, 0);
       XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 4, 1);
       XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 8, 2);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 12, 3);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 16, 4);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 20, 5);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 24, 6);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 28, 7);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 32, 8);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 36, 9);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 40, 0);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 44, 0);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 48, 12);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 52, 13);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 56, 0);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_DATA_BASE + 60, 0);

/* Write Filter input port*/

    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_FILTER_BASE + 0, 1);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_FILTER_BASE + 4, 1);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_FILTER_BASE + 8, 1);
    XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_FILTER_BASE + 12, 1);

/*Start module*/
       u32 Data = XConv_ReadReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_AP_CTRL) & 0x80;
       XConv_WriteReg(XPAR_CONV_0_S_AXI_CONV_IO_BASEADDR,
XCONV_CONV_IO_ADDR_AP_CTRL, Data | 0x01);

while(1)
/* switch on led connected to H17*/
*led = 1;
return 0;
}
```

References:

[1] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems, pages 1–9, 2012.