# High-Level Synthesis of the Max Pooling Layer using Vivado HLS

Venkatesh Mahadevan
Department of Electrical and Computer
*University of Toronto, Toronto, Canada*
*venkatesh.mahadevan@utoronto.ca*

***Abstract*—This paper describes the max pooling algorithm that is often used in Convolutional Neural Networks to identify maximum values in each layer and thus reducing the computation for the upper layers. The source code used for implementing the same is described, along with details on the different architectures that were explored in order to find one that is the most optimal in terms of area and performance. The latter is then integrated with the MicroBlaze processor for use on the Zedboard. The paper concludes by identifying the possible interactions that can occur between MicroBlaze and the IP and the significance of the same in our project.**

***Keywords* — Vivado, MicroBlaze, CNN, HLS, architecture**.

## I. Introduction

Neural networks are information processing paradigms that are inspired by the way biological systems such as the brain process information. They are composed of a large number of highly interconnected elements (called neurons) that work in unison to solve specific problems. Since the brain is composed of millions of neurons, one would naturally conclude that the parallel processing power of the same would be enormous. However, if we were to compare the brain's neurons with individual cores in a parallel computing cluster, we would find that the speed at which neurons could compute would be orders of magnitude slower than their hardware based counterparts. The question then arises as to how the brain is able to achieve a high throughput in terms of processing information in the form of signals coming from various mediums, and how once could incorporate it to make computers even faster.

## II. ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (or ANN's) are an example of how the behaviour of the brain can be modelled in software. Their function is similar to the adjustment of synaptic connections in the human brain. These adjustments happen millions of times a second, which is what bestows our brain with its immense processing power. In addition, extensive research done in the scientific community has shown that different parts of our brain perform different functions. Their response to stimuli is often dictated by feedback and error correction. This idea of response via feedback gives rise to an ANN paradigm known as Convolutional Neural Networks (or CNN's). CNN's are a type of feed-forward ANN's that comprise of individual neurons that are tiled so as to respond to overlapping regions in the visual field. These are variations of multi-layer perceptrons, which were thought to be the basic processing units for perception in the early days of neural networks. They exploit spatially-local correlation through the enforcement of local connectivity patterns between neurons of adjacent layers, Simply put, the outputs of neurons from previous layers become inputs to the neurons in the subsequent layers, The output is determined by the activation functions associated with each layer. A simplified diagram of the same is given below

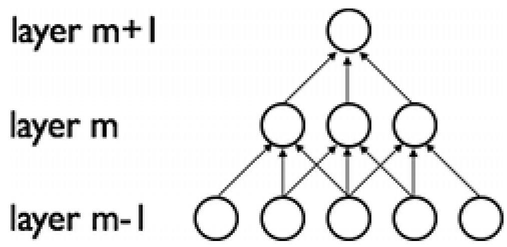


**Fig 2.1: Simplified model of CNN**

CNN's consist of various layers, each with its own unique processing abilities. They are outlined below:

1) **Convolutional layers (CONV):** This layer implements a set of filters to identify characteristic elements of the individual data.. A filter defined by certain dimensions forms a kernel, and the kernel coefficients form the layer's synaptic weights. By sliding a certain number of such filters over the input data, a certain number of output feature maps are formed.
2) **Pooling layers (POOL):** These layers compute the max or average over a number of neighbor points. Its effect is to reduce the input layer dimensionality, which allows coarse grain (larger scale) features to emerge. It does not use any learned synaptic weights for computation..
3) **Local Response Normalization Layers (LRN):** This layer implements competition between neurons at the same location, but in different neighbor (feature) maps. Their effect can be considered to be similar to lateral inhibition found in biological neurons.
4) **Classifier Layers (CLASS):** The outputs from the previous layers is then fed to multiple classifier layers. Multi-layer perceptrons are often used for these layers. Their main function is to correlate the different features extracted from the filtering normalization and pooling steps, and create the output categories for image or video processing.

This report primarily focuses on the pooling layers, with special emphasis being placed on the max pooling operation. The following sections describe the algorithm, the main blocks used to implement the same, and background about the code used to achieve the functionality required.

## III. MAX POOLING ALGORITHM

Mathematically, the max pooling algorithm can be represented as shown on the next page ($k_x$ and $k_y$ are the mask dimensions, out is the output and in the input):

:

$$out(x; y) = \max_{0<=kx<=Kx;0<=ky<=Ky} in(x + kx; y + ky) \quad (Eqn.\ 3.1)$$

As described previously, the max pooling layer receives input from the convolutional layer. Using a mask of suitable size, an attempt is made to find the maximum value in each region of the input that has the same size as the mask. These values are then passed to the LRN layer for further processing. The pseudo-code for max pooling can be described as follows:

```
for col from 0 to input_width in steps of mask_col_size
    for row from 0 to input_height in steps of mask_col_rows
        initialize maximum value and index
        for pcol from 0 to mask_width and col+pcol less than input width in unit steps
            for prow from 0 to mask_row and row+prow less than input height in unit steps
                find maximum value in region
                    if value > maximum ; maximum = value
        set output array entry to maximum found above
    iterate
```

**Fig 3.1. Pseudo-code listing for max pooling**

Since the max pooling layer exhibits a dependency of the layers above and below it, and since those layers would be designed at different points in time, an attempt was made to mimic the behaviour of the layer if it were subject to actual input. The next section describes the code chosen to implement and exercise the max pooling algorithm.

## IV. CODE DESCRIPTION

To exercise the algorithm, we use a filter-based approach. An input image with three color channels (*red*, *green* and *blue*) is setup and fed to a module that converts it into a different set of color channels (one luma called *'y'* and two chrominance *'u'* and *'v'*). This is then scaled by an appropriate scaling factor and passed to the max pooling layer described above. The max pooling layer then extracts the maximum values, and these are then converted back to the original color channels (*red*, *green* and *blue*). A flow chart for the code is shown below:

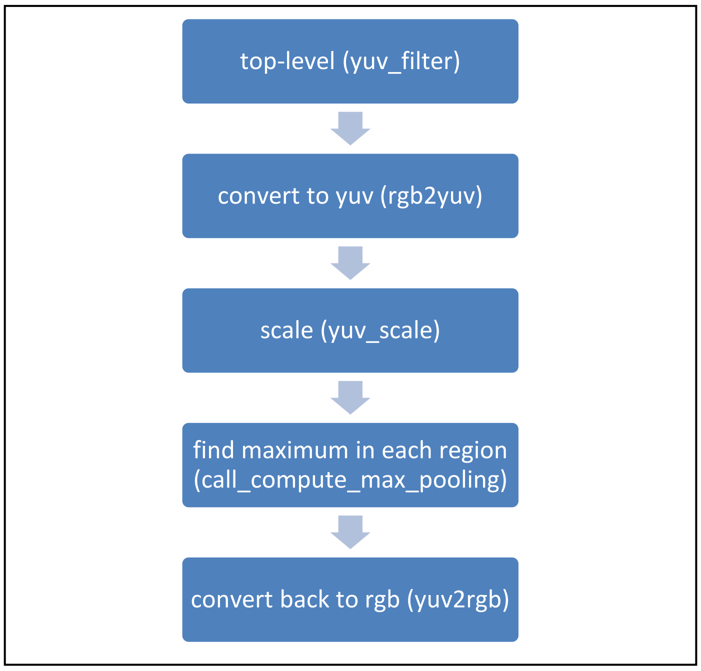

top-level (yuv_filter)
convert to yuv (rgb2yuv)
scale (yuv_scale)
find maximum in each region (call_compute_max_pooling)
convert back to rgb (yuv2rgb)

**Fig 4.1 Flow chart for top-level code**

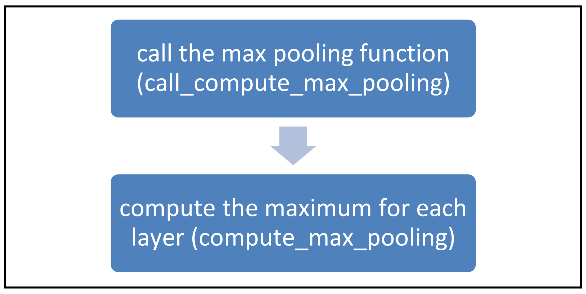

call the max pooling function (call_compute_max_pooling)
compute the maximum for each layer (compute_max_pooling)

**Fig 4.2 Flow chart for code used for max pooling**

The top-level function (*yuv_filter*) receives a pointer to the input image, a pointer to the output image, and scaling factors for the *y*, *u* and *v* color channels. The input and the output images are both of type *struct* with the following three member variables:
1. Width (of type *unsigned short*)
2. Height (of type *unsigned short*)
3. A color channel *struct*.
The color channel *struct* is composed of the following three member variables (all of type *char*):
1. First color channel
2. Second color channel
3. Third color channel.

Each of the color channels is of the same size as the input image or the output image, depending on where it is used. *yuv_filter* calls *rgb2yuv*, which converts the rgb color space to the yuv color space. The output from *rgb2yuv* is then scaled by *yuv_scale* using the scaling factors provided above, and fed to the *call_compute_max_pooling* function which returns the maximum values. These values are then converted to rgb using the *yuv2rgb* function. The testbench then compares the calculated values with the values from the hardware based implementation, and displays an error message if any discrepancies exist. The pseudo-code for *yuv_filter* is shown below:

```
convert from rgb to yuv
    scale the output from above using the provided
     scaling factors
       compute the maximum value from the output above
         convert the maximum values from yuv back to rgb
```

**Fig 4.3 Pseudo-code for *yuv_filter***

The *rgb2yuv* function takes a pointer to the input image and a pointer to the resultant *yuv* image. It then sets the width and height of the output image to those of the input image. A matrix comprising of constant coefficients is then constructed for the purpose of converting from the *rgb* to *yuv* color space. It then iterates over the dimensions of the input image, extracts the *r*, *g* and *b* components and multiplies them by the above coefficients and uses bitwise operations for translation into the corresponding *yuv* component. These values are then assigned to be the corresponding *yuv* entries, which are later used by *yuv_scale*. The pseudo-code for *rgb2yuv* is described below:

```
for row from 0 to input_width in unit steps
    for col from 0 to input_height in unit steps
        extract R, G, B components
        multiply by coefficient matrix and shift to form Y,U,V components
```

**Fig 4.4 Pseudo-code for *rgb2yuv***

The *yuv_scale* function then takes a pointer to the above output, and a pointer to where its output would be generated.

The width and height of its output are set to those of the input image. Then, iterating over the dimensions of its input, it extracts the *y*, *u* and *v* components and scales by the appropriate scaling factors and shifts at the same time. These are then assigned to the output, and later used by *call_compute_max_pooling*. The pseudo-code listing for the yuv_scale function is shown below:

```
for row from 0 to input_width in unit steps
   for col from 0 to input_height in unit steps
         extract Y, U, V components
         scale by scaling factors and shift to form scaled
Y,U,V components
```

**Fig 4.5 Pseudo-code for *yuv_scale***

The *call_compute_max_pooling* function is a top-level wrapper for *compute_max_pooling*. It takes in the scaled yuv image and a pointer to its output, which contains the maximum *y*, *u* and *v* values. It then declares its own input and output arrays (intermediate),which are used to separate the three color channels. These are then fed to *compute_max_pooling*, and the result from the latter is combined into the output array. The pseudo-code listing for *call_compute_max_pooling* is shown below:

```
for row from 0 to input_width in unit steps
   for col from 0 to input_height in unit steps
         extract scaled Y, U, V components

call compute_max_pooling with these components

for row from 0 to output_width in unit steps
   for col from 0 to output_height in unit steps
         assign the maximum of each color component to
the output array
```

**Fig 4.6 Pseudo-code for *call_compute_max_pooling***

The *compute_max_pooling* function takes in a pointer to the input channel and a pointer to its output channel. It then iterates over then dimensions of the input channel, and slides the mask over the same. The maximum value in the region of the input channel which is covered by the mask is then computed, and assigned to the output channel. The pseudo-code listing for *compute_max_pooling* is shown in Fig 3.1, and therefore not repeated here.

The *yuv2rgb* function behaves in a similar fashion as *rgb2yuv* but in the reverse order. It takes a pointer to the maximum values in the scaled input image and a pointer to the resultant *rgb* image. It then sets the width and height of the output image to those of the input image. A matrix comprising of constant coefficients is then constructed for the purpose of converting from the *yuv* to *rgb* color space. It then iterates over the dimensions of the input image, extracts the *y*, *u* and *v* components and multiplies them by the above coefficients and uses bitwise operations for translation into the corresponding *rgb* component. These values are then assigned to be the corresponding *rgb* output entries. The pseudo-code for yuv2rgb is shown next:

```
for row from 0 to output_width in unit steps
   for col from 0 to output_height in unit steps
         extract Y, U,V  components
         multiply by coefficient matrix and shift to form
R,G,B components
```

**Fig4.7 Pseudo-code for *yuv2rgb***

A detailed listing of the code for all functions can be found in Appendix A

## V INPUT AND OUTPUT TYPES

The previous section described the inputs and outputs for various functions. The type of input and output chosen can have a great impact of the results of high-level synthesis conducted by Vivado. To recap, the top-level inputs and outputs are pointers to *structs*, whose member variables are of type *unsigned short* and *unsigned char*. Therefore, they will be synthesized as type *ap_memory* by Vivado. Furthermore, since they need to be connected to MicroBlaze via the AXI interconnect, the '*INTERFACE*' directive is used with the *mode* set to '*s_axilite*' and bundle set to *'yuv_filter_io'*. The same procedure is used for the *ap_start*, *ap_done* and *ap_idle* signals at the top-level. This results in the creation of address maps for the above signals that can be accessed via software. An attempt was made to use the *'array_partition'* directive on the inputs of sub-functions, but was not successful due to the fact that Vivado 2014.1 ran into internal memory errors. A snapshot of the directives used in the final solution is shown in AppendixC, Part 5, subsection 1

The next section goes on to describe the testbench used to check the functionality of the max pooling algorithm.

## VI TESTBENCH DESCRIPTION

The testbench has been designed to check the output from the software implementation against the output from the hardware implementation. If the two match for every channel, the testbench outputs the "*Test passed*" message. If the two do not match, the "*Test Failed*" message is output to the screen. The pseudo-code for the testbench is as follows:

```
create space for the input and output image.
Read the input image from the file
call yuv_filter and pass the image to it with the scale factor
Invoke the hardware implementation if HW_COSIM has
been defined
Check the values of all channels and increment error count
if mismatch occurs.
Output the corresponding message base on the previous
```

**Fig 6.1 Pseudo-code for *testbench***

A detailed listing of the testbench code and the results obtained on running the same can be found in Appendix B.
The next section describes the various architectures that were explored and the performance comparison of the same.

## VI. ARCHITECTURAL EXPLORATION

Given the ability of Vivado to create various solutions, an attempt was made to generate a few solutions by incremental application of directives. In the process of doing so, five such solutions were generated, each better than the other in terms of both area and performance. Below, a description of each solution is given, and a table comparing the solutions can be found in Appendix C.

1. **Solution 1**:

In solution 1, no other directive other than the *'loop tripcount*' directive is applied to all the loops within the design. The reason for doing so is to get an initial estimate of the loop latencies and the area occupied Further iterations would be used to improve upon these estimates. The reason for using the *'loop tripcount*' directive was to allow Vivado to calculate the loop latency for loops with variable bounds. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis and implementation phase in shown in Appendix C, part 1.

2. **Solution 2**:

In solution 2, the *'loop_tripcount'* directive is used along with the '*pipeline*' directive. The latter is applied to the innermost loops in the design in order to reduce the overall latency. The result of applying the above is that the latency is now loop_body_latency + loop_iteration_count, instead of being loop_body_latency * loop_iteration_count. *call_compute_max_pooling* has been inlined using the HLS '*inline*' as well, and automatic inlining has been disabled. The resource utilization increases, as Vivado attempts to reduce the initiation interval by utilizing more resources. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis and implementation phase in shown in Appendix C, part 2.

3. **Solution 3**:

In solution 3, the '*dataflow*' directive is applied to the top-level *'yuv_filter'* function, along with the directives listed in solution 2. Performing the above optimization results in the insertion of FIFO's between functions to ensure the next function can begin operation before the *'yuv_filter*' has finished. Since all of the memory accesses in the design are fully sequential in nature, the use of FIFO's is justified. In addition, the use of the '*dataflow*' directive results in reduced utilization of FF's and LUT's, with the exception of an additional DSP48E. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis and implementation phase in shown in Appendix C, part 3.

4. **Solution 4**:

In solution 4, all of the directives from solution 3 carry over. In addition, the '*dataflow*' directive has been applied to *'call_compute_max_pooling'* as well, since it calls *'compute_max_pooling'*, once for each channel. This allows for the use of FIFO's in the case of intermediate function calls, as the '*dataflow*' directive is only valid within the scope of the function it is applied to Since the latter had 4 levels of nesting in terms of loops, the '*pipeline*' directive was removed from the innermost loop and applied to the outermost loop (*OUTER_DATA_LOOP*). This results in loop flattening. In other words, by pipelining an outer loop, all the inner loops are automatically unrolled (if legal) and there is no need to explicitly unroll the inner loops. The inner loops now operate concurrently, reducing the overall latency in the design. Surprisingly, the resource utilization of FF's and LUT's decreases as well, while that of DSP48E remains the same. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis and implementation phase in shown in Appendix C, part 4.

5. **Solution 5 (also called Solution 6 in the tables shown in the Appendix)**:

In solution 5, all of the directives from solution 4 carry over. Special interface directives are also applied to the primary inputs and outputs to connect the system to MicroBlaze for Zedboard. An attempt was made to apply the *'array_partition*' directive on lower-level arrays, but it resulted in tool crashes. Also, the application of *'array_reshape*' resulted in an increase in latency and initiation intervals. along with increased resource utilization. Since it did not improve upon the previous solutions, the application of these directives was not considered when choosing this design. The overall latency and initiation intervals for this solution were the same as that for solution 4. The resource utilization remained the same as well. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis and implementation phase in shown in Appendix C, part 5.

The next section describes the reasons for the chosen architecture and the duration it took to perform the exploration.

## VII. CHOSEN ARCHITECTURE

After comparing all the generated solutions, solution 5 was chosen since it had the lowest resource utilization when compared to the others. Although solution 3 had a faster clock that solution 5, the number of resources in solution 3 were far more than that in solution 5, swinging the decision in favor of the latter. Overall, the entire architecture exploration phase took under 2 days due to fast turnaround times achieved via incremental solution development. If the tool had not crashed when using the *'array_partition'* directive, the number of external I/O pins generated by Vivado HLS could have been greatly reduced from its current value of 50% and memory bandwidth could have been improved. This would have reduced routing time and resource utilization in the final design. Solution 2 proved to be the most non-optimal in terms of resource usage, as the negative impacts of using the 'pipeline' directive without the '*dataflow*' directive (as evidenced in later solutions) was clearly visible (see Appendix C, Part 6).

## VIII. EXPORT TO HDL AND COMPARISON OF RESULTS

Solution 5 was chosen as the optimal solution and was exported to HDL using VHDL as the chosen language. A comparison of the clock and resource utilization before and after the export phase is shown below:

**Table 8.1 Comparison of clock period before/after export**

| Clock | Before export | After export |
|---|---|---|
| Target | 10.00 | 10.00 |
| Achieved | 8.18 | 8.911 |

**Table 8.2 Comparison of resource utilization before/after export**

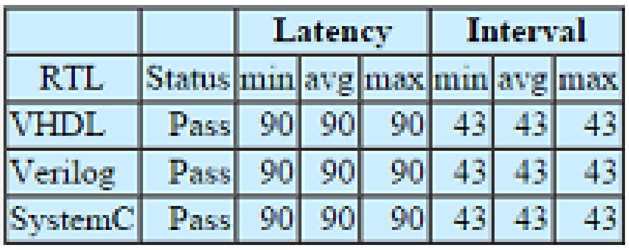
| Resource | Before export | After export |
|---|---|---|
| LUT | 2333 | 1817 |
| FF | 1536 | 998 |
| DSP | 16 | 19 |
| BRAM | 0 | 15 |

Note that SLICE and SRL are not shown in the report before export, and hence they are not taken into consideration when comparing  resource usage here.

We see a significant different in resource usage before/after export. The clock period has increased by only 8.93%, while LUT resource has gone down (22.11 %) and FF usage has gone down as well (35.02%). DSP usage has gone up (18.75%) while BRAM usage has gone up by 1500%  (quite large!). Thus the clock estimates appear to be closer than the resource estimates. The slight increase in clock period along with BRAM usage seems to suggest the exporting has replaced FF's and LUT's with memories and pipelined the latter so as to avoid impacting the quality of  the synthesis and implementation results while meeting timing requirements.

## IX.  RTL/C COSIMULATION RESULTS

Since we had added the -*DHW_COSIM* flag to our testbench, Vivado understands that the user would like to run RTL/C co-simulation.  The code in the testbench also instantiates the IP only if the above flag is defined. Hence, we run  the above from Vivado for all 3 languages (Verilog, VHDL and System C ). The results from the same are shown below:



|  |  | Latency |  |  | Interval |  |  |
|---|---|---|---|---|---|---|---|
| RTL | Status | min | avg | max | min | avg | max |
| VHDL | Pass | 90 | 90 | 90 | 43 | 43 | 43 |
| Verilog | Pass | 90 | 90 | 90 | 43 | 43 | 43 |
| SystemC | Pass | 90 | 90 | 90 | 43 | 43 | 43 |

**Fig 9.1 RTL/C Co-simulation results**

Thus we see that the co-simulation passes in all cases for the selected solution.

## X.  EXPORTING DESIGN AS IP-XACT

After adding the appropriate resource directives as stated in Section 5, the design is exported as an IP-XACT adapter for integration with MicroBlaze. The synthesis, implementation and bitstream generation results are shown in Appendix D.

## XI. INTEGRATION WITH MICROBLAZE

After exporting from Vivado HLS, the IP is integrated with MicroBlaze and targeted for the Zedboard  The main idea was to store the input and golden output in local memory. An external switch would be used to start the computation. The results which were received would be compared to the golden values stored in local memory. If the two would match, one of LED's would light up. If there was a mismatch, another LED would light up. However , it was later understood that it would not be directly possible to interface with the MicroBlaze processor, and hence it was not possible to test the design for the time being. Other interactions may be possible, and can only be realized when integrated with the entire neural network.

A block diagram of the entire system can be found in Appendix E.

## XII. CONCLUSION

In this report, we have tried to implement the max pooling layer on an FPGA using HLS. The algorithm and the testbench used have been presented, along with the various architectures that were explored, and the decisions behind choosing the best. The IP has been exported and integrated with MicroBlaze successfully. It is hoped that the learning outcomes resulting from this assignment will help when working on our project, by allowing us to make informed decisions.

## XIII, ACKNOWLEDGMENTS

This assignment has largely been possible due to guidance from the source code available from lab 2 of the Xilinx tutorials.

## XIV. REFERENCES

1. Xilinx tutorials
2. http://deeplearning.net/tutorial/lenet.html
3. Various research papers.

# APPENDICES

## APPENDIX A

This appendix is a detailed listing of the C source files and the code contained therein.

1. yuv_filter.h ( this is the file containing the declaration of various functions described in Section IV and the associated variables.)

```
#ifndef RGB2YUV_H_
#define RGB2YUV_H_

#ifdef BIT_ACCURATE
#include "autopilot_tech.h"

typedef int9  rgb2yuv_coef_t;
typedef int11 yuv2rgb_coef_t;
typedef int9  yuv_intrnl_t;
typedef uint8 yuv_scale_t;

#else // Use native C types

typedef signed short  rgb2yuv_coef_t ;
typedef signed short  yuv2rgb_coef_t;
typedef signed short  yuv_intrnl_t;
typedef unsigned char yuv_scale_t;

#endif

#include "image_aux.h"
#include "compute_max_pooling.h"

#define CLIP(x) (((x)>255) ? 255 : (((x)<0) ? 0 : (x)))

void rgb2yuv (in_image_t*, in_image_t*);
void yuv2rgb (out_image_t*, out_image_t*);

void yuv_scale (
  in_image_t *in,
  in_image_t *out,
  yuv_scale_t Y_scale,
  yuv_scale_t U_scale,
  yuv_scale_t V_scale
  );

void yuv_filter (
  in_image_t *in,
  out_image_t *out,
  yuv_scale_t Y_scale,
  yuv_scale_t U_scale,
  yuv_scale_t V_scale
  );

  void call_compute_max_pooling(
  in_image_t* scale,
  out_image_t* maxout
);
```

2. yuv_filter.c (This file contains the definition of all routines as described in Section IV except *compute_max_pooling*. It has been split into parts for easier display. This is Part 1.)

```
#include "yuv_filter.h"
#include <stdio.h>
#include <stdlib.h>
#include "compute_max_pooling.h"

// The top-level function
void yuv_filter (
      in_image_t *in,
      out_image_t *out,
      yuv_scale_t Y_scale,
      yuv_scale_t U_scale,
      yuv_scale_t V_scale
      )
{
// Internal image buffers
#ifndef __SYNTHESIS__
   in_image_t *yuv = (in_image_t *)malloc(sizeof(in_image_t));
   in_image_t *scale = (in_image_t *)malloc(sizeof(in_image_t));
   out_image_t *maxout = (out_image_t *)malloc(sizeof(out_image_t));
#else // Workaround malloc() calls w/o changing rest of code
   in_image_t _yuv;
   in_image_t _scale;
   out_image_t _maxout;
   in_image_t *yuv = &_yuv;
   in_image_t *scale = &_scale;
   out_image_t *maxout = &_maxout;
#endif

   // convert to yuv
   rgb2yuv   (in, yuv);
   // scale
   yuv_scale (    yuv, scale, Y_scale, U_scale, V_scale);
   // compute max
   call_compute_max_pooling (scale,  maxout);
   // convert to rgb
   yuv2rgb   ( maxout, out);
}

// Convert RGB image to Y'UV format
void rgb2yuv (
      in_image_t *in,
      in_image_t *out
      )
{
   image_dim_t x, y;
   image_dim_t width, height;
   image_pix_t R, G, B, Y, U, V;
   const rgb2yuv_coef_t Wrgb[3][3] = {
      { 66, 129,  25},
      {-38, -74, 112},
      {122, -94, -18},
   };
  width = in->width;
   height = in->height;
   out->width = width;
   out->height = height;
```

## 2. yuv_filter.c (This file contains the definition of all routines as described in Section IV except *compute_max_pooling*. It has been split into parts for easier display. This is Part 2.)

```
RGB2YUV_LOOP_X:
   for (x=0; x<width; x++) {
 #pragma HLS loop_tripcount min=0 max=4
RGB2YUV_LOOP_Y:
      for (y=0; y<height; y++) {
 #pragma HLS loop_tripcount min=0 max=4
         R = in->channels.ch1[x][y];
         G = in->channels.ch2[x][y];
         B = in->channels.ch3[x][y];
         Y = ((Wrgb[0][0] * R + Wrgb[0][1] * G + Wrgb[0][2] * B + 128) >> 8) +  16;
         U = ((Wrgb[1][0] * R + Wrgb[1][1] * G + Wrgb[1][2] * B + 128) >> 8) + 128;
         V = ((Wrgb[2][0] * R + Wrgb[2][1] * G + Wrgb[2][2] * B + 128) >> 8) + 128;
         out->channels.ch1[x][y] = Y;
         out->channels.ch2[x][y] = U;
         out->channels.ch3[x][y] = V;
      }
   }
}

void yuv2rgb (
      out_image_t *in,
      out_image_t *out
      )
{
   image_dim_t x,y;
   image_dim_t width, height;
   image_pix_t R, G, B;
   image_pix_t Y, U, V;
   yuv_intrnl_t C, D, E;
   const yuv2rgb_coef_t Wyuv[3][3] = {
      {298,    0,  409},
      {298, -100, -208},
      {298,  516,    0},
   };

   width = in->width;
   height = in->height;
   out->width = width;
   out->height = height;

 YUV2RGB_LOOP_X:
   for (x=0; x<width; x++) {
 #pragma HLS loop_tripcount min=0 max=2
YUV2RGB_LOOP_Y:
      for (y=0; y<height; y++) {
 #pragma HLS loop_tripcount min=0 max=2
         Y = in->channels.ch1[x][y];
         U = in->channels.ch2[x][y];
         V = in->channels.ch3[x][y];
         C = Y - 16;
         D = U - 128;
         E = V - 128;
         R = CLIP(( Wyuv[0][0] * C                  + Wyuv[0][2] * E + 128) >> 8);
         G = CLIP(( Wyuv[1][0] * C + Wyuv[1][1] * D + Wyuv[1][2] * E + 128) >> 8);
         B = CLIP(( Wyuv[2][0] * C + Wyuv[2][1] * D                  + 128) >> 8);
         out->channels.ch1[x][y] = R;
         out->channels.ch2[x][y] = G;
         out->channels.ch3[x][y] = B;
      }
   }
}
```

## 2. yuv_filter.c (This file contains the definition of all routines as described in Section IV except *compute_max_pooling*. It has been split into parts for easier display. This is Part 3.)

```
void yuv_scale (
      in_image_t *in,
      in_image_t *out,
      yuv_scale_t Y_scale,
      yuv_scale_t U_scale,
      yuv_scale_t V_scale
      )
{
   image_dim_t x,y;
   image_dim_t width, height;
   image_pix_t Y, U, V;
   yuv_intrnl_t Yn, Un, Vn;

   width = in->width;
   height = in->height;
   out->width = width;
   out->height = height;

YUV_SCALE_LOOP_X:
   for (x=0; x<width; x++) {
 #pragma HLS loop_tripcount min=0 max=4
YUV_SCALE_LOOP_Y:
      for (y=0; y<height; y++) {
 #pragma HLS loop_tripcount min=0 max=4
         Y = in->channels.ch1[x][y];
         U = in->channels.ch2[x][y];
         V = in->channels.ch3[x][y];
         Yn = (Y * Y_scale) >> 7;
         Un = (U * U_scale) >> 7;
         Vn = (V * V_scale) >> 7;
         out->channels.ch1[x][y] = Yn;
         out->channels.ch2[x][y] = Un;
         out->channels.ch3[x][y] = Vn;
      }
   }
}

void call_compute_max_pooling(
               in_image_t* scale,
               out_image_t* maxout
    )
{


   image_dim_t x,y;
   image_dim_t width, height;
   image_pix_t Y, U, V;


   width = scale->width;
   height = scale->height;
   maxout->width = OWIDTH;
   maxout->height = OHEIGHT;
```

2. yuv_filter.c (This file contains the definition of all routines as described in Section IV except *compute_max_pooling*. It has been split into parts for easier display. This is Part 4.)

```
int ycomp[IWIDTH*IHEIGHT];
   int ucomp[IWIDTH*IHEIGHT];
   int vcomp[IWIDTH*IHEIGHT];

   int yout[OWIDTH*OHEIGHT];
   int uout[OWIDTH*OHEIGHT];
   int vout[OWIDTH*OHEIGHT];

   int count=0;
YUV_COPY_LOOP_X:
   for (x=0; x<width; x++) {
 #pragma HLS loop_tripcount min=0 max=4
YUV_COPY_LOOP_Y:
      for (y=0; y<height; y++) {
 #pragma HLS loop_tripcount min=0 max=4
         ycomp[count] = scale->channels.ch1[x][y]-'0';
         //printf("Y COMPONENT BEFORE IS %d \n", scale->channels.ch1[x][y]-'0');
         ucomp[count] = scale->channels.ch2[x][y]-'0';
         vcomp[count] = scale->channels.ch3[x][y]-'0';
         count++;
      }
   }


   compute_max_pooling(ycomp,yout);
   compute_max_pooling(ucomp,uout);
   compute_max_pooling(vcomp,vout);

   count=0;
MAX_YUV_COPY_LOOP_X:
   for (x=0; x<OWIDTH; x++) {
 #pragma HLS loop_tripcount min=0 max=2
MAX_YUV_COPY_LOOP_Y:
      for (y=0; y<OHEIGHT; y++) {
 #pragma HLS loop_tripcount min=0 max=2
         maxout->channels.ch1[x][y]=yout[count]+'0';
         //printf("Y COMPONENT BEFORE IS %d \n", scale->channels.ch1[x][y]-'0');
         maxout->channels.ch2[x][y]=uout[count]+'0';
         maxout->channels.ch3[x][y]=vout[count]+'0';
         count++;
      }
   }
}
```

## 3. compute_max_pooling.h (This file contains the declaration of compute_max_pooling and associated variables as described in Section IV.)

```
#ifndef __COMPUTE_MAX_POOLING_H__
#define __COMPUTE_MAX_POOLING_H__

#include <math.h>
#include <limits.h>
#include "image_aux.h"


// Uncomment this line to compare TB vs HW C-model and/or RTL
//#define HW_COSIM

#define NUM_MASK_ROWS 2
#define NUM_MASK_COLS 2
#define IDX2C(i,j,ld) (((j)*(ld))+(i))

// Prototype of top level function for C-synthesis
void compute_max_pooling(int data[IWIDTH*IHEIGHT], int out[OWIDTH*OHEIGHT]);

#endif // __COMPUTE_MAX_POOLING_H__ not defined
```

## 4. compute_max_pooling.c (This file contains the definition of compute_max_pooling as described in Section IV.)

```
#include "compute_max_pooling.h"

void compute_max_pooling(int data[IWIDTH*IHEIGHT], int out[OWIDTH*OHEIGHT]) {
  int m;
  int idx;
  int count = 0;
  int col, row, pcol, prow;

OUTER_DATA_LOOP:for (col = 0; col < IWIDTH; col += NUM_MASK_COLS) {
    INNER_DATA_LOOP:for (row = 0; row < IHEIGHT; row += NUM_MASK_ROWS) {

                        m = INT_MIN;
                        idx = -1;
        OUTER_MASK_LOOP:for (pcol = 0; (pcol < NUM_MASK_COLS && col + pcol < IWIDTH); ++pcol) {
            INNER_MASK_LOOP:for (prow = 0; (prow < NUM_MASK_ROWS && row + prow < IHEIGHT); ++prow)
{
                                if (data[IDX2C(row + prow, col + pcol, IHEIGHT)] > m) {
                                idx = IDX2C(row + prow, col + pcol, IHEIGHT);
                                 m = data[idx];
                                   }
                              }
                          }

         out[count] = m;
         //printf("Y COMPONENT AFTER IS %d\n", m);
         count++;
      }
  }
}
```

## 5. image_aux.h  (This file contains the declaration of all routines and variables. This is used by the testbench for reading an input image.)

```
#ifndef IMAGE_AUX_H_
#define IMAGE_AUX_H_

#ifdef BIT_ACCURATE
#include "autopilot_tech.h"

typedef uint8  image_pix_t;
typedef uint11 image_dim_t;

#else // Use native C types

typedef unsigned char  image_pix_t;
typedef unsigned short image_dim_t;

#endif // ifdef BIT_ACCURATE

#define IWIDTH 4
#define IHEIGHT 4

#define OWIDTH 2
#define OHEIGHT 2

typedef struct {
   image_pix_t ch1[IWIDTH][IHEIGHT];
   image_pix_t ch2[IWIDTH][IHEIGHT];
   image_pix_t ch3[IWIDTH][IHEIGHT];
} in_channel_t;

typedef struct {
   image_pix_t ch1[OWIDTH][OHEIGHT];
   image_pix_t ch2[OWIDTH][OHEIGHT];
   image_pix_t ch3[OWIDTH][OHEIGHT];
} out_channel_t;

typedef struct {
   in_channel_t channels;
   image_dim_t width;
   image_dim_t height;
} in_image_t;

typedef struct {
   out_channel_t channels;
   image_dim_t width;
   image_dim_t height;
} out_image_t;

void image_read(in_image_t *in_image);

#endif
```

6. image_aux.c  (This file contains the definition of routines declared in image_aux.h. This is used by the testbench for reading an input image and is not synthesized.)

```
#include <stdio.h>
#include "image_aux.h"

void image_read (
      in_image_t *in_image
      ) {
   image_dim_t width, height;
   int x, y;
   FILE       *fp;

   fp=fopen("C:\\Users\\Owner\\Desktop\\ECE1373HLS\\labsource\\labs\\lab2\\test_data\\input1.dat","r");
   int tmp;
   fscanf(fp, "%d", &tmp);
   width = tmp;
   fscanf(fp, "%d", &tmp);
   height = tmp;

   in_image->width  = width;
   in_image->height = height;
   for (x=0;x<width;x++) {
      for (y=0;y<height;y++) {
         fscanf(fp, "%d", &tmp);
         in_image->channels.ch1[x][y] = tmp;
         fscanf(fp, "%d", &tmp);
         in_image->channels.ch2[x][y] = tmp;
         fscanf(fp, "%d", &tmp);
         in_image->channels.ch3[x][y] = tmp;
      }
   }
   fclose(fp);

}
```

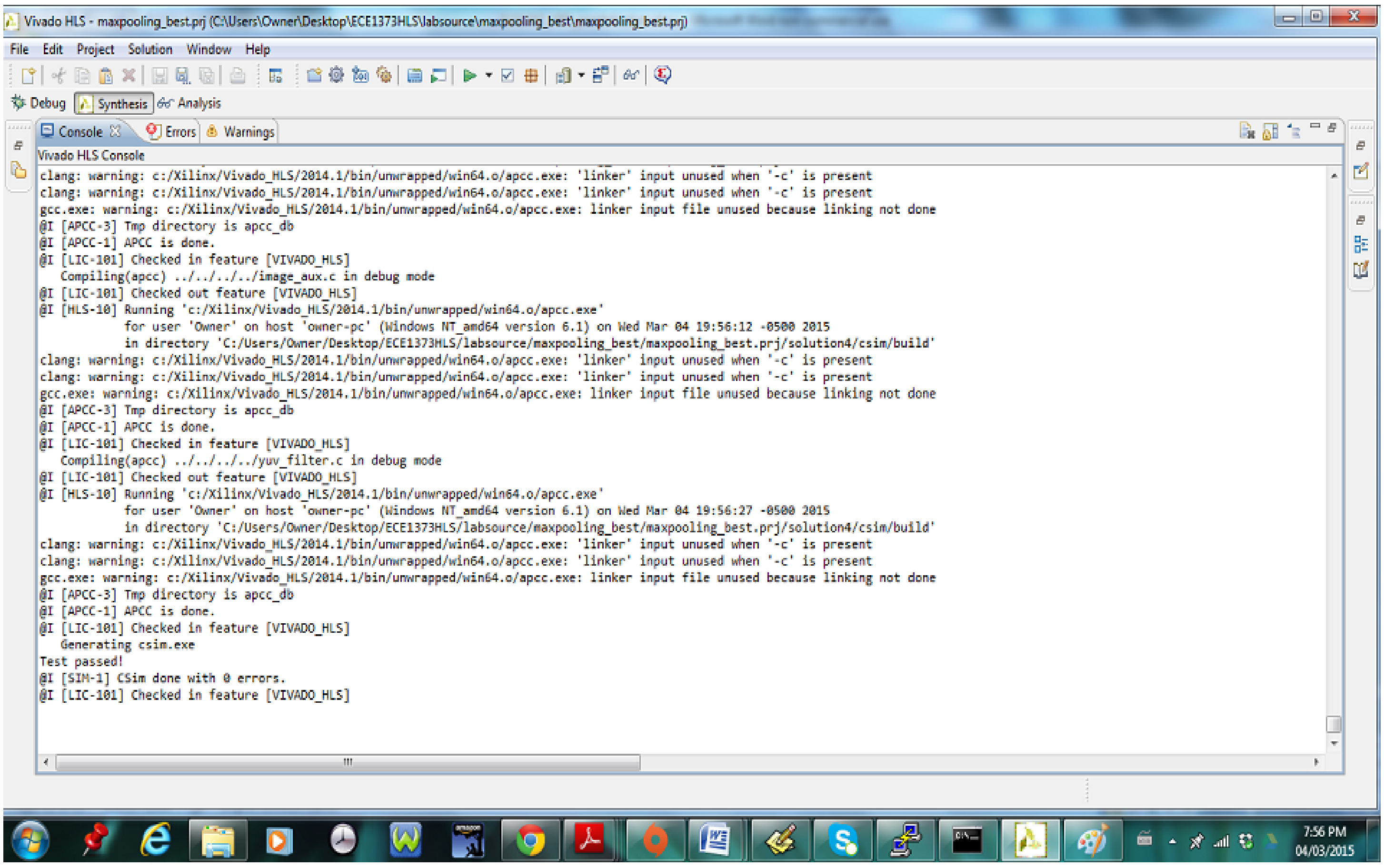
## APPENDIX B

This appendix is a detailed listing of the testbench and the results of C simulation using the same.

1. main.c (This is the main file that reads in the input image and calls yuv_filter. The result is checked with the software version to ensure consistency).

```
#include "image_aux.h"
#include <stdio.h>
#include <stdlib.h>
#include "yuv_filter.h"

int main()
{
   in_image_t *img_rgb = (in_image_t *)malloc(sizeof(in_image_t));
   out_image_t *sw_img_restore = (out_image_t *)malloc(sizeof(out_image_t));
   out_image_t *hw_img_restore = (out_image_t *)malloc(sizeof(out_image_t));

   // Read input image
   image_read(img_rgb);


    int err_cnt = 0;
    int x,y;

    // Create output image
   // Scale should be <=128 (128 corresponds to a scale of 1.0)
   yuv_filter(img_rgb, sw_img_restore, 128, 128, 128);

#ifdef HW_COSIM
   // Run the AutoESL yuv filter block
   yuv_filter(img_rgb, hw_img_restore, 128, 128, 128);
#endif

   // check result matrix
#ifdef HW_COSIM

        // Check HW result against SW
     for (x=0; x<OWIDTH; x++) {
      for (y=0; y<OHEIGHT; y++) {
        if (   (sw_img_restore->channels.ch1[x][y] != hw_img_restore->channels.ch1[x][y]) ||
               (sw_img_restore->channels.ch2[x][y] != hw_img_restore->channels.ch2[x][y]) ||
               (sw_img_restore->channels.ch3[x][y] != hw_img_restore->channels.ch3[x][y]) ) {
               err_cnt++;
           }
      }
   }

#endif

#ifdef HW_COSIM
   if (err_cnt)
      printf("ERROR: %d mismatches detected!\n", err_cnt);
   else
      printf("Test passed!\n");
#endif
   return err_cnt;
}
```

## 2. Snapshot of  C simulation showing success:


Vivado HLS - maxpooling_best.prj (C:\Users\Owner\Desktop\ECE1373HLS\labsource\maxpooling_best\maxpooling_best.prj)
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Console
Errors
Warnings
Vivado HLS Console
7:56 PM
04/03/2015

```
clang: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: 'linker' input unused when '-c' is present
clang: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: 'linker' input unused when '-c' is present
gcc.exe: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: linker input file unused because linking not done
@I [APCC-3] Tmp directory is apcc_db
@I [APCC-1] APCC is done.
@I [LIC-101] Checked in feature [VIVADO_HLS]
   Compiling(apcc) ../../../../image_aux.c in debug mode
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe'
            for user 'Owner' on host 'owner-pc' (Windows NT_amd64 version 6.1) on Wed Mar 04 19:56:12 -0500 2015
            in directory 'C:/Users/Owner/Desktop/ECE1373HLS/labsource/maxpooling_best/maxpooling_best.prj/solution4/csim/build'
clang: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: 'linker' input unused when '-c' is present
clang: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: 'linker' input unused when '-c' is present
gcc.exe: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: linker input file unused because linking not done
@I [APCC-3] Tmp directory is apcc_db
@I [APCC-1] APCC is done.
@I [LIC-101] Checked in feature [VIVADO_HLS]
   Compiling(apcc) ../../../../yuv_filter.c in debug mode
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe'
            for user 'Owner' on host 'owner-pc' (Windows NT_amd64 version 6.1) on Wed Mar 04 19:56:27 -0500 2015
            in directory 'C:/Users/Owner/Desktop/ECE1373HLS/labsource/maxpooling_best/maxpooling_best.prj/solution4/csim/build'
clang: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: 'linker' input unused when '-c' is present
clang: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: 'linker' input unused when '-c' is present
gcc.exe: warning: c:/Xilinx/Vivado_HLS/2014.1/bin/unwrapped/win64.o/apcc.exe: linker input file unused because linking not done
@I [APCC-3] Tmp directory is apcc_db
@I [APCC-1] APCC is done.
@I [LIC-101] Checked in feature [VIVADO_HLS]
   Generating csim.exe
Test passed!
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

## 3. Example of input file used for testing (The first 2 rows are width and height, the next rows correspond to the R,G,B values of each pixel for each location)

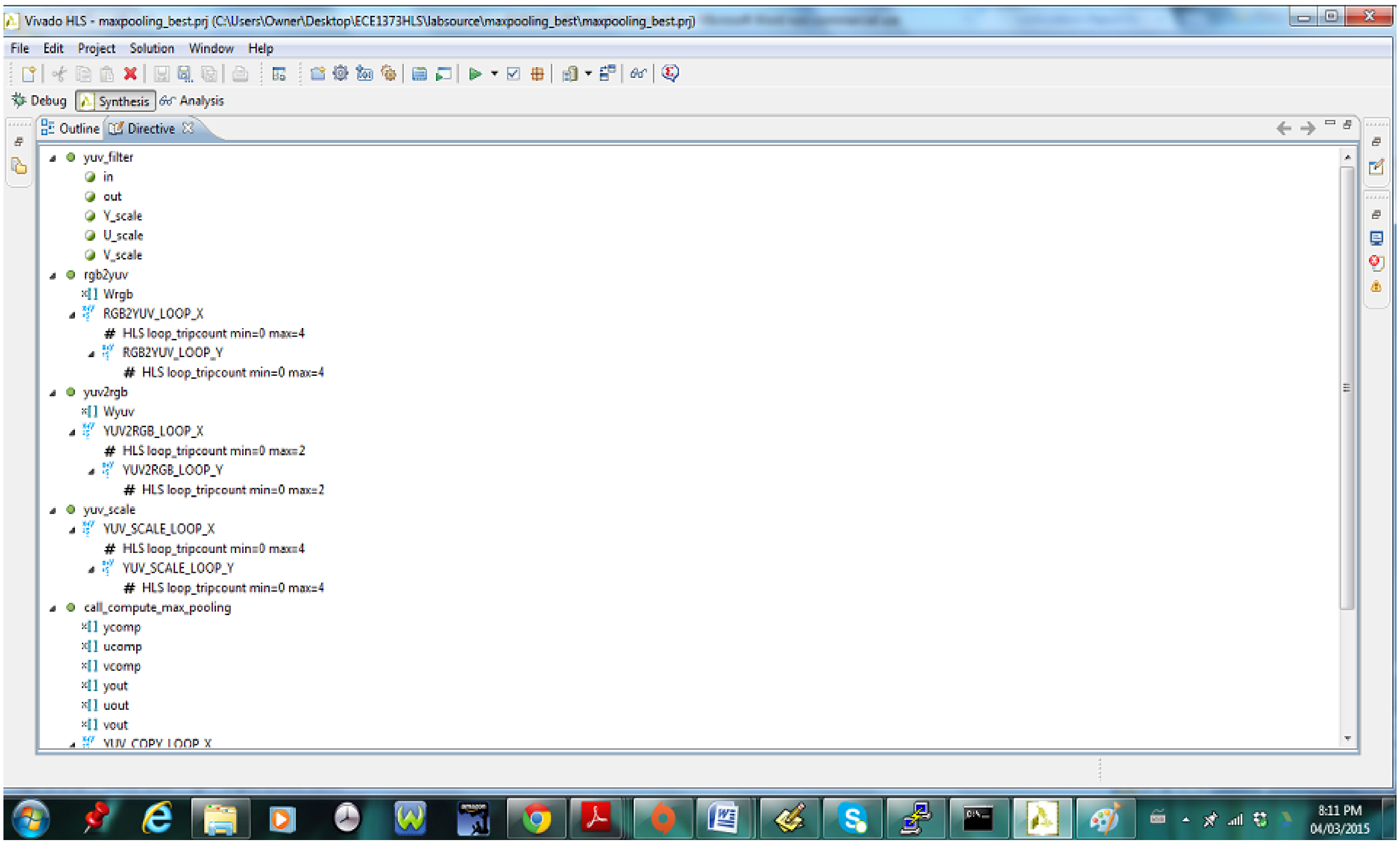
```
4
4
2
3
4
5
6
7
8
9
10
11
12
13
14
…
48
49
```

# APPENDIX C

This appendix contains a listing of the directives used at the top-level, and synthesis and implementation reports for each of the solutions.

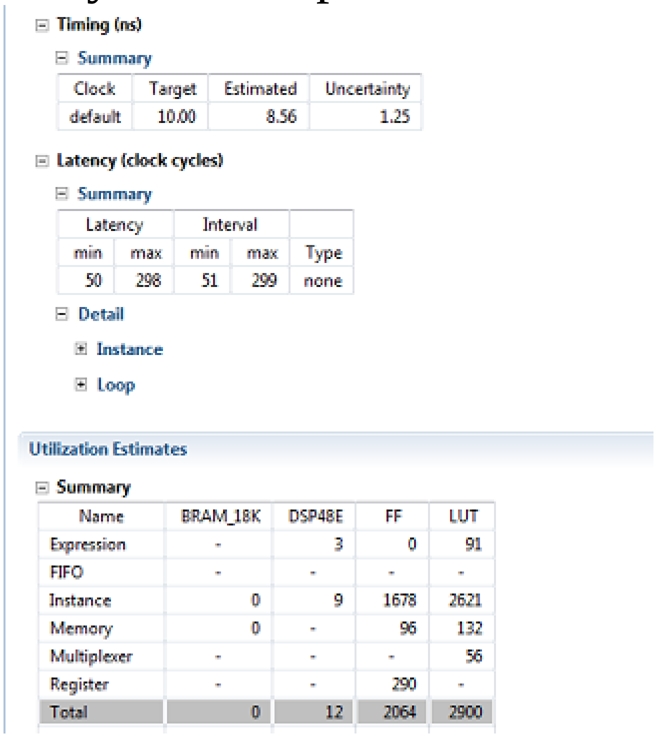
Part 1:

1. Directives used for solution 1:


Vivado HLS - maxpooling_best.prj (C:\Users\Owner\Desktop\ECE1373HLS\labsource\maxpooling_best\maxpooling_best.prj)
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Outline
Directive
8:11 PM
04/03/2015

```
yuv_filter
    in
    out
    Y_scale
    U_scale
    V_scale
rgb2yuv
    Wrgb
    RGB2YUV_LOOP_X
        # HLS loop_tripcount min=0 max=4
        RGB2YUV_LOOP_Y
            # HLS loop_tripcount min=0 max=4
yuv2rgb
    Wyuv
    YUV2RGB_LOOP_X
        # HLS loop_tripcount min=0 max=2
        YUV2RGB_LOOP_Y
            # HLS loop_tripcount min=0 max=2
yuv_scale
    YUV_SCALE_LOOP_X
        # HLS loop_tripcount min=0 max=4
        YUV_SCALE_LOOP_Y
            # HLS loop_tripcount min=0 max=4
call_compute_max_pooling
    ycomp
    ucomp
    vcomp
    yout
    uout
    vout
    YUV_COPY_LOOP_X
```

2. Synthesis report:



**Timing (ns)**

**Summary**

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 10.00 | 8.56 | 1.25 |

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 50 | 298 | 51 | 299 | none |

**Detail**

Instance

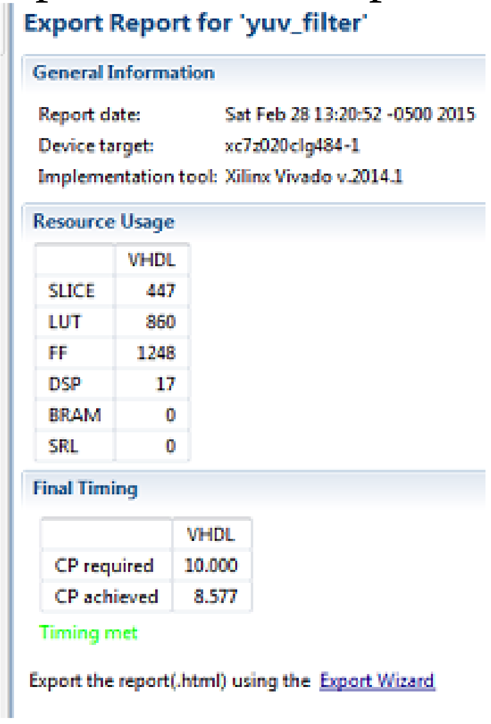
Loop

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | 3 | 0 | 91 |
| FIFO | - | - | - | - |
| Instance | 0 | 9 | 1678 | 2621 |
| Memory | 0 | - | 96 | 132 |
| Multiplexer | - | - | - | 56 |
| Register | - | - | 290 | - |
| Total | 0 | 12 | 2064 | 2900 |

3. Implementation report:



**Export Report for 'yuv_filter'**

**General Information**

| | |
|---|---|
| Report date: | Sat Feb 28 13:20:52 -0500 2015 |
| Device target: | xc7z020clg484-1 |
| Implementation tool: | Xilinx Vivado v.2014.1 |

**Resource Usage**

| | VHDL |
|---|---|
| SLICE | 447 |
| LUT | 860 |
| FF | 1248 |
| DSP | 17 |
| BRAM | 0 |
| SRL | 0 |

**Final Timing**

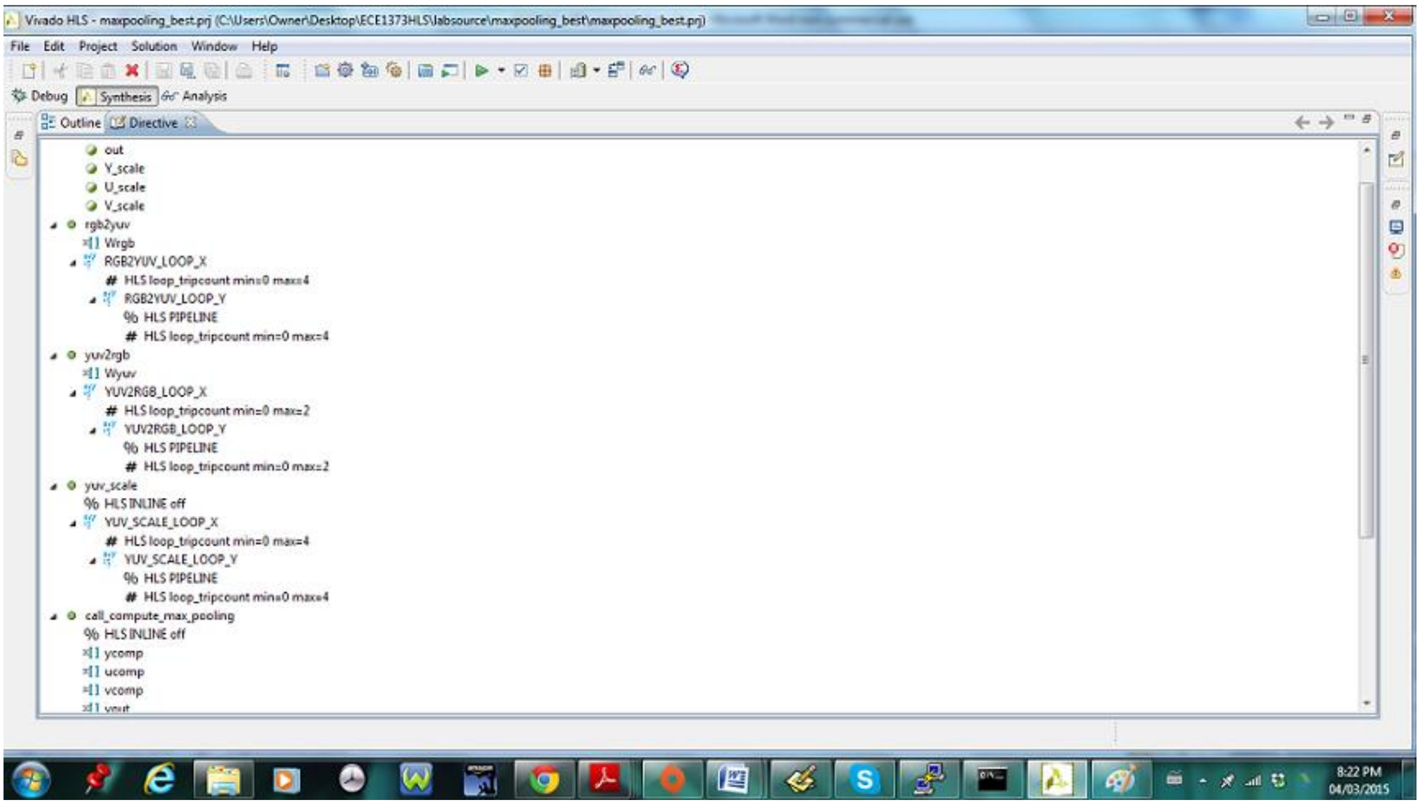
| | VHDL |
|---|---|
| CP required | 10.000 |
| CP achieved | 8.577 |

Timing met

Export the report(.html) using the Export Wizard

## Part 2:

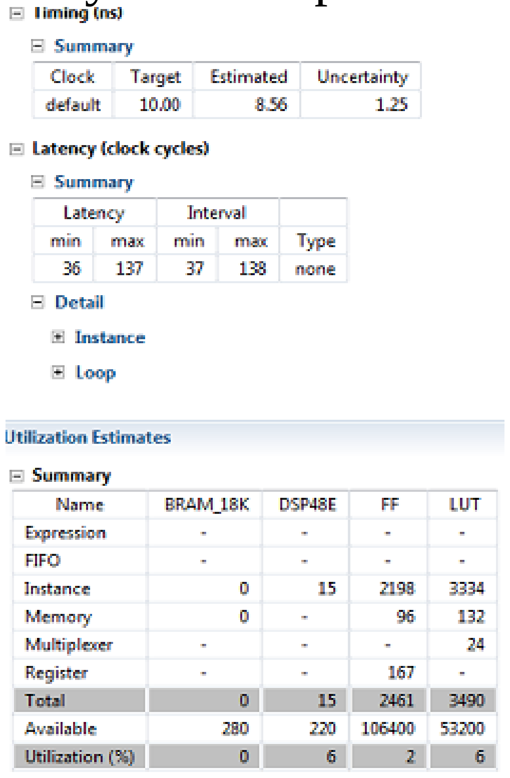
## 1. Directives used for solution 2:



Vivado HLS - maxpooling_best.prj (C:\Users\Owner\Desktop\ECE1373HLS\labsource\maxpooling_best\maxpooling_best.prj)

```
    out
    Y_scale
    U_scale
    V_scale
rgb2yuv
    Wrgb
    RGB2YUV_LOOP_X
        # HLS loop_tripcount min=0 max=4
        RGB2YUV_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=4
yuv2rgb
    Wyuv
    YUV2RGB_LOOP_X
        # HLS loop_tripcount min=0 max=2
        YUV2RGB_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=2
yuv_scale
    % HLS INLINE off
    YUV_SCALE_LOOP_X
        # HLS loop_tripcount min=0 max=4
        YUV_SCALE_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=4
call_compute_max_pooling
    % HLS INLINE off
    ycomp
    ucomp
    vcomp
    vout
```

## 2. Synthesis report:



Timing (ns)

Summary

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 10.00 | 8.56 | 1.25 |

Latency (clock cycles)

Summary

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 36 | 137 | 37 | 138 | none |

Detail

Instance

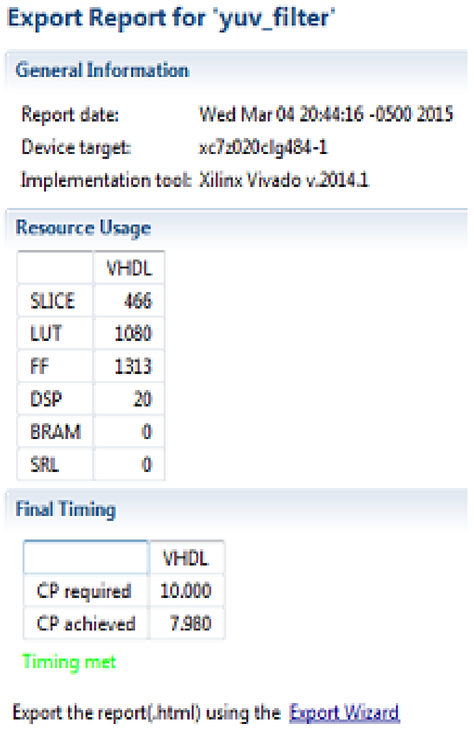
Loop

Utilization Estimates

Summary

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | - | - |
| FIFO | - | - | - | - |
| Instance | 0 | 15 | 2198 | 3334 |
| Memory | 0 | - | 96 | 132 |
| Multiplexer | - | - | - | 24 |
| Register | - | - | 167 | - |
| Total | 0 | 15 | 2461 | 3490 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 6 | 2 | 6 |

## 3. Implementation report:



Export Report for 'yuv_filter'

General Information

| | |
|---|---|
| Report date: | Wed Mar 04 20:44:16 -0500 2015 |
| Device target: | xc7z020clg484-1 |
| Implementation tool: | Xilinx Vivado v.2014.1 |

Resource Usage

| | VHDL |
|---|---|
| SLICE | 466 |
| LUT | 1080 |
| FF | 1313 |
| DSP | 20 |
| BRAM | 0 |
| SRL | 0 |

Final Timing

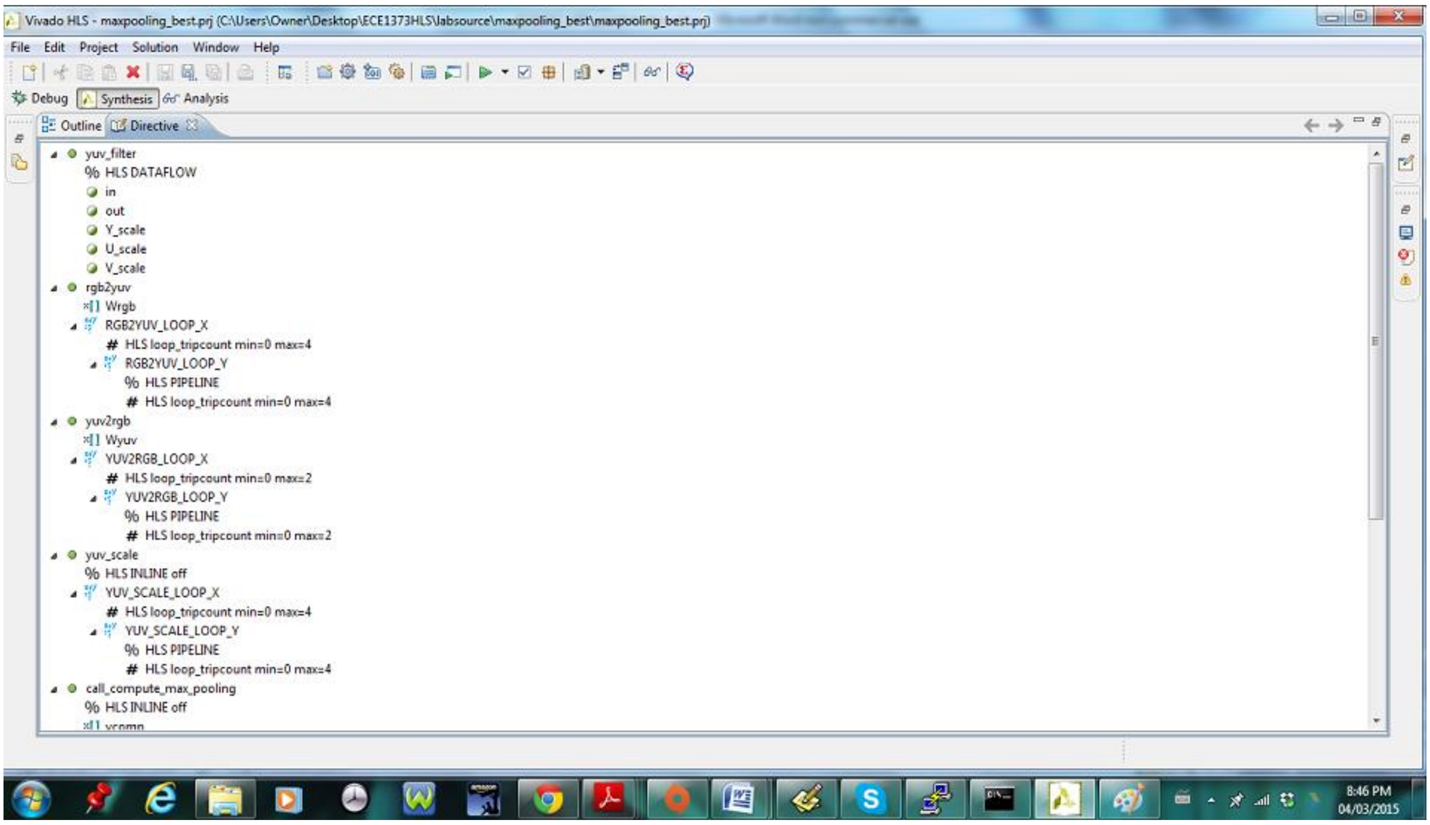
| | VHDL |
|---|---|
| CP required | 10.000 |
| CP achieved | 7.980 |

Timing met

Export the report(.html) using the Export Wizard

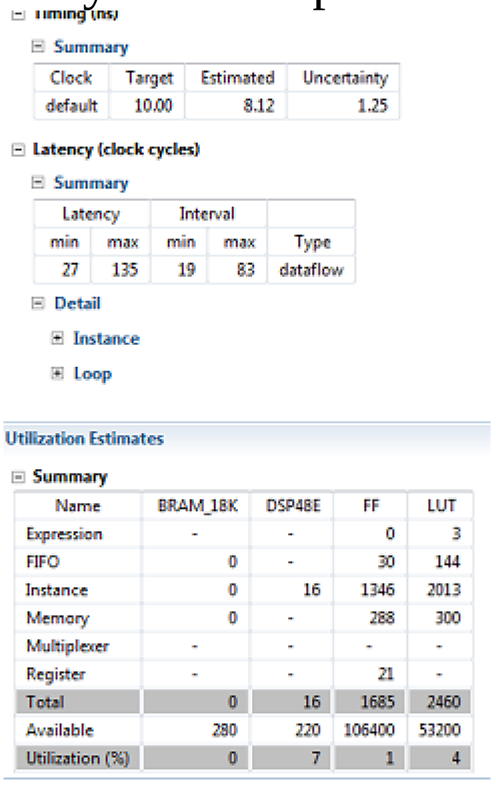
Part 3:

1. Directives used for solution 3:


Vivado HLS - maxpooling_best.prj (C:\Users\Owner\Desktop\ECE1373HLS\labsource\maxpooling_best\maxpooling_best.prj)

```
yuv_filter
    % HLS DATAFLOW
    in
    out
    Y_scale
    U_scale
    V_scale
rgb2yuv
    Wrgb
    RGB2YUV_LOOP_X
        # HLS loop_tripcount min=0 max=4
        RGB2YUV_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=4
yuv2rgb
    Wyuv
    YUV2RGB_LOOP_X
        # HLS loop_tripcount min=0 max=2
        YUV2RGB_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=2
yuv_scale
    % HLS INLINE off
    YUV_SCALE_LOOP_X
        # HLS loop_tripcount min=0 max=4
        YUV_SCALE_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=4
call_compute_max_pooling
    % HLS INLINE off
    vcomp
```

2. Synthesis report:


Timing (ns)

Summary

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 10.00 | 8.12 | 1.25 |

Latency (clock cycles)

Summary

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 27 | 135 | 19 | 83 | dataflow |

Detail

Instance

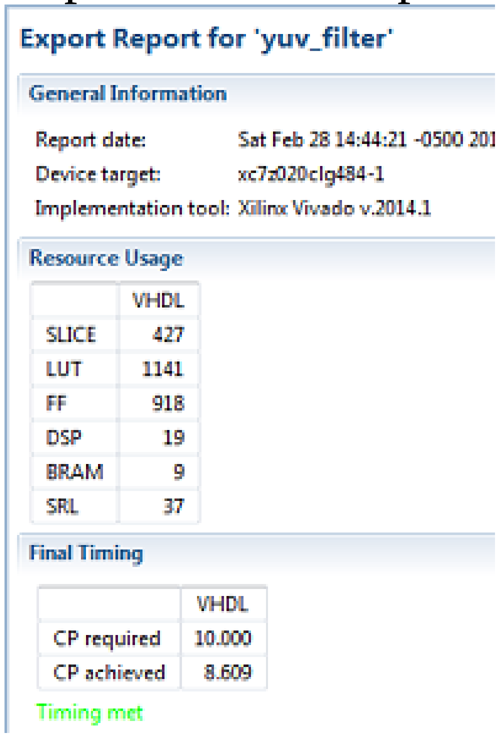
Loop

Utilization Estimates

Summary

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 3 |
| FIFO | 0 | - | 30 | 144 |
| Instance | 0 | 16 | 1346 | 2013 |
| Memory | 0 | - | 288 | 300 |
| Multiplexer | - | - | - | - |
| Register | - | - | 21 | - |
| Total | 0 | 16 | 1685 | 2460 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 7 | 1 | 4 |

3. Implementation report:


Export Report for 'yuv_filter'

General Information

| | |
|---|---|
| Report date: | Sat Feb 28 14:44:21 -0500 201 |
| Device target: | xc7z020clg484-1 |
| Implementation tool: | Xilinx Vivado v.2014.1 |

Resource Usage

| | VHDL |
|---|---|
| SLICE | 427 |
| LUT | 1141 |
| FF | 918 |
| DSP | 19 |
| BRAM | 9 |
| SRL | 37 |

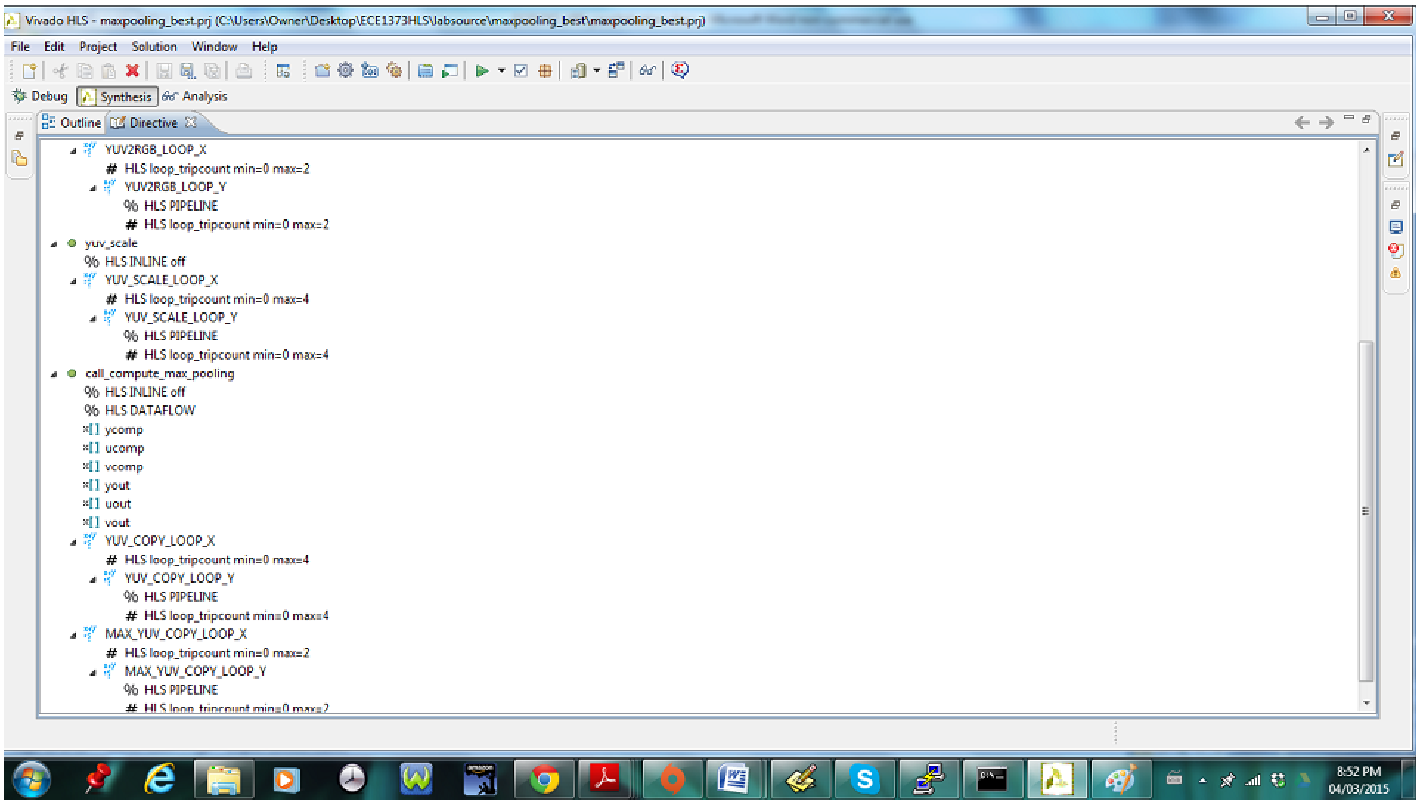
Final Timing

| | VHDL |
|---|---|
| CP required | 10.000 |
| CP achieved | 8.609 |

Timing met

## Part 4:

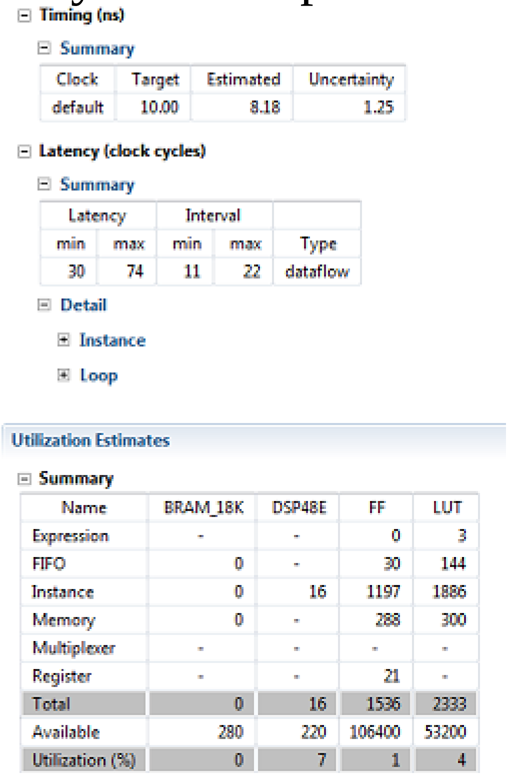
## 1. Directives used for solution 4:



Vivado HLS - maxpooling_best.prj (C:\Users\Owner\Desktop\ECE1373HLS\labsource\maxpooling_best\maxpooling_best.prj)

```
    YUV2RGB_LOOP_X
        HLS loop_tripcount min=0 max=2
      YUV2RGB_LOOP_Y
          HLS PIPELINE
          HLS loop_tripcount min=0 max=2
  yuv_scale
      HLS INLINE off
    YUV_SCALE_LOOP_X
        HLS loop_tripcount min=0 max=4
      YUV_SCALE_LOOP_Y
          HLS PIPELINE
          HLS loop_tripcount min=0 max=4
  call_compute_max_pooling
      HLS INLINE off
      HLS DATAFLOW
      ycomp
      ucomp
      vcomp
      yout
      uout
      vout
    YUV_COPY_LOOP_X
        HLS loop_tripcount min=0 max=4
      YUV_COPY_LOOP_Y
          HLS PIPELINE
          HLS loop_tripcount min=0 max=4
    MAX_YUV_COPY_LOOP_X
        HLS loop_tripcount min=0 max=2
      MAX_YUV_COPY_LOOP_Y
          HLS PIPELINE
          HLS loop_tripcount min=0 max=2
```

## 2. Synthesis report:



**Timing (ns)**

**Summary**

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 10.00 | 8.18 | 1.25 |

**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 30 | 74 | 11 | 22 | dataflow |

**Detail**

Instance

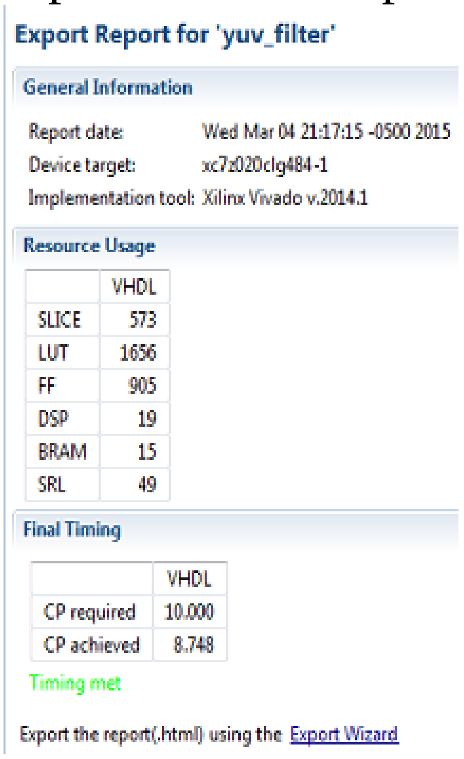
Loop

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 3 |
| FIFO | 0 | - | 30 | 144 |
| Instance | 0 | 16 | 1197 | 1886 |
| Memory | 0 | - | 288 | 300 |
| Multiplexer | - | - | - | - |
| Register | - | - | 21 | - |
| Total | 0 | 16 | 1536 | 2333 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 7 | 1 | 4 |

## 3. Implementation report:



**Export Report for 'yuv_filter'**

**General Information**

| | |
|---|---|
| Report date: | Wed Mar 04 21:17:15 -0500 2015 |
| Device target: | xc7z020clg484-1 |
| Implementation tool: | Xilinx Vivado v.2014.1 |

**Resource Usage**

| | VHDL |
|---|---|
| SLICE | 573 |
| LUT | 1656 |
| FF | 905 |
| DSP | 19 |
| BRAM | 15 |
| SRL | 49 |

**Final Timing**

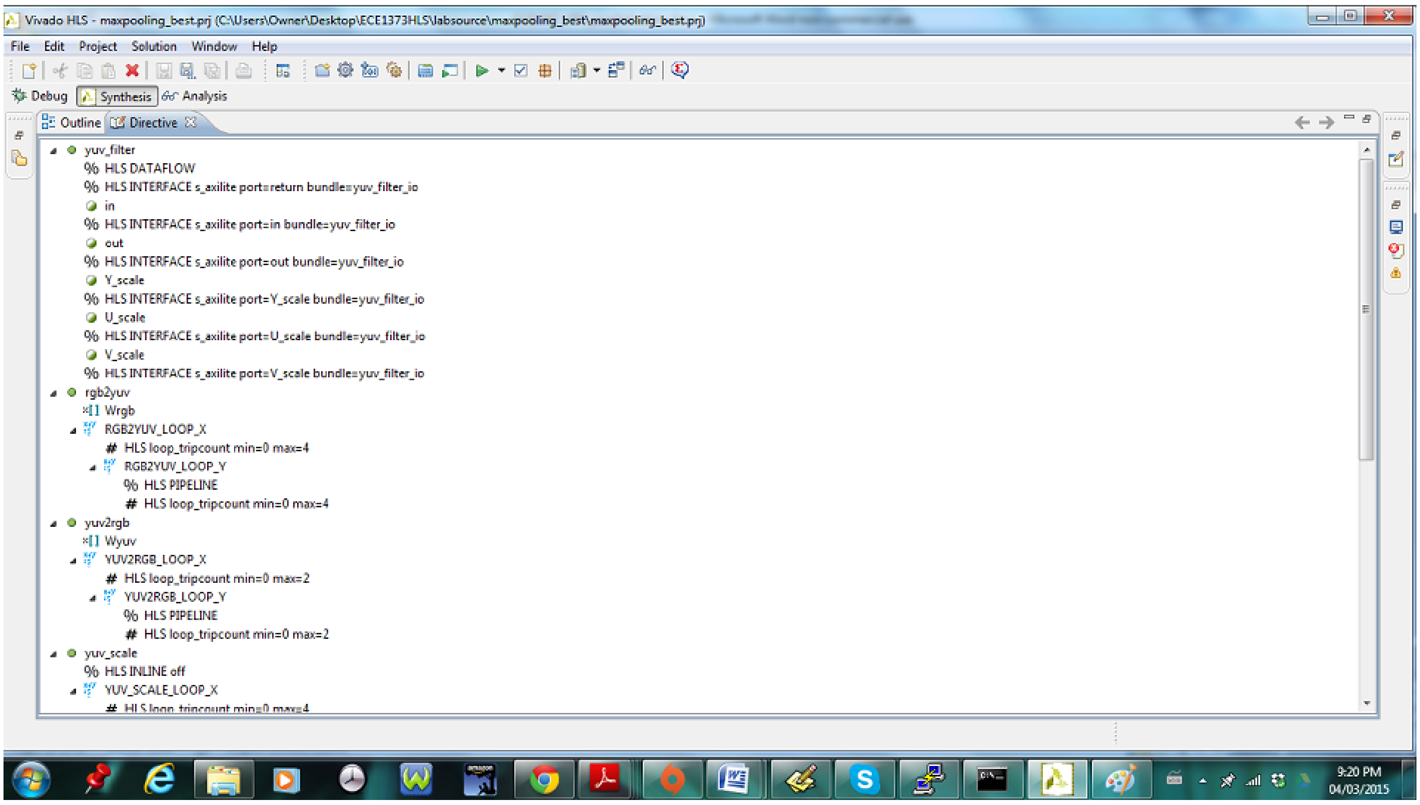
| | VHDL |
|---|---|
| CP required | 10.000 |
| CP achieved | 8.748 |

Timing met

Export the report(.html) using the Export Wizard

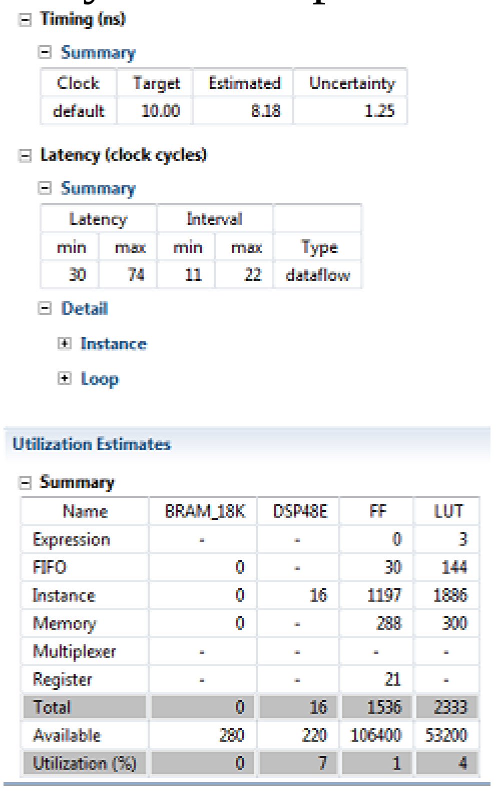
## Part 5:

## 1. Directives used for solution 5:


Vivado HLS - maxpooling_best.prj (C:\Users\Owner\Desktop\ECE1373HLS\labsource\maxpooling_best\maxpooling_best.prj)
File Edit Project Solution Window Help
Debug Synthesis Analysis
Outline Directive
9:20 PM
04/03/2015

```
yuv_filter
    % HLS DATAFLOW
    % HLS INTERFACE s_axilite port=return bundle=yuv_filter_io
    in
    % HLS INTERFACE s_axilite port=in bundle=yuv_filter_io
    out
    % HLS INTERFACE s_axilite port=out bundle=yuv_filter_io
    Y_scale
    % HLS INTERFACE s_axilite port=Y_scale bundle=yuv_filter_io
    U_scale
    % HLS INTERFACE s_axilite port=U_scale bundle=yuv_filter_io
    V_scale
    % HLS INTERFACE s_axilite port=V_scale bundle=yuv_filter_io
rgb2yuv
    Wrgb
    RGB2YUV_LOOP_X
        # HLS loop_tripcount min=0 max=4
        RGB2YUV_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=4
yuv2rgb
    Wyuv
    YUV2RGB_LOOP_X
        # HLS loop_tripcount min=0 max=2
        YUV2RGB_LOOP_Y
            % HLS PIPELINE
            # HLS loop_tripcount min=0 max=2
yuv_scale
    % HLS INLINE off
    YUV_SCALE_LOOP_X
        # HLS loop_tripcount min=0 max=4
```

## 2. Synthesis reports:



Timing (ns)

Summary

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 10.00 | 8.18 | 1.25 |

Latency (clock cycles)

Summary

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 30 | 74 | 11 | 22 | dataflow |

Detail

Instance

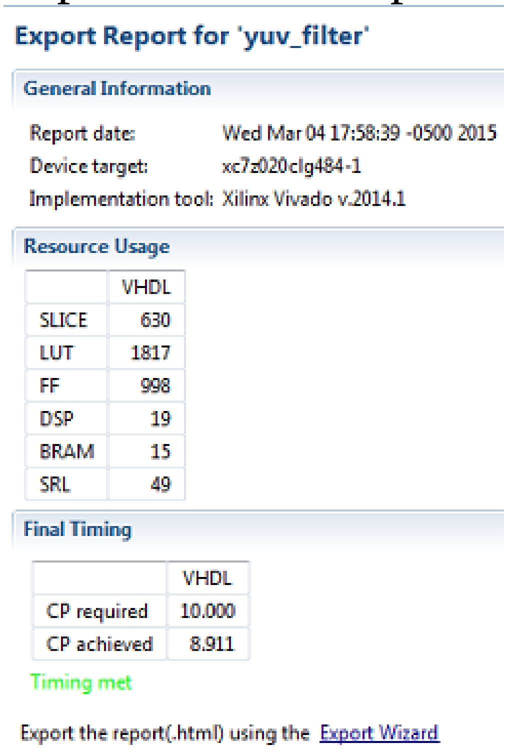
Loop

Utilization Estimates

Summary

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 3 |
| FIFO | 0 | - | 30 | 144 |
| Instance | 0 | 16 | 1197 | 1886 |
| Memory | 0 | - | 288 | 300 |
| Multiplexer | - | - | - | - |
| Register | - | - | 21 | - |
| Total | 0 | 16 | 1536 | 2333 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 7 | 1 | 4 |

## 3. Implementation reports:



Export Report for 'yuv_filter'

General Information

| | |
|---|---|
| Report date: | Wed Mar 04 17:58:39 -0500 2015 |
| Device target: | xc7z020clg484-1 |
| Implementation tool: | Xilinx Vivado v.2014.1 |

Resource Usage

| | VHDL |
|---|---|
| SLICE | 630 |
| LUT | 1817 |
| FF | 998 |
| DSP | 19 |
| BRAM | 15 |
| SRL | 49 |

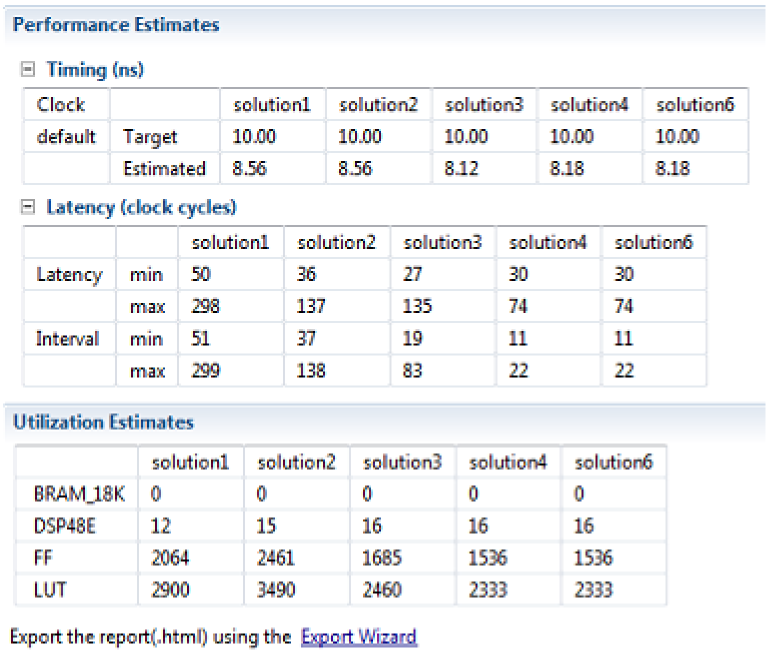
Final Timing

| | VHDL |
|---|---|
| CP required | 10.000 |
| CP achieved | 8.911 |

Timing met

Export the report(.html) using the Export Wizard

Part 6:
Performance and resource estimate comparison across solutions (Here solution 5 is renamed to solution 6).



Performance Estimates

⊟ Timing (ns)

| Clock | | solution1 | solution2 | solution3 | solution4 | solution6 |
|---|---|---|---|---|---|---|
| default | Target | 10.00 | 10.00 | 10.00 | 10.00 | 10.00 |
| | Estimated | 8.56 | 8.56 | 8.12 | 8.18 | 8.18 |

⊟ Latency (clock cycles)

| | | solution1 | solution2 | solution3 | solution4 | solution6 |
|---|---|---|---|---|---|---|
| Latency | min | 50 | 36 | 27 | 30 | 30 |
| | max | 298 | 137 | 135 | 74 | 74 |
| Interval | min | 51 | 37 | 19 | 11 | 11 |
| | max | 299 | 138 | 83 | 22 | 22 |

Utilization Estimates

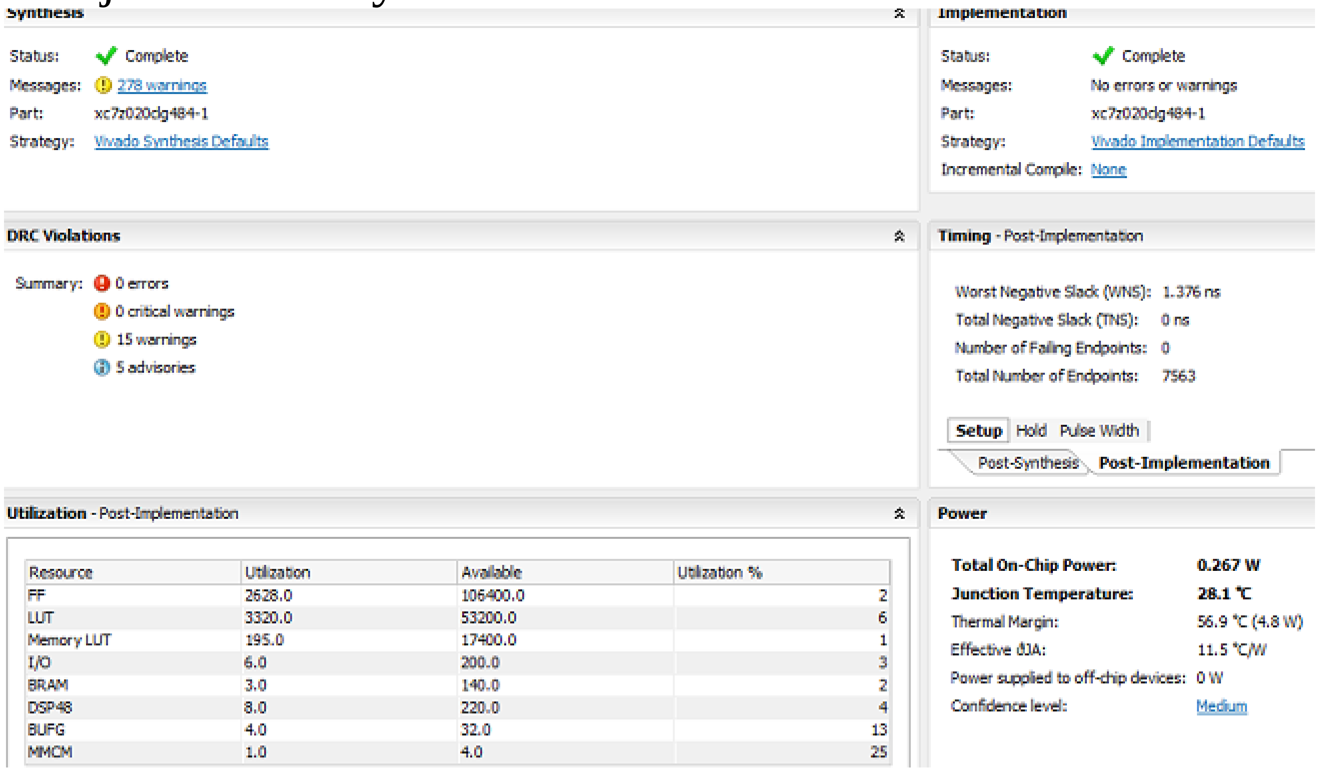
| | solution1 | solution2 | solution3 | solution4 | solution6 |
|---|---|---|---|---|---|
| BRAM_18K | 0 | 0 | 0 | 0 | 0 |
| DSP48E | 12 | 15 | 16 | 16 | 16 |
| FF | 2064 | 2461 | 1685 | 1536 | 1536 |
| LUT | 2900 | 3490 | 2460 | 2333 | 2333 |

Export the report(.html) using the Export Wizard

## APPENDIX D

This appendix contains a snapshot of the project summary and the bitstream generation reports.

1. Project Summary:



**Synthesis**

| | |
|---|---|
| Status: | ✓ Complete |
| Messages: | 278 warnings |
| Part: | xc7z020clg484-1 |
| Strategy: | Vivado Synthesis Defaults |

**Implementation**

| | |
|---|---|
| Status: | ✓ Complete |
| Messages: | No errors or warnings |
| Part: | xc7z020clg484-1 |
| Strategy: | Vivado Implementation Defaults |
| Incremental Compile: | None |

**DRC Violations**

Summary: 0 errors
0 critical warnings
15 warnings
5 advisories

**Timing** - Post-Implementation

| | |
|---|---|
| Worst Negative Slack (WNS): | 1.376 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 7563 |

**Utilization** - Post-Implementation

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| FF | 2628.0 | 106400.0 | 2 |
| LUT | 3320.0 | 53200.0 | 6 |
| Memory LUT | 195.0 | 17400.0 | 1 |
| I/O | 6.0 | 200.0 | 3 |
| BRAM | 3.0 | 140.0 | 2 |
| DSP48 | 8.0 | 220.0 | 4 |
| BUFG | 4.0 | 32.0 | 13 |
| MMCM | 1.0 | 4.0 | 25 |

**Power**

| | |
|---|---|
| **Total On-Chip Power:** | **0.267 W** |
| **Junction Temperature:** | **28.1 °C** |
| Thermal Margin: | 56.9 °C (4.8 W) |
| Effective ϑJA: | 11.5 °C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Medium |

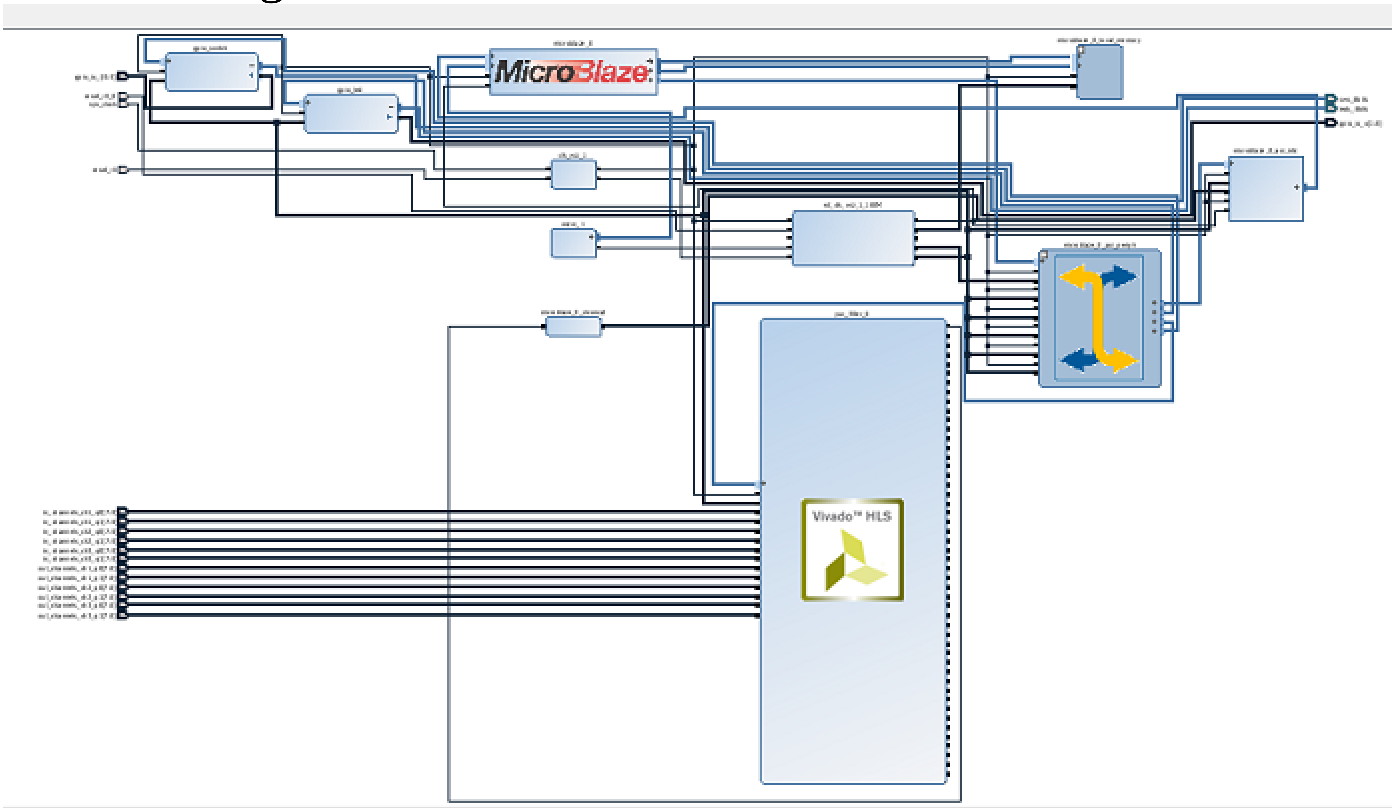
2. Bitstream generation:

```
Running DRC as a precondition to command write_bitstream
INFO: [Drc 23-27] Running DRC with 2 threads
INFO: [Vivado 12-3199] DRC finished with 0 Errors, 15 Warnings, 5 Advisories
INFO: [Vivado 12-3200] Please refer to the DRC report (report_drc) for more information.
Loading data files...
Loading site data...
Loading route data...
Processing options...
Creating bitmap...
Creating bitstream...
Writing bitstream ./system_wrapper.bit...
INFO: [Vivado 12-1842] Bitgen Completed Successfully.
INFO: [Project 1-118] WebTalk data collection is enabled (User setting is ON. Install Setting is ON.).
INFO: [Common 17-186]
'C:/Users/Owner/Desktop/ECE1373HLS/labsource/maxpooling_best/max_pooling_image/max_pooling_image.run
s/impl_1/usage_statistics_webtalk.xml' has been successfully sent to Xilinx on Tue Mar 03 15:17:35 2015. For
additional details about this file, please refer to the WebTalk help file at
C:/Xilinx/Vivado/2014.1/doc/webtalk_introduction.html.
INFO: [Common 17-83] Releasing license: Implementation
write_bitstream: Time (s): cpu = 00:00:52 ; elapsed = 00:02:12 . Memory (MB): peak = 1272.715 ; gain = 362.668
INFO: [Common 17-206] Exiting Vivado at Tue Mar 03 15:17:35 2015...
```

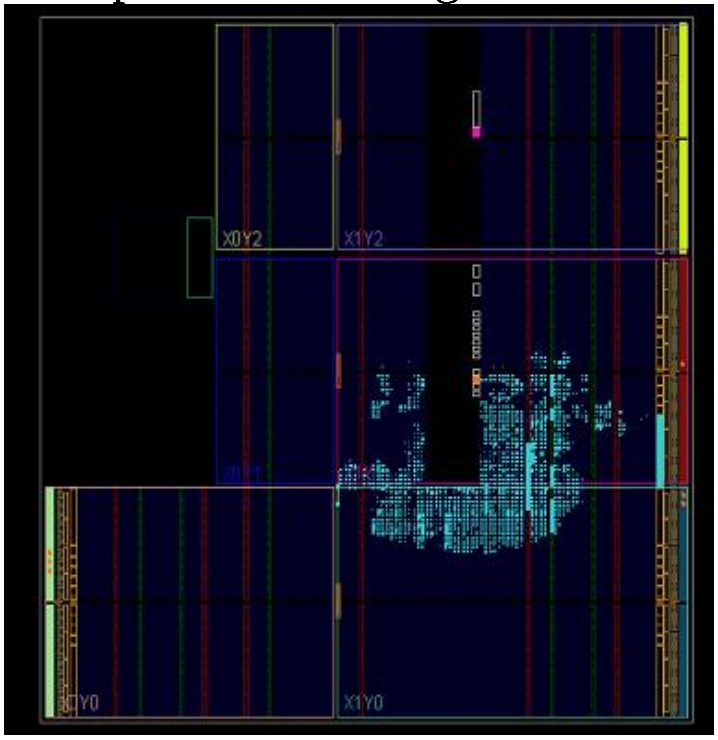
## APPENDIX E

This appendix contains the block diagram, the implemented design, and the constraints file.

1. Block diagram:



2. Implemented design:



3. Constraints in XDC:

```
## Clock signal
set_property -dict { PACKAGE_PIN L18
IOSTANDARD LVCMOS33 } [get_ports {
sys_clock }];
create_clock -add -name sys_clk_pin -period
10.00 -waveform {0 5} [get_ports {sys_clock}];

## Switches
set_property -dict { PACKAGE_PIN AB12
IOSTANDARD LVCMOS33 } [get_ports {
gpio_io_i[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { LOC AB11 IOSTANDARD
LVCMOS33 } [get_ports { reset_rtl_0 }];
#IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { LOC AB9 IOSTANDARD
LVCMOS33 } [get_ports { reset_rtl }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]

set_property CLOCK_DEDICATED_ROUTE
FALSE [get_nets
system_i/clk_wiz_1/inst/clk_in1_system_clk_wi
z_1_0];
```