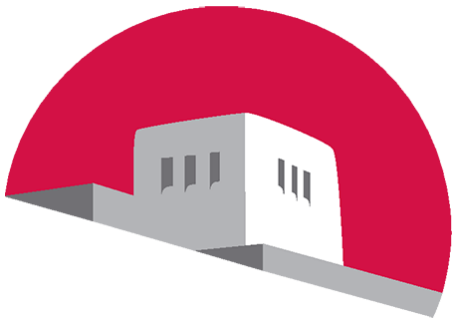
Department Of Electrical And Computer Engineering



School Of Engineering

University Of New Mexico

Tutorial on Partial Reconfiguration of Image Processing Blocks using Vivado and SDK

Nishmitha Naveenchandra Kajekar

nishmi@unm.edu

Department of Electrical and Computer Engineering

University of New Mexico, Albuquerque, NM 87131

Abstract

Partial Reconfiguration (PR) is the modification of an operating FPGA design by loading a partial configuration file which will reduce configuration time and save memory. This is a tutorial which describes how to create and implement two filter design i.e. Sobel Edge Detector and Gaussian Filter that is partially reconfigurable using a modular design technique. MATLAB is used for converting image file to text file and text file back to image file, Vivado IPI and Software Development Kit to create a reconfigurable peripheral using ARM Cortex-A9 processor system on Zynq. Finally a bit-level verification is performed to verify the results. Web tutorials for Partial Reconfiguration is also developed which explains the user step-by-step process on performing PR.

Keywords

Dynamic Partial Reconfiguration, Vivado, SDK, Image Processing, Web Tutorials

1. Introduction

FPGA technology provides the flexibility of on-site programming and re-programming without going through re-fabrication with a modified design. Partial Reconfiguration (PR) takes this flexibility one step further, allowing the modification of an operating FPGA design by loading a partial configuration file. The two Reconfigurable modules used in this tutorial are Sobel Edge Detection and Gaussian Filter.

Sobel operator performs a 2-D spatial gradient measurement on an image. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. The Sobel edge detector uses a pair of 3x3 convolution masks, one estimating the gradient in the x-direction (columns) and the other estimating the gradient in the y-direction (rows). A convolution mask is usually much smaller than the actual image. As a result, the mask is slid over the image, manipulating a square of pixels at a time.

The Gaussian smoothing operator is a 2-D convolution operator that is used to `blur' images and remove detail and noise. In this sense it is similar to the mean filter, but it uses a different kernel that represents the shape of a Gaussian (`bell-shaped') hump. The idea of Gaussian smoothing is to use this 2-D distribution as a `point-spread' function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we can perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero more than about three standard deviations from the mean, and so we can truncate the kernel at this point.

In this tutorial, Vivado IPI and Software Development Kit is used to create a reconfigurable peripheral using ARM Cortex-A9 processor system on Zynq. Vivado IPI is used to create a top-level design, which includes the Zynq processor system as a sub-module. During the PR flow, one Reconfigurable Partition having two Reconfigurable Modules (Sobel and Gaussian filter). Then multiple Configurations are created and run the Partial Reconfiguration implementation flow to generate full and partial bitstreams. ZedBoard is used to verify the design in hardware using a SD card to initially configure the FPGA, and then partially reconfigure the device using the PCAP under user software control.

2. Background

Vivado Design Suite is a software suite produced by Xilinx and introduced in April 2012 for synthesis and analysis of HDL designs which took over the Xilinx ISE with additional features for system on a chip development and offers a new approach for ultra-high productivity with next generation C/C++ and IP-based design. Xilinx recommends Vivado Design Suite for new designs with Ultrascale, Virtex-7, Kintex-7, Artix-7, and Zynq-7000.

The tutorial makes use of ZedBoard for the implementation and verification of the design. ZedBoard is a development kit used by the designers interested in exploring designs using Xilinx Zynq®-7000 All Programmable SoC. The board supports wide range of features and makes it an ideal kit for beginners for prototyping a design.

3. Motivation

Dynamically Reconfigurable Architecture Systems for Time-Varying Image Constraints (DRASTIC) work in ivPCL which is mainly focused on the development of adaptive video processing systems that can change in response to content, their environment, or user needs has motivated to work on this tutorial which will help in getting an idea on how to perform PR in the field of image processing.

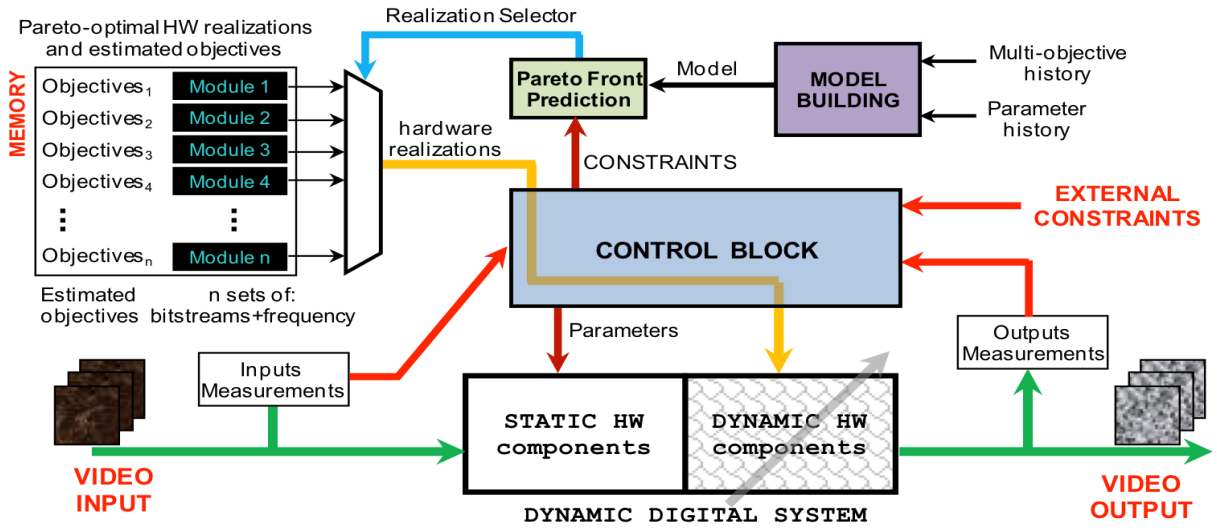


Figure1. DRASTIC Video Processing System

4. Requirements

Software Tools:

- Vivado Design Suite 2015.2
- Xilinx Software Development Kit 2015.2
- MATLAB
- Terminal program (Teraterm)

Hardware Tools:

- ZedBoard (Zynq™ Evaluation and Development)

Licensing:

- Xilinx Partial Reconfiguration

5. Design Description

The purpose of the project is to implement a design that can be dynamically reconfigurable using PCAP resource and PS sub-system. In the initial stage MATLAB is used to convert the image from RGB to grayscale and then write the image into text file. Then this text file is used for perform the required operation. Once the required operation is performed, then the text file is read back into an image file.

For the design to be dynamically reconfigurable the reconfigurable partition is loaded with two reconfigurable modules in the system consisting of one peripheral i.e. myfilter function having to two unique feature which is Sobel Edge Detection and Gaussian Filter.

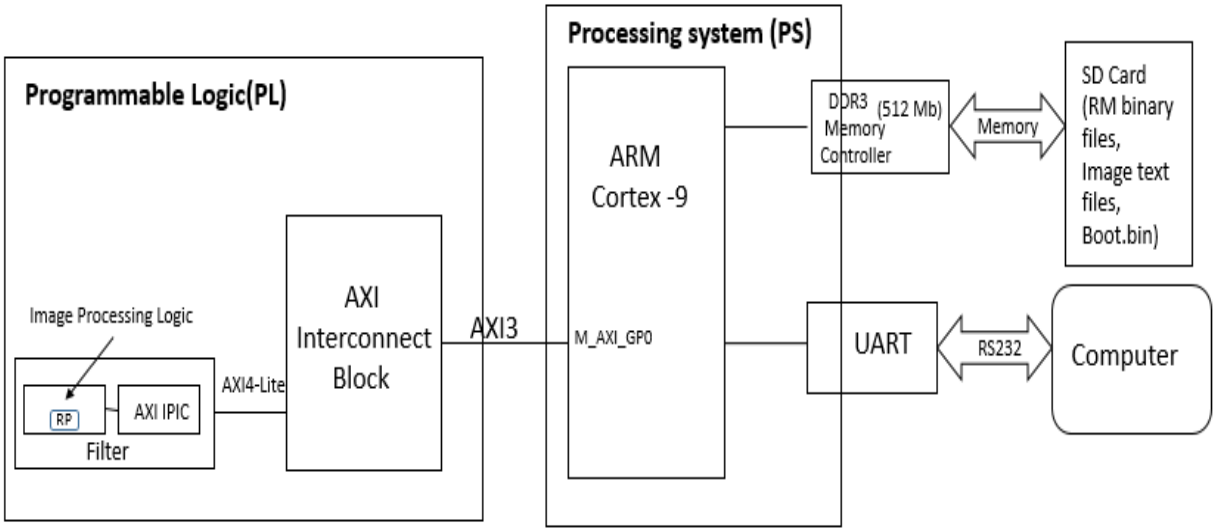


Figure 2. The Main Design

The Design Directory Structure is as follows:

- Bitstreams
  - Generated full and partial bitstreams
- Checkpoint
  - Checkpoints for static and reconfigurable modules
- Implement
  - Implemented configuration for modules
- Sources
  - Contains myfilter processor core in ip repo directory
  - Source files for Sobel and Gaussian filter in reconfig_modules directory
  - Software application in TestApp directory
  - Floorplan constraints is in xdc directory.
- Synth
  - Synthesized checkpoints

Along with it the Design directory also contains few TCL files which will perform tasks like creating process system, performing synthesis, implementation and generation of bitstreams for the design.

6. Design Procedure

The project design procedure is separated into different steps. The Figure 3 gives the general information about the detailed instructions followed in the project.

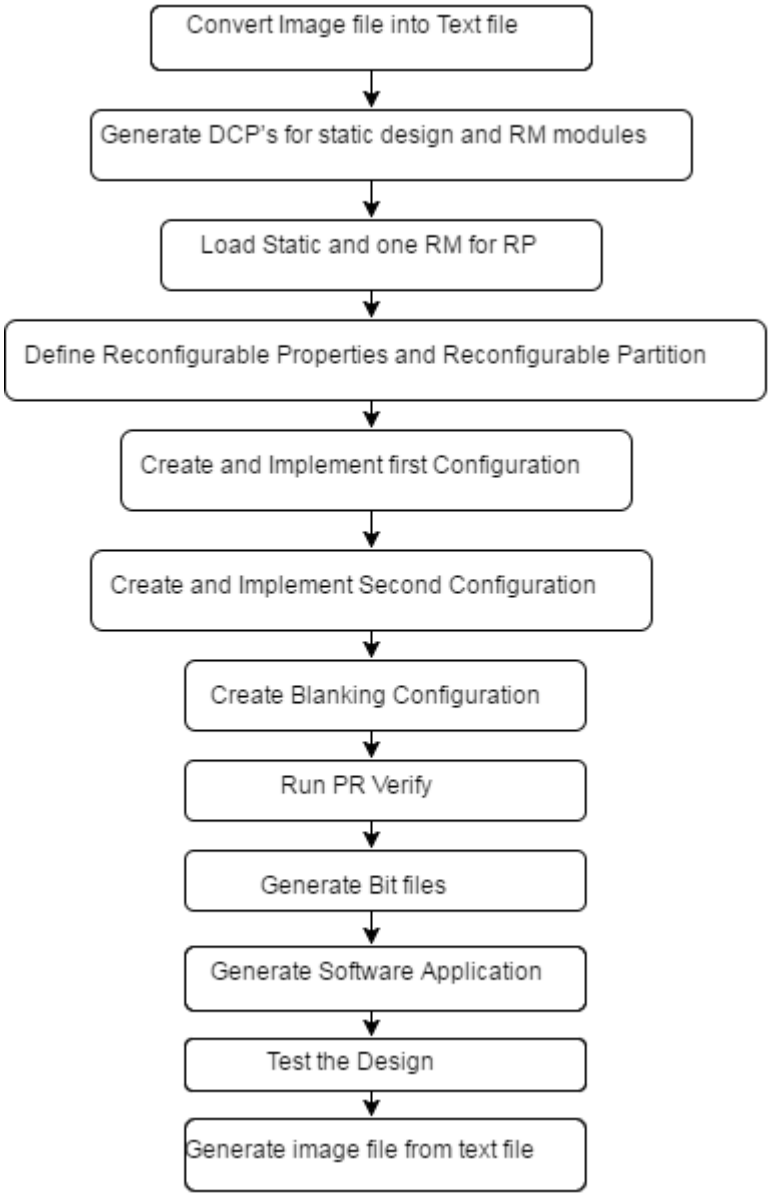


Figure 3. Design Procedure Flow

The initial step of project is to convert image file into text file using MATLAB. Then on executing the ps7_create.tcl the block diagram required to create static DCP will be created. Synthesize the

static module and then synthesize reconfigurable modules too by executing source synth_reconfig_modules.tcl. Once the synthesis is done load one of the RM to the static design and then design the pblock, then implement the design and save the checkpoint. To implement another RM, first clear the previously loaded RM from pblock then load another RM and implement that design too. After both the RM's are implemented load the static DCP and then implement the blanking configuration. Now the design is PR verified to ensure that the static implementation, including interfaces to reconfigurable regions, is consistent across all Configurations. Then generate bitstreams for all the configuration by sourcing generate_bitstreams.tcl.

Once the Bitstreams is generated then generate the software application for the design by using Xilinx SDK application. Then test the design using ZedBoard and verify the design using MATLAB. For the detailed explanation is refer Appendix A.

7. Result

A website on explaining step-by-step process on how to do partial reconfiguration on ZedBoard along with video based Instructions is developed.

After testing the design the results are shown below, it's also bit-level verified on MATLAB for correctness.

Figure 4. Original Image

Results by Testing the Design on ZedBoard:

Figure 5(a). After Applying Sobel Filter

Figure 5(b). After Applying Gaussian Filter

Results by Testing the Design on MATLAB:

Figure 5(c). After Applying Sobel Filter

Figure 5(d). After Applying Gaussian Filter

8. Conclusion

This tutorial helps in understanding steps involved in creating a processor system using Vivado IPI. Generating Full bit stream as well as partial reconfiguration bitstreams along with bin files. Generating the boot image as well as verifying the functionality using ZedBoard

9. Future Work

To perform Partial configuration on large images and on real-time videos.To be able to perform PR for heterogeneous video computing.

Appendix

A. Procedure Guidelines

a. Convert Image file into Text file

i. Open MATLAB Application by selecting Start> All Programs > MATLAB

ii. Write a program to convert image file into text file. I have basically converted the final output image.txt file to have one grayscale pixel per line. The pixels are in row order. The first pixel is from the row 1 col 1 of the array. The second pixel (on the second line) is from row 1 col 2 and so on. (Use the given imagetxt.m file in the OtherRequiredFiles.zip folder.)

iii. Save the image file (image.txt) as sobel.txt and gaussian.txt into the SD card.

b. Generate DCP's for static design and RM modules

i. Start Vivado and execute the Tcl script to create the processor design checkpoint for the static design with one RP.

ii. Open Vivado by selecting Start > All Programs > Xilinx Design Tools > Vivado 2015.2 >Vivado 2015.2

iii. In the Tcl Shell window enter the following command to change to the lab directory to which you have extracted the PRLab.zip and hit Enter.

```
cd E:/PRLab
```

Note: E:/PRLab is the default directory used throughout this tutorial. If you change the directory then you need to change the ps7_create.tcl accordingly.

iv. Generate the PS design executing the provided Tcl script.

```
source ps7_create.tcl
```

This script will create the block design called system, instantiate ZYNQ PS and the provided myfilter IP will then be instantiated.

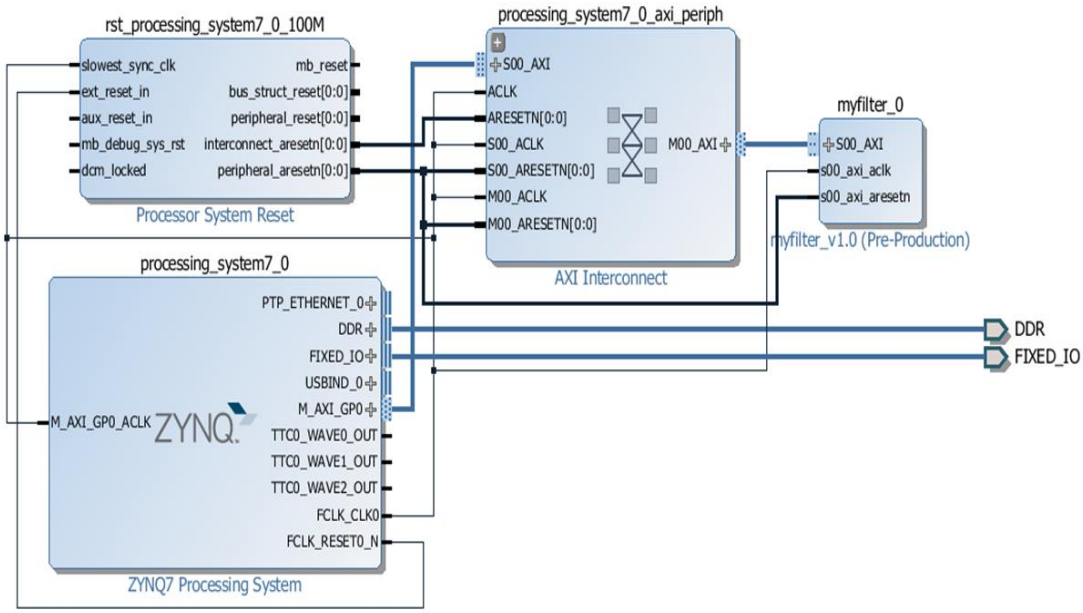


Figure 6. System Block Design

v. Click on Sources tab>right click on filter_design > Select Generate Output Products> Click Generate.

   Note: No changes to be made in the Generate Output Products window.

vi. Click on Sources> right click on filter_design > Select Create HDL wrapper> Click OK.

   Note: No changes to be made in the Create HDL wrapper, let the settings be on Let Vivado manage wrapper and auto update.

vii. Click Run Synthesis under the Synthesis group in the Flow Navigator to run the synthesis process. Wait for the synthesis to complete. When done click cancel.

viii. Using the windows explorer, copy the filter_design_wrapper.dcp file from Filter\Filter.runs\synth_1 into the Synth\Static directory under the current lab directory.

ix. Close the project by typing the close_project command in the Tcl console or selecting File > Close Project.

We need to generate the dcp for each of the RMs. The generated dcps should be stored in appropriate directories so they can be accessed correctly. The dcp files for RM must be in separate directories as their dcp file names will be same for a given RP.

x. In the Tcl Shell window enter the following command to change to the lab directory and hit Enter.

```
cd E:/PRLab
```

xi. Synthesize each of the RMs (two) executing the provided Tcl script.

```
source synth_reconfig_modules.tcl
```

This script will synthesize the HDL files for each RM in an out of context mode and write the design checkpoint (dcp) in the respective destination folder under the Synth directory. After each RM's dcp is generated, the respective design is closed.

c. Load Static and one RM for RP

i. In the Tcl Shell window enter the following command to change to the lab directory and hit Enter.

```
cd E:/PRLab
```

ii. Load the static design using the open_checkpoint command.

```
open_checkpoint
Synth/Static/filter_design_wrapper.dcp
```

You can see the design structure in the Netlist pane with one black box for the filter0 module.

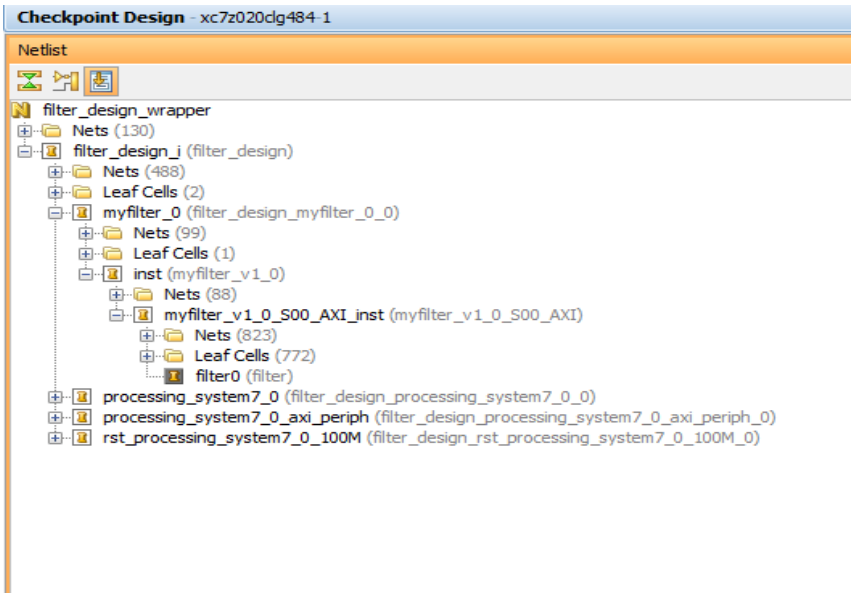


Figure 7. Static Design with Black box

Select the filter0 instance and then select the Properties tab in the Cell Properties window. Note that the IS_BLACKBOX checkbox is checked.

iii. Load one RM for the RP by using the read_checkpoint command.

```
read_checkpoint -cell
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_AX
I_inst/filter0
Synth/reconfig_modules/filter_sobel/sobel_synth.dcp
```

Now you can now see the design structure in the Netlist pane with a reconfigurable module loaded for the filter0 module. Select the filter0 instance and then select the Properties tab in the Cell Properties window. Note that the IS_BLACKBOX checkbox is not checked since a RM design is loaded. Also Note that in the Statistics tab it shows the resources used by loading the RM.

d. Define Reconfigurable Properties and Reconfigurable Partitions

- Define each of the loaded RMs (submodules) as partially reconfigurable by setting the HD.RECONFIGURABLE property using the following commands:

```
set_property HD.RECONFIGURABLE 1 [get_cells
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_A
XI_inst/filter0]
```

- Write the dcp file in the provided Checkpoint directory by using following command:

```
write_checkpoint Checkpoint/top_link_add.dcp
```

Depending on the number of resources used by the all the RMs for RP the RP region must be defined.

- You can execute the following command to define the RP region

```
read_xdc Sources/xdc/fplan.xdc
```

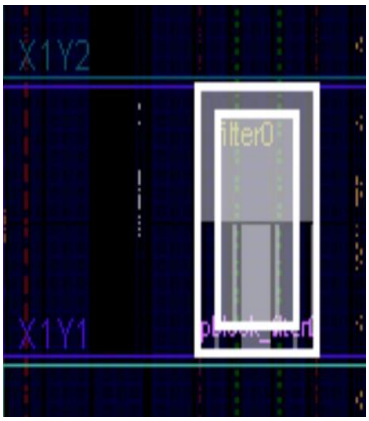


Figure 8. Pblock Design

- Select Tools > Report > Report DRC. Deselect All Rules, select Partial Reconfiguration, and then click OK to run the PR-specific design rules. You should not see any error.

e. Create and Implement first Configuration

i. Optimize, place and route the design by executing the following commands.

```
opt_design
place_design
route_design
```

ii. Save the full design checkpoint.

```
write_checkpoint -force
Implement/Config_sobel/top_route_design.dcp
```

iii. Save checkpoints for the reconfigurable module.

```
write_checkpoint -force -cell
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_AX
```

```
I_inst/filter0
Checkpoint/filter0_sobel_route_design.dcp
```

At this point, a fully implemented partial reconfiguration design from which full and partial bitstreams can be generated is ready for Sobel Filter.

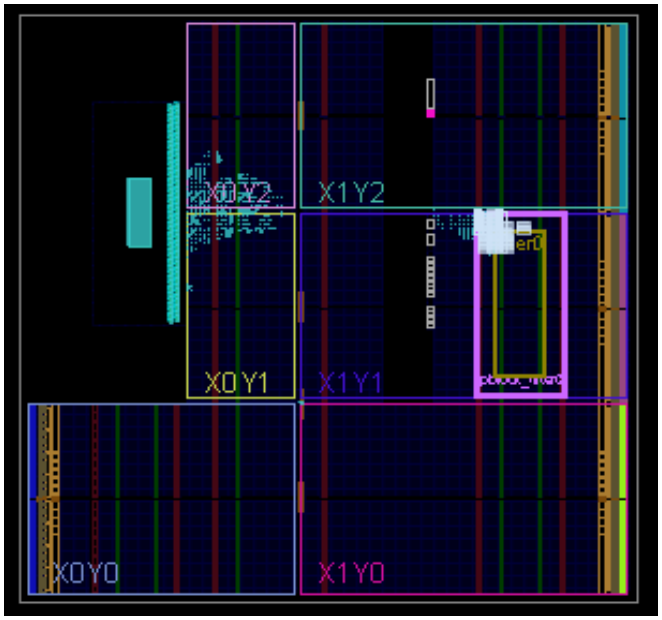


Figure 9. Implemented First Configuration

After the first configuration is created, the static logic implementation will be reused for the rest of the configurations. So it should be saved. But before you save it, the loaded RM should be removed.

iv. Clear out the existing RMs executing the following command

```
update_design -cell
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_AX
I_inst/filter0 -black_box
```

v. Lock down all placement and routing by executing the following command

```
lock_design -level routing
```

vi. Write out the remaining static-only checkpoint by executing the following command

```
write_checkpoint -force
Checkpoint/static_route_design.dcp
```

f. Create and Implement Second Configuration

i. With the locked static design open in memory, read in post-synthesis checkpoint for the second reconfigurable module.

```
read_checkpoint -cell
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_AX
I_inst/filter0
Synth/reconfig_modules/filter_gaussian/gaussian_synt
h.dcp
```

ii. Optimize, place and route the design by executing the following commands.

```
opt_design
place_design
route_design
```

iii. Save the full design checkpoint.

```
write_checkpoint -force
Implement/Config_gaussian/top_route_design.dcp
```

iv. Save the checkpoint for the reconfigurable module.

```
write_checkpoint -force -cell
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_AX
I_inst/filter0
Checkpoint/filter0_gaussian_route_design.dcp
```

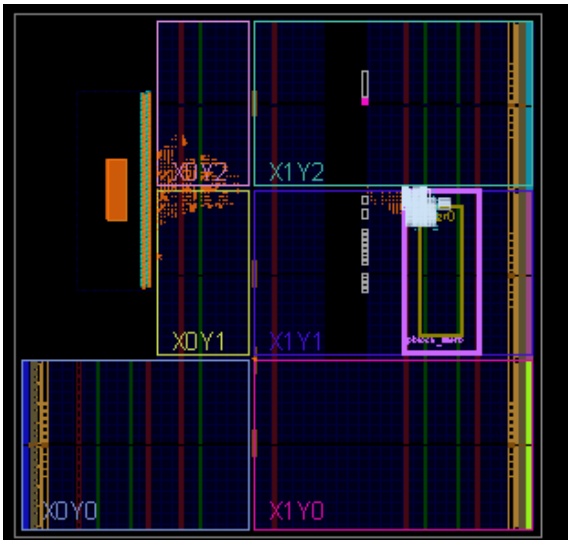


Figure 10. Implemented Second Configuration

v. Close the project

```
close_project
```

g. Create Blanking Configuration

i. Open the static route checkpoint.

```
open_checkpoint Checkpoint/static_route_design.dcp
```

ii. For creating the blanking configuration, use the update_design -buffer_ports command to insert LUTs tied to constants to ensure the outputs of the reconfigurable partition are not left floating.

```
update_design -buffer_ports -cell
filter_design_i/myfilter_0/inst/myfilter_v1_0_S00_AX
I_inst/filter0
```

iii. Now place and route the design. There is no need to optimize the design.

```
place_design
route_design
```

iv. Save the checkpoint

```
write_checkpoint –force
Implement/Config_blank/top_route_design.dcp
```

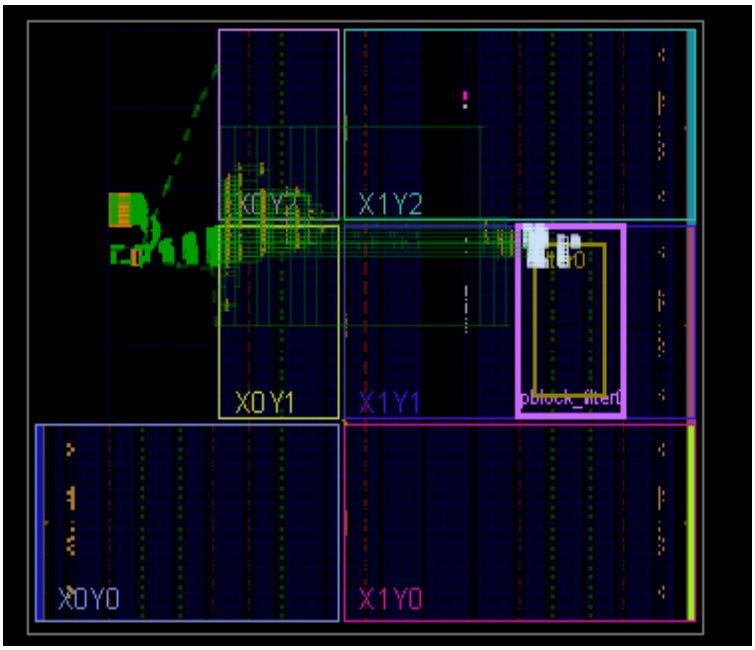


Figure 11. Implemented Blank Configuration

v. Close the project

```
close_project
```

h. Run PR Verify

You must ensure that the static implementation, including interfaces to reconfigurable regions, is consistent across all Configurations. To verify this, you run the PR_Verify utility by the following commands:

```
pr_verify -initial
Implement/Config_sobel/top_route_design.dcp -additional
{Implement/Config_gaussian/top_route_design.dcp
Implement/Config_blank/top_route_design.dcp}

close_project
```

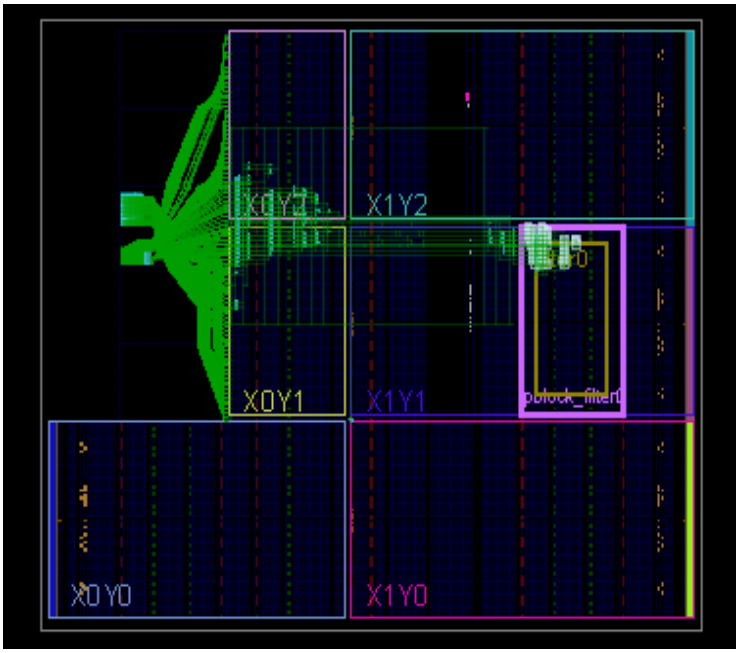


Figure 12. PR Verified Design

i. Generate Bit files

Generate the .bit and .bin files by executing the provided Tcl script.

```
source generate_bitstreams.tcl
```

Notice that bitstreams will be created in the bitstreams folder.

- Config_sobel.bit – This is the power-up, full design bitstream.
- Config_sobel_pblock_filter0_partial.bit – This is the partial bit file for the sobel module.
- sobel.bin – This is the partial bit file for the sobel module in the bin format.
- Config_gaussian.bit – This is the power-up, full design bitstream.
- Config_gaussian_pblock_filter0_partial.bit – This is the partial bit file for the gaussian module.
- gaussian.bin – This is the partial bit file for the sobel module in the bin format.
- blanking.bit – Blank bitstream.

j. Generate Software Application

i. Click on the Open Project link, browse to Filter.xpr and click OK to open the design created in Step 2.

ii. Select File > Export > Export Hardware (In the Export Hardware form, do not check the Include bitstream checkbox) and click OK.

iii. Select File > Launch SDK> OK

iv. Create a Board Support Package

- In SDK, select File > New > Board Support Package. Click Finish with default settings.
- Select xilffs as the FAT file support is necessary to read the partial bit files.
- Click OK.

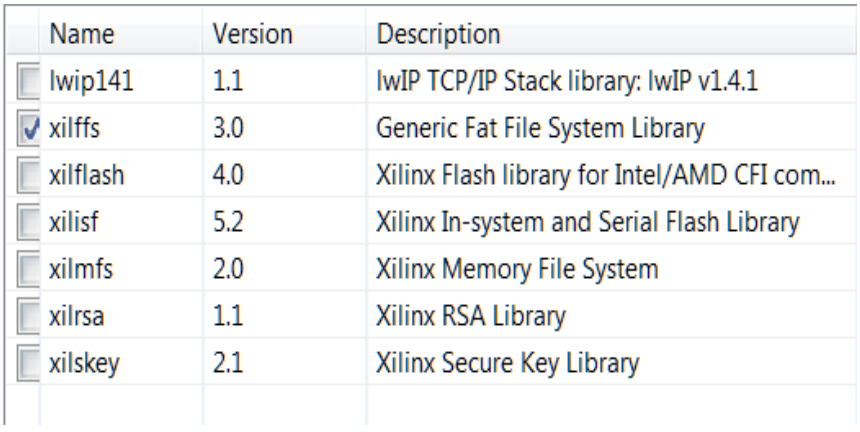


Figure 13. Library support

v. Create an application for the design

- Select File > New > Application Project. Enter FilterTest as the Project Name, and for Board Support Package, choose Use Existing (standalone_bsp_0 should be the only option). Click next, and select Empty Application and click Finish.
- Expand the FilterTest entry in the project view, right-click the src folder, and select Import. Expand General category and double-click on File System. Browse to E:\PR\PR_lab \Sources\TestApp\src (Under extracted folder) and click OK. Select TestApp.c and click finish to add the file to the project. The program should compile successfully.
- Open the Bitstreams folder and verify that the bit length size in the program listed matches the 4x times the .bin file size in bytes (as the program uses the size in words). If different, then change in the program and save it.

vi. Create a zynq_fsbl application

- Select File > New > Application Project. Enter zynq_fsbl as the Project Name, and for Board Support Package, choose Create New. Click next, select Zynq FSBL, and click Finish. This will create the first stage bootloader application called zynq_fsbl.elf

vii. Create a Zynq boot image

- Select Xilinx Tools > Create Zynq Boot Image. Click the Browse button of the Output BIF file path field, browse to E:\PR\PR_lab (Project directory), and then click Save with the output as the default filename.
- Click on the Add button of the Boot image partitions, click the Browse button in the Add Partition form, browse to (Project folder created in the Step 2) E:\PR\PR_lab\Filter\Filter.sdk\zynq_fsbl\Debug directory, select zynq_fsbl.elf and click Open. Note the partition type is bootloader.
- Click again on the Add button of the Boot Image partitions, click the Browse button in the Add Partition form, browse to (Project directory)E:\PR\PR_lab\ Bitstreams directory, select blanking.bit and click Open. Note the partition type is datafile.

- Click again on the Add button of the Boot Image partitions, click the Browse button in the Add Partition form, browse to (Project folder created in Step2) E:\PR\PR_lab\Filter\Filter.sdk\FilterTest\Debug directory, select FilterTest.elf and click Open. Note the partition type is datafile.
- Click Create Image.
- Close the SDK program by selecting File > Exit.

k. Test the Design

Connect the board with micro-USB cable connected to the UART. Place the board in the SD boot mode. Copy the generated BOOT.bin and the bin files on the SD card along with it make sure the SD card has the image text files then place the SD card in the board. Power On the board. Start a terminal emulator program i.e. TeraTerm. Select an appropriate COM port (you can find the correct COM number using the Control Panel). Set the COM port for 115200 baud rate communication.

You will see a in the terminal asking to select the filter operation. If you Enter 1 then Sobel Filter is performed, else if 2 is entered then Gaussian Filter is performed else if 0 is entered then it stops further action.

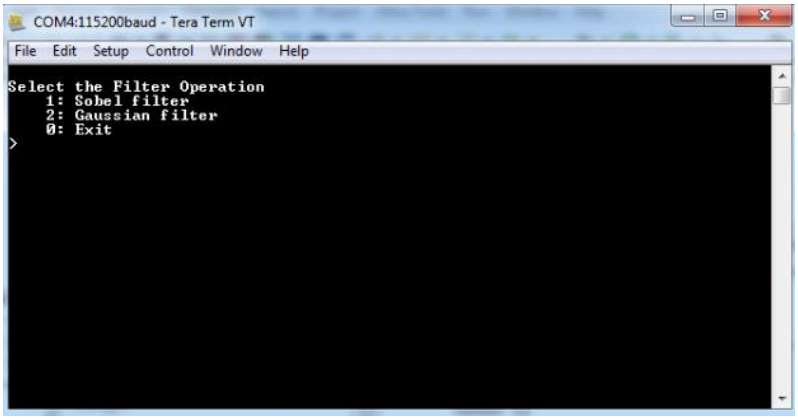


Figure 14. Tera Term

l. Generate image file from text file

i. Create a matlab file to convert text file to image file.

ii. Use the given files in the OtherRequiredFiles.zip folder.

B. Sobel Filter Design

The Sobel Edge Detection Operator is a 3x3 mask.

Horizontal Gradient Operator =

-1	0	1
-2	0	2
-1	0	1

Vertical Gradient Operator =

-1	-2	-1
0	0	0
1	2	1

The mask is convolved over the image to obtain the edge of the image. The mask is convolved and the center element is replaced as the mask operates on the image. The borders of the images are usually blacked out, since they cannot be computed with a 3x3 mask, or partially computed.

a. Software Development

```
// sobel filter and sd card file read/write
for (i=1; i<=image_size-3; i++){
   for (j=1; j<=image_size-3; j++) {
      k=i*image_size + j;
      *(filter_addr+0) = image[k-(image_size+1)]; //p0
      *(filter_addr+1) = image[k-1];              //p1
     *(filter_addr+2) = image[k+(image_size-1)];  //p2
     *(filter_addr+3) = image[k-image_size];      //p3
     *(filter_addr+4) = image[k+image_size];      //p5
```

```
        *(filter_addr+5) = image[k-(image_size-1)];   //p6
        *(filter_addr+6) = image[k+1];                //p7
       *(filter_addr+7) = image[k+(image_size+1)];  //p8
       sum[i][j] = *(filter_addr+8);                 //q
       if (sum[i][j] > 200) sum[i][j] = 255;
       else sum[i][j] = 0;
       }
   }
```

b. Verilog

```
module filter(
    input [11:0]p0,
    input [11:0]p1,
    input [11:0]p2,
    input [11:0]p3,
    input [11:0]p5,
    input [11:0]p6,
    input [11:0]p7,
    input [11:0]p8,
    output [11:0]q);

wire signed [13:0] gx,gy
wire signed [13:0] abs_gx,abs_gy;//it is used to find the absolute value of gx and gy
wire [13:0] sum;//the max value is 255*8. Here no sign bit needed.

assign gx=((p2-p0)+((p5-p3)<<1)+(p8-p6));//sobel mask for gradient in horiz. direction
```

```
assign gy=((p6-p0)+((p7-p1)<<1)+(p8-p2));//sobel mask for gradient in vertical direction

assign abs_gx = (gx[10]? ~gx+1 : gx);// to find the absolute value of gx.
assign abs_gy = (gy[10]? ~gy+1 : gy);// to find the absolute value of gy.

assign sum = (abs_gx+abs_gy); // finding the sum

assign q[7:0]  = (|sum[13:8])?8'hff : sum[7:0];// to limit the max value to 255
assign q[11:8] = 0;

endmodule
```

C. Gaussian Filter Design

The Gaussian smoothing can be performed using standard convolution methods. The 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then convolving with another 1-D Gaussian in the y direction. The kernel used here for computation is

Kernel = [1 17 78 128 78 17 1]

This 1-D x component kernel that would be used to compute horizontal filter. The 1-D y component is exactly the same but is oriented vertically to compute vertical filter.

a. Software Development

```
// Gaussian filter and sd card file read/write
//Horizontal filter
for (i=4;i<=image_size-8;i++){
  for (j=4; j<=image_size-8;j++) {
      k=i*image_size + j;
      *(filter_addr+0) = image[k-3];        //p0
       *(filter_addr+1) = image[k-2];        //p1
```

```
        *(filter_addr+2) = image[k-1];         //p2
        *(filter_addr+3) = image[k];           //p3
        *(filter_addr+4) = image[k+1];         //p5
        *(filter_addr+5) = image[k+2];         //p6
        *(filter_addr+6) = image[k+3];         //p7
        *(filter_addr+7) = 0;                  //p8
        image_H[i][j] = *(filter_addr+8);    //q
        }
    }
    //Vertical filter
    for (i=4;i<=image_size-8;i++){
        for (j=4; j<=image_size-8;j++) {
        k=i*image_size + j;
        *(filter_addr+0) = image_V[k-3*image_size];  //p0
        *(filter_addr+1) = image_V[k-2*image_size];  //p1
        *(filter_addr+2) = image_V[k-image_size];      //p2
        *(filter_addr+3) = image_V[k];                 //p3
        *(filter_addr+4) = image_V[k+image_size];     //p5
        *(filter_addr+5) = image_V[k+2*image_size]; //p6
        *(filter_addr+6) = image_V[k+3*image_size]; //p7
        *(filter_addr+7) = 0;                          //p8
        image_VH[i][j] = *(filter_addr+8);             //q
        }
    }
```

b. Verilog

```
module filter(
   input [11:0] p0,
   input [11:0] p1,
   input [11:0] p2,
   input [11:0] p3,
```

```
    input [11:0] p5,
    input [11:0] p6,
    input [11:0] p7,
    input [11:0] p8,
    output [11:0] q);

parameter unsigned[11:0]coefa = 1;
parameter unsigned[11:0]coefb = 17;
parameter unsigned[11:0]coefc = 78;
parameter unsigned[11:0]coefd = 128;
parameter unsigned[11:0]coeff = 78;
parameter unsigned[11:0]coefg = 17;
parameter unsigned[11:0]coefh = 1;

wire  unsigned[19:0]delay_tapa = p0 * coefa;
wire  unsigned[19:0]delay_tapb = p1 * coefb;
wire  unsigned[19:0]delay_tapc = p2 * coefc;
wire  unsigned[19:0]delay_tapd = p3 * coefd;
wire  unsigned[19:0]delay_tapf = p5 * coeff;
wire  unsigned[19:0]delay_tapg = p6 * coefg;
wire  unsigned[19:0]delay_taph = p7 * coefh;
wire  unsigned[19:0]delay_tapi = p8;

wire  unsigned[31:0]addresult = delay_tapa + delay_tapb + delay_tapc + delay_tapd +
delay_tapf + delay_tapg + delay_taph + delay_tapi;

assign q = addresult[19:8];
endmodule
```

References

1. Vivado Design Suite Tutorial: Partial Reconfiguration (UG947)
2. Partial Reconfiguration User Guide (UG702) - For ISE Design Tool
3. Vivado Design Suite Tcl Command Reference Guide (UG835)
4. Vivado Design Suite User Guide: Designing with IP (UG896)
5. Partial Reconfiguration User Guide (UG909)
6. Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite (XAPP1231)
7. Xilinx University Program on Partial Reconfiguration Flow on Zynq using Vivado
8. Image Processing Learning Resources - Gaussian Smoothening
   http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm
9. Edge Detection in FPGA using Sobel Operator by Arun Prabhakar
10. Tutorials developed and taught by Prof. Daniel Llamocca
    http://www.secs.oakland.edu/~llamocca/EmbSysZynq.html