XAPP890 (v1.0) September 25, 2012

Zynq All Programmable SoC Sobel Filter Implementation Using the Vivado HLS Tool

Author: Fernando Martinez Vallina, Christian Kohn, and Pallav Joshi

Summary

This application note describes how to generate the Sobel edge detection filter in the Zynq™-7000 All Programmable SoC ZC702 Base Targeted Reference Design (TRD) using the Vivado™ High-Level Synthesis (HLS) tool. The techniques described in this application note present the fundamental flow for integrating an IP block generated by the Vivado HLS tool into a Zynq AP SoC-based system.

Introduction

The Vivado HLS tool provides a methodology for migrating algorithms from a processor onto the FPGA logic. In the context of Zynq devices, this means moving code from the ARM® dual-core Cortex™-A9 processor to the FPGA logic for acceleration. The code implemented with the HLS tool in hardware represents the computational bottleneck of the algorithm. This bottleneck can be discovered through code profiling. For information about profiling processor code, refer to *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design* [Ref 1].

For this application note, the computational bottleneck is the Sobel edge detection algorithm running at 60 frames per second on a resolution of 1080p. This application note describes how to take a C description of the algorithm, generate RTL with the HLS tool, and integrate the resulting block into a hardware system design. This document describes the following aspects of generating the Sobel edge detection core for the ZC702 Base TRD:

- Sobel Edge Detection Algorithm Overview
- Implementing the Algorithm in the HLS Tool
- Creating a Linux Driver for the IP Core Generated by the HLS Tool
- Integrating the System into the Zynq Base TRD

Programming Environment Specifics

This application note assumes that the user has some general knowledge of the Vivado HLS tool and XPS. For more information on these tools, see *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design* [Ref 1] and *Vivado Design Suite User Guide: High-Level Synthesis* [Ref 2].

The design in this application note requires the use of the Vivado HLS tool 2012.2 and ISE® Design Suite: System Edition 14.2.

Required Hardware

The reference design in this application note requires the Zynq-7000 ZC702 board. For more information on this hardware platform, refer to the Zynq Base TRD wiki page [Ref 3].

Sobel Edge Detection Algorithm Overview

Sobel edge detection is a classical algorithm in the field of image and video processing for the extraction of object edges. Edge detection using Sobel operators works on the premise of computing an estimate of the first derivative of an image to extract edge information [Ref 4]. By computing the x and y direction derivatives of a specific pixel against a neighborhood of surrounding pixels, it is possible to extract the boundary between two distinct elements in an image. Due to the computational load of calculating derivatives using squaring and square root operators, fixed coefficient masks have been adopted as a suitable approximation in computing the derivative at a specific point. In the case of Sobel, the masks used are shown in Table 1.

Table 1: **Sobel Operator Masks**

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

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

The visual effect of applying a Sobel operation on a frame of video or an image is shown in Figure 1.

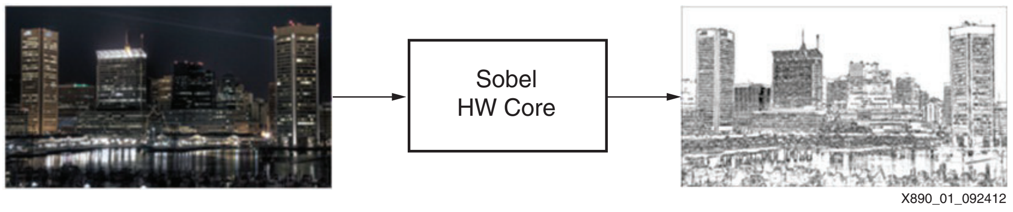


Figure 1: **Sobel Edge Detection**

There are different ways of implementing the C code for Sobel edge detection depending on the target execution platform and performance requirements. For the implementation described in this document, the starting point for the HLS tool is the following code:

```
for(i = 0; i < height, i++){
  for(j=0; j < width; j++){
    x_dir = 0;
    y_dir = 0;
    if((i > 0 && (i < (height-1)) && (j > 0) && (j < (width-1))){
    for(rowOffset = -1; rowOffset <= 1; rowOffset++){
      for(colOffset = -1; colOffset <=1; colOffset++){
        x_dir = x_dir + input_image[i+rowOffset][j+colOffset]* Gx[1+rowOffset][1+colOffset];
        y_dir = y_dir + input_image[i+rowOffset][j+colOffset]* Gy[1+rowOffset][1+colOffset];
      }
    }
  edge_weight = ABS(x_dir) + ABS(y_dir);
  output_image[i][j] = edge_weight;
}
```

This C code shows that applying the masks of Figure 1 to an image creates a 2-dimensional traversal in memory along a 3 x 3 window. Because this window accesses different rows in the image, the benefits of memory locality to reduce memory bandwidth requirements are limited. Furthermore, to achieve the target performance of 60 frames per second at 1080p, the inner for-loops computing x_dir and y_dir must be fully unrolled.

Complete unrolling of the inner for-loop structure in the context of the HLS tool has the effect of exposing the entire datapath for the computation of the gradient at a pixel to the compiler. It also exposes all the memory accesses required to complete the computation as fast as possible. In this case, achieving maximum performance requires nine read operations to the memory storing the input image to construct the 3 x 3 window on which the gradient is computed. This creates a memory bandwidth bottleneck, which must be addressed at the algorithmic level.

As described in *Implementing Memory Structures for Video Processing in the Vivado HLS Tool* [Ref 5], the algorithm targeted at the HLS tool must have a notion of a proper memory architecture for achieving performance. In the context of Sobel edge detection, a proper memory architecture requires only a single access to the global memory storing the input image and a local buffer storage to create the 3 x 3 computation window. Line buffers and other memory structures must be part of the C code given to the HLS tool in order to generate the correct design. *Implementing Memory Structures for Video Processing in the Vivado HLS Tool* provides the details on how to create line buffers and memory structures for image and video processing. The code in the reference design for this application note has been modified to express tiered memory architecture to meet the performance requirements of the design.

Implementing the Algorithm in the HLS Tool

The instructions in this section describe how to generate the Sobel edge detection IP core using the HLS tool. For more information on the tool directives used to optimize the design, refer to *Vivado Design Suite User Guide: High-Level Synthesis* [Ref 2].

1. Unzip the reference design file (`xapp890.zip`).
2. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Vivado 2012.2** > **Vivado HLS** to open the Vivado HLS tool (Figure 2).

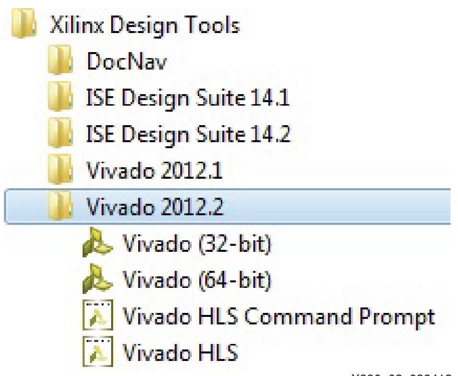
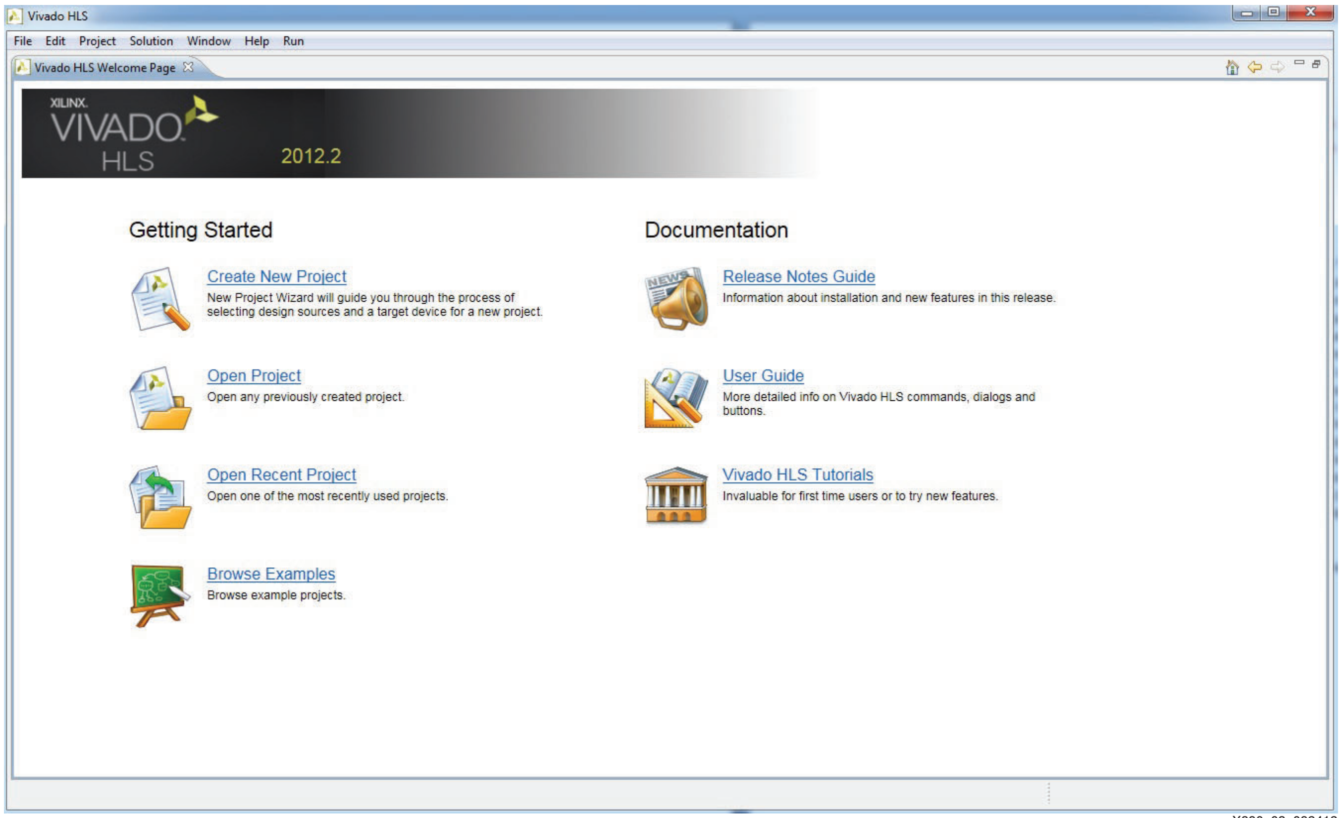


Figure 2: **Vivado HLS Tool in Windows Start Menu**

3. On the Vivado HLS welcome page, click **Open Project** in the Getting Started group (Figure 2).



New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Open any previously created project.

Open one of the most recently used projects.

Browse example projects.

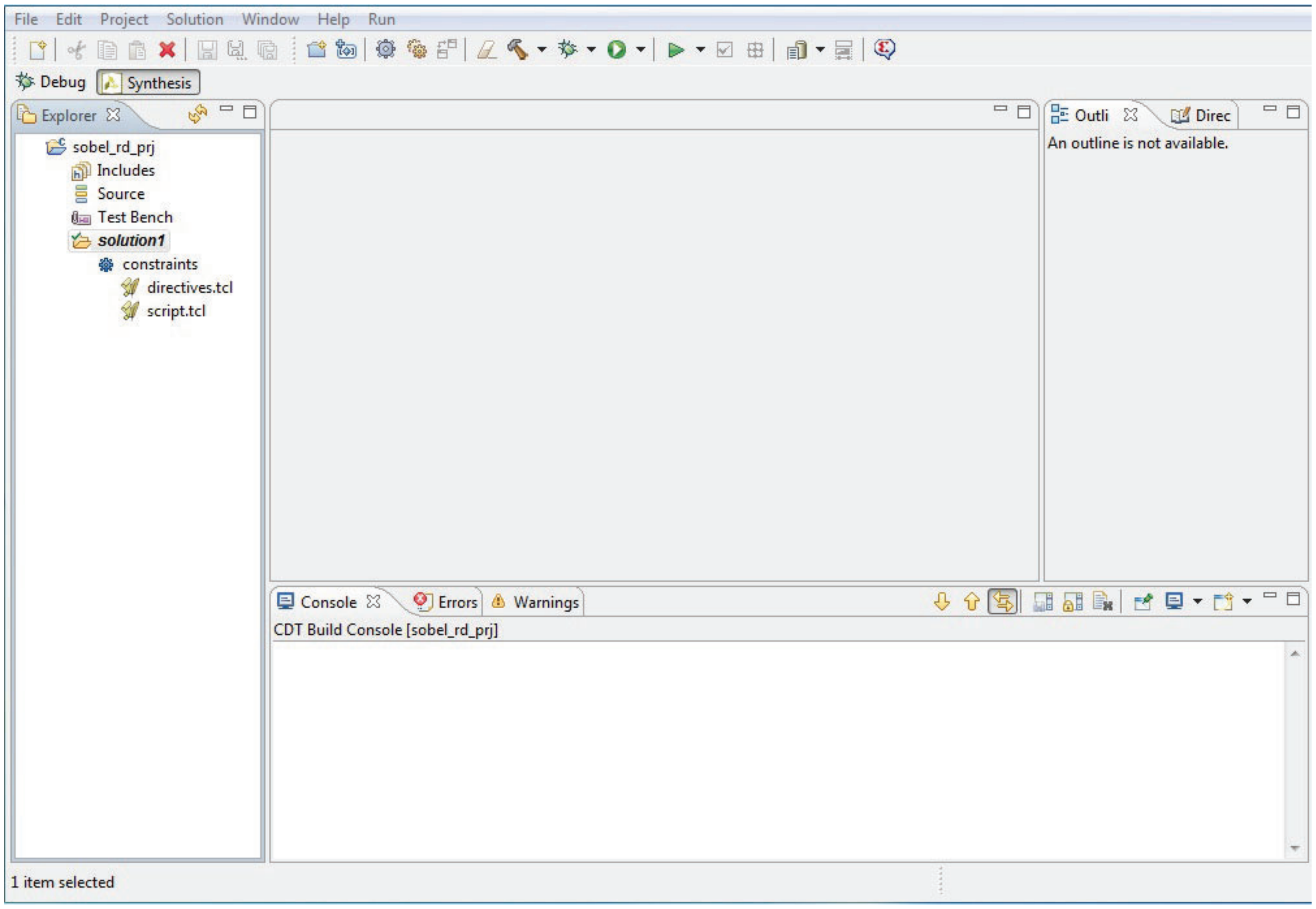
Information about installation and new features in this release.

More detailed info on Vivado HLS commands, dialogs and buttons.

Invaluable for first time users or to try new features.

Figure 3: **Vivado HLS Welcome Page**

4. Browse to the `xapp890\sobel_rd_prj` directory and click **OK**. The Vivado HLS window looks like Figure 4.



An outline is not available.

CDT Build Console [sobel_rd_prj]

X890_04_092412

Figure 4: **Vivado HLS Window**

5. Click the **Synthesis** button to generate the RTL for the algorithm.

6. Click the **Export** button to package the RTL for EDK.

7. In the dialog box, select **Pcore for EDK** and click **OK** (Figure 5).

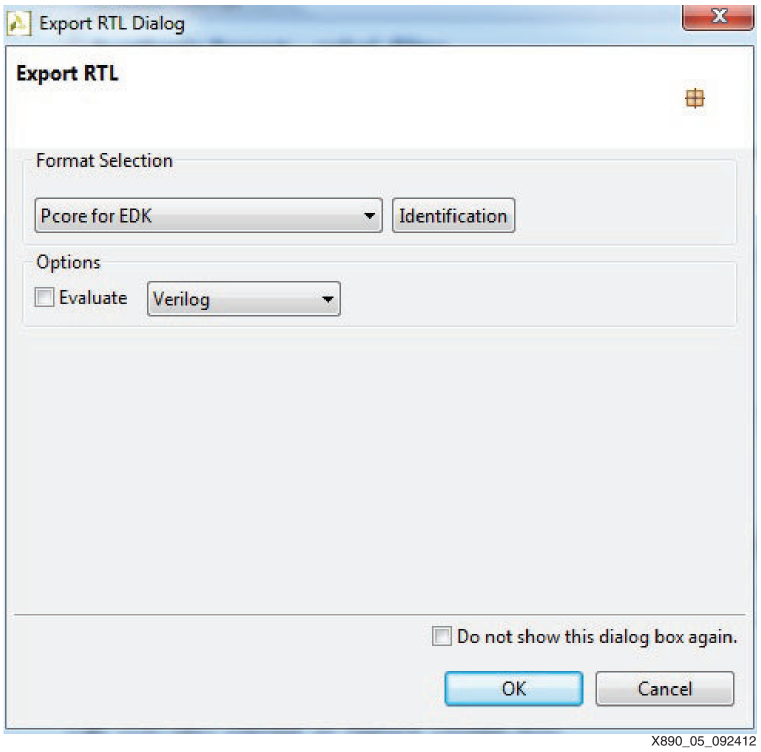


Figure 5: **Export RTL Dialog Box**

8. The pcore for the Zynq TRD is located in the directory `xapp890\sobel_rd_prj\solution1\impl\pcores`.

Creating a Linux Driver for the IP Core Generated by the HLS Tool

The HLS tool creates a standalone or bare-metal driver for all generated IP cores. A complete description of the bare metal driver can be found in *Processor Control of Vivado HLS Designs* [Ref 6]. This section describes how to create the Linux driver used in the Zynq Base TRD to control the Sobel edge detection core created in Vivado HLS.

In the Linux operating system, the virtual memory space is divided into kernel space and user space. The kernel space is the portion of memory reserved for the execution of the core operating system functions: kernel, kernel extensions, and device drivers. In contrast, the user space is a portion of memory where user application programs are executed. Depending on the system load, user space programs can be swapped in and out of memory as needed.

In the context of a Zynq AP SoC running under Linux, the user application does not have direct access to the hardware accelerators in the FPGA logic. The application gains access to the hardware by interacting with the kernel through the use of kernel application program interfaces (APIs) or system calls. These APIs or system calls interact with the Linux device driver to enable access to the hardware accelerator from the user application. Along with providing access, the Linux device driver provides a programming abstraction layer for the target hardware accelerator. As in the case of the bare-metal drivers generated by HLS tool, the Linux device driver presents the user with an API to start, configure, stop, and interact with the hardware accelerator. The fine grain details of register mapping and communication protocols at the hardware level are hidden within the driver.

The character driver created for the Sobel edge detection IP block from the HLS tool can be characterized into four major sections:

- Probing
- Interrupt Handling

- File Operations Implementation
- IOCTL Implementation

Probing

The purpose of probing is to acquire the device information from the operating system device tree. The information obtained in this process includes register base address and offsets, and the interrupt behavior of the core. The device tree is read by using the Open Firmware API declared in the `linux/of_*.h` header files. The code example below is from the Linux device source tree file (dts). The device tree compiler (dtc) converts the dts file into a device tree blob (dtb), which is then accessed by the Open Firmware API:

```
axi_sobel_0: axi-sobel@0x400D0000 {
  compatible = "xlnx,axi-sobel-1.00.a";
  reg = < 0x400D0000 0xFFFF>;
  interrupts = <0 55 0>;
  interrupt-parent = <&gic>;
};
```

For the user application to access the hardware IP, the results of the device tree read operation must be mapped into the kernel virtual memory space. This mapping between the real physical address from the device tree and the virtual address in the kernel memory space is executed by the ioremap kernel utility. An example of how to use ioremap is:

```
/* Get the Sobel IP controller reg space virtual address */
retval = of_address_to_resource(dev->of_node, 0, $r_mem);

drvdata->mem_start = r_mem.start;
drvdata->mem_end = r_mem.end;
drvdata->mem_size = r_mem.end - r_mem.start + 1;

drvdata->base_address = ioremap(drvdata->mem_start, drvdata->mem_size);
```

The final stage of probing is to bind the hardware interrupt line with the kernel. This binding is achieved by using the request_irq API to register the interrupt service routine (ISR) of the device with the operating system kernel. An example of how to use the request_irq API is:

```
/* find the IRQ line, if it exists in the device tree */
drvdata->irq = irq_of_parse_and_map(dev->of_node, 0);
retval = request_irq(drvdata->irq, filter_intr_handler, IRQF_SHARED,
    "Xilinx-filter-controller", drvdata);
```

Interrupt Handling

When the interrupt line in the hardware IP is asserted, the operating system kernel calls the device's register ISR to handle the event. For the Sobel edge detection IP, the interrupt line corresponds to the ap_done signal in the HLS tool generated RTL. This signal is triggered after the core has completed processing of a video frame of 1920 x 1080 pixels. This signal from the hardware triggers the kernel registered device ISR.

The purpose of the ISR is to determine the source of the interrupt by polling status registers in the hardware device. After the source of the interrupt has been identified, the ISR is responsible for taking proper action and resetting the hardware device to a state where the next interrupt can be generated.

In the case of the Sobel IP block, the interrupt is caused by the ap_done signal at the end of a frame. The ISR for this device clears the status of the core and restarts it so that processing of the following frame can begin. The ISR used for this device is:

```
static irqreturn_t filter_intr_handler(int irq, void *data)
{
  struct xFilter_drvdata *drvdata = data;
  u32 stat;
  unsigned long flags;

  spin_lock_irqsave(&drvdata->lock, flags);

  // Disable the interrupt
  XFILTER_WR(drvdata->base_address, XFILTER_REG_IER, 0);

  // Check which interrupt
  stat = XFILTER_RD(drvdata->base_address, XFILTER_REG_ISR);

  if(stat == 1)// Do the sanity check
  {
    // Clear & Ack the interrupts
    XFILTER_WR(drvdata->base_address, XFILTER_REG_ISR, 1);

    // Restart frame processing or set completed status
    if (drvdata->mode == E_xFilterContinuousRunning)
    {
      spin_unlock_irqrestore(&drvdata->lock, flags);
      xfilter_start_ioctl(data);
    }
    else
    {
      spin_unlock_irqrestore(&drvdata->lock, flags);
      tasklet_schedule(&drvdata->tasklet);
    }

    return IRQ_HANDLED;
  }
  else
  {
    // Assert failed
    printk("xFilter Error : In ISR without ap done !!");
  }
  spin_unlock_irqrestore(&drvdata->lock, flags);
  return IRQ_HANDLED;
}
```

File Operations Implementation

File operations (fops) are the file operations registered with the kernel at the time the character driver is registered. The functions registered in the fops structure provide mapping to the user side system calls open, close, read, write, and ioctl. The structure also registers the major and minor numbers for the device:

```
static const strut file_operations xsobel_fops = {
  .owner = THIS_MODULE,
  .open = xsobel_open,
  .unlocked_ioctl = xsobel_ioctl,
  .release = xsobel_release,
};
```

The combination of the major and minor numbers is used when the device node is created under `/dev/<myDevice>`. With this information, the kernel knows how to route system calls against `/dev/<myDevice>`. An example of how to initialize the Sobel IP for the operating system is:

```
dev_t devt;

devt = MKDEV(XSOBEL_MAJOR, XSOBEL_MINOR + xfilter_dev_id);
```

```
  cdev_init(&drvdata->cdev, &xsobel_fops);
  drvdata->cdev.owner = THIS_MODULE;
  retval = cdev_add(&drvdata->cdev, devt, 1);
  if (retval) {
    dev_err(dev, "cdev_add() failed\n");
    goto failed2;
  }
```

IOCTL Implementation

Although fops implementation handles the generic system calls that can interact with the Sobel IP, there are also device specific commands that must be executed by the application. These commands, which are wrapped in the ioctl system call, control unique functions related to the Sobel IP created by the HLS tool.

The ioctl() system call has a "command" argument in the form of an unsigned integer to differentiate between several IP-specific tasks in the driver. The unsigned integer value of these commands is user defined and must be communicated to the application interacting with the driver. The ioctl commands defined for the Sobel IP are:

1. **Configure**: This command sets the resolution of the video frame in terms of rows and columns. It also selects the operating mode of the core from these choices:
   a. **Continuous**: In this mode, the ISR for the Sobel IP triggers the processing start for the next frame.
   b. **On Demand**: In this mode, the system processes only one frame per call to the IP from the user application. The ISR does not trigger the subsequent frame. In this mode, the user application is responsible for using the wait ioctl command to check on the status of the device.
2. **Start**: This ioctl command triggers the start of the edge detection operation.
3. **Stop**: This ioctl command stops continuous mode processing. The ISR instructions enabling automatic restart of the hardware device are disabled. This call is only applicable when the device has been configured in continuous mode.
4. **Wait**: This ioctl command is a blocking call, which returns only after the last frame triggered by the start ioctl has been processed. This command is applicable to the On Demand configuration mode.

The ioctl implementation for this design is:

```
static long xsobel_ioctl(struct file *file, unsigned int cmd, unsigned long
arg)
{
  struct inode *inode = file->f_dentry->d_inode;
  struct xFilter_drvdata *drvdata = container_of(inode->i_cdev, struct
      xFilter_drvdata, cdev);

  struct xFilterConfig data;


  switch (cmd) {
  case XFILTER_INIT:
    if (copy_from_user(&data, (void *)arg, sizeof(data)))
      return -EFAULT;

    xfilter_init_ioctl(drvdata, &data);
    break;

  case XFILTER_START:
    xfilter_start_ioctl(drvdata);
    break;
```

```
        case XFILTER_STOP:
          xfilter_stop_ioctl(drvdata);
          break;

        case XFILTER_WAIT_FOR_COMPLETION:
          xfilter_wait_ioctl(drvdata);
          break;

        default:
          break;

        }

        return 0;
      }
```

Integrating the System into the Zynq Base TRD

This tutorial is built on top of the ZC702 Base TRD, a video processing application for the embedded domain. The Zynq-7000 AP SoC consists of two elements: an ARM dual-core Cortex-A9 MPCore-based processing system (PS) and a video processing pipeline implemented in programmable logic (PL). The AP SoC allows the user to implement a video processing algorithm that performs edge detection (Sobel engine) on an image either as a software program running on the PS or as a hardware accelerator inside the PL. The Base TRD demonstrates how the user can seamlessly switch between a software and a hardware implementation and evaluate the cost and benefit of each implementation. The TRD also demonstrates the value of offloading computation-intensive tasks onto PL, thereby freeing the CPU resources to be available for user-specific applications.

Figure 6 shows a block diagram of the hardware architecture. For additional information, refer to *Zynq-7000 AP SoC Base Targeted Reference Design User Guide* [Ref 7] and *Zynq-7000 AP SoC ZC702 Evaluation Kit Getting Started Guide* [Ref 8].

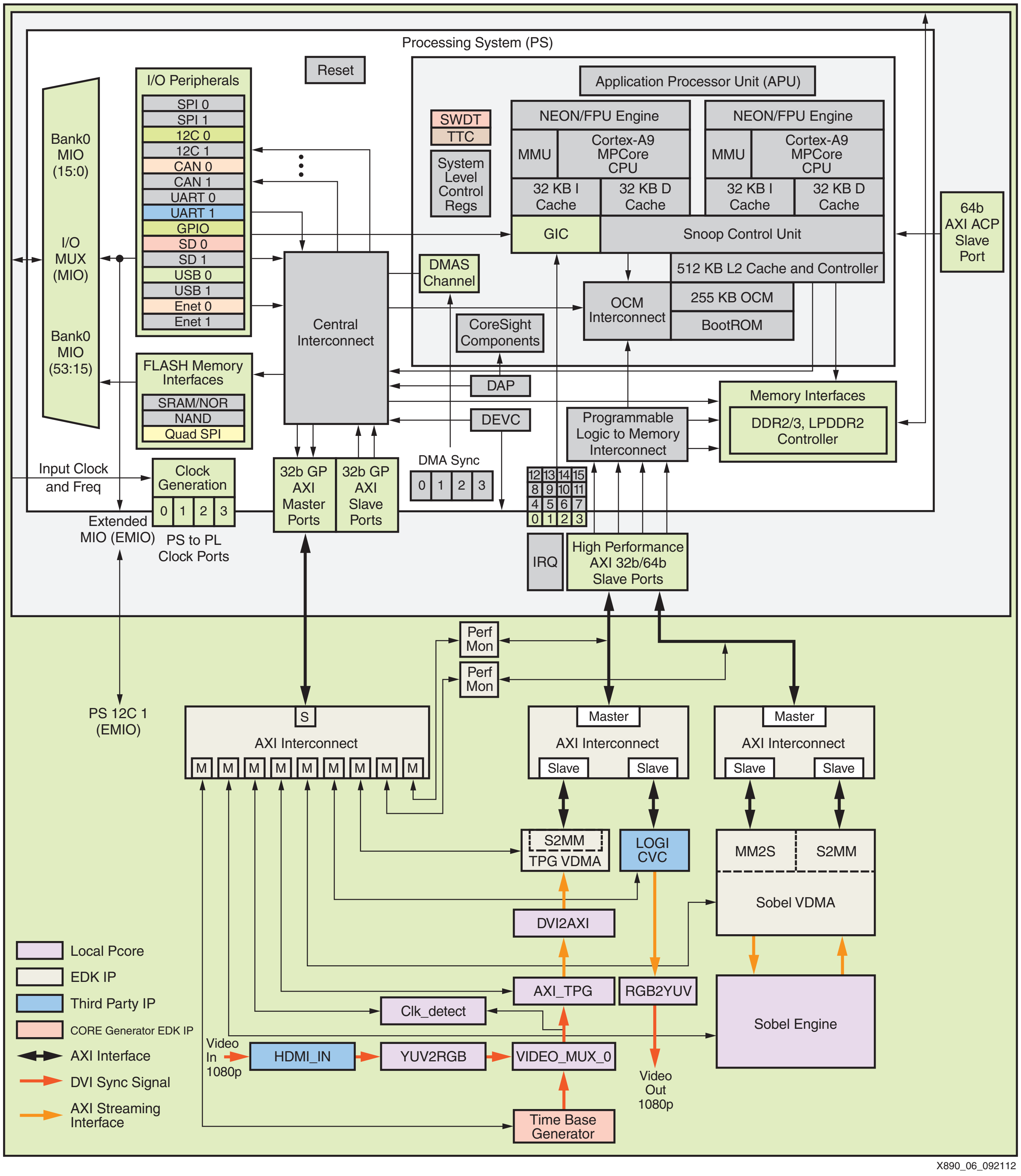


Figure 6: **Block Diagram of the Hardware Architecture**

Sobel pcore Interface and System Integration

The generated Sobel engine pcore is connected to the system using one AXI4-Lite interface, two AXI4-Stream interfaces, and one dedicated port each for clock, reset, and interrupt. These are described as follows:

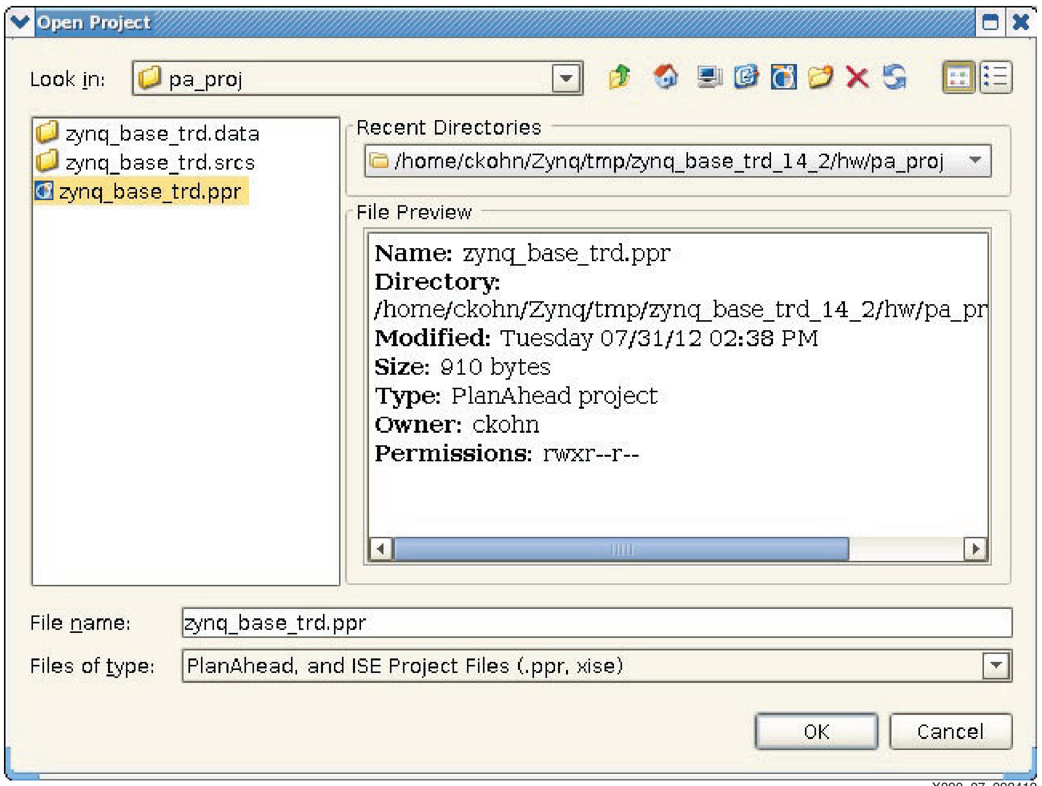
- The **AXI4-Lite interface** is connected to the GP0 port of the PS through an AXI4 Interconnect block. It is used by the CPU to configure main core parameters like the video dimensions at run time.
- The **input and output AXI4-Stream interfaces** are connected through a Video DMA (VDMA) and an AXI Interconnect block to the HP2 port of the PS. The video frame to be processed is pulled out of memory by the VDMA and sent to the Sobel engine via its input interface. The processed frame is pushed back into memory using the output interface.
- The **reset line** is connected to the PS general-purpose I/O (GPIO) controller and can be controlled by software.
- The **clock signal** is connected to the PL internal clock generator and runs at 150 MHz.
- The **interrupt port** is connected to the PS general interrupt controller (GIC). Whenever a full video frame is processed, the Sobel engine asserts its interrupt line. The software application is responsible for clearing the interrupt and restarting the Sobel Engine to process the next frame.

Inserting the Generated pcore into the Hardware System

The instructions in this section describe how to replace the existing Sobel engine block with the Vivado HLS-generated one inside the Zynq Base TRD PlanAhead™ or XPS project.

1. Download the Zynq Base TRD package (`zynq_base_trd_14_2.zip`) from the wiki page [Ref 3] and unzip the package.
2. Copy the pcore directory generated in *Vivado Design Suite Tutorial: High-Level Synthesis* [Ref 9] to the local Base TRD pcores repository, located at `zynq_base_trd_14_2\hw\pa_proj\zynq_base_trd.srcs\sources_1\edk\xps_proj\pcores`.
3. Select **Start** > **All Programs** > **Xilinx Design Tools** > **Xilinx Design Suite 14.2** > **PlanAhead** > **PlanAhead** to open the PlanAhead tool.
4. On the PlanAhead tool welcome page, click **Open Project** in the Getting Started group.

5. Browse to the `zynq_base_trd_14_2\hw\pa_proj` directory, select `zynq_base_trd.ppr`, and click **OK** (Figure 7).



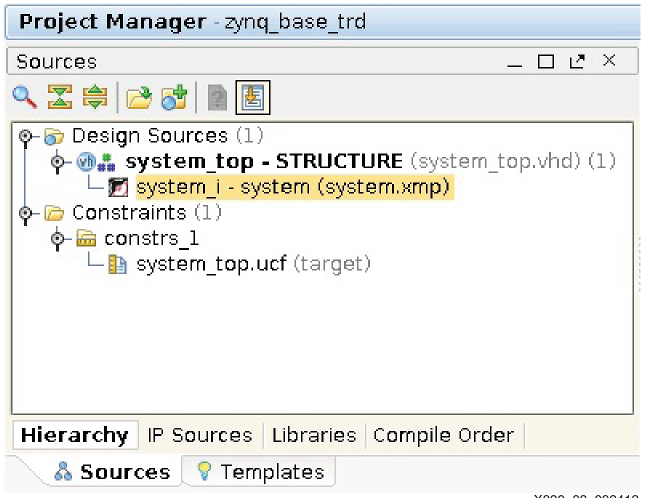
Name: zynq_base_trd.ppr
Directory:
/home/ckohn/Zynq/tmp/zynq_base_trd_14_2/hw/pa_pr
Modified: Tuesday 07/31/12 02:38 PM
Size: 910 bytes
Type: PlanAhead project
Owner: ckohn
Permissions: rwxr--r--

File name: zynq_base_trd.ppr

X890_07_092412

Figure 7: **PlanAhead Open Project Dialog Box**

6. Expand the **system_top** module in the Sources window and double-click **system_i - system (system.xmp)** to open the embedded XPS project (Figure 8).



X890_08_092412

Figure 8: **PlanAhead Project Manager**

The Xilinx Platform Studio (XPS) software opens.

7. In the Platform window, select the **Project** tab and double-click on **system.mhs** under Project Files to open the microprocessor hardware specification (MHS) file (Figure 9).

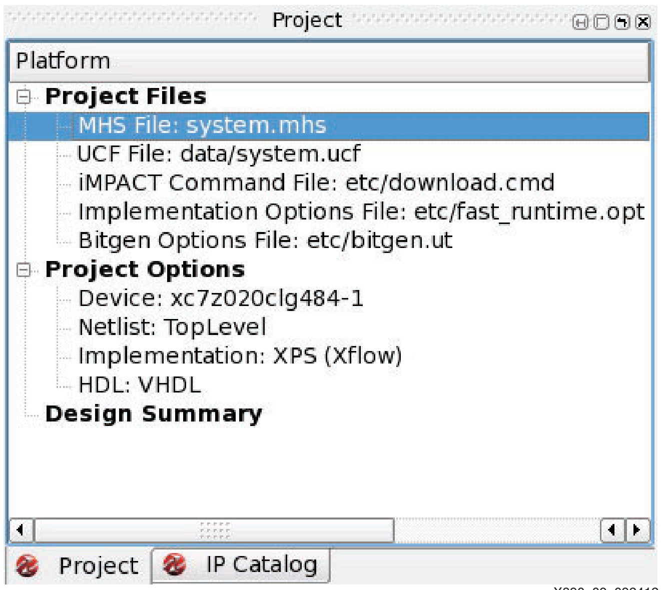
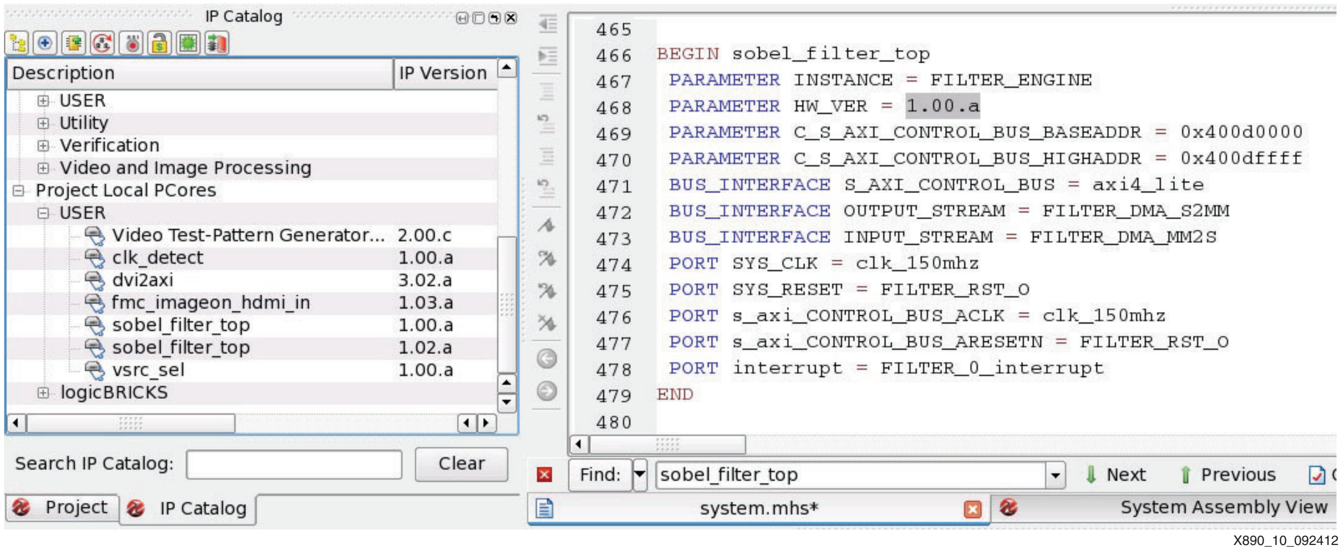

Project
Platform
Project Files
MHS File: system.mhs
UCF File: data/system.ucf
iMPACT Command File: etc/download.cmd
Implementation Options File: etc/fast_runtime.opt
Bitgen Options File: etc/bitgen.ut
Project Options
Device: xc7z020clg484-1
Netlist: TopLevel
Implementation: XPS (Xflow)
HDL: VHDL
Design Summary
Project
IP Catalog
X890_09_092412

Figure 9: **XPS Project Tab**

8. In the Platform window, select the **IP Catalog** tab.
9. Expand **USER** under Project Local PCores (Figure 10).

   Two versions of the sobel_filter_top module are listed here: 1.02.a is the currently used version and 1.00.a is the one copied over in step 2.



```
BEGIN sobel_filter_top
 PARAMETER INSTANCE = FILTER_ENGINE
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_S_AXI_CONTROL_BUS_BASEADDR = 0x400d0000
 PARAMETER C_S_AXI_CONTROL_BUS_HIGHADDR = 0x400dffff
 BUS_INTERFACE S_AXI_CONTROL_BUS = axi4_lite
 BUS_INTERFACE OUTPUT_STREAM = FILTER_DMA_S2MM
 BUS_INTERFACE INPUT_STREAM = FILTER_DMA_MM2S
 PORT SYS_CLK = clk_150mhz
 PORT SYS_RESET = FILTER_RST_O
 PORT s_axi_CONTROL_BUS_ACLK = clk_150mhz
 PORT s_axi_CONTROL_BUS_ARESETN = FILTER_RST_O
 PORT interrupt = FILTER_0_interrupt
END
```

Figure 10: **XPS IP Catalog Tab and `system.mhs` File**

10. In the `system.mhs` source file window, scroll down to the sobel_filter_top module and change the **Parameter HW_VER** from **1.02.a** to **1.00.a**, as shown in Figure 10.
11. Select **File > Save** to save the file.
12. Close the XPS window to return to the PlanAhead tool.

13. Select **Flow > Generate Bitstream** and click **Yes** when prompted to launch synthesis and implementation first. The bitstream is generated at `zynq_base_trd_14_2\hw\pa_proj\zynq_base_trd.runs\impl_1\system_top.bit`.

 Note: During implementation, a message window opens, reporting three critical warnings. Ignore these warnings and click **OK** to continue.

Refer to the Zynq Base TRD wiki page [Ref 3] for the complete tool flow. The above instructions are a replacement for step 3.1 on the wiki page (Building the Bitstream). Steps 3.2 through 9.2 show how to build the remainder of the required software components. Step 10 provides instruction on how to run the design on the ZC702 hardware platform.

Reference Design

The reference design files for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=193509

Table 2: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Xilinx
Target devices (stepping level, ES, production, speed grades)	Zynq-7000 AP SoC
Source code provided	Yes
Source code format	C
Design uses code and IP from existing Xilinx application note and reference designs, CORE Generator software, or third party	Yes
Simulation	
Functional simulation performed	Yes
Timing simulation performed	No
Test bench used for functional and timing simulations	Yes
Test bench format	C
Simulator software/version used	ModelSim 6.6d
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/version used	Vivado HLS 2012.2
Implementation software tools/versions used	EDK 14.2
Static timing analysis performed	No
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	ZC702 board

Conclusion

This application note describes how to generate the Sobel IP block in the Zynq TRD using the Vivado HLS tool. The techniques described in this application note apply to all designs in which a Vivado tool-generated IP block is integrated into a Zynq AP SoC.

References

This application note uses the following references:

1. UG683, *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design.*
2. UG902, *Vivado Design Suite User Guide: High-Level Synthesis.*
3. Zynq Base TRD wiki page: http://wiki.xilinx.com/zynq-base-trd-14-2
4. Gonzalez, R. and R. Woods. 2001. *Digital Image Processing, Second Edition*. Prentice Hall.
5. XAPP793, *Implementing Memory Structures for Video Processing in the Vivado HLS Tool.*
6. XAPP745, *Processor Control of Vivado HLS Designs.*
7. UG925, *Zynq-7000 EPP Base Targeted Reference Design User Guide.*
8. UG926, *Zynq-7000 All Programmable SoC: ZC702 Evaluation Kit and Video and Imaging Kit (ISE Design Suite 14.2) Getting Started Guide.*
9. UG871, *Vivado Design Suite Tutorial: High-Level Synthesis.*

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
09/25/12	1.0	Initial Xilinx release.

Notice of Disclaimer