1300 Henley Court
Pullman, WA 99163
509.334.6306
www.digilentinc.com

Embedded Linux® Hands-on Tutorial for the ZYBO™

Revised December 5, 2014

Overview

The purpose of this document is to provide step-by-step instructions for customizing your hardware, compiling the Linux Kernel, and writing driver and user applications. This documentation intends to integrate knowledge and skills in FPGA logic circuit design, standalone software programming, Linux operating system and user application development, and apply them to the ZYBO. We will start from the *ZYBO Base System Design* (available on the ZYBO product page of the Digilent website). The system architecture for the *ZYBO Base System Design* is shown in Fig. 1.

In the *ZYBO Base System Design*, we connect UART1 to USB-UART, SD0 to the SD Card Slot, USB0 to the USB-OTG port, Enet0 to the Giga-bit Ethernet Port, and Quad SPI to the on-board QSPI Flash. These cores are hard IPs inside the Processing System (PS) and connect to on-board peripherals via Multiplexed I/O (MIO) pins. The use of PS GPIO is connected to Btn 4 and 5. In the Programmable Logic (PL), we have an HDMI TX Controller, VDMA, and GPIO IP cores to talk to the ADV7511 HDMI transmitter chip and I2S and GPIO IP cores for ADAU1761 audio codec. More details of the hardware design can be found in the documentation inside the *ZYBO Base System Design* package.

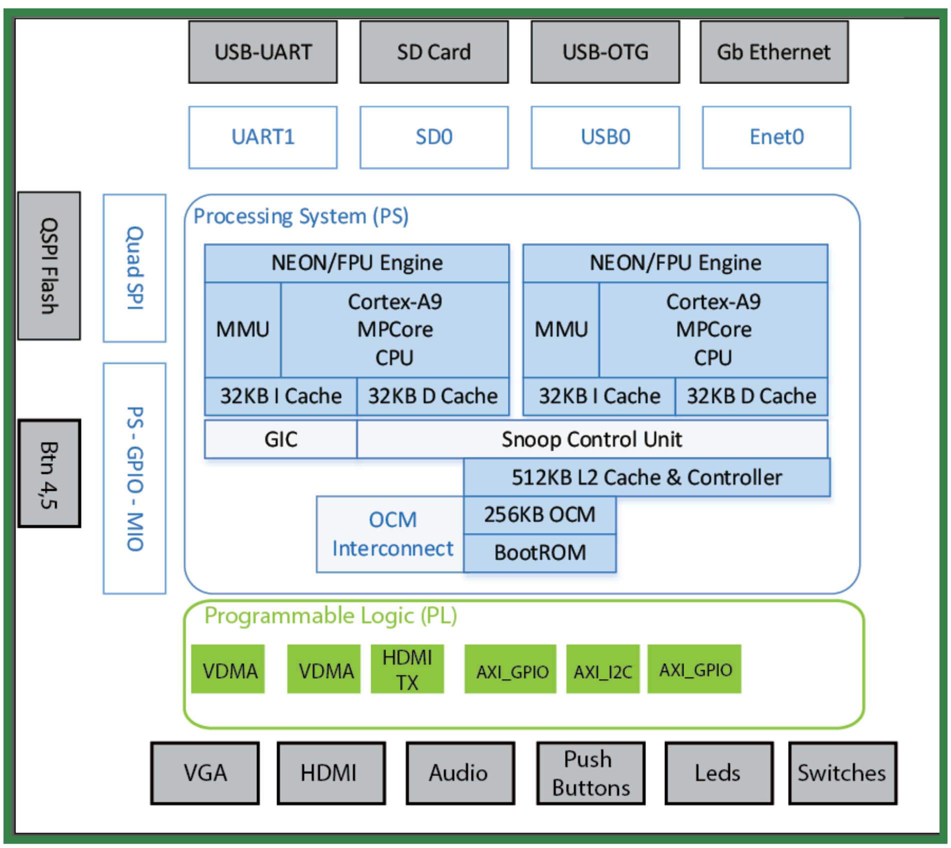

Figure 1. Reference Basic Hardware System Architecture for ZYBO.

In this tutorial, we are going to detach the LEDs from the AXI GPIO core and implement our own `myLed` core for it in PL, as shown in Fig. 2. We will then add our own LED controller into the device tree, write a driver for it, and develop user applications to control the status of the LEDs.

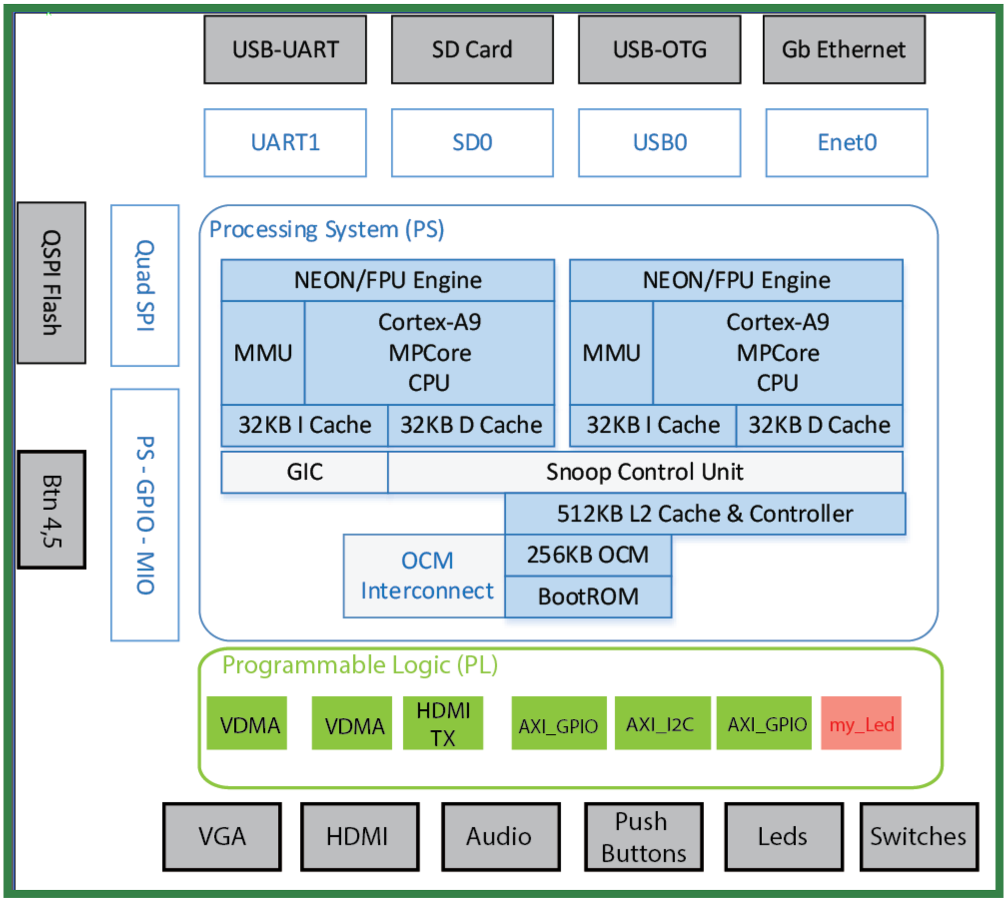

Figure 2. Hardware System Architecture of the system we are going to implement in this Tutorial.

Before going through this tutorial, we recommend that you read *Getting Started with Embedded Linux - ZYBO*. You can follow this tutorial with the *Embedded Linux Development Guide* (available on the Digilent website Embedded Linux Page). The guide will provide you with the knowledge you may need in each step of the development.

In this tutorial, we are going to use Xilinx® Vivado™ 2014.1 WebPACK™ in a Linux environment. All of the screenshots and codes are done using Vivado Design Suite 2014.1 in CentOS 6 x86_64.

That's it for the background information on this tutorial, now it's time to get our hands dirty with some real design!

1 Hardware Customization

1.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx website Download Page.
- *ZYBO Base System Design:* available at the Digilent website on the ZYBO Page.

1.2 Instructions

1. Download the *ZYBO Base System Design* from the Digilent website and unzip it into our working directory, as in Fig. 3 (our working directory is named ***tutorial*** throughout this document). For more information on the hardware design, please refer to *Project Guide* under doc folder.

```
[kfranz@localhost Tutorial]$ unzip /home/kfranz/Downloads/zybo_base_system.zip
Archive:  /home/kfranz/Downloads/zybo_base_system.zip
  inflating: zybo_base_system/ProjectGuide.txt
   creating: zybo_base_system/sd_image/
  inflating: zybo_base_system/sd_image/BOOT.bin
   creating: zybo_base_system/source/
   creating: zybo_base_system/source/ise/
   creating: zybo_base_system/source/ise/hw/
   creating: zybo_base_system/source/ise/hw/data/
  inflating: zybo_base_system/source/ise/hw/data/ps7_constraints.ucf
```

Figure 3. Unzip the ZYBO_Base_System.

2. Source Vivado 2014.1 settings and open the design with Vivado Design Suite. You will see the Vivado window pop up as shown in Fig. 4.

   Note: There are four settings files available in the Vivado toolset: settings64.sh for use on 64-bit machines with bash; settings32.sh for use on 32-bit machines with bash; settings32.csh for use on 32-bit machines with C Shell; and settings64.csh for use on 64-bit machines with C Shell.

```
[kfranz@localhost Tutorial]$ source /opt/Xilinx/Vivado/2014.1/settings64.sh
[kfranz@localhost Tutorial]$ vivado zybo_base_system/source/vivado/hw/zybo_bsd/z
ybo_bsd.xpr

****** Vivado v2014.1 (64-bit)
  **** SW Build 881834 on Fri Apr  4 14:00:25 MDT 2014
  **** IP Build 877625 on Fri Mar 28 16:29:15 MDT 2014
    ** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

start_gui
```

Figure 4. Open the Project.

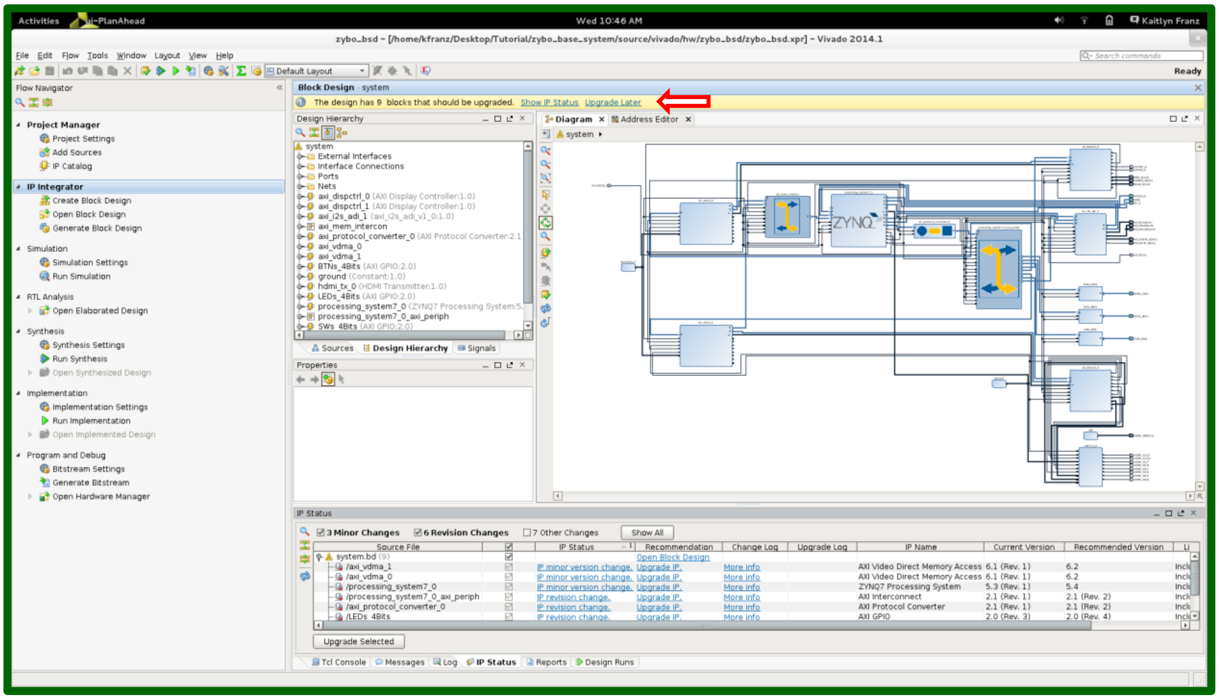

Figure 5. Vivado platform studio GUI.

3. We are going to detach LEDs from the GPIO core in the PS first. So we need to click on the IP integrator and open the **Block Diagram** as shown in Fig. 5. Then we need to delete the current LED IP as shown in Fig. 6. We will handle the modification of external pin location configuration (**xdc** file) in later steps.

   Note: In Fig. 6 there is a yellow bar indicating the need for an upgrade. To upgrade, hit **show IP status**, make sure all are selected and hit **Upgrade Selected**.

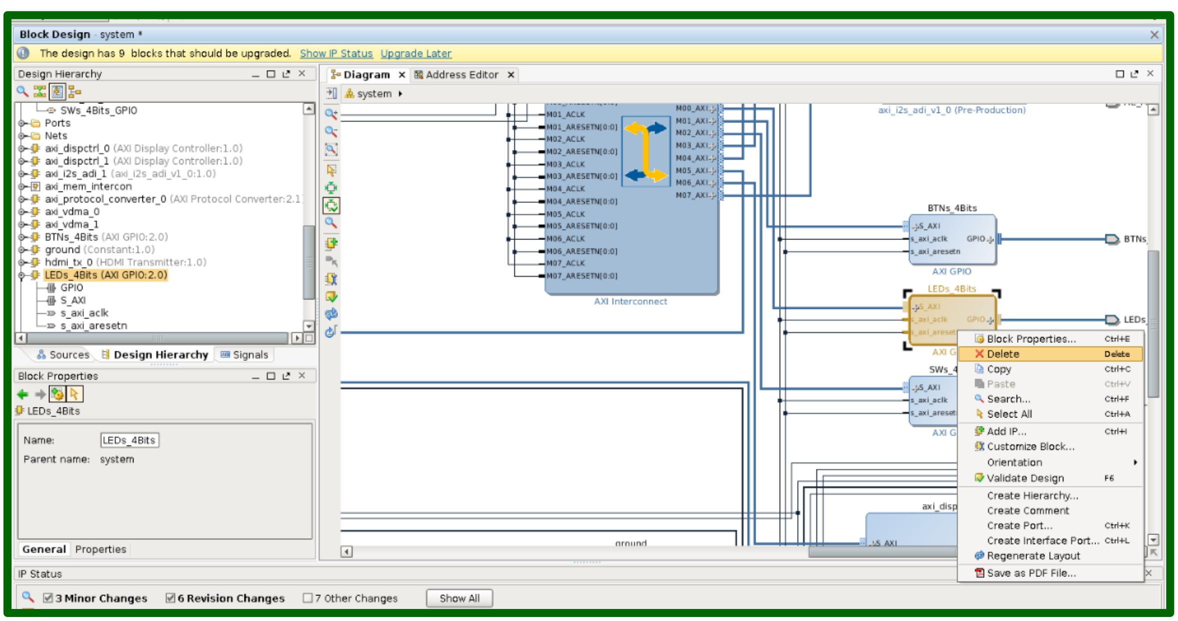

Figure 6. Delete existing LED IP.

4. (Vivado 2014.1 only) Before we can start implementing our `myLed` IP Core, we need to name the vendor that will automatically be applied in the IP packager. In Vivado 2014.1, this is not automatically done for the user. To do this, first go to the **Project Settings** under **Project Manager** on the left side of the window (Fig. 7) and the project settings window will pop up. In the Project Settings window, select **IP** (Fig. 8). Notice that the vendor is chosen as “(none)”, this will cause a Vivado internal exception. You can name the Vendor whatever you like (Fig. 9).

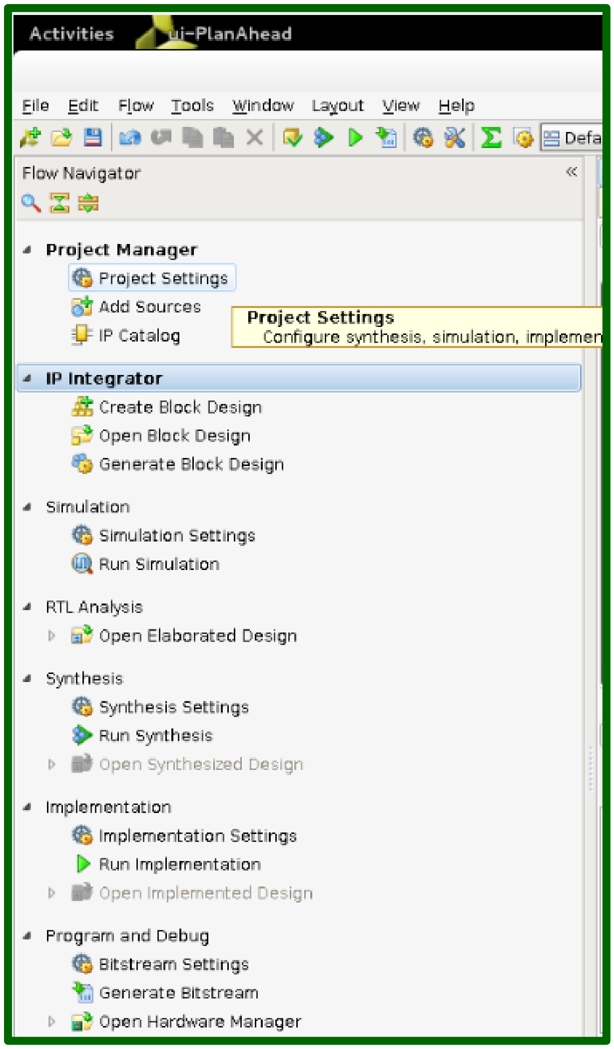

Figure 7. Project settings.

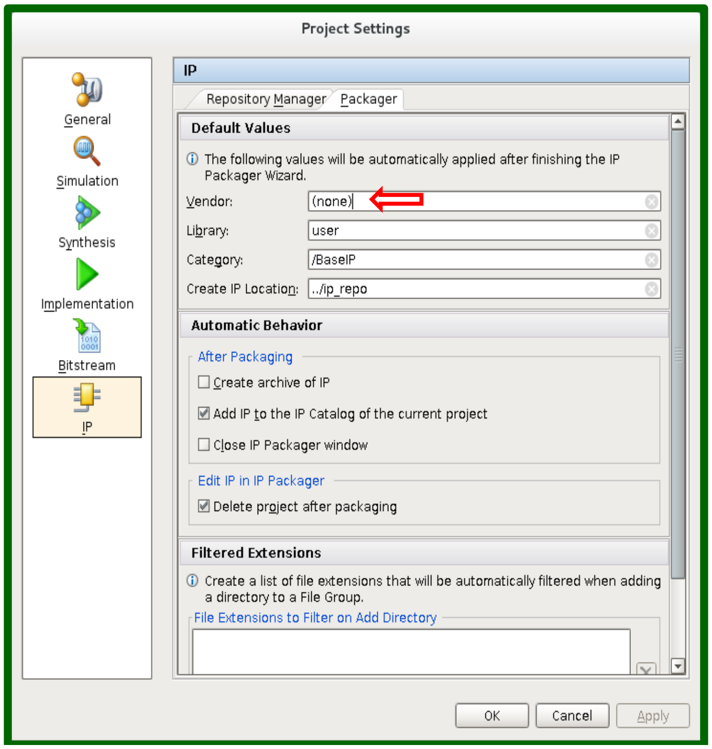

Figure 8. Unnamed vendor.

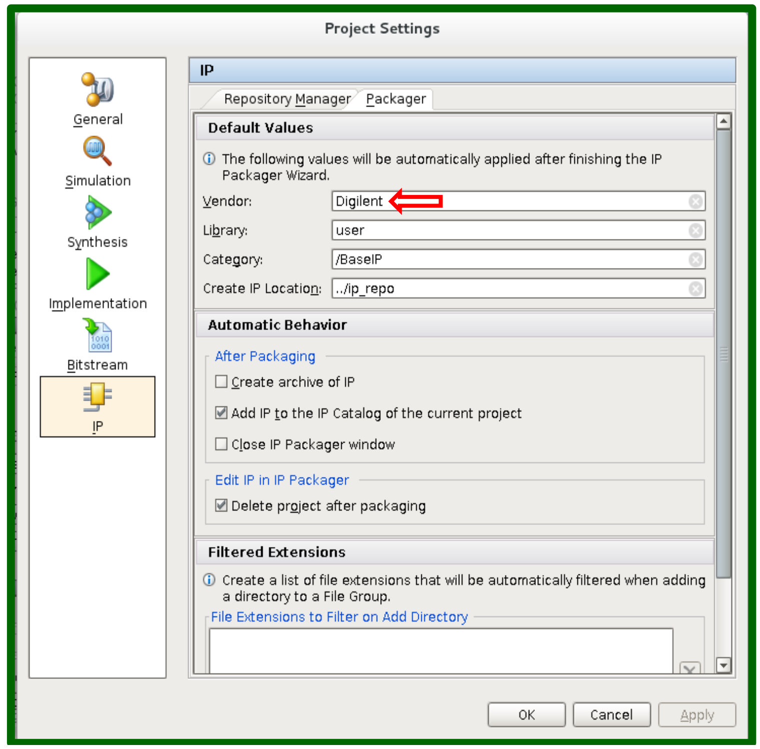

Figure 9. Named vendor.

5. Now we can start implementing our `myLed` IP Core. Click **Tools -> Create and Package IP…** from the menu (as shown in Fig. 10). The Create and Package New IP window will pop up (as shown in Fig. 11), Click **Next**. In the next window, name the new IP and click next again (Fig. 12).

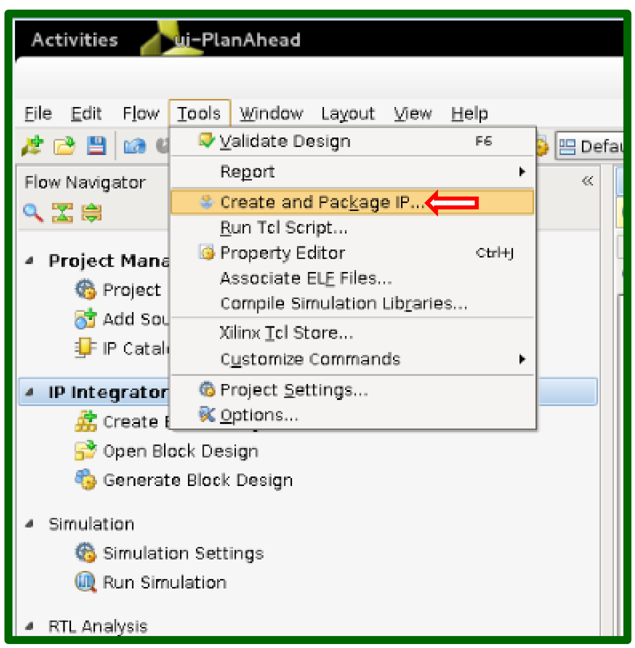

Figure 10. Create and Package IP.

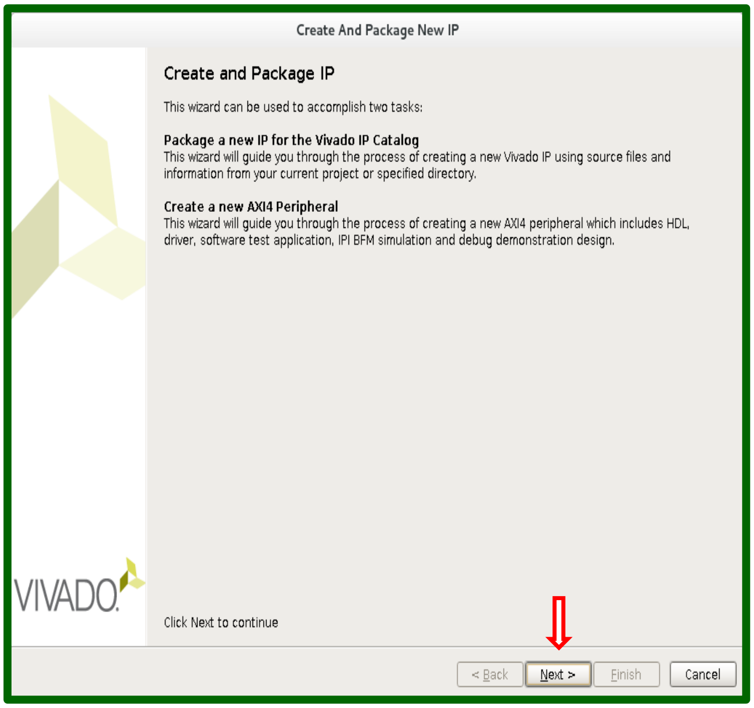

Figure 11. IP Options.

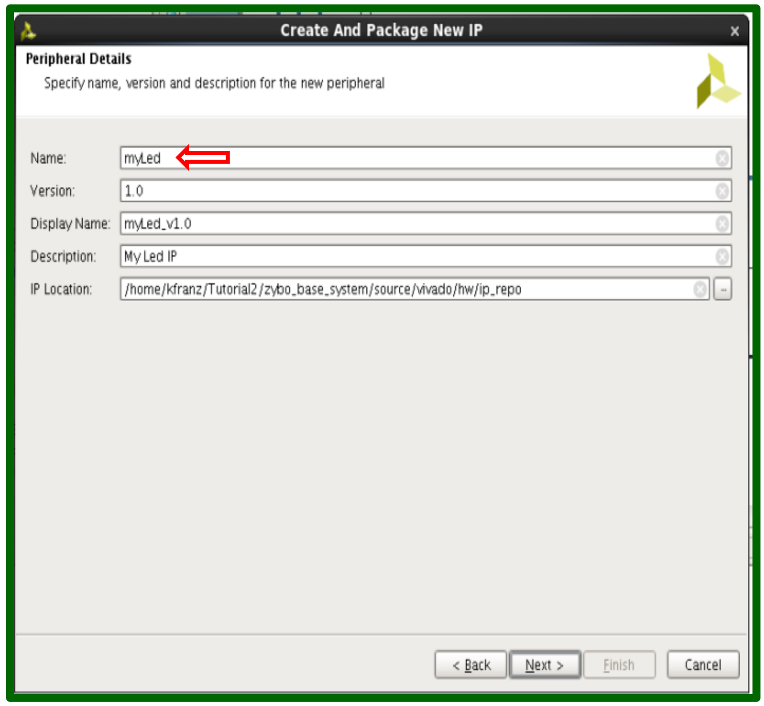

Figure 12. Peripheral Details.

6. The next window will be the Add Interfaces Window. This will create the AX14 Interface for the `myLed` peripheral (Fig. 13). Make sure the interface type is **Lite**, the mode is **Slave**, the data width is **32 bits** and the number of registers is **4**. Change the Name to **S_AXI** rather than S00_AXI. We only need 1 register but the minimum we can select is 4. Click next to proceed.

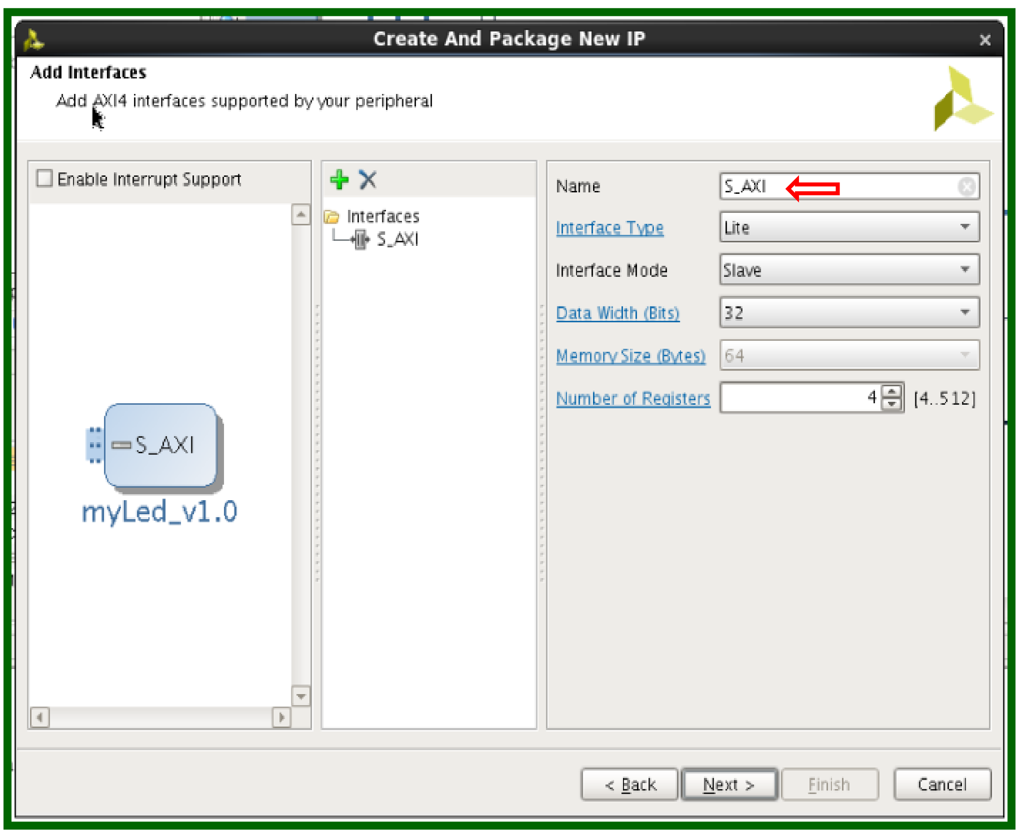

Figure 13. Add Interface

7. The next window will prompt the finishing steps to create the IP (Fig. 14). Change the Radio button to select **Edit IP** and hit finish. We need to add user logic to the IP so that our slave is connected to the LED output.

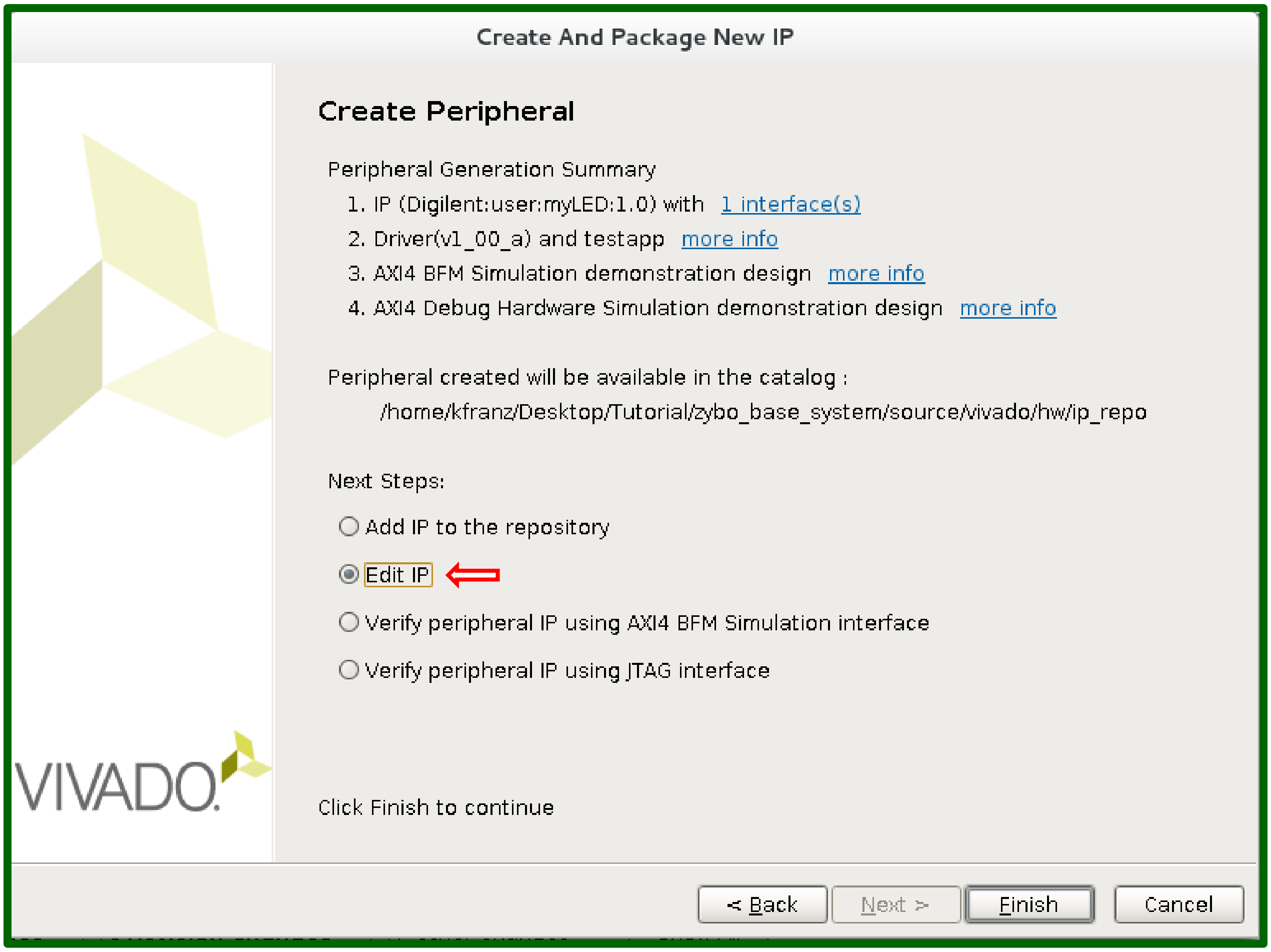

Figure 14. Edit IP.

8. After selecting finish, the Create and Package IP window will disappear and the next window you will see is the edit_myLed window (Fig. 15). This is where we will add our user logic.

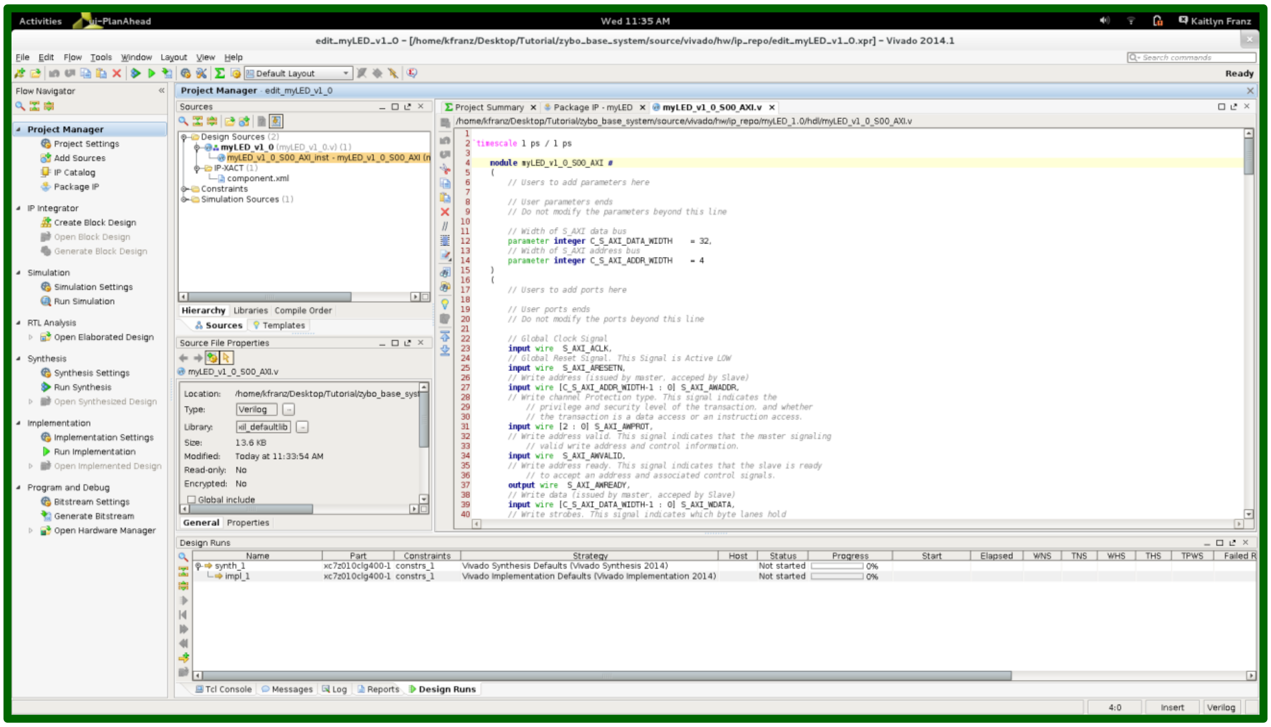

Figure 15. Edit_myLed.

9. In the Project Manager, click the circle next to `myLed_v1_0` and highlight **myLed_v1_0_S_AXI** (Fig. 16). This contains the user logic inside of the `myLed` IP. We need to add two lines of code to complete the user logic for this module. First, we need to create a user port called led (Fig. 17). Next, we need to connect the internal slave to this user port. We will connect slv_reg0[3:0] as we have four LEDs (Fig. 18).

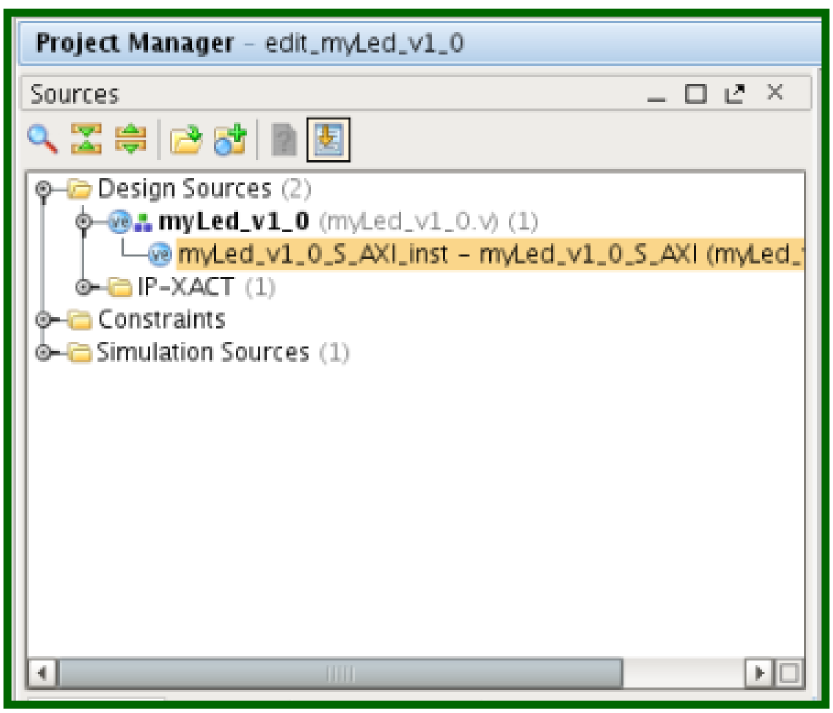

Figure 16. Select user logic file.

```

`timescale 1 ps / 1 ps

    module myLed_v1_0_S00_AXI #
    (
        // Users to add parameters here

        // User parameters ends
        // Do not modify the parameters beyond this line

        // Width of S_AXI data bus
        parameter integer C_S_AXI_DATA_WIDTH    = 32,
        // Width of S_AXI address bus
        parameter integer C_S_AXI_ADDR_WIDTH    = 4
    )
    (
        // Users to add ports here
        output wire [3:0]led,
        // User ports ends
        // Do not modify the ports beyond this line

```

Figure 17. Add user port.

```
            end
        end
    end

    // Add user logic here
    assign led = slv_reg0[3:0];
    // User logic ends

    endmodule
```

Figure 18. Add user logic.

10. Next, we need to connect the user logic to `myLed`. In the project manager select the file **myLed_v_0**. To complete the IP, there are two lines of code we need to add to this file. Under the comment that says "Users to add ports here," add a port for the LEDs (Fig. 19). Connect the led output from the previous file containing the user logic to `myLed` (Fig. 20).

```
`timescale 1 ps / 1 ps

    module myLed_v1_0 #
    (
        // Users to add parameters here

        // User parameters ends
        // Do not modify the parameters beyond this line


        // Parameters of Axi Slave Bus Interface S00_AXI
        parameter integer C_S00_AXI_DATA_WIDTH  = 32,
        parameter integer C_S00_AXI_ADDR_WIDTH  = 4
    )
    (
        // Users to add ports here
        output [3:0] led,
        // User ports ends
```

Figure 19. Add External.

```
// Instantiation of Axi Bus Interface S00_AXI
    myLed_v1_0_S00_AXI # (
        .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
    ) myLed_v1_0_S00_AXI_inst (
        .led(led),
        .S_AXI_ACLK(s00_axi_aclk),
        .S_AXI_ARESETN(s00_axi_aresetn),
        .S_AXI_AWADDR(s00_axi_awaddr),
        .S_AXI_AWPROT(s00_axi_awprot),
        .S_AXI_AWVALID(s00_axi_awvalid),
        .S_AXI_AWREADY(s00_axi_awready),
        .S_AXI_WDATA(s00_axi_wdata),
        .S_AXI_WSTRB(s00_axi_wstrb),
        .S_AXI_WVALID(s00_axi_wvalid),
```

Figure 20. Connect myLed to User Logic.

11. Now that our IP is created and the user logic is defined, we need to package our IP. Under **Project Manager** on the left side of the window, select **Package IP**. A new tab will open that is called Package IP. On the left side of this tap there are a series of labels. We need to complete those that do not have green check marks.

    First, select **IP customization Parameters**. At the top of that window select the option to **merge changes from IP Customization Parameters Wizard,** as in Fig. 21.

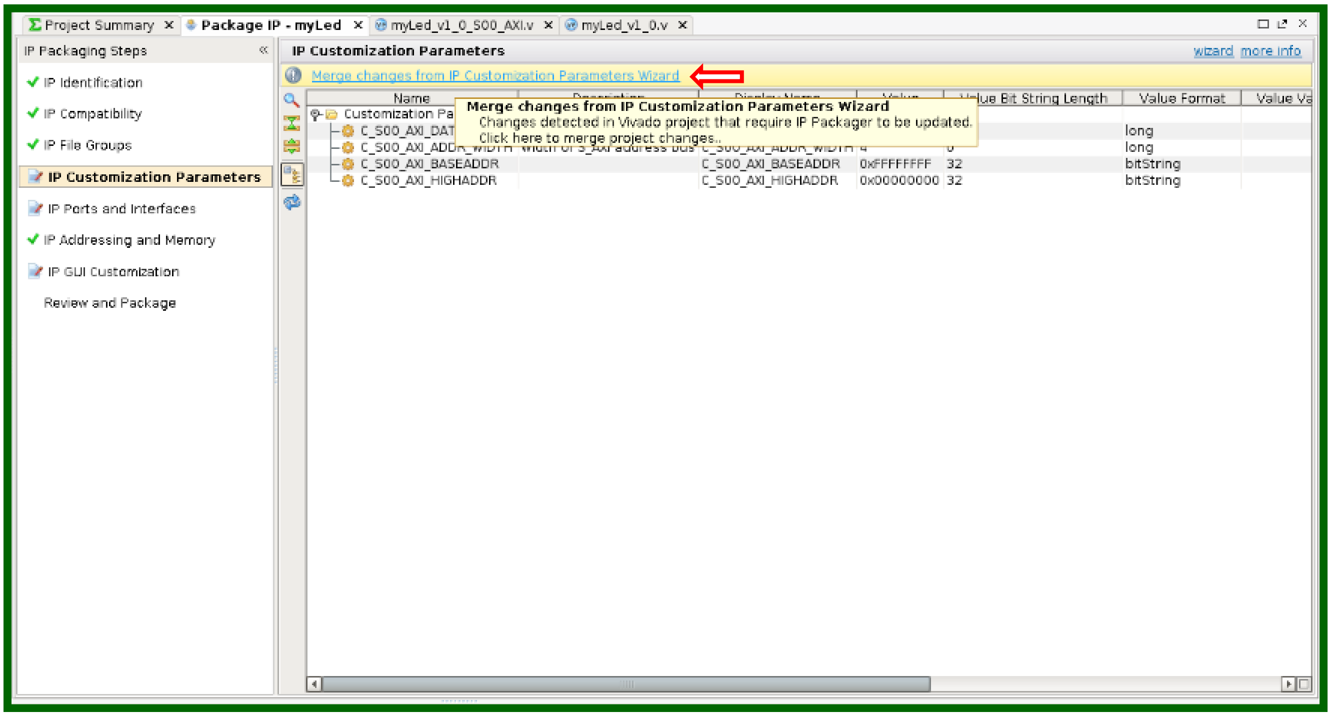

Figure 21. Customization Parameters.

Next, select the IP Ports and Interfaces. Notice that your new LED IP is there (Fig. 22).

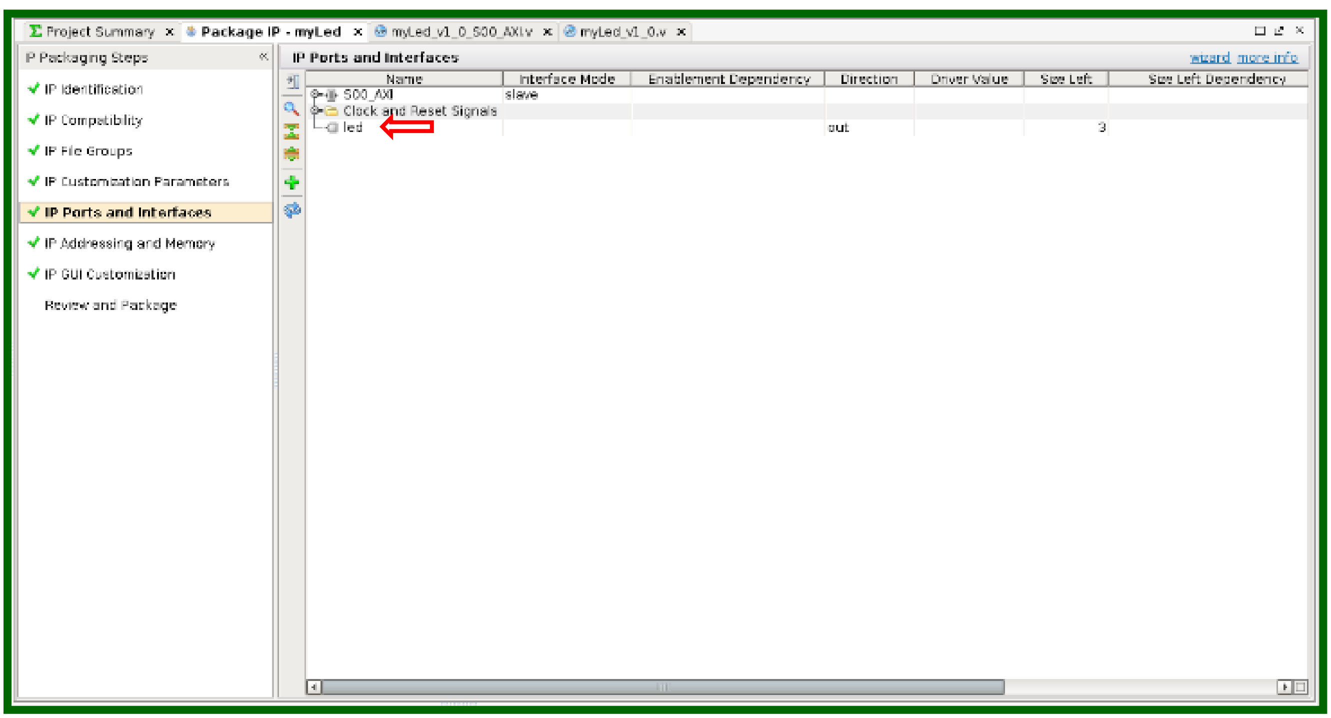

Figure 22. IP Ports and Interfaces.

Next, select **IP GUI Customization**. Our IP GUI is fine as is, so we won't make any changes here (Fig. 23).

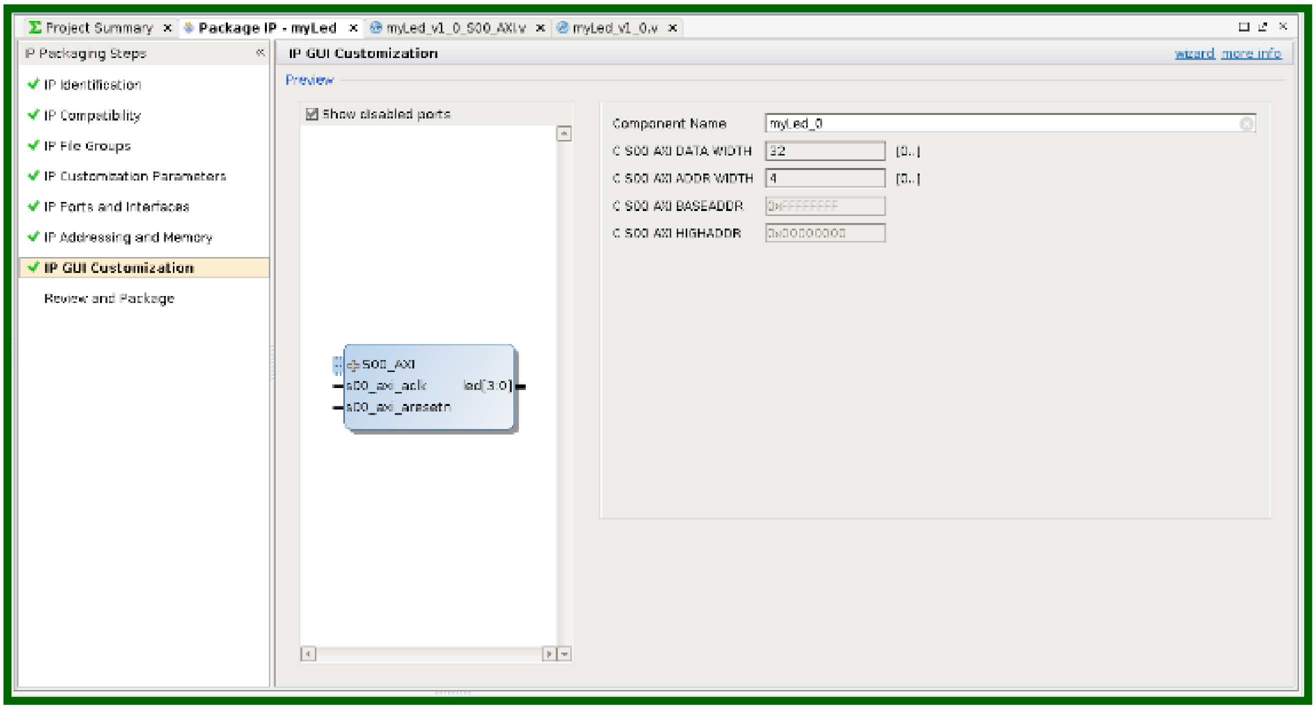

Figure 23. GUI Customization.

Now we can **Review and Package** our `myLed` IP. Select Review and Package IP and press the **Re-Package IP button**. Our IP is now completed and packaged.

12. We are going to add our IP to our design. Right click anywhere on the block design and click **Add IP** (as shown in Fig. 24). Select the correct IP, **myLed_v1.0**, and press enter (Fig. 23).

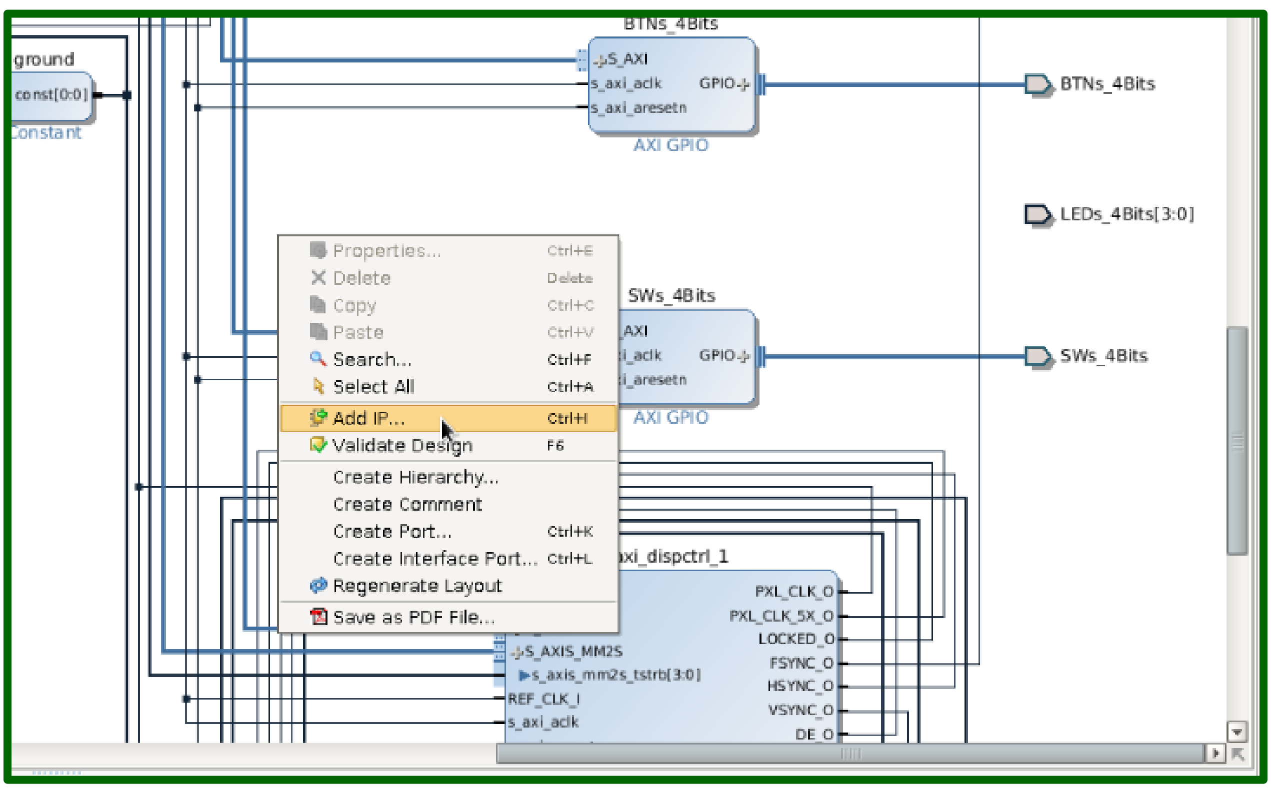

Figure 24. Add IP.

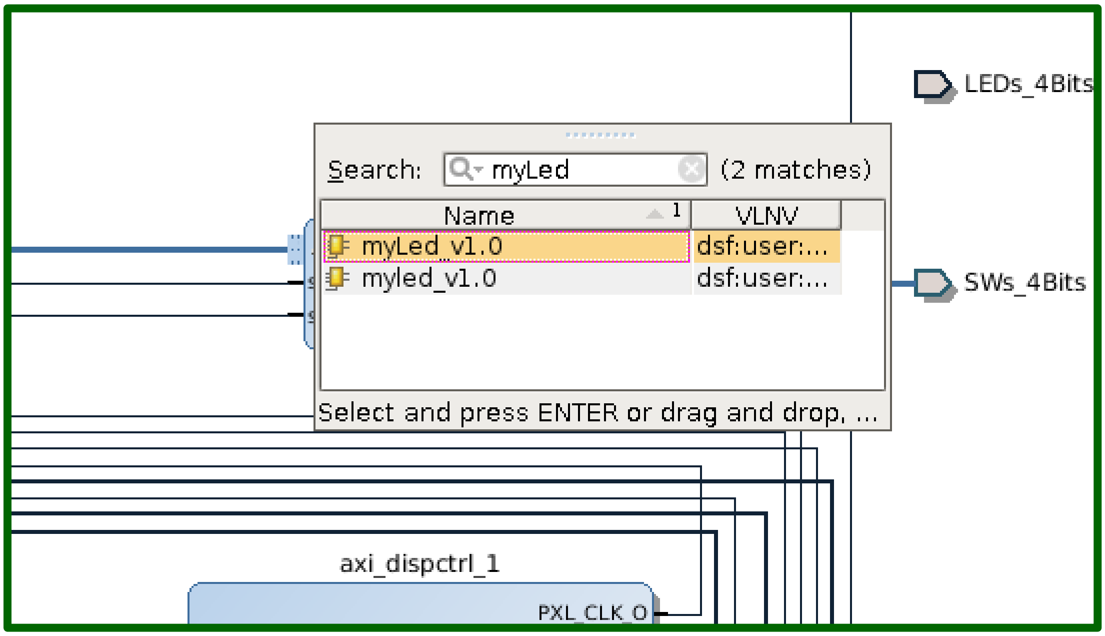

Figure 25. Select IP.

13. The AXI4-Lite bus of `myLed` IP Core needs to be connected to the processing system. At the top of the window, click the blue text that says **Run connection automation** (Fig. 26). This will connect the inputs of the `myLed` IP Core. You should see that S_AXI is now connected to the first output of the AXI Interconnect.

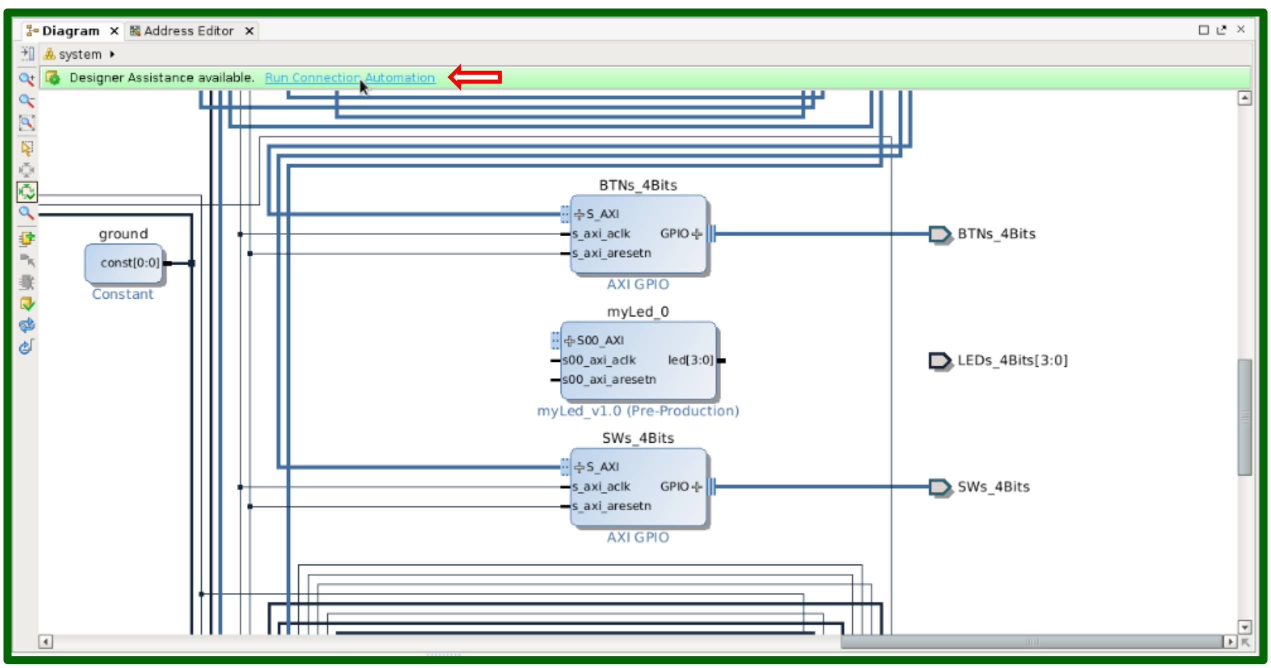

Figure 26. Connect IP.

14. Next, we need to connect the `myLed` IP to an external port. The `myLed` IP Core that we implemented will not connect to the existing LEDs_4Bits port, so we need to make a new external port called led. Click on the existing LED port and press delete. To create the new port, right click and select **create port** (Fig. 27). Name the port, select output, select vector [3:0] and press enter.

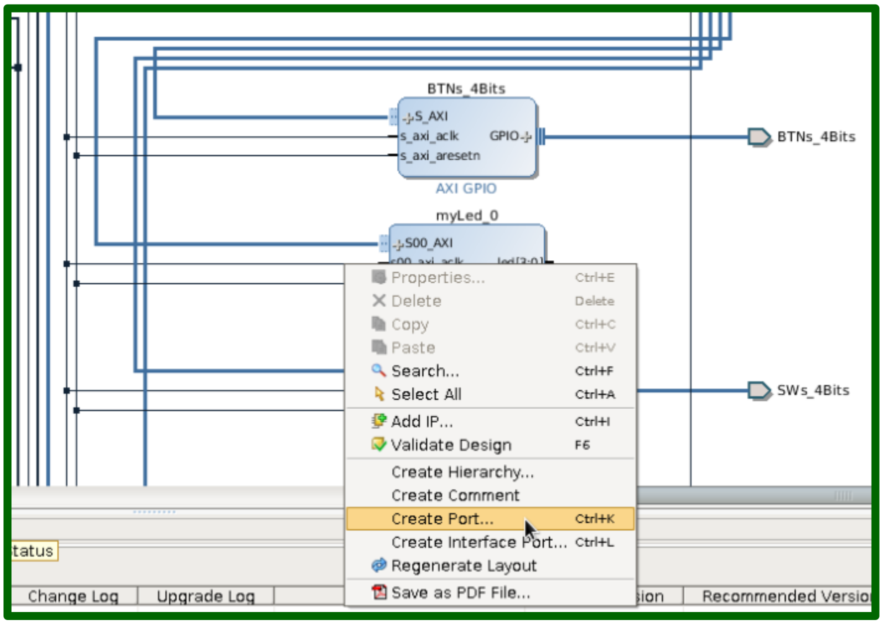

Figure 27. Create Port.

Next, connect the LED port to the `myLed` IP by clicking on the port and dragging a connection to `myLed` (Fig. 28).

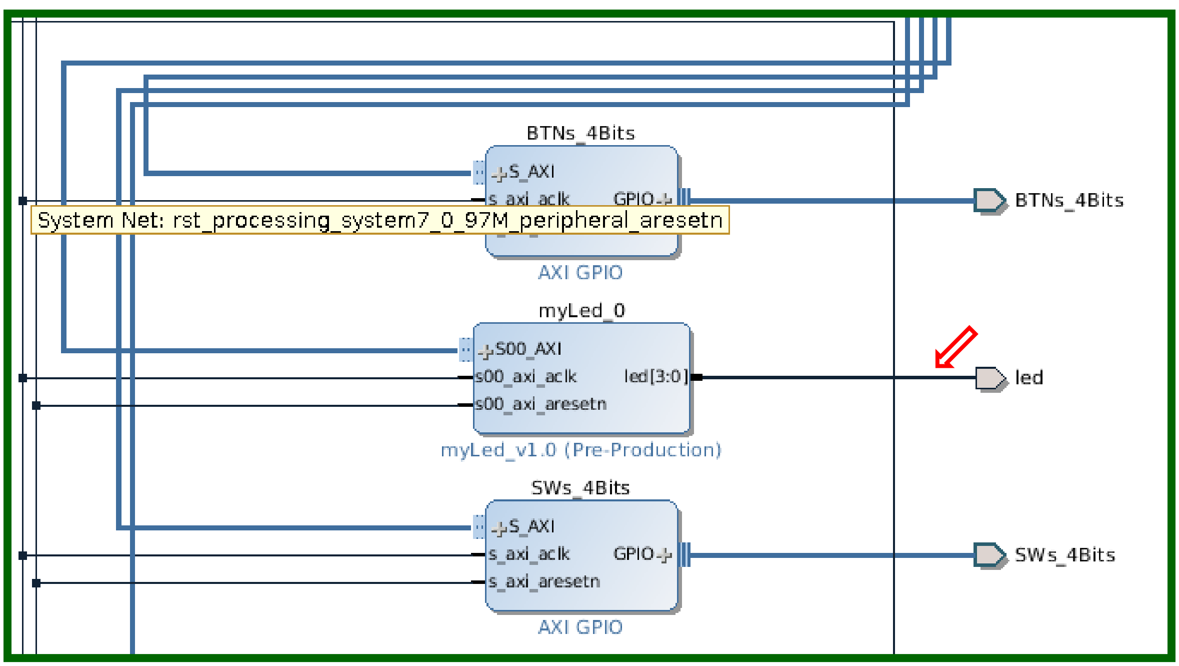

Figure 28. Connect IP to Port.

15. The final step is to specify the pin numbers for `myled_0_LED_pin` to physically connect our customized IP Core to the on-board LEDs. In the Project Manager, expand the Constraints section and select the **base.xdc** file (Fig. 29). Within that file, change the names of the external LED pins so that they match the name of our external led port (Fig. 30).

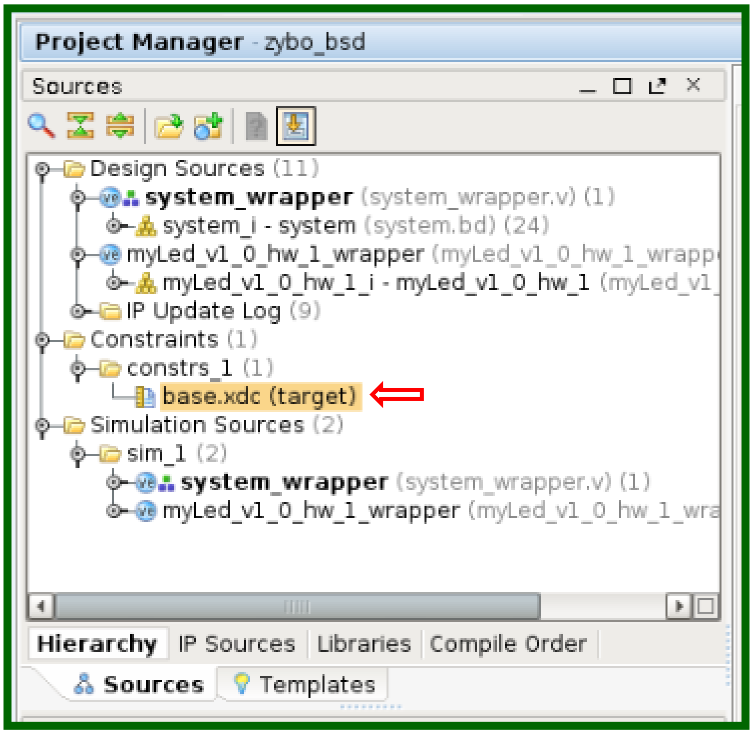

Figure 29. Open XDC File.

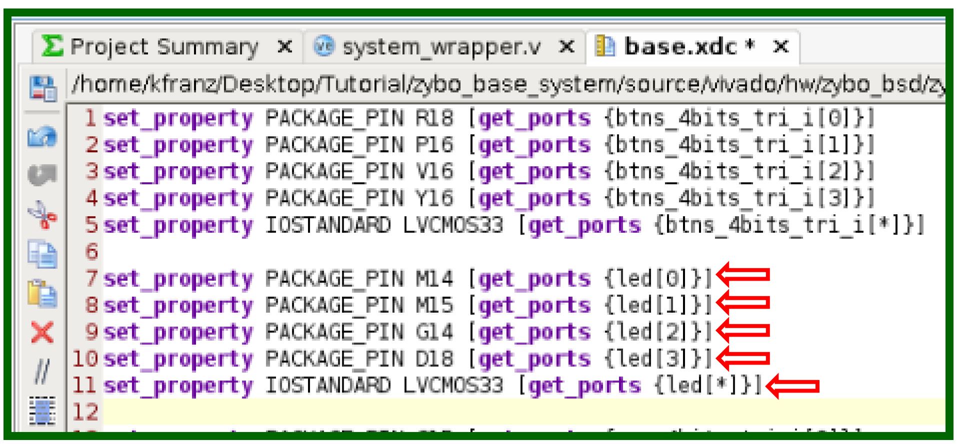

Figure 30. Connect Port led to the LEDs on the ZYBO Board.

16. Regenerate the bitstream for the hardware design by clicking on **Generate Bitstream** under Program and Debug on the left side of the window.

2 Compile U-Boot (Optional)

2.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website Download Page.
- *ZYBO Base System Design:* available at the Digilent Website on the ZYBO Page.

2.2 Instructions (Use the Master-Next Branch Until Further Notice)

1. Get the source code for U-Boot from the Digilent Git repository. There are two ways to retrieve the source code:

   Using *git* command: If you have Git installed in your distribution, you can clone the repository to your computer by command Git clone: `https://github.com/DigilentInc/u-boot-Digilent-Dev.git`. The whole Git Repository is around 55MB, as shown in Fig. 31. If you want to get a separate branch, follow Fig. 32. The next contains the U-boot that is not yet released. The clone URL referenced above can be found on the Digilent GitHub page, as seen in Fig. 33.

```
[kfranz@DIGILENT_LINUX ~]$ git clone https://github.com/DigilentInc/u-boot-Digil
ent-Dev.git
Initialized empty Git repository in /home/kfranz/u-boot-Digilent-Dev/.git/
remote: Reusing existing pack: 253078, done.
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 253088 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (253088/253088), 69.54 MiB | 4.88 MiB/s, done.
Resolving deltas: 100% (201962/201962), done.
```

Figure 31. U-Boot repository.

```
[kfranz@DIGILENT_LINUX ~]$ git clone -b master-next https://github.com/DigilentI
nc/u-boot-Digilent-Dev.git
Initialized empty Git repository in /home/kfranz/u-boot-Digilent-Dev/.git/
remote: Reusing existing pack: 253078, done.
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 253088 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (253088/253088), 69.54 MiB | 5.24 MiB/s, done.
Resolving deltas: 100% (201962/201962), done.
[kfranz@DIGILENT_LINUX ~]$
```

Figure 32. Next-repository.

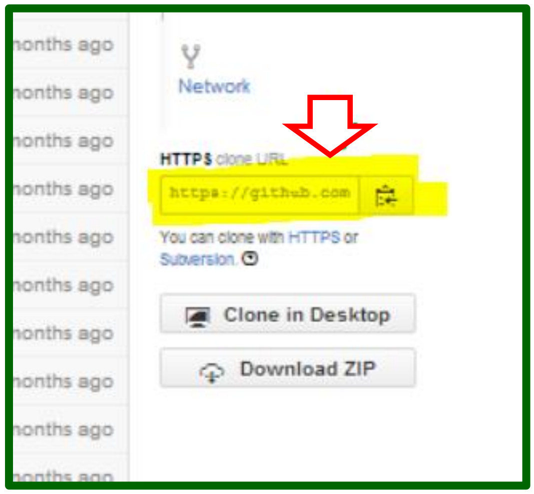

Figure 33. Clone URL.

2. To compile U-Boot, we need cross-compile tools which are provided by Vivado 2014.1. Those tools have a prefix `arm-xilinx-linux-gnueabi-` to the standard names for the **GCC** tool chain. The prefix references the platforms that are used. The ZYBO board has two arm cores, so we reference arm. In order to use the cross-platform compilers, please make sure the Vivado 2014.1 settings have been sourced. If not, please refer to step 1 above. To configure and build U-Boot for ZYBO, follow Fig. 34.

```
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi-
zynq_zybo_config
Configuring for zynq_ZYBO board...
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$ make CROSS_COMPILE=arm-xilinx-linux-gnueabi-
Generating include/autoconf.mk
Generating include/autoconf.mk.dep
. . .
arm-xilinx-linux-gnueabi-ld -gc-sections -Ttext 0x1000000 -o demo crt0.o demo.o libgenwrap.o
lent-Dev/arch/arm/lib/eabi_compat.o -L /opt/Xilinx/SDK/2014.1/gnu/arm/lin/bin/../lib/gcc/arm
arm-xilinx-linux-gnueavi-objcopy -O binary demo demo.bin 2>/dev/null
make[1]: Leaving directory /home/kfranz/Tutorial/u-boot-Digilent-Dev/examples/api'
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$
```

Figure 34. Compile U-Boot.

3. After the compilation, the **ELF** (**E**xecutable and **L**inkable **F**ile) generated is named `u-boot`. We need to add a `.elf` extension to the file name so that Xilinx SDK can read the file layout and generate `BOOT.BIN`. In this tutorial, we are going to move the `u-boot.elf` to the `sd_image` folder and substitute the `u-boot.elf` that comes along with the *ZYBO Base System Design Package*, as shown in Fig. 35.

```
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$ cp u-boot ../zybo_base_system/sd_image/u-boot.elf
[kfranz@DIGILENT_LINUX u-boot-Digilent-Dev]$
```

Figure 35. Add .elf.

3 Generate BOOT.BIN

3.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website Download Page.
- *ZYBO Base System Design:* available at the Digilent Website on the ZYBO Page.
- Finished the hardware customization from **Section 1** and `u-boot.elf` from **Section 2** (Section 2 optional).

3.2 Instructions

1. Export the hardware design (after Section 1, step 16) to Xilinx SDK by clicking on **File -> Export -> Export Hardware for SDK…**, as shown in Fig. 36.

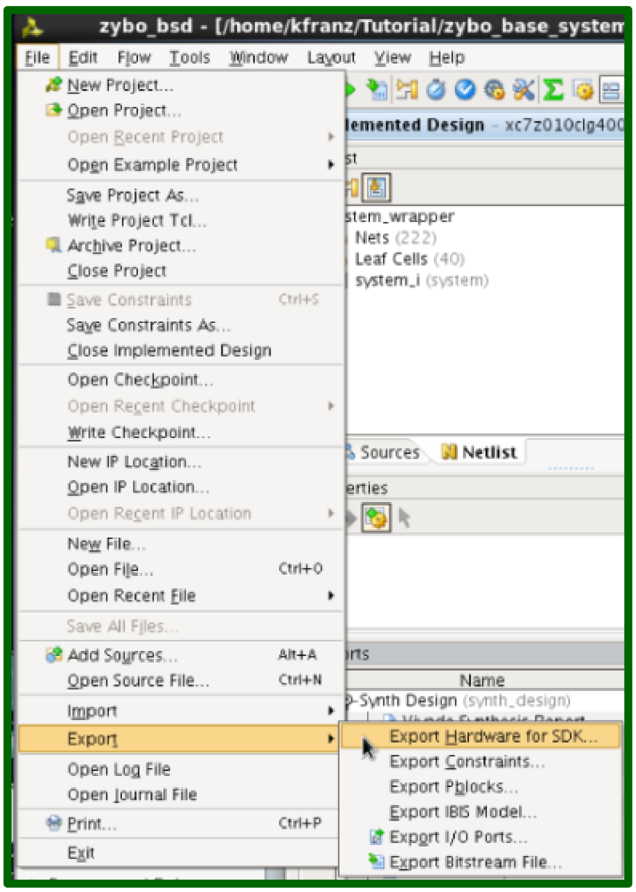

Figure 36. Export Hardware Design to SDK.

2. Leave the workspace as **<Local to Project>**. Make sure that the "Launch SDK" box is checked and click **OK**, as shown in Fig. 37.

   Note: If you are using Vivado 2014.1, you may have to export twice.

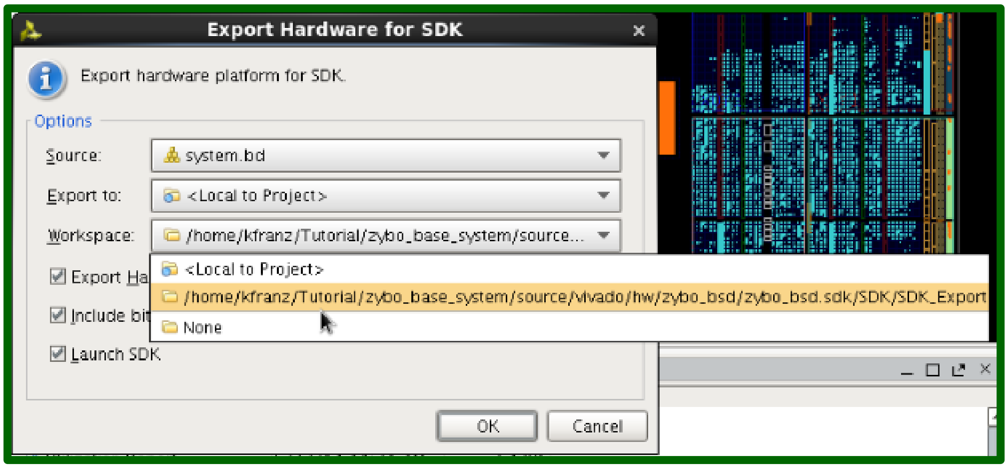

Figure 37. Set SDK Workspace Path.

3. After SDK launches, the hardware platform project is already present in Project Explorer on the left of the SDK main window, as shown in Fig. 38. We now need to create a First Stage Bootloader (FSBL). Click **File->New->Project...**, as shown in Fig. 39.

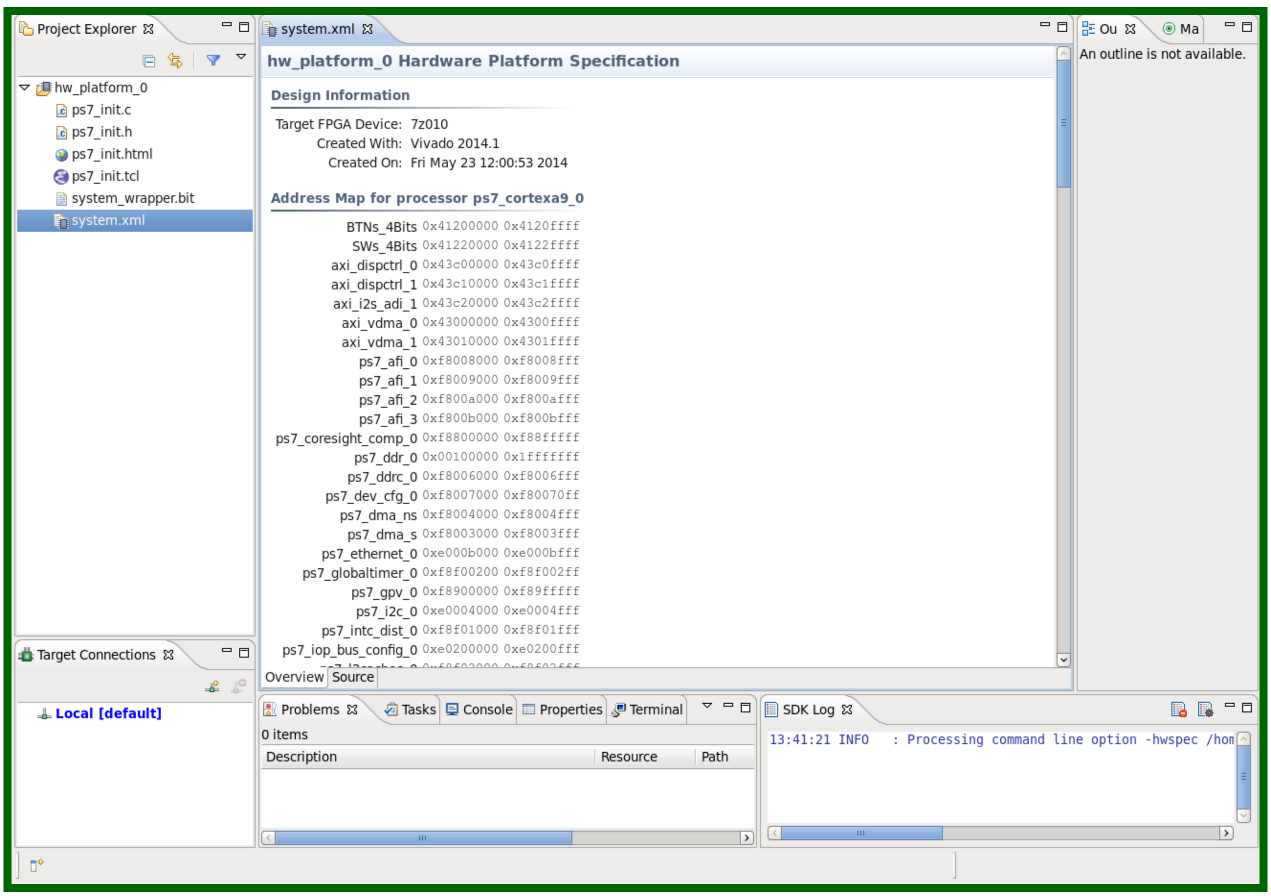

Figure 38. Export hardware design to SDK.

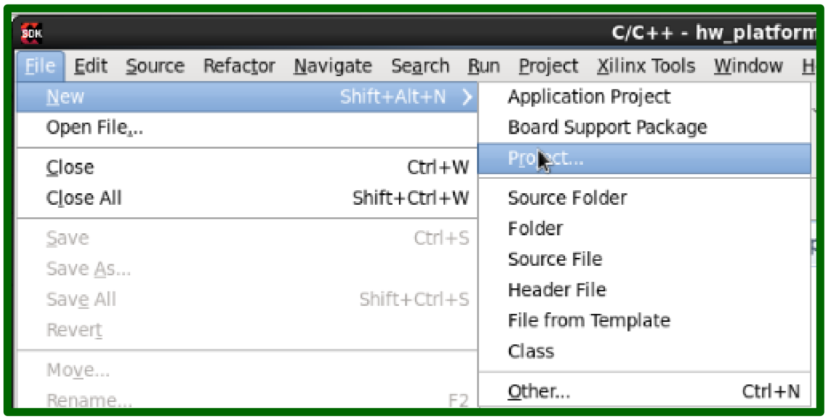

Figure 39. Create new project in SDK.

4. In the New Project window, select **Xilinx->Application Project**, and then Click **Next** (Fig. 40).

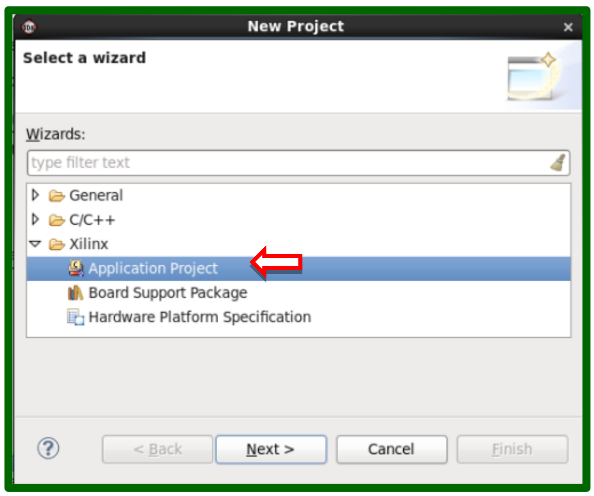

Figure 40. Select Application Project Wizard.

5. We will name the project FSBL. Select **hw_platform_0** for **Target Hardware** because it is the hardware project we just exported. Select **standalone** for **OS Platform**. Click **Next**, as shown in Fig. 41.

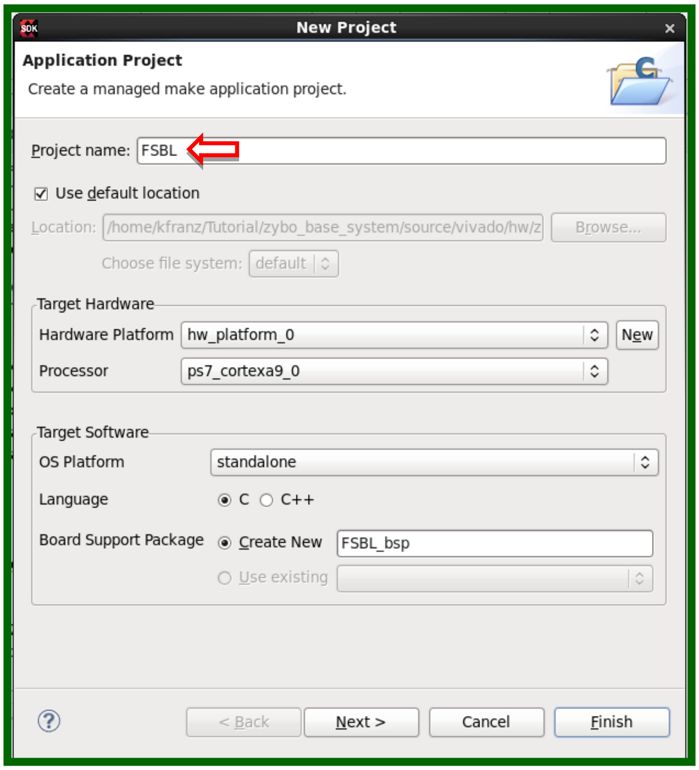

Figure 41. New Application Project.

6. Select **Zynq FSBL** as template, and click Finish as shown in Fig. 42.

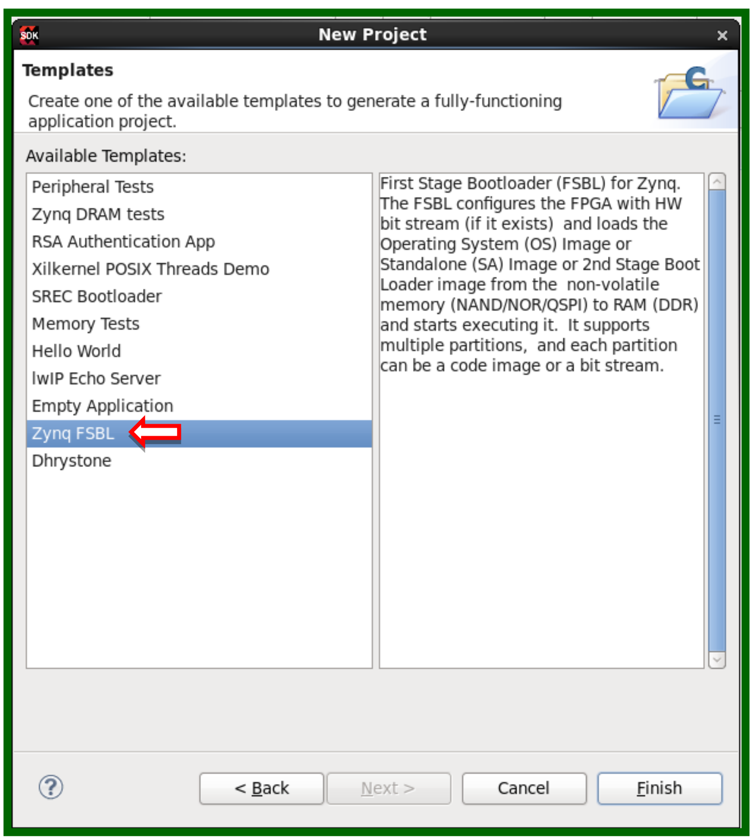

Figure 42. Select Zynq FSBL as template.

7. For the ZYBO, we need to set the mac address for the Ethernet in the fsbl hook. We want the mac address for the Ethernet to remain constant when we turn the ZYBO board off and on. You can swap the `fsbl_hooks.c` file in the FSBL project with the `fsbl_hooks.c` under `source/vivado/SDK/fsbl` in the *ZYBO Base System Design* (Fig. 43).

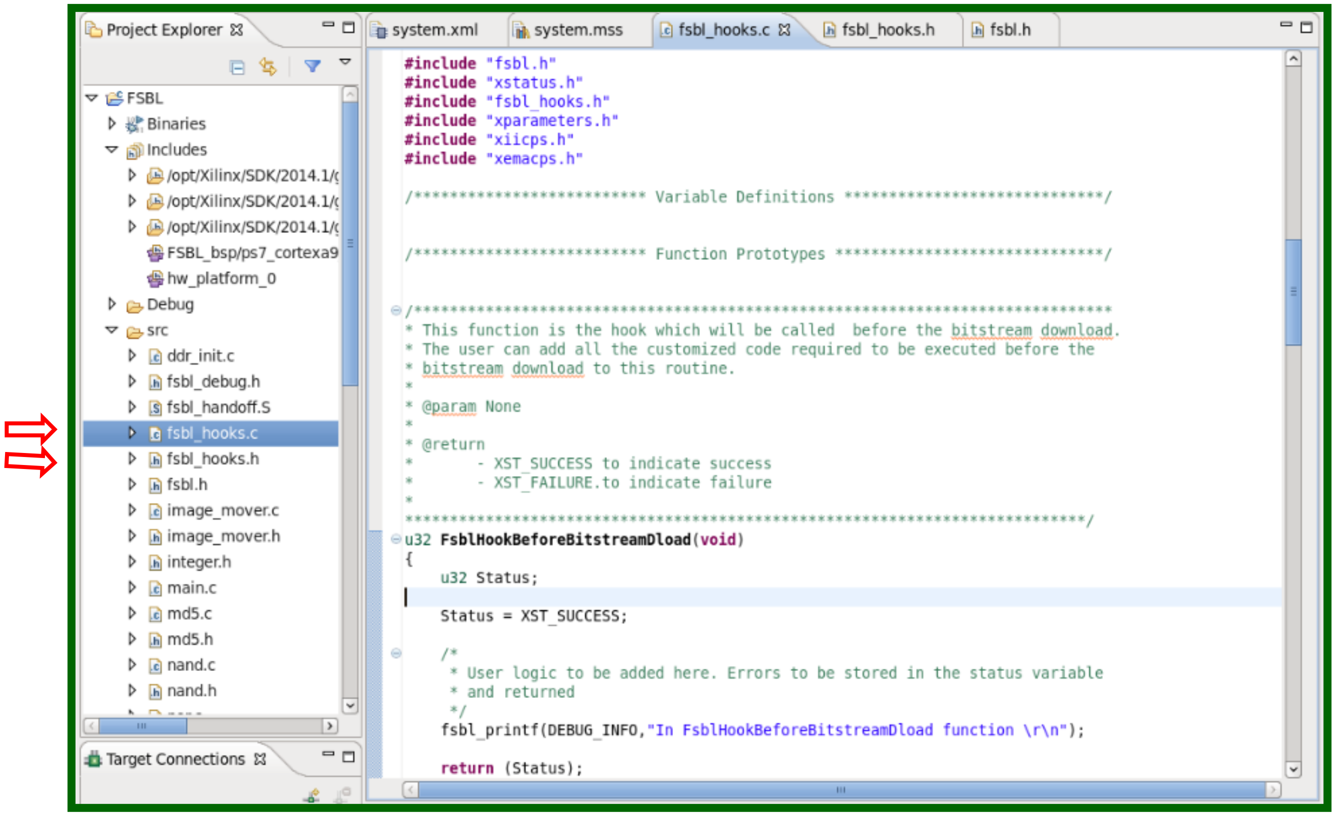

Figure 43. fsbl_hooks.c

8. After you have saved the changes to `fsbl_hooks.c`, the project will rebuild itself automatically. If it does not rebuild, click **Project->Clean** to clean the project files, and **Project->Build All** to rebuild all the projects. The compiled ELF file is located in:

```
ZYBO_base_system/source/vivado/hw/ZYBO_bsd.sdk/SDK/SDK_Export/FSBL/Debug
```

9. Now we have all of the files ready to create `BOOT.BIN`. Click **Xilinx Tools -> Create Zynq Boot Image**, as shown in Fig. 44.

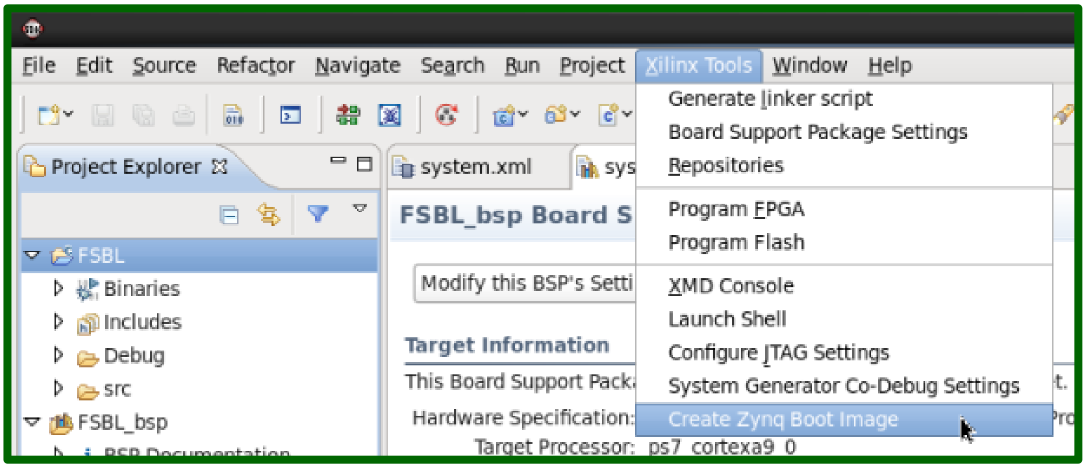

Figure 44. Create Zynq Boot Image.

10. In the Create Zynq Boot Image window (as shown in Fig. 45), Click **Browse** to set the path for **FSBL elf**. Click **Add** to add the `system.bit` file found at: `/ZYBO_base_system/source/vivado/hw/ZYBO_bsd/ZYBO_bsd.sdk/SDK/SDK_Export/hw_platform_0/`.Click **Add** to add the `u-boot.elf` file found at: `ZYBO_base_system/sd_image/`. *It is very important that the 3 files are added in this order,* or else the FSBL will not work properly (the proper order can be seen in Fig. 45). It is also very important that you set `FSBL.elf` as the bootloader and `system.bit` and `u-boot.elf` as data files. In this tutorial, the `sd_image` folder is set as output folder for the BIN file. Click **Create Image**.

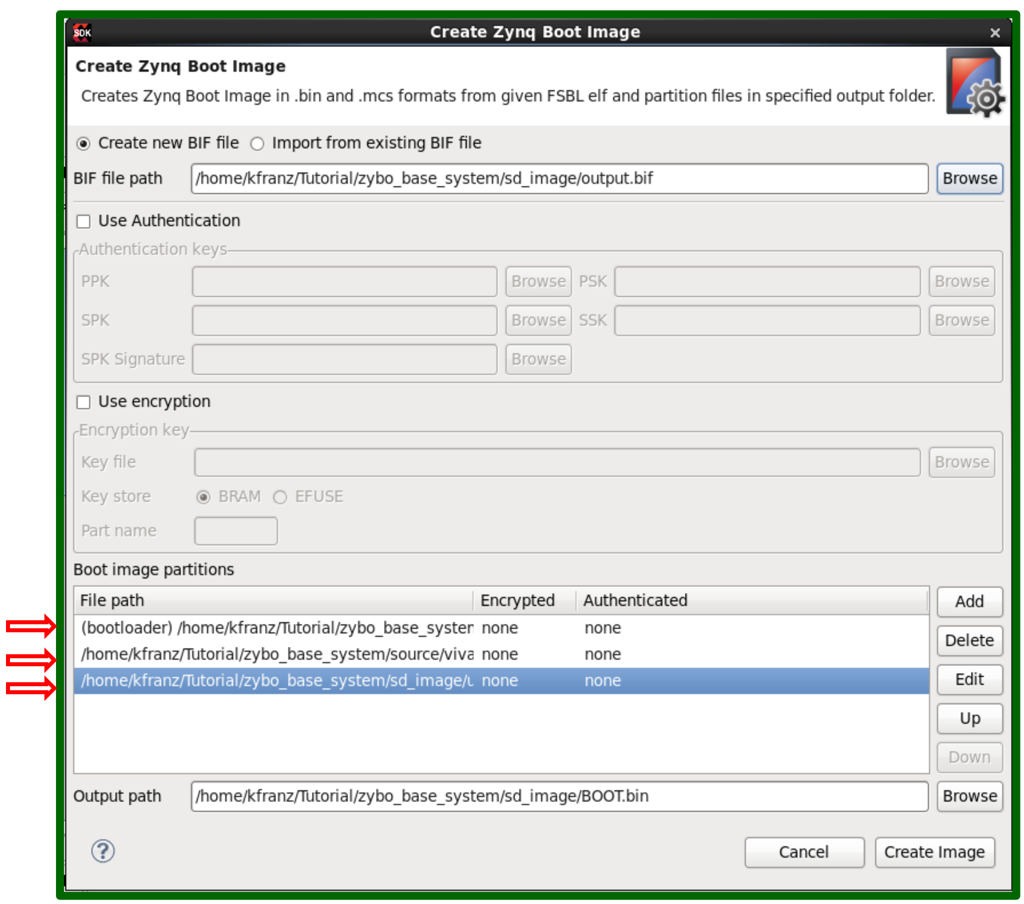

Figure 45. Create Zynq Boot Image Configuration.

11. The created BIN file was named BOOT.bin.

4 Compile Linux Kernel

4.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website [Download Page].
- *ZYBO Base System Design:* available at the Digilent Website on the [ZYBO Page].

4.2 Instructions (Use the Master-Next Branch Until Further Notice)

1. Get the Linux kernel source code from Digilent Git repository. There are two ways to retrieve the source code:

   Using *git* command: If you have Git installed in your distribution, you can clone the repository to your computer by command `git clone https://github.com/DigilentInc/Linux-Digilent-Dev.git`
   The whole Git Repository is around 850MB, as shown in Fig. 46.

```
[kfranz@DIGILENT_LINUX ~]$ git clone https://github.com/DigilentInc/Linux-Digile
nt-Dev.git
Initialized empty Git repository in /home/kfranz/Linux-Digilent-Dev/.git/
remote: Counting objects: 3586185, done.
remote: Compressing objects: 100% (549192/549192), done.
remote: Total 3586185 (delta 3007223), reused 3586185 (delta 3007223)
Receiving objects: 100% (3586185/3586185), 864.81 MiB | 2.76 MiB/s, done.
Resolving deltas: 100% (3007223/3007223), done.
[kfranz@DIGILENT_LINUX ~]$
```

Figure 46. Clone Kernel.

2. We will start to configure the kernel with the default configuration for ZYBO. The configuration is located at `arch/arm/configs/xylinx_zynq_defconfig`. To use the default configuration, you can follow Fig. 47.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- xilinx_zynq_defconfig
#
# configuration written to .config
#
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

Figure 47. Default Configuration.

3. Follow Fig. 48 to compile the Linux Kernel.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
scripts/kconfig/conf --silentoldconfig Kconfig
  CHK     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
```

```
  Kernel: arch/arm/boot/Image is ready
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  Building modules, stage 2.
  MODPOST 23 modules
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

Figure 48. Compile Kernel.

4. After the compilation, the kernel image is located at arch/arm/boot/zImage. However, in this case the kernel image has to be a uImage (unzipped) rather than a zimage. To make the uimage, follow Fig. 49.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-  UIMAGE_LOADADDR=0x8000 uImage
  CHK     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: `include/generated/mach-types.h' is up to date.
```

```
Data Size:    3819232 Bytes = 3729.72 kB = 3.64 MB
Load Address: 00008000
Entry Point:  00008000
  Image arch/arm/boot/uImage is ready
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

Figure 49. Create uImage.

Note: Depending on your distribution of Linux, you may get an error regarding the path of the mkimage. If this is the case, you can change the path following Fig. 50.

```
[kfranz@DIGILENT_LINUX Tutorial]$ PATH=$PATH:/home/kfranz/Tutorial/u-boot-Digilent-Dev/tools/
[kfranz@DIGILENT_LINUX Tutorial]$ echo $PATH
```

Figure 50. Change Path.

5 Test Kernel Image with Pre-built File System

5.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website Download Page.
- *Linux Kernel Source Code*: available at Digilent GitHub repository https://github.com/DigilentInc/Linux-Digilent-Dev. (Use the Master-Next Branch Until Further Notice)
- Pre-built File System Image: ramdisk Image is available in ZYBO Linux Reference Design.
- BOOT.BIN from **Section 3**, uImage from **Section 4**.

5.2 Instructions

1. To boot the Linux operating system on the ZYBO, you need BOOT.BIN, a Linux kernel image (uImage), a device tree blob (DTB file), and a file system. BOOT.BIN has been created in Section 3 and uImage has been compiled in Section 4. We will now compile the DTB file. The default device tree source file is

located in the Linux Kernel source at arch/arm/boot/dts/zynq-ZYBO.dt**s.**

RAMDISK: modify the device tree source file according to Fig. 51. For Zynq, only the ramdisk image has to be wrapped in a u-boot header in order for u-boot to boot with it. This is shown in Fig. 52.

```
48    chosen {
49        /* bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
    rootfstype=ext4 rootwait devtmpfs.mount=1"; */
50        bootargs = "console=ttyPS0,115200 root=/dev/ram rw initrd=0x800000,8M
    init=/init earlyprintk rootwait devtmpfs.mount=1";
51        linux,stdout-path = "/axi@0/serial@e0001000";
52    };
```

Figure 51. Edit device tree.

```
[kfranz@DIGILENT_LINUX Tutorial]$ ./u-boot-Digilent-Dev/tools/mkimage -A arm -T
ramdisk -c gzip -d ./ramdisk8M.image.gz uramdisk.image.gz
Image Name:
Created:      Mon Jun  9 12:39:18 2014
Image Type:   ARM Linux RAMDisk Image (gzip compressed)
Data Size:    3694108 Bytes = 3607.53 kB = 3.52 MB
Load Address: 00000000
Entry Point:  00000000
[kfranz@DIGILENT_LINUX Tutorial]$
```

Figure 52. Make Uramdisk Image.

2. Generate DTB file, as shown in Fig. 53.

```
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$ ./scripts/dtc/dtc -I dts -O dtb -o .
./devicetree.dtb arch/arm/boot/dts/zynq-zybo.dts
[kfranz@DIGILENT_LINUX Linux-Digilent-Dev]$
```

Figure 53. Generate DTB File.

3. **(RAMDISK)** Copy BOOT.BIN, devicetree.dtb, uImage and uramdisk.image.gz to the first partition of an SD card, as shown in Fig. 54.

```
[kfranz@DIGILENT_LINUX Tutorial]$ ls
devicetree.dtb  linux-digilent-dev  u-boot-digilent  ZYBO_base_system
[kfranz@DIGILENT_LINUX Tutorial]$ cp ZYBO_base_system/sd_image/BOOT.BIN /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$ cp ZYBO_base_system/sd_image/ uramdisk.image.gz /BOOT.BIN
/media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$ cp ./devicetree.dtb /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$ cp Linux-Digilent-Dev/arch/arm/boot/uImage /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX Tutorial]$
```

Figure 54. Ramdisk.

4. Plug the SD card into the ZYBO. To boot from the SD card, jumper 7 needs to be configured for USB, as shown on the ZYBO board, and Jumper 5 must be connected to SD. Connect UART port to PC with a micro USB cable and set the UART terminal on PC to 115200 baud rate, 8 data bits, 1 stop bit, no parity, and no flow control. After powering on the board, the console (shown in Fig. 55) should be seen at the UART terminal if you use RamDisk. More information about these file systems can be found in *Getting Started with Embedded Linux - ZYBO.*

```
[    1.170000] fb0:  frame buffer device
[    1.170000] drm: registered panic notifier
[    1.170000] [drm] Initialized analog_drm 1.0.0 20110530 on minor 0
[    1.260000] EXT4-fs (ram0): couldn't mount as ext3 due to feature incompatibilities
[    1.310000] EXT4-fs (ram0): mounting ext2 file system using the ext4 subsystem
[    1.310000] EXT4-fs (ram0): warning: mounting unchecked fs, running e2fsck is recommended
[    1.320000] EXT4-fs (ram0): mounted filesystem without journal. Opts: (null)
[    1.320000] VFS: Mounted root (ext2 filesystem) on device 1:0.
[    1.330000] devtmpfs: mounted
[    1.330000] Freeing init memory: 152K
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Configure static IP 192.168.1.10
[    1.510000] GEM: lp->tx_bd ffdfa000 lp->tx_bd_dma 19bd7000 lp->tx_skb d8070480
[    1.520000] GEM: lp->rx_bd ffdfb000 lp->rx_bd_dma 19bd8000 lp->rx_skb d8070580
[    1.520000] GEM: MAC 0x00350a00, 0x00002201, 00:0a:35:00:01:22
[    1.530000] GEM: phydev d8b6f400, phydev->phy_id 0x1410dd1, phydev->addr 0x0
[    1.530000] eth0, phy_addr 0x0, phy_id 0x01410dd1
[    1.540000] eth0, attach [Marvell 88E1510] phy driver
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
++ Starting OLED Display
[    1.580000] pmodoled-gpio-spi [zed_oled] SPI Probing
++ Exporting LEDs & SWs
rcS Complete
zynq>
```

Figure 55. Ramdisk, UART Console after boot up.

6 Modify Device Tree and Compose Kernel Driver

6.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website Download Page.
- Linux Kernel Source Code: available at Digilent GitHub repository https://github.com/Digilentinc/Linux-Digilent-Dev (Use the Master-Next Branch Until Further Notice)

6.2 Instructions

1. Create a directory named "drivers" in the Tutorial folder, as shown in Fig. 56. Inside the driver's directory, we will compose the driver for the `myLed` controller.

```
[kfranz@DIGILENT_LINUX Tutorial]$ mkdir drivers
[kfranz@DIGILENT_LINUX Tutorial]$ ls
BOOT.bin             output.bif               vivado.jou
drivers              ps_clock_registers.log   vivado.log
Linux-Digilent-Dev   u-boot-Digilent-Dev      zybo_base_system
[kfranz@DIGILENT_LINUX Tutorial]$
```

Figure 56. Driver Directory.

2. We need a Makefile so that we can compile the driver. The Makefile is created in Fig. 57.

```
[kfranz@DIGILENT_LINUX Tutorial]$ cd drivers
[kfranz@DIGILENT_LINUX drivers]$ vim Makefile
```

Figure 57. Create Makefile.

After creating the file, hit I to change to insert mode and insert the following text (Fig. 58).

```
obj-m := myled.o

                                                    all:
       make -C ../Linux-Digilent-Dev/ M=$(PWD) modules

   clean:

       make -C ../Linux-Digilent-Dev/ M=$(PWD) clean
```

Figure 58. Makefile.

Note: make sure the spacing in the Makefile is made up of tabs, not spaces, where necessary. Then hit esc to exit insert mode and :x to save the file and exit vim editor.

3. We will start with a simple driver that creates a file named myled under the Linux /proc file system. The status of the on-board LEDs can be changed by writing a number to the file. The driver is coded in Fig. 59.

```
[kfranz@DIGILENT LINUX drivers]$ vim myled.c
```

Figure 59. Create myled.c

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/uaccess.h>    /* Needed for copy_from_user */
#include <asm/io.h>         /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h>  /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h>  /* Needed for Platform Driver Functions */

/* Define Driver Name */
#define DRIVER_NAME "myled"

unsigned long *base_addr;   /* Vitual Base Address */
struct resource *res;       /* Device Resource Structure */
unsigned long remap_size;   /* Device Memory Size */

/* Write operation for /proc/myled
 * -----------------------------------
 *  When user cat a string to /proc/myled file, the string will be stored in
 *  const char __user *buf. This function will copy the string from user
 *  space into kernel space, and change it to an unsigned long value.
 *  It will then write the value to the register of myled controller,
 *  and turn on the corresponding LEDs eventually.
 */
static ssize_t proc_myled_write(struct file *file, const char __user * buf,
                size_t count, loff_t * ppos)
{
    char myled_phrase[16];
    u32 myled_value;

    if (count < 11) {
        if (copy_from_user(myled_phrase, buf, count))
            return -EFAULT;

        myled_phrase[count] = '\0';
    }

    myled_value = simple_strtoul(myled_phrase, NULL, 0);
    wmb();
    iowrite32(myled_value, base_addr);
    return count;
}

/* Callback function when opening file /proc/myled
 * ------------------------------------------------------
 *  Read the register value of myled controller, print the value to
 *  the sequence file struct seq_file *p. In file open operation for /proc/myled
 *  this callback function will be called first to fill up the seq_file,
 *  and seq_read function will print whatever in seq_file to the terminal.
 */
static int proc_myled_show(struct seq_file *p, void *v)
{
    u32 myled_value;
    myled_value = ioread32(base_addr);
    seq_printf(p, "0x%x", myled_value);
    return 0;
}
```

Figure 60. myled.c

```
/* Open function for /proc/myled
 * ------------------------------------
 *  When user want to read /proc/myled (i.e. cat /proc/myled), the open function
 *  will be called first. In the open function, a seq_file will be prepared and the
 *  status of myled will be filled into the seq_file by proc_myled_show function.
 */
static int proc_myled_open(struct inode *inode, struct file *file)
{
    unsigned int size = 16;
    char *buf;
    struct seq_file *m;
    int res;

    buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
    if (!buf)
        return -ENOMEM;

    res = single_open(file, proc_myled_show, NULL);

    if (!res) {
        m = file->private_data;
        m->buf = buf;
        m->size = size;
    } else {
        kfree(buf);
    }

    return res;
}

/* File Operations for /proc/myled */
static const struct file_operations proc_myled_operations = {
    .open = proc_myled_open,
    .read = seq_read,
    .write = proc_myled_write,
    .llseek = seq_lseek,
    .release = single_release
};

/* Shutdown function for myled
 * -----------------------------------
 *  Before myled shutdown, turn-off all the leds
 */
static void myled_shutdown(struct platform_device *pdev)
{
    iowrite32(0, base_addr);
}

```

Figure 60. myled.c (Cont.)

```
/* Remove function for myled
 * ----------------------------------
 *  When myled module is removed, turn off all the leds first,
 *  release virtual address and the memory region requested.
 */
static int myled_remove(struct platform_device *pdev)
{
    myled_shutdown(pdev);

    /* Remove /proc/myled entry */
    remove_proc_entry(DRIVER_NAME, NULL);

    /* Release mapped virtual address */
    iounmap(base_addr);

    /* Release the region */
    release_mem_region(res->start, remap_size);

    return 0;
}

/* Device Probe function for myled
 * ------------------------------------
 *  Get the resource structure from the information in device tree.
 *  request the memory regioon needed for the controller, and map it into
 *  kernel virtual memory space. Create an entry under /proc file system
 *  and register file operations for that entry.
 */
static int myled_probe(struct platform_device *pdev)
{
    struct proc_dir_entry *myled_proc_entry;
    int ret = 0;

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!res) {
        dev_err(&pdev->dev, "No memory resource\n");
        return -ENODEV;
    }

    remap_size = res->end - res->start + 1;
    if (!request_mem_region(res->start, remap_size, pdev->name)) {
        dev_err(&pdev->dev, "Cannot request IO\n");
        return -ENXIO;
    }

    base_addr = ioremap(res->start, remap_size);
    if (base_addr == NULL) {
        dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n",
            (unsigned long)res->start);
        ret = -ENOMEM;
        goto err_release_region;
    }

```

Figure 60. myled.c (Cont.)

```
    myled_proc_entry = proc_create(DRIVER_NAME, 0, NULL,
                                   &proc_myled_operations);
    if (myled_proc_entry == NULL) {
        dev_err(&pdev->dev, "Couldn't create proc entry\n");
        ret = -ENOMEM;
        goto err_create_proc_entry;
    }

    printk(KERN_INFO DRIVER_NAME " probed at VA 0x%08lx\n",
           (unsigned long) base_addr);

    return 0;

 err_create_proc_entry:
    iounmap(base_addr);
 err_release_region:
    release_mem_region(res->start, remap_size);

    return ret;
}

/* device match table to match with device node in device tree */
static const struct of_device_id myled_of_match[] = {
    {.compatible = "dglnt,myled-1.00.a"},
    {},
};

MODULE_DEVICE_TABLE(of, myled_of_match);

/* platform driver structure for myled driver */
static struct platform_driver myled_driver = {
    .driver = {
           .name = DRIVER_NAME,
           .owner = THIS_MODULE,
           .of_match_table = myled_of_match},
    .probe = myled_probe,
    .remove = myled_remove,
    .shutdown = myled_shutdown
};

/* Register myled platform driver */
module_platform_driver(myled_driver);

/* Module Infomations */
MODULE_AUTHOR("Digilent, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": MYLED driver (Simple Version)");
MODULE_ALIAS(DRIVER_NAME);
```

Figure 60. myled.c (Cont.)

4. Compile and generate the driver module using make (as shown in Fig. 61). Don't forget to source Vivado settings.

```
[kfranz@DIGILENT_LINUX drivers]$ make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
make -C ../Linux-Digilent-Dev/ M=/home/kfranz/Tutorial/drivers modules
make[1]: Entering directory `/home/kfranz/Tutorial/Linux-Digilent-Dev'
  CC [M]  /home/kfranz//Tutorial/drivers/myLed.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/kfranz/Tutorial/drivers/myLed.mod.o
  LD [M]  /home/kfranz/Tutorial/drivers/myLed.ko
make[1]: Leaving directory `/home/kfranz/Tutorial/Linux-Digilent-Dev'
[kfranz@DIGILENT_LINUX drivers]$
```

Figure 61. Compile Driver.

5. We need to add the **myLed** device node into the device tree. Make a copy of the default device tree source in the `drivers` folder, and modify it according to Fig. 62. The compatibility string of the node is the same as we define in the driver source code (myled.c: line 182). The reg property defines the physical address and size of the node. The address here should match with the address of the myLed IP Core in the address editor tab of the Vivado design, as shown in Fig. 63.

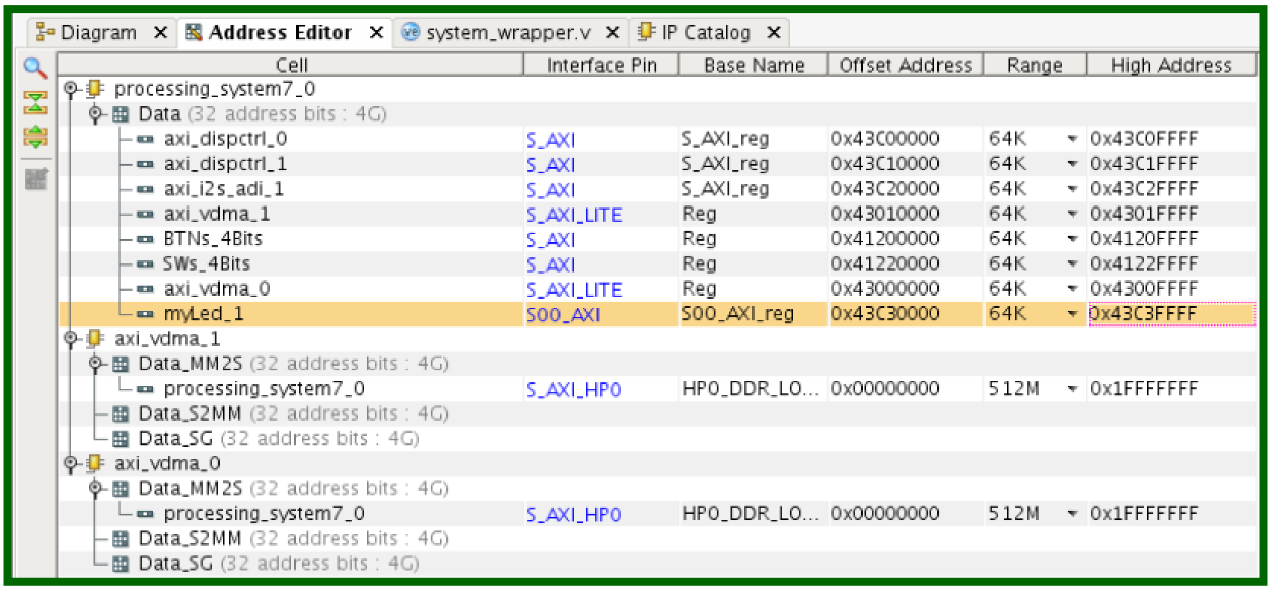

Figure 62. Physical Address for myLed IP Core.

```
[kfranz@DIGILENT_LINUX drivers]$ cp ../Linux-Digilent-Dev/arch/arm/boot/dts/zynq-ZYBO.dts ./
[kfranz@DIGILENT_LINUX drivers]$ vim zynq-ZYBO.dts
```

Figure 63. Edit device tree.

```
            spi-speed-hz = <4000000>;
            spi-sclk-gpio = <&ps7_gpio_0 59 0>;
            spi-sdin-gpio = <&ps7_gpio_0 60 0>;
        };

        myled {
            compatible = "dglnt,myled-1.00.a";
            reg = <0x43c30000 0x10000>;
        };
    };
};
```

Figure 64. zynq-ZYBO.dts

6. Recompile the device tree blob as shown in Fig. 65.

```
[kfranz@DIGILENT_LINUX drivers]$ ../Linux-Digilent-Dev/scripts/dtc/dtc -I dts -O dtb -o
devicetree.dtb zynq-ZYBO.dts
DTC: dts->dtb  on file "zynq-ZYBO.dts"
[kfranz@DIGILENT_LINUX drivers]$
```

Figure 65. Compile DTB.

7. Copy these two files to the first partition of the SD card, as shown in Fig. 66. We are ready to test our driver on-board now.

```
[kfranz@DIGILENT_LINUX drivers]$ ls
devicetree.dtb   Makefile       Module.symvers  myled.ko     myled.mod.o
zynq-ZYBO.dts  modules.order  myled.c         myled.mod.c  myled.o
[kfranz@DIGILENT_LINUX drivers]$ cp myled.ko /media/ZYBO_BOOT/d
[kfranz@DIGILENT_LINUX drivers]$ cp devicetree.dtb /media/ZYBO_BOOT/
[kfranz@DIGILENT_LINUX drivers]$
```

Figure 66. Copy files to SD.

8. Plug the SD card into the ZYBO and we can start testing our driver. Use the **insmod** command to install the driver module into the kernel. After the driver is installed, an entry named **myled** will be created under the `/proc` file system. Writing `0x0F` to `/proc/myled` will light up **LED 0~3**. You can either remove the driver with command **rmmod** or power off the system by command **poweroff**. In both cases, all of the LEDs will be turned off, as shown in Fig. 67. For instructions on using the terminal with the ZYBO, please refer to **Section 5, Step 4** or the Section **Boot from SD** in *Getting Started with Embedded Linux – ZYBO*.

```
U-Boot 2012.04.01-dirty (June 30 2014 - 12:52:36)

DRAM:  512 MiB
WARNING: Caches not enabled
MMC:   SDHCI: 0
Using default environment
...
reading uImage

2457328 bytes read
reading devicetree.dtb

9728 bytes read
reading uramdisk.image.gz

3694108 bytes read
## Starting application at 0x00008000 ...
Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0
[    0.000000] Linux version 3.6.0-digilent-13.01-00002-g06b3889 (kfranz@DIGILENT_LINUX)
(gcc version 4.6.3 (Sourcery CodeBench Lite 2012.03-79) ) #1 SMP PREEMPT Sun June 30
23:54:12 PST 2014
...
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt/
zynq> cd /mnt/
zynq> ls
BOOT.BIN             devicetree.dtb       uramdisk.image.gz
myled.ko              uImage
zynq> insmod myled.ko
[  122.160000] myled probed at va 0xe0d20000
zynq> ls /proc
1             567           9             fs            partitions
10            582           asound        interrupts    scsi
11            588           buddyinfo     iomem         self
12            594           bus           ioports       slabinfo
13            595           cmdline       irq           softirqs
14            596           config.gz     kallsyms      stat
15            6             consoles      kmsg          swaps
2             608           cpu           kpagecount    sys
3             614           cpuinfo       kpageflags    sysvipc
317           615           crypto        loadavg       timer_list
318           621           device-tree   locks         tty
333           641           devices       meminfo       uptime
4             642           diskstats     misc          version
429           643           dma           modules       vmallocinfo
440           647           dri           mounts        vmstat
441           652           driver        mtd           zoneinfo
5             653           execdomains   myled
515           7             fb            net
548           8             filesystems   pagetypeinfo
zynq> echo 0x0F > /proc/myled
zynq> cat /proc/myled
0x0f
zynq> mkdir -p /lib/modules/`uname -r`
zynq> cp myled.ko /lib/modules/`uname -r`
zynq> rmmod myled
```

Figure 67. RAMDISK

7 User Application

7.1 Prerequisites

- Vivado 2014.1 WebPACK: available at the Xilinx Website Download Page.

7.2 Instructions

1. In this section, we will write a user application that makes the LEDs blink by writing to /proc/myled. Create a directory named **user_app** in the Tutorial folder, as shown in Fig. 68. Inside the **user_app** directory, we will compose the led_blink.c, as shown in Fig. 69.

```
[kfranz@DIGILENT_LINUX Tutorial]$ mkdir user_app
[kfranz@DIGILENT_LINUX Tutorial]$ ls
devicetree.dtb  drivers   linux-digilent  u-boot-digilent  user_app  ZYBO_base_system
[kfranz@DIGILENT_LINUX Tutorial]$
```

Figure 68. User_app

```
[kfranz@DIGILENT_LINUX user_app]$ vim led_blink.c
```

Figure 69. led_blink

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    FILE* fp;
    while(1) {
        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
            printf("Cannot open /proc/myled for write\n");
            return -1;
        }
        fputs("0x0F\n", fp);
        fclose(fp);
        sleep(1);
        fp = fopen("/proc/myled", "w");
        if(fp == NULL) {
            printf("Cannot open /proc/myled for write\n");
            return -1;
        }
        fputs("0x00\n", fp);
        fclose(fp);
        sleep(1);
    }
    return 0;
}
```

Figure 70. led_blink.c

2. Compose a Makefile and compile led_blink.c into led_blink.o, as shown in Figs. 71-73.

```
[kfranz@DIGILENT_LINUX user_app]$ vim Makefile
```

Figure 71. Makefile.

```
CC = arm-xilinx-linux-gnueabi-gcc
CFLAGS = -g

all : led_blink

led_blink : led_blink.o
    ${CC} ${CFLAGS} $^ -o  $@

clean :
    rm -rfv *.o
    rm -rfv led_blink

.PHONY : clean
```

Figure 72. Makefile.

```
[kfranz@DIGILENT_LINUX user_app]$ make
arm-xilinx-linux-gnueabi-gcc -g   -c -o led_blink.o led_blink.c
arm-xilinx-linux-gnueabi-gcc -g -o led_blink led_blink.o
[kfranz@DIGILENT_LINUX user_app]$ ls
led_blink  led_blink.c  led_blink.o  Makefile
[kfranz@DIGILENT_LINUX user_app]$
```

Figure 73. Compile led_blink.

3. Insert the SD card into the computer, and copy the binary file `led_blink` onto the first partition of SD card, as shown in Fig. 74.

```
[kfranz@DIGILENT_LINUX user_app]$ cp led_blink /media/ZYBO_BOOT/
```

Figure 74. Move led_blink.

```
...
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt/
zynq> cd /mnt/
zynq> ls
BOOT.BIN              devicetree.dtb        led_blink
myled.ko              ramdisk8M.image.gz    zImage
zynq> insmod myled.ko
[  122.160000] myled probed at va 0x8000
zynq> ./led_blink
^C
zynq> mkdir -p /lib/modules/`uname -r`
zynq> cp myled.ko /lib/modules/`uname -r`
zynq> rmmod myled
```

Figure 75. RAMDISK.