

Using the Embedded MicroBlaze Processor

Acknowledgement

This module is derived from labs developed by Xilinx.

Goal

- Use the Vivado tool to build a basic MicroBlaze system. This will consist of a MicroBlaze processor, on-chip memory, GPIO, and a UART.
- Be able to use a C program in the SDK to interact with the processor.
- Use some software debugging tools in an embedded processor environment.

Requirements

- Xilinx Vivado software
- Xilinx SDK software
- Xilinx Nexys 4 DDR board and a programming cable
- Enough disk space for the project files

Background

Soft processors, such as MicroBlaze, are implemented using programmable logic (FPGA LUTs and registers) and can be custom configured to suit the software application they run. In contrast, **hard processors**, such as the Intel i7 CPU, are implemented as application-specific integrated circuits (ASICs), and their functions are fixed after manufacture. Soft processors are common in embedded systems built with FPGAs, and they usually provide high-level control logic for the system. They can be quickly and easily programmed with a C-based language, and have the advantage of customization. For example, the user can choose between various instruction and data caches, as well as choose to include custom instructions for faster execution of an application.

Introduction

In this tutorial you will create a simple MicroBlaze system for an Artix-7 FPGA using the Vivado IP integrator.

The MicroBlaze system includes native Xilinx IP such as the:

- MicroBlaze processor
- AXI Timer
- UARTLite
- Debug Module (MDM)
- Proc Sys Reset
- Interrupt Controller
- Local memory bus (LMB)

These are the basic building blocks used in a typical MicroBlaze system.

In addition to creating the system described above, this tutorial also describes the development of a small application that you develop in the Xilinx Software Development Kit (SDK) in the Vivado Design Suite. The application code developed in the SDK prints “Hello World” on a terminal.

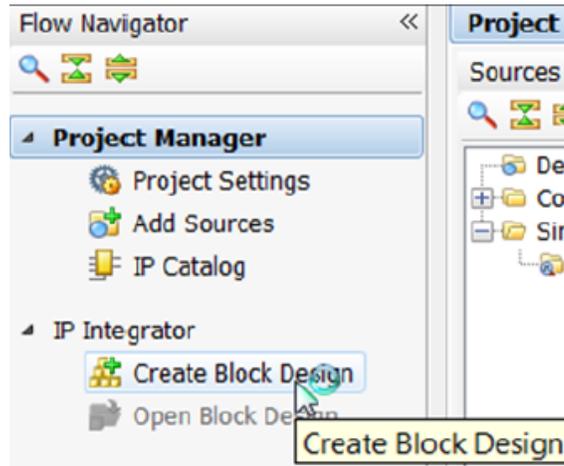
This tutorial targets the Xilinx Nexys 4 DDR FPGA Evaluation Board, and uses the 2016.2 version of Vivado Design Suite.

1. Create a Project

1. Invoke the Vivado IDE
2. From the Getting Started page, select **Create New Project**
3. In the **Project Name** dialog box, type the project name and location.
4. In the **Project Type** dialog box, select RTL Project.
5. In the **Add Sources** dialog box, ensure that the Target language is set to Verilog.
6. Click **Next** in the **Add Constrains** and **Add Existing IP** dialog box
7. Choose xc7a100tcsq324-1 in the **Default Part** dialog box.
8. Click **Finish** to finish creating the project.

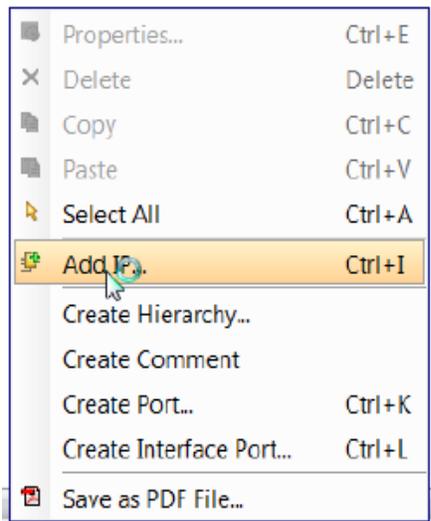
2. Create an IP Integrator Design

1. From Navigator > IP Integrator, select **Create Block Design**

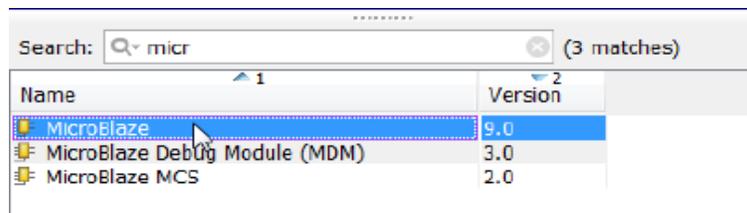


2. Specify the design name and click OK.

3. Right click anywhere in the Diagram and select **Add IP**.

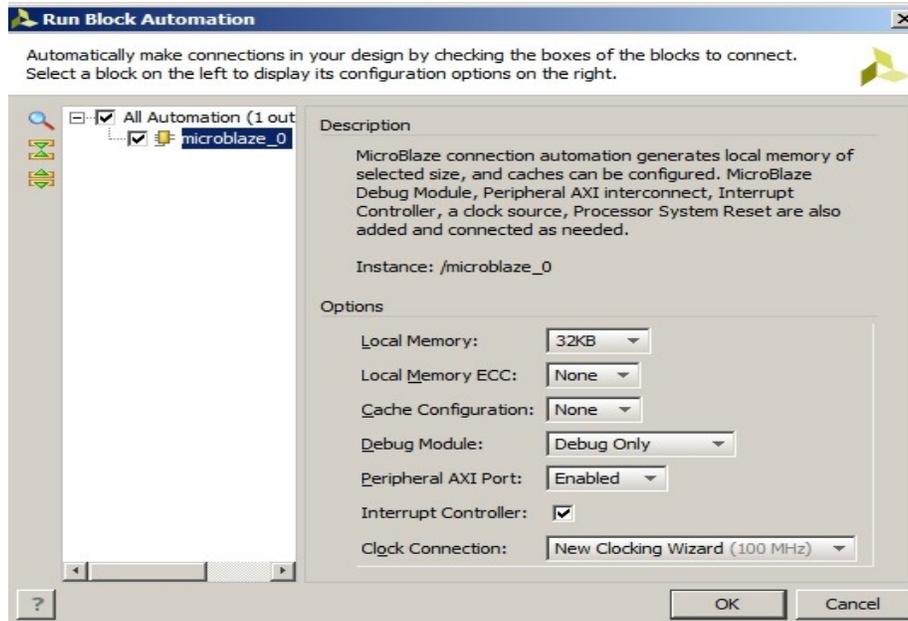


4. In the Search field, type **microblaze** to find the MicroBlaze IP, then click Enter.



Customize the MicroBlaze Processor

1. In the Diagram view, click the *Run Block Automation* link beside *Designer Assistance* available



2. From the pulldown menu, set Local Memory to 32 KB.

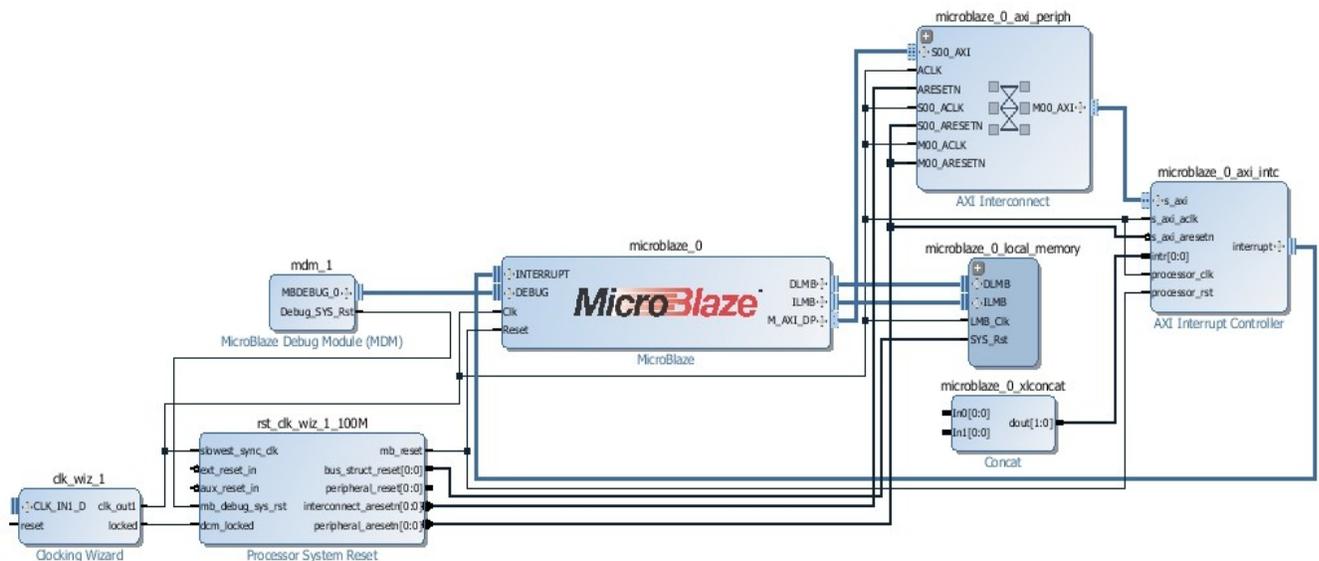
3. Leave the Debug Module option to its default state **Debug Only**.

4. Leave the Peripheral AXI Port option as **Enabled**.

5. Check the **Interrupt Controller** option.

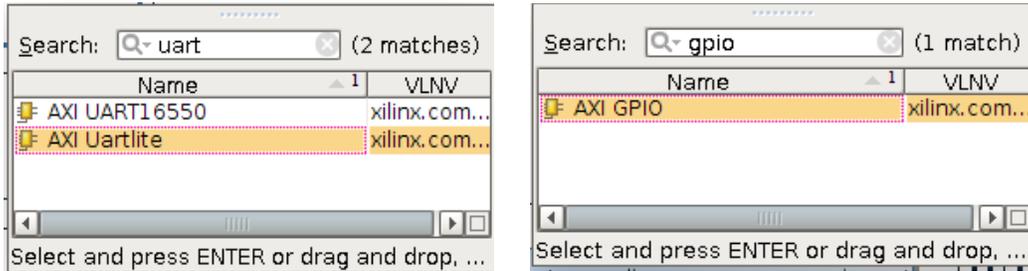
6. Select the Clock Connection option of **New Clocking Wizard (100 Mhz)**. This will create a clock signal in the block design.

The generated basic MicroBlaze system should look like the following:



Add peripherals: AXI Uartlite, GPIOs

1. Right click anywhere in the block diagram, Add IP and search for and select the **AXI Uartlite**.
2. Repeat step 1 but search for **AXI GPIO**.
3. Repeat step 2 so that you have two GPIO blocks in your design. The reason why there are two GPIO blocks is that one of them is for switches and another one is for LEDs so that you can control the LEDs using the switches.



4. Right click one of the GPIO block and click **Block Properties**. Change the name to **gpio_led**.
5. Right click the other GPIO block and click **Block Properties**. Change the name to **gpio_switch**.

Block Configurations and Connections:

Interrupts

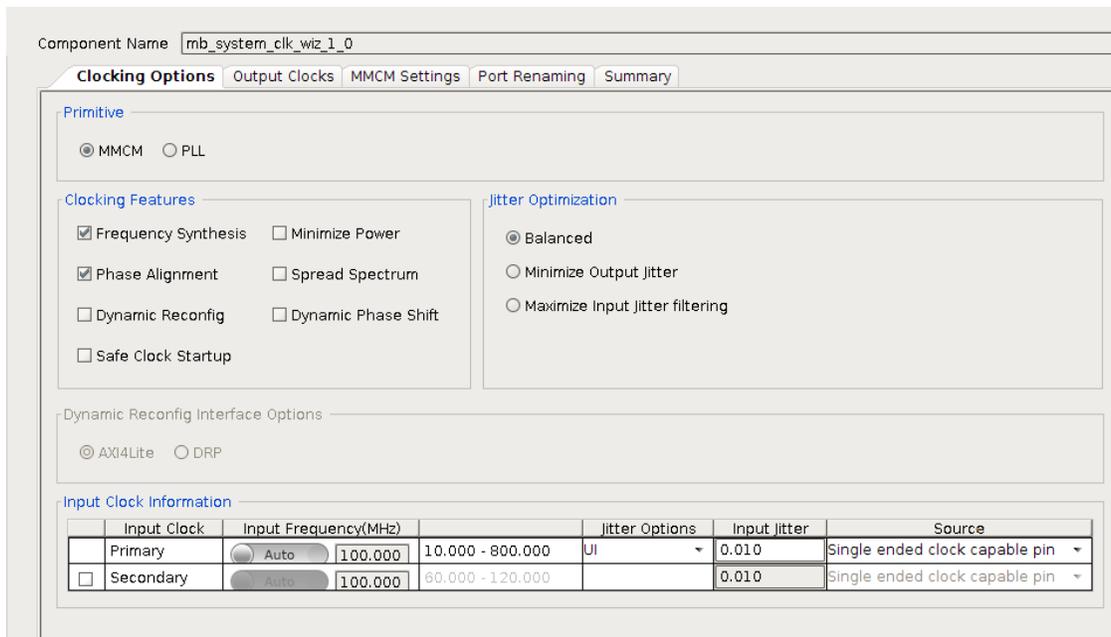
1. Connect the **interrupt** signal of the AXI Uartlite to the **ln0** signal of Concat, which connects to the interrupt input of the interrupt controller. Concat is used to concatenate individual signals into a bus signal.

Move the mouse to the **interrupt** signal and left click on it, drag it to ln0 and release the mouse. You'll see a green check mark when you do it. That means you can make the connection.

Note that there is one input of the Concat block that is not connected and there is no other input to connect to the block.

Leaving the unconnected input open will result in critical warnings and ultimately an error when you reach the SDK. There are several ways to deal with this particular situation. To reduce the number of ports, double click on the Concat block and change the number of ports to 1 in the Re-customize IP window. Click OK.

Clock Wizard:

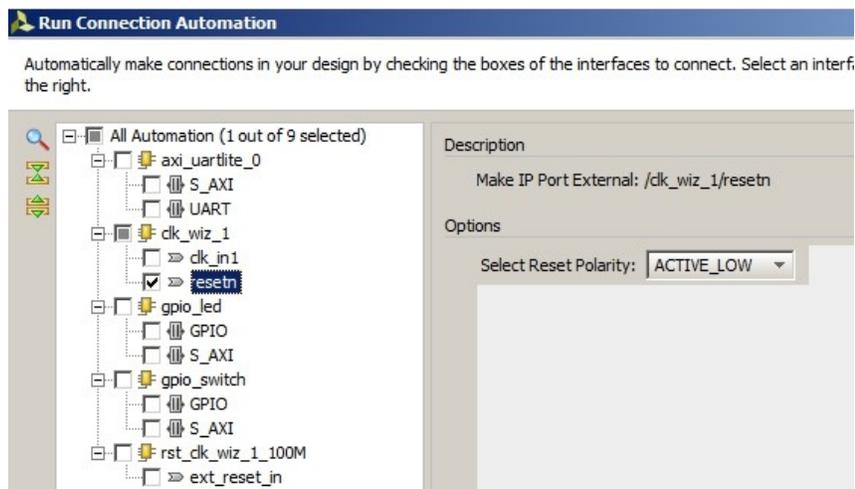


2. Double click the **Clocking Wizard** and change the Source of Primary clock input to **Single ended clock capable pin**. The clock supplied to the FPGA will use a single wire instead of a differential signal requiring two wires, which is more often used for very high speed clocks.

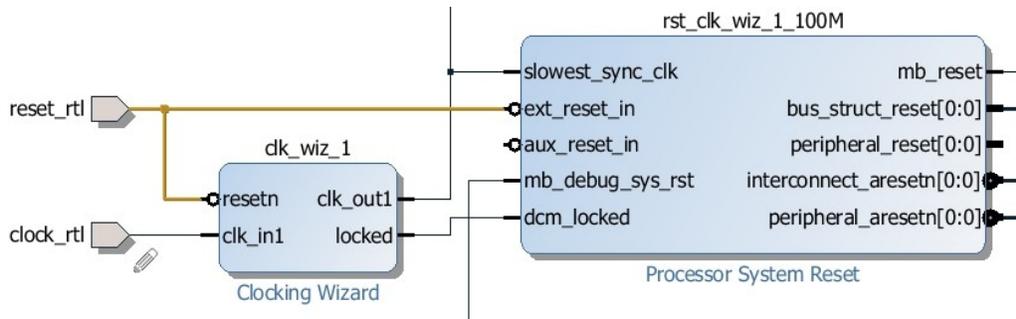
In the **Output Clocks** tab, change the **Reset Type** to Active Low and Press OK.

Reset Connections:

1. Click on **Run Connection Automation** and choose `clk_wiz_1/resetn`. Make the clock reset **active low** and press OK. This will generate a new `reset_rtl` pin in the block diagram.



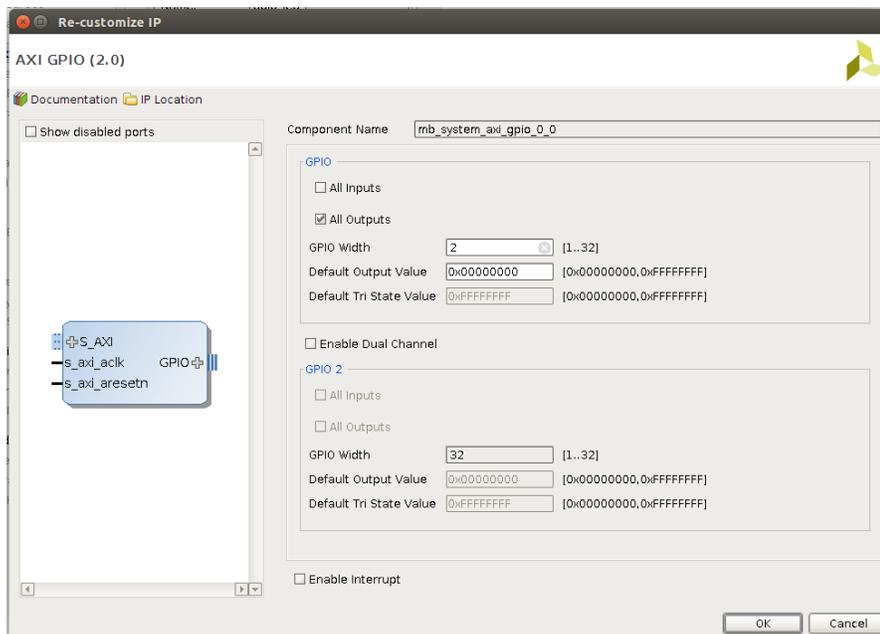
2. Connect the **ext_reset_in** pin of the Processor System Reset block to the **reset_rtl** pin.



GPIO Configuration:

Double click on **gpio_led** to bring up the configuration window. Check the **All Outputs** and change the **GPIO Width** to 2 as we only need to use 2 LEDs for this lab. If you need more LEDs, you can change the width to the value you wish.

Double click on **gpio_switch** block and configure the block to accept two *inputs* in a similar manner.



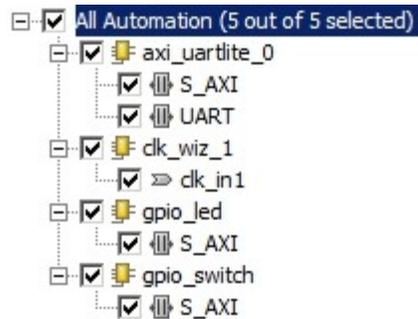
Create GPIO Ports:

Expand the GPIO port on **gpio_switch** by clicking the plus sign. Right click on **gpio_io_i** and choose **make external**.

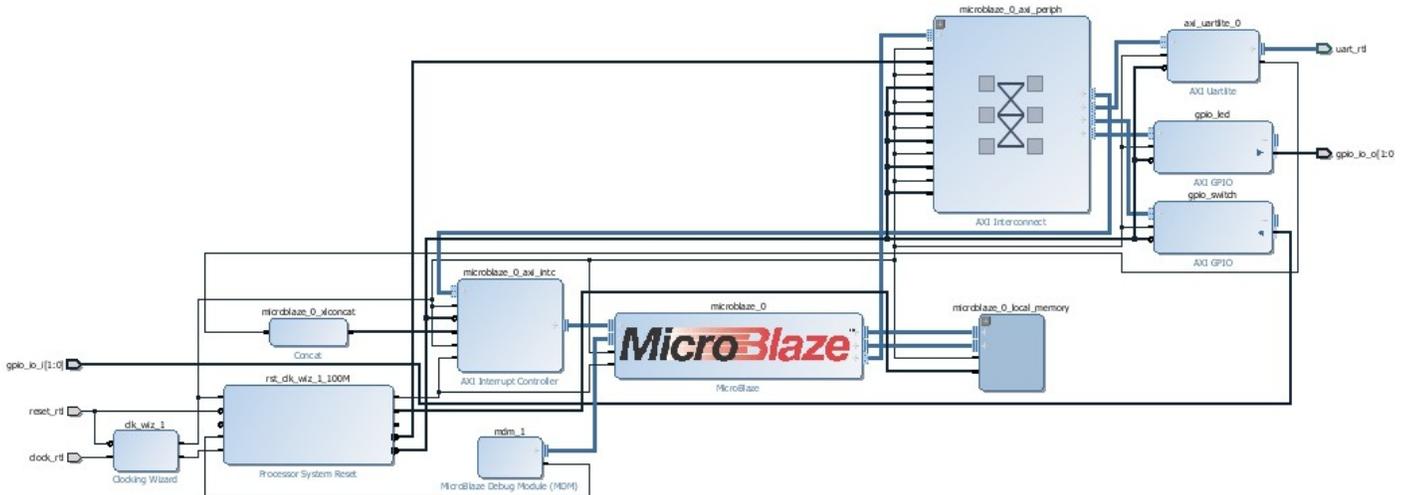
Expand the GPIO port on **gpio_led** by clicking the plus sign. Right click on **gpio_io_o** and choose **make external**.

AXI and Clock Connections:

To handle the remaining connections we will use connection automation. Click on **Run Connection Automation** and select **All Automation**. This will connect the GPIO and UART controllers to the AXI Interconnect and create an external pin for the clock source.



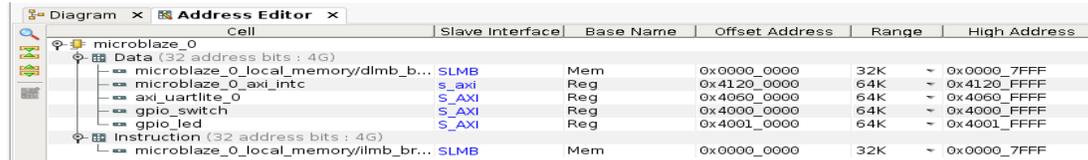
You should now have the following complete system with a MicroBlaze processor, three AXI peripherals, interrupt controller, local memory, debug module and clock and reset generators. Ensure you can identify the function of each block in your design.



Select Tools > Validate Design. Or simply just press F6. If you followed every step above, it should say validation successful.

Memory-Mapping

Click on **Address Editor** to examine the Address Map. This table shows how different peripherals are mapped into the address spaces of different bus masters. In this design, the master is the MicroBlaze soft processor. For instance, notice below that the gpio switch peripheral can be accessed starting from the address 0x4000_0000 from the Data port of the MicroBlaze.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
microblaze_0_local_memory/dlmb_b...	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
microblaze_0_axi_intc	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
gpio_switch	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
gpio_led	S_AXI	Reg	0x4001_0000	64K	0x4001_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_br...	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF

Creating Constraints

1. Click on **Add Sources** in the Flow Navigator box and choose **Add or Create Constraints** then click Next.
2. Click on **Create File** and give it a name. Click **OK** and then **Finish**.
3. Open up the constraint file you just created and copy the code below, then save it. You can find the file in the Sources tab under Constraints. Double click on the file name to edit it. Note that the reset signal is the CPU RESET button on the board. You can also import the XDC file packaged with this tutorial.

```
## Clock signal
```

```
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMOS33 } [get_ports { clock_rtl }];  
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock_rtl}];
```

```
## Reset (CPU_RESET)
```

```
set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 } [get_ports { reset_rtl }];
```

```
##Switches
```

```
set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 } [get_ports { gpio_io_i[0] }];  
set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 } [get_ports { gpio_io_i[1] }];
```

```
## LEDs
```

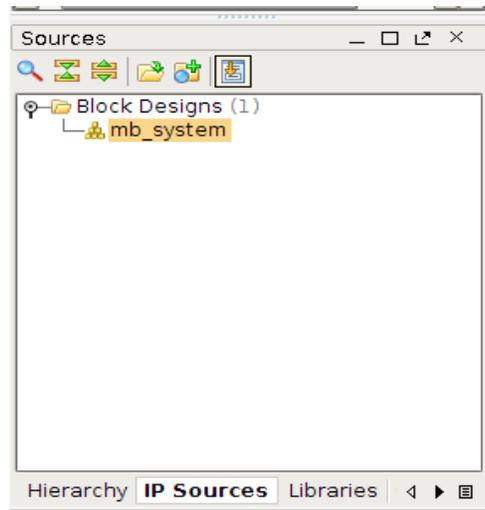
```
set_property -dict { PACKAGE_PIN H17    IOSTANDARD LVCMOS33 } [get_ports { gpio_io_o[0] }];  
set_property -dict { PACKAGE_PIN K15    IOSTANDARD LVCMOS33 } [get_ports { gpio_io_o[1] }];
```

```
##USB-RS232 Interface
```

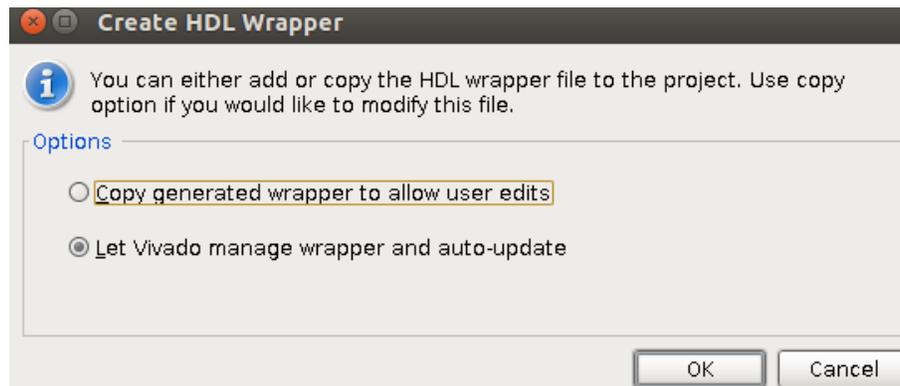
```
set_property -dict { PACKAGE_PIN C4     IOSTANDARD LVCMOS33 } [get_ports { uart_rtl_rxd }];  
set_property -dict { PACKAGE_PIN D4     IOSTANDARD LVCMOS33 } [get_ports { uart_rtl_txd }];
```

Wrap and build the design

1. In the **Sources** box, click **IP Sources**. Right click your design and choose **Create HDL Wrapper** to wrap your design. In this example, the design is called mb_system so right click on mb_system.



2. A window pops up as shown below. Leave the options as default then click OK.



3. Navigate through the hierarchy of the design in the **Hierarchy** tab. You will notice that a top-level verilog file is created by the HDL wrapper. As the design flow continues, HDL files will be generated for each block in the IP Integrator design.

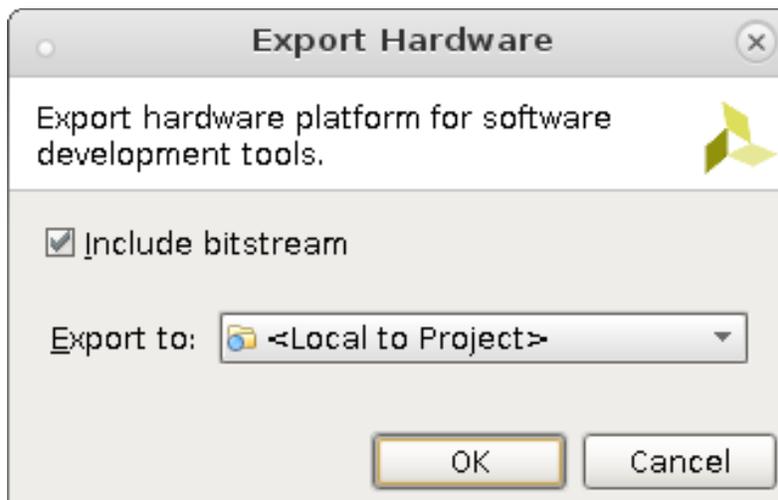
4. Click on **Run Synthesis**.

5. Click on **Run Implementation**.

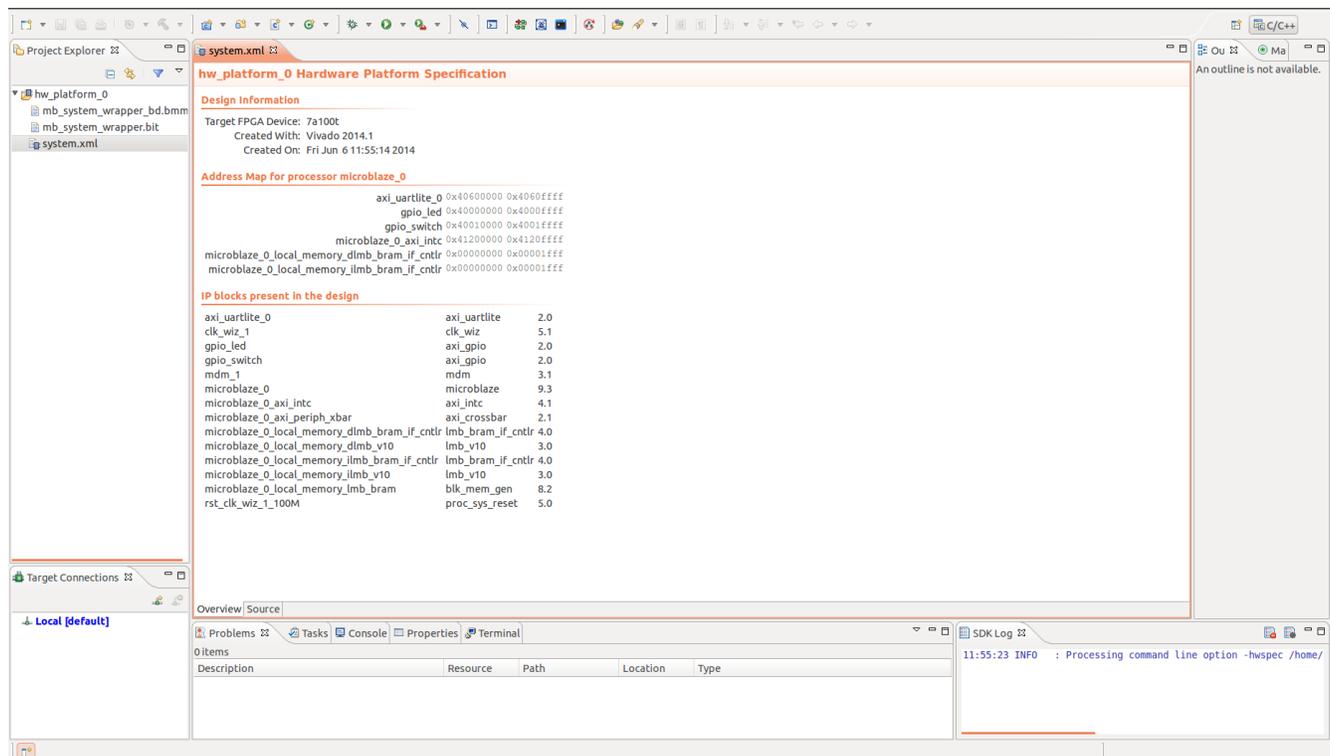
6. Click on **Generate Bitstream**.

3. Export to SDK

After generating the bitstream, click on **File** on the top and choose **Export**, then choose **Export Hardware**. Make sure you check **Include bitstream**. If the option is gray, simply open your Implemented Design under **Implementation** in the Flow Navigator and do the export again. Click on **File** on the top and choose Launch SDK so that after export, the SDK program will launch.

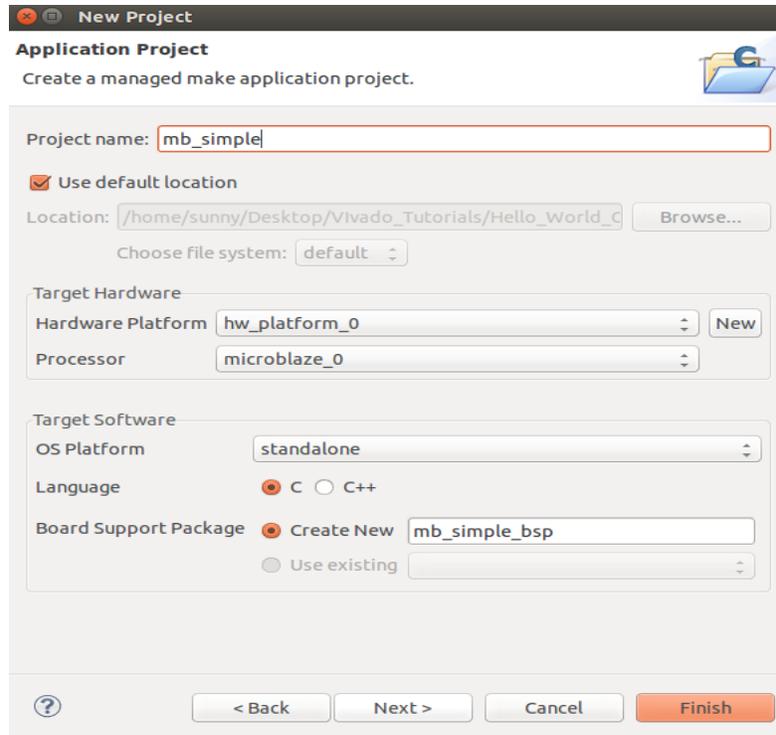


After the SDK first launches, double click system.hdf and you should see a description of the peripherals connected to the MicroBlaze processor.



Creating a Hello World program

1. Select File > New > Application Project
2. Enter the **Project name** and choose the **OS Platform** to be **standalone**. Click Next.

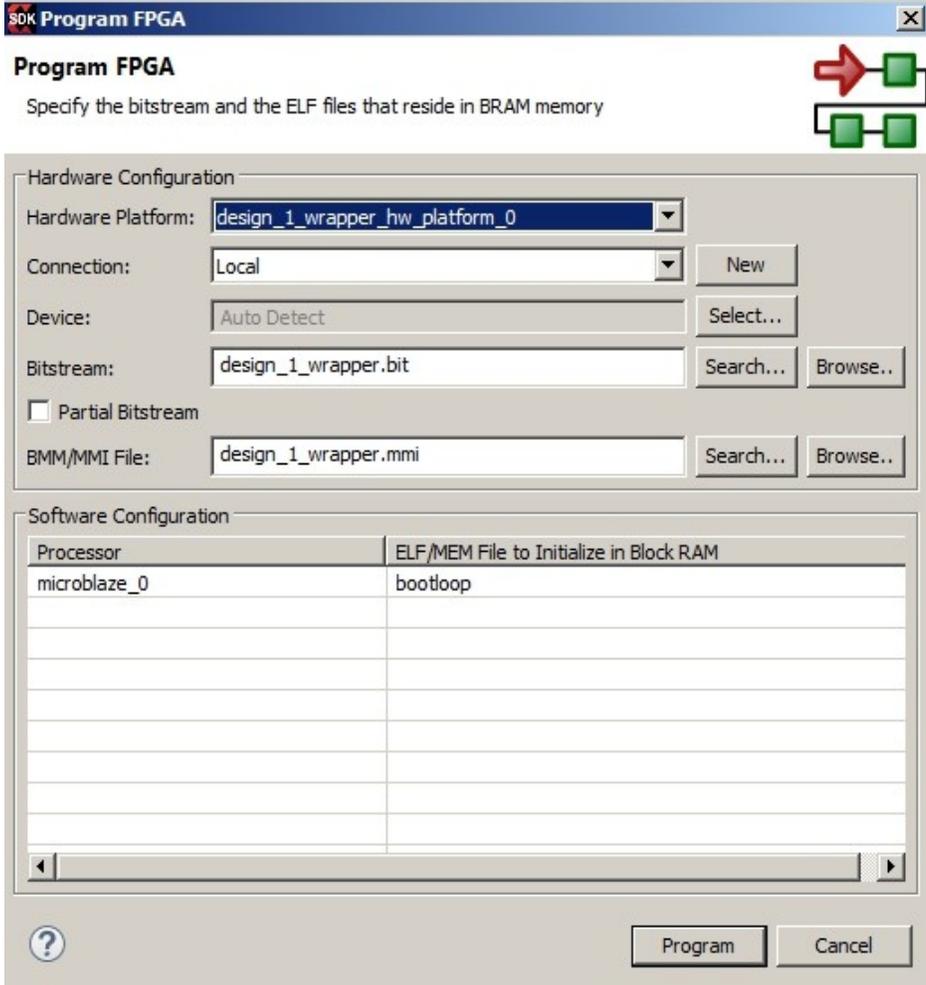


The screenshot shows the 'New Project' dialog box in Xilinx Vivado. The dialog is titled 'Application Project' and contains the following fields and options:

- Project name:** mb_simple
- Use default location
- Location:** /home/sunny/Desktop/Vivado_Tutorials/Hello_World_C (with a 'Browse...' button)
- Choose file system:** default
- Target Hardware:**
 - Hardware Platform:** hw_platform_0 (with a 'New' button)
 - Processor:** microblaze_0
- Target Software:**
 - OS Platform:** standalone
 - Language:** C C++
 - Board Support Package:** Create New mb_simple_bsp Use existing

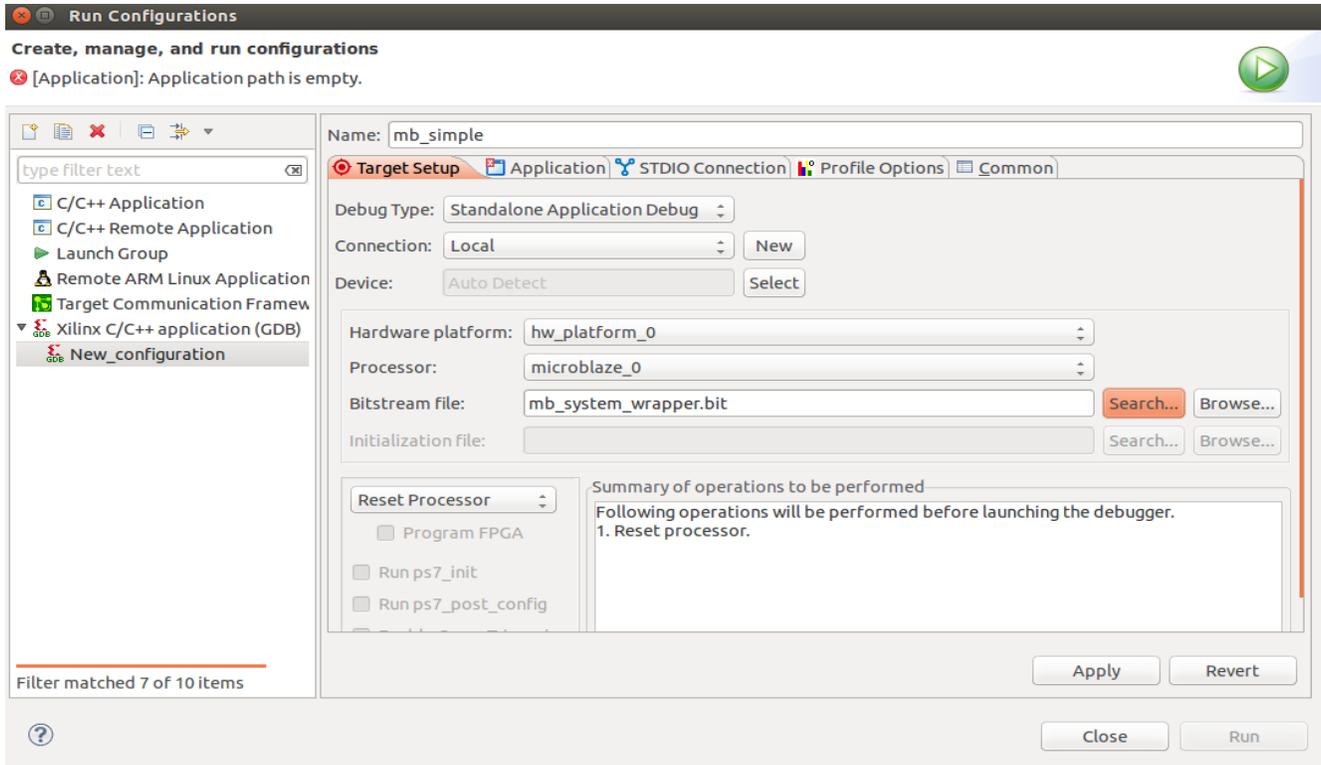
At the bottom of the dialog, there are buttons for '< Back', 'Next >', 'Cancel', and 'Finish'.

3. Choose **Hello World** as your template. Click Finish.
4. Connect your board to the computer and turn it on.
5. Click **Xilinx Tools** at the top -> **Program FPGA** to program your board.

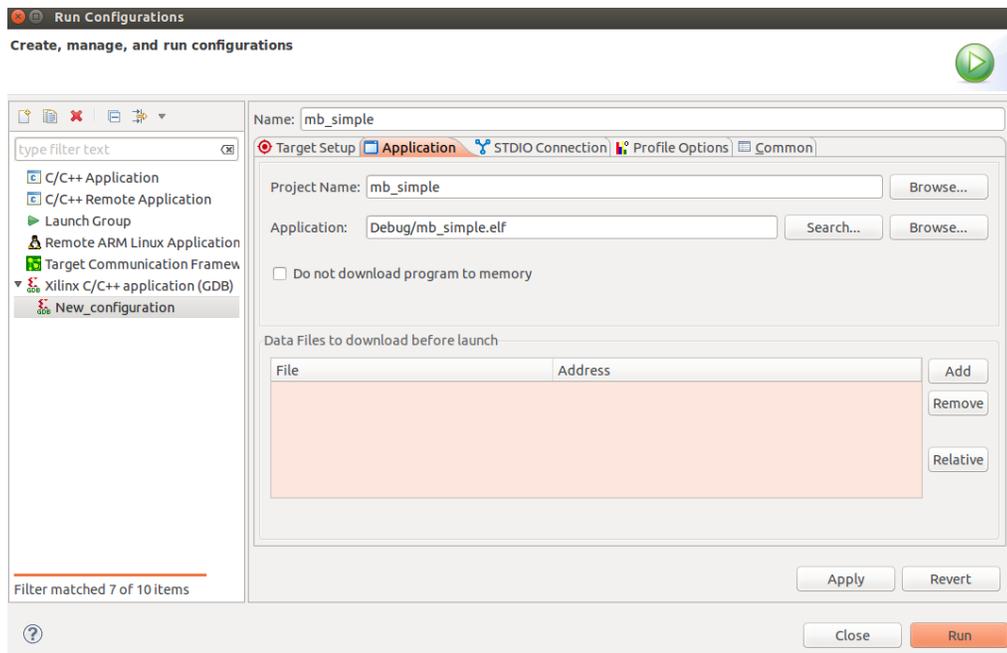


Run the C Program

1. Click on **Run** on the top -> **Run Configurations** to create a new run session.
2. Choose **Xilinx C/C++ application(GDB)** and click new on the top left corner to create a new launch.

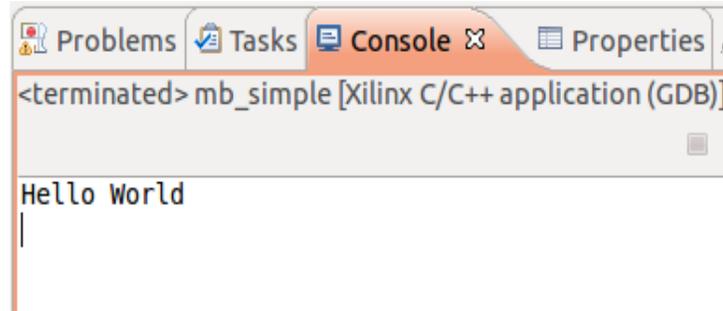


3. In **Application**, browse the Project Name and choose your project. In this example, it is mb_simple. You will notice that the Application file is automatically selected under Debug folder.



4. In **STDIO Connection**, check **Connect STDIO to Console** and select the largest COM port (Typically COM6). Leave the BAUD Rate at 9600. Leave everything else as default and click **Run**.

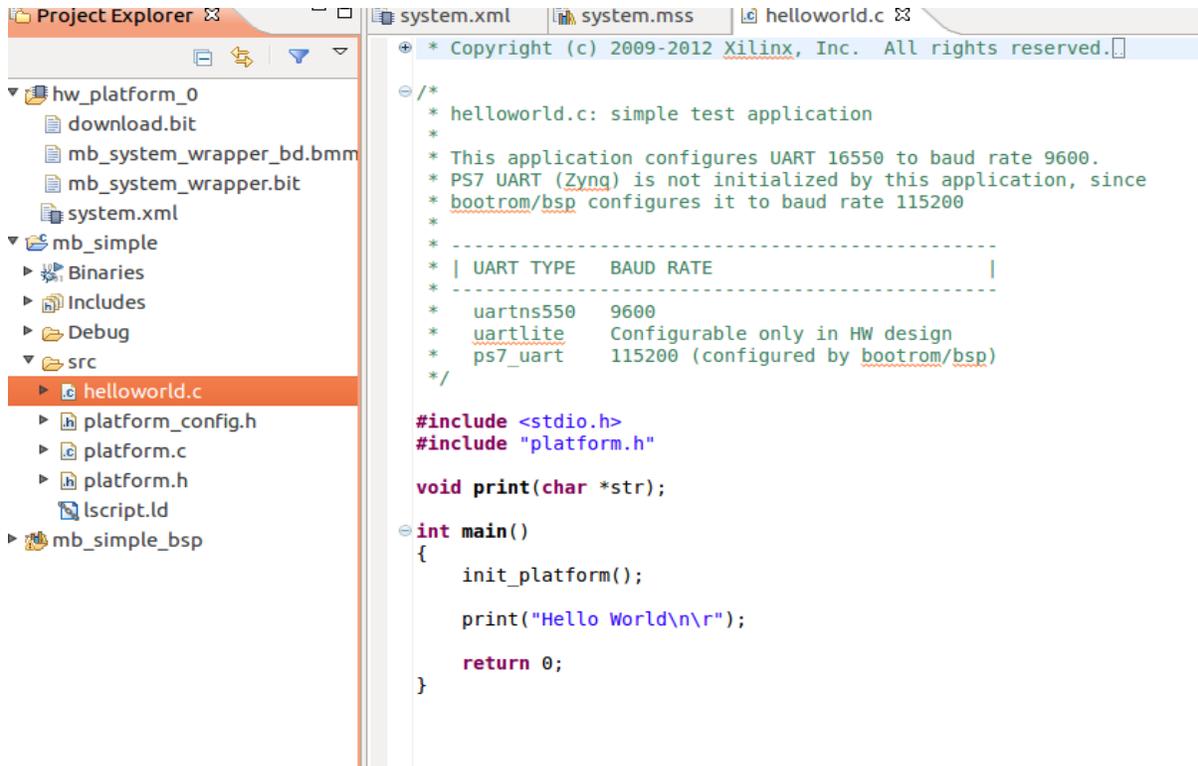
You will see Hello World is printed in the Console.



You may also leave the *Connect STDIO to Console* unchecked and instead establish a serial connection using the Terminal tab to view the output in that window.

Controlling LEDs using Switches

1. Open up your helloworld.c file under source file



2. Copy the following code to helloworld.c

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

volatile unsigned int * led = (unsigned int *)0x40010000;
volatile unsigned int * swt = (unsigned int *)0x40000000;

int main()
{
    init_platform();
    print("Hello World\n\r");

    while(1)
        *led = *swt;
    return 0;
}
```

In this example, the base address of the LED controller is 0x40010000 and the base address for the switches is 0x40000000. The infinite while loop keeps checking the value of the switches and assigns them to the LEDs. This way you can control the LEDs from the switches. **Double check the addresses** of your GPIO peripherals either by double clicking *system.hdf* in SDK or by looking at the Address Editor in the IP Integrator tool in Vivado.

3. Click **Run**.

4. Change the value of the switches and see the changes of leds.

Experiment with the program and block design to implement different logic between the LEDs and switches.