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 2014.1 version of Vivado Design Suite. To test your system on a Nexys 4 board, you must use a terminal emulation program such as TeraTerm, Hyperterminal, minicom or the Terminal in the SDK. You must also ensure that you have the device drivers for the board installed correctly.

1. Create a Project

1. Invoke the Vivado IDE

2. From the Getting Started page, select **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.

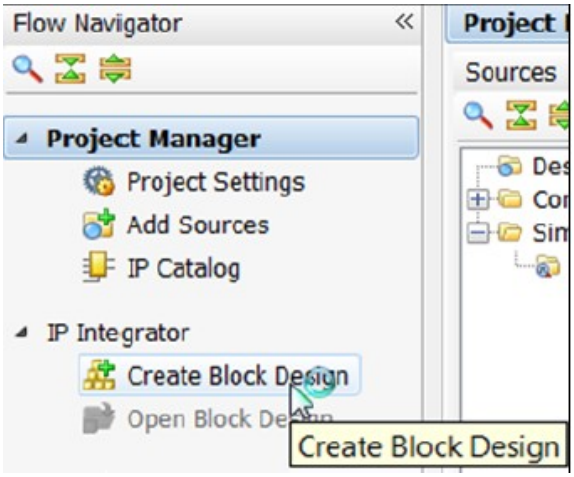
5. In the **Add Sources** dialog box, ensure that the Target language is set to Verilog.

6. Choose xc7a100tcsg324-1 in the **Default Part** dialog box.

7. Click **Finish** to finish creating the project.

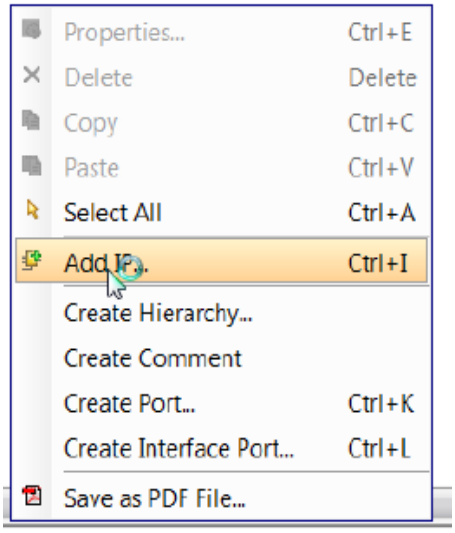
2. Create an IP Integrator Design

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

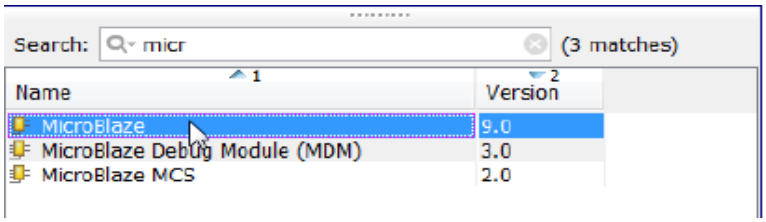


2. Specify the design name and click OK.

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.



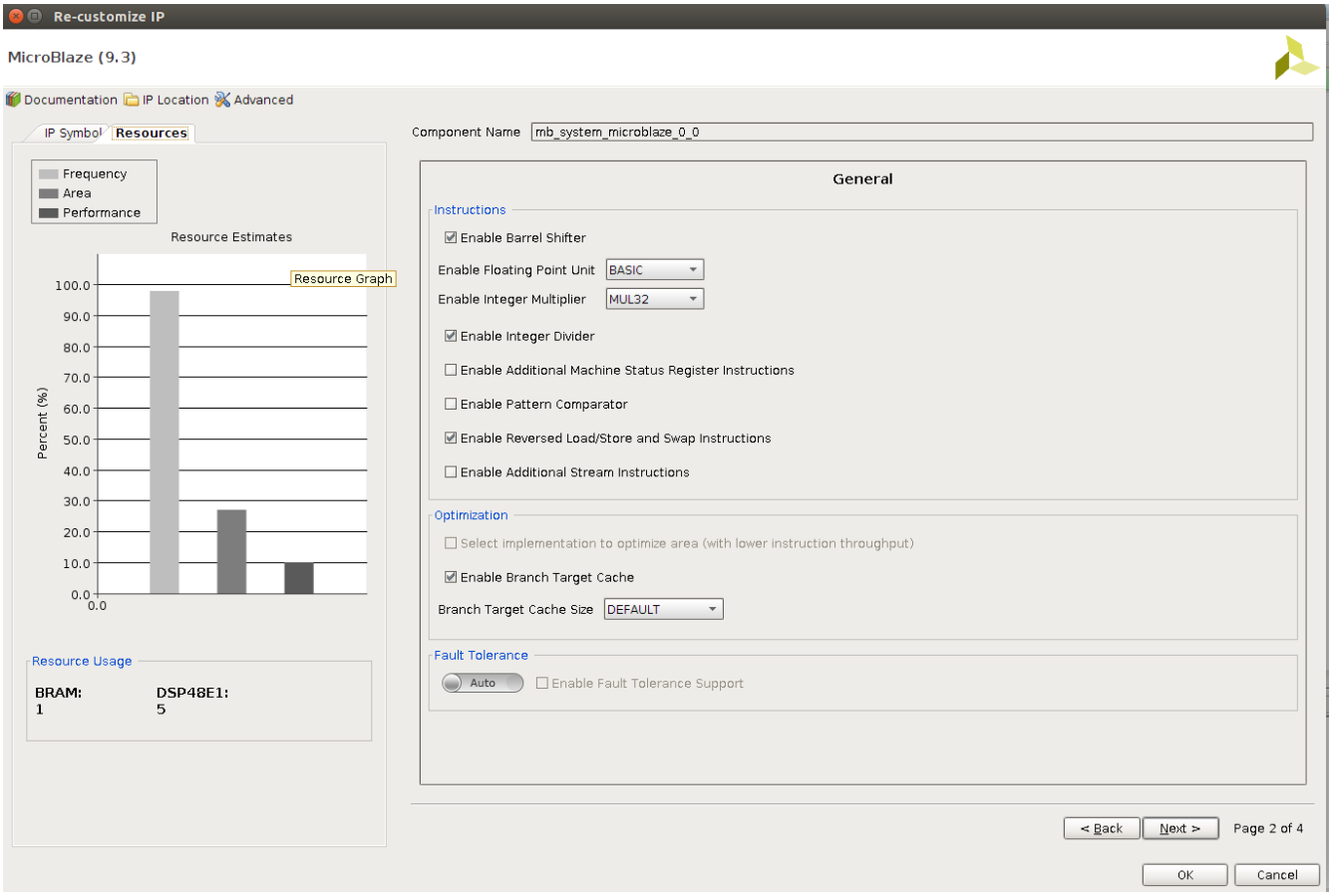
Customize the MicroBlaze Processor

1. In the IP canvas, double-click the MicroBlaze Processor diagram. The Re-customize IP dialog box opens up.

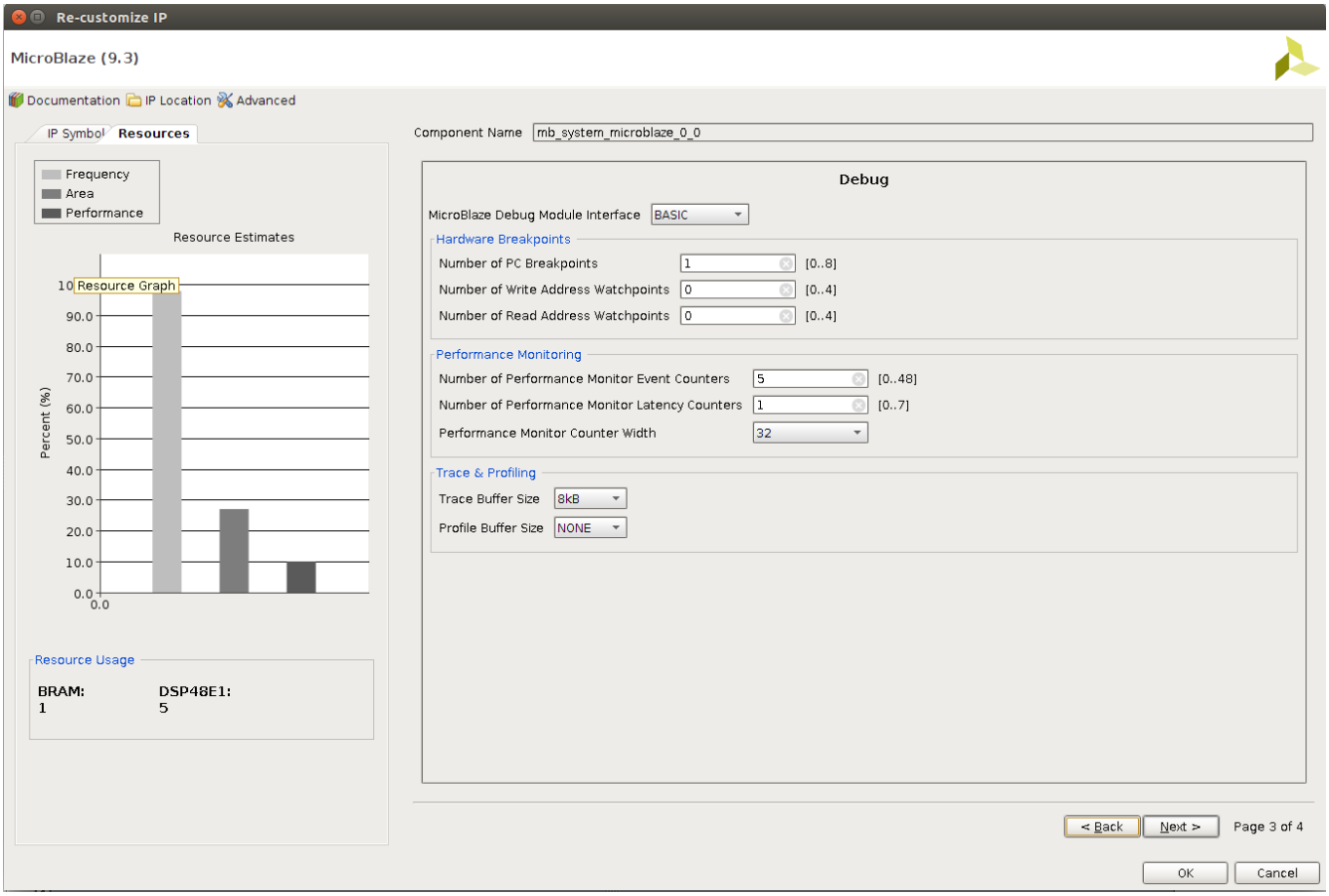
2. Leave everything in page 1 as default, and click **Next**.

3. On page 2 of the Re-customize IP dialog box:

- Check the **Enable Barrel Shifter** option.
- From the pulldown menu, in option **Enable Floating Point Unit** select **BASIC**.
- From the pulldown menu, in option **Enable Integer Multiplier** select **MUL32** (32-bit).
- Check the **Enable Integer Divide** option.
- Check the **Enable Branch Target Cache** option.
- Click **Next**.

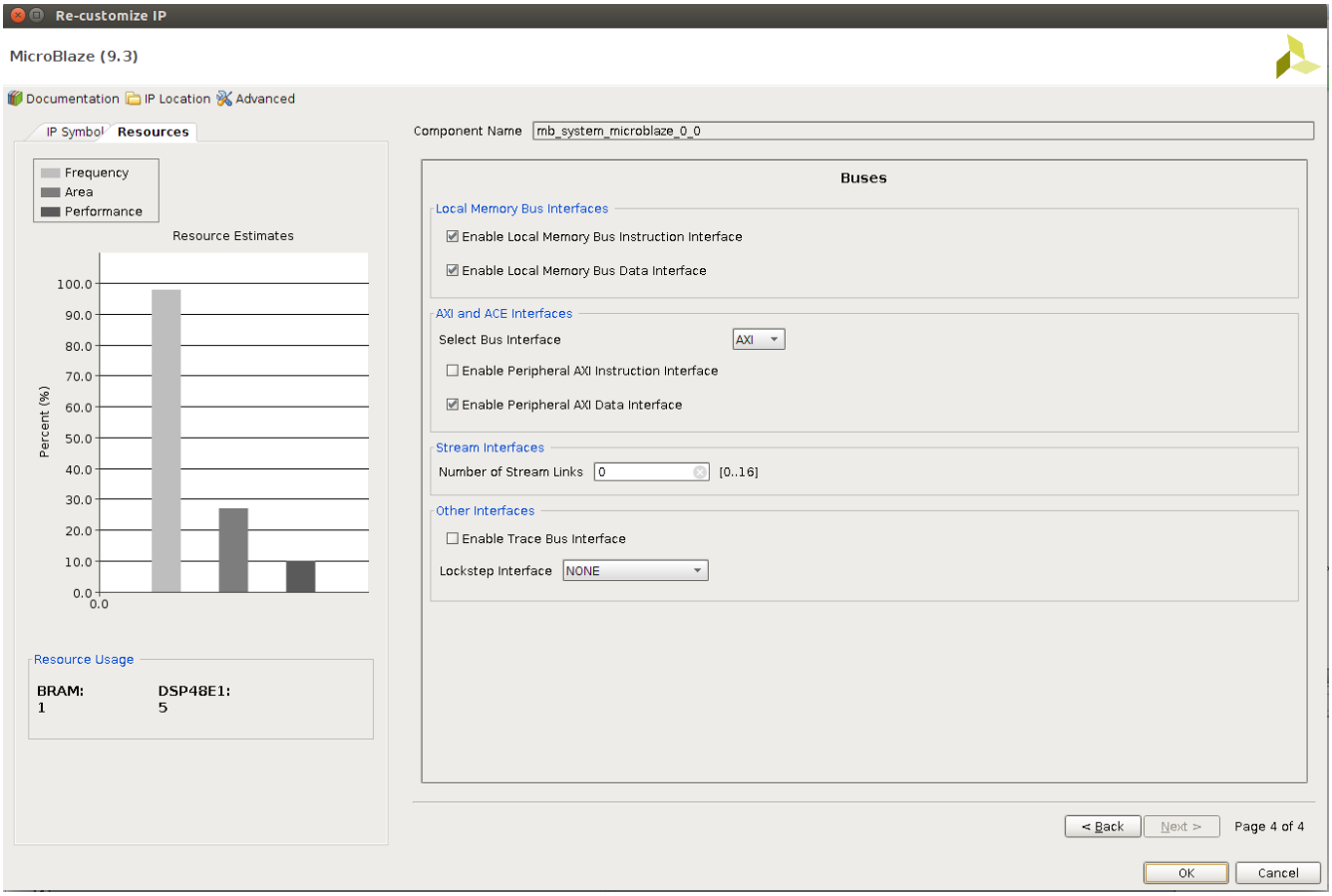


4. On Page 3 of the Re-customize IP dialog box, ensure that the MicroBlaze Debug Module is enabled (i.e. BASIC), and click Next.

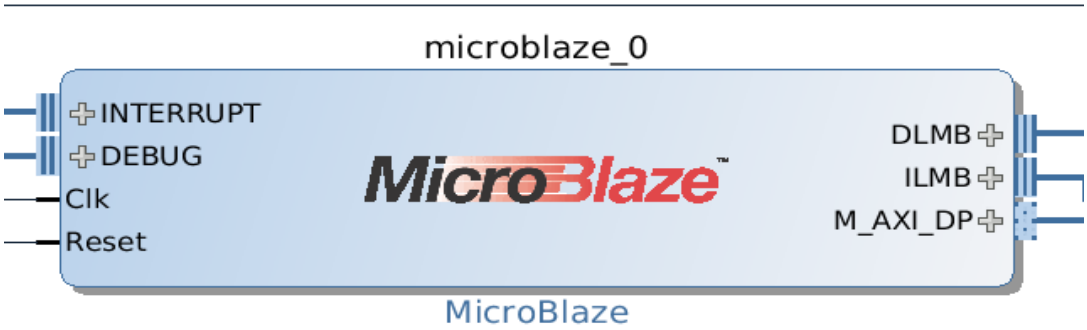


Note: The MicroBlaze Debug Module core enables JTAG-based debugging of one or more MicroBlaze processors.

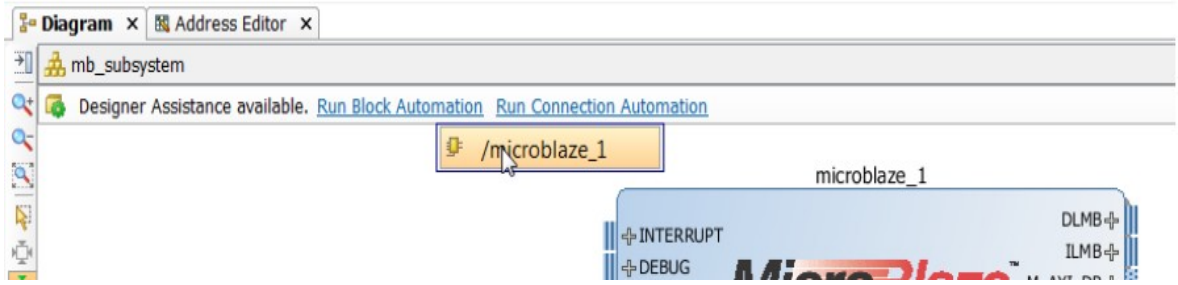
5. On Page 4 of the Re-customize IP dialog box, ensure that the Enable Peripheral AXI Data Interface option is checked, and click OK to re-configure the MicroBlaze processor



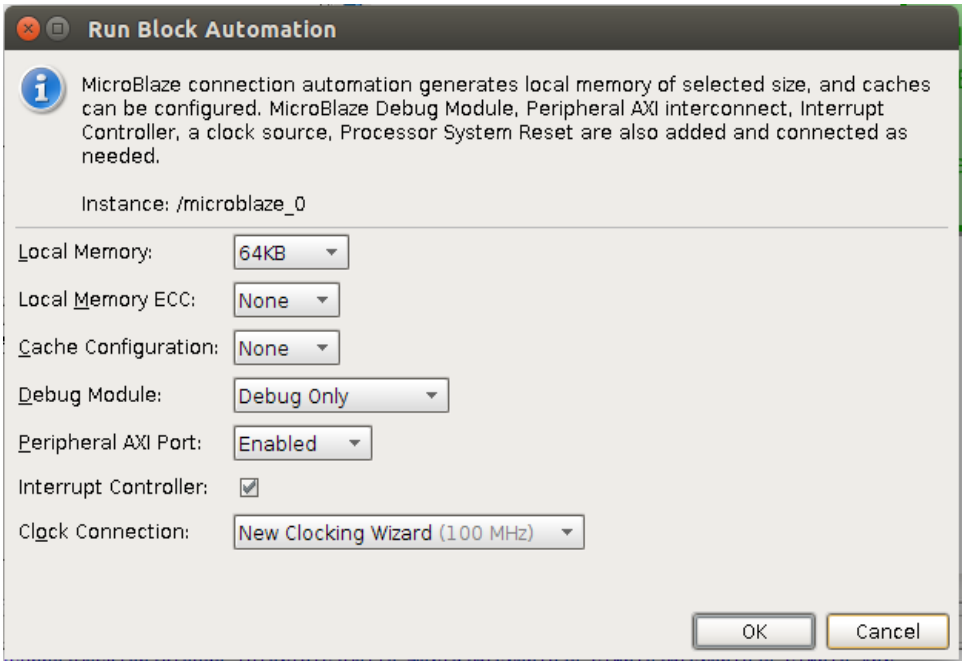
Now, the MicroBlaze processor diagram should look the figure below.



6. Click **Run Block Automation**.



The Run Block Automation dialog box opens up.



7. From the pulldown menu, set Local Memory to 8 **KB**.

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

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

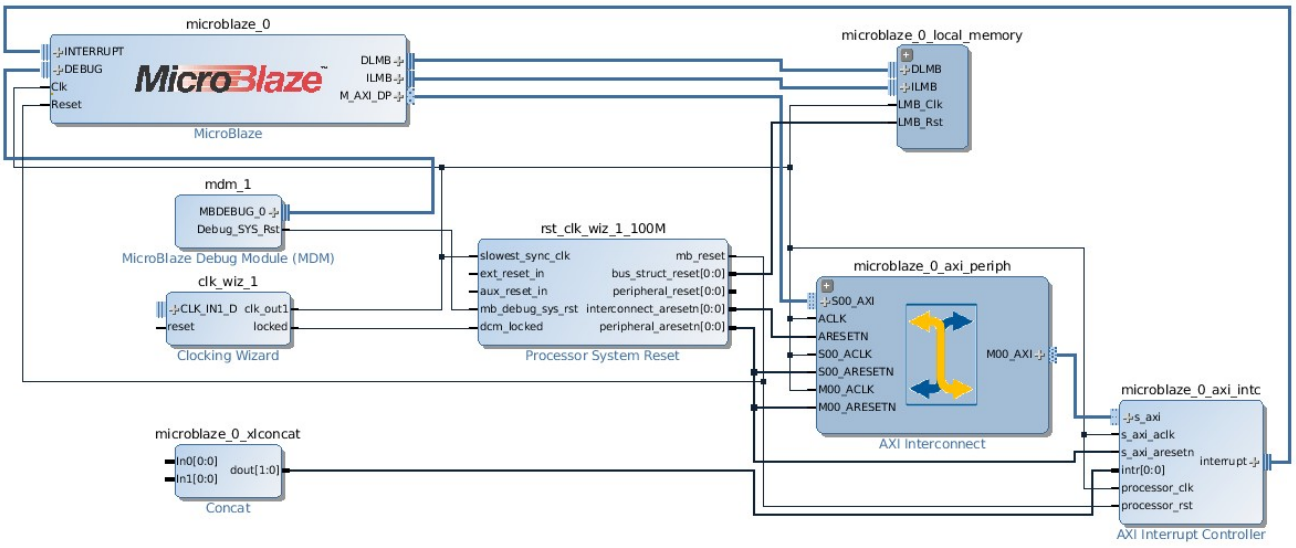
Note: By enabling the AXI Port, an IP called AXI Interconnect will be added during Block Automation. The IP connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices.

10. Check the **Interrupt Controller option**.

11. 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 now look like below.

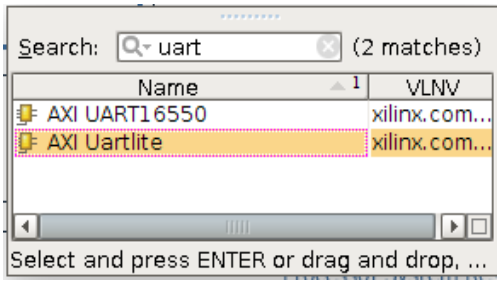


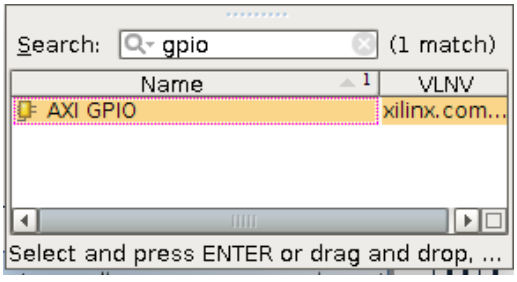
Add peripherals: AXI Uartlite, GPIOs

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.**

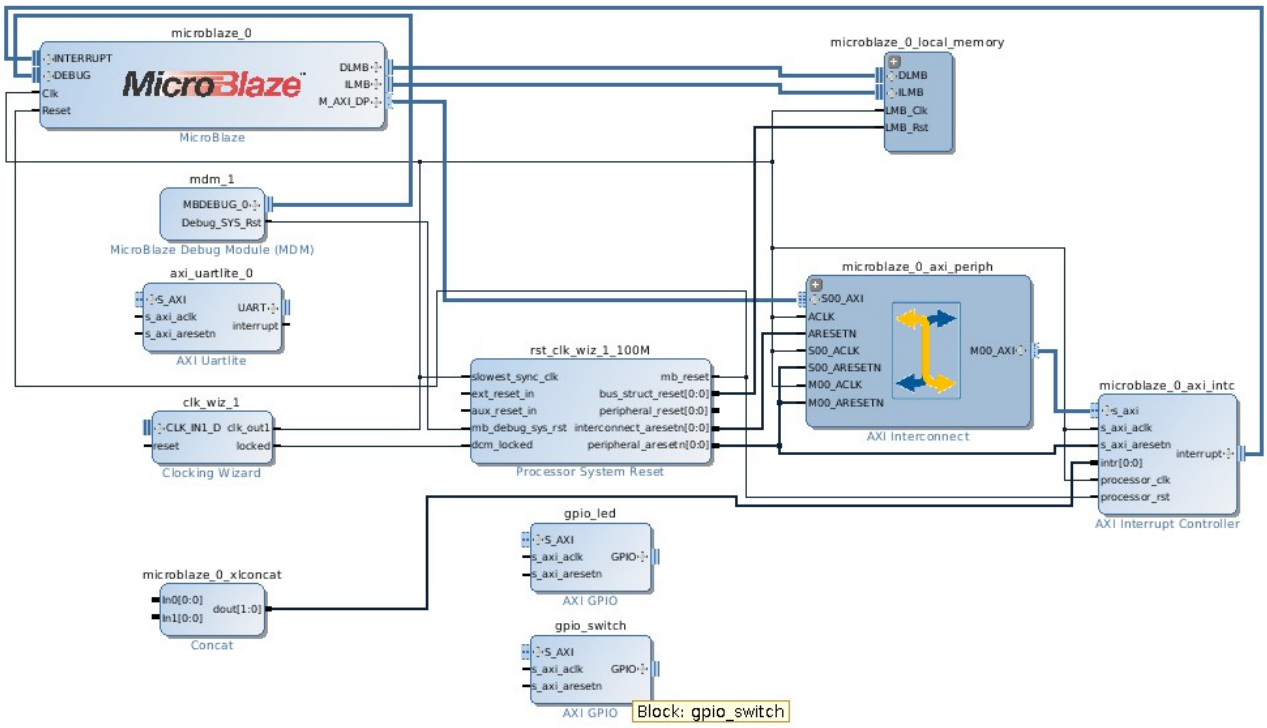
3. Repeat step 2 so that you have two GPIO blocks in your design. The reason why there is two GPIO blocks is that one of them is for switches and another one is for leds so that you can control the leds by switches.





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**.



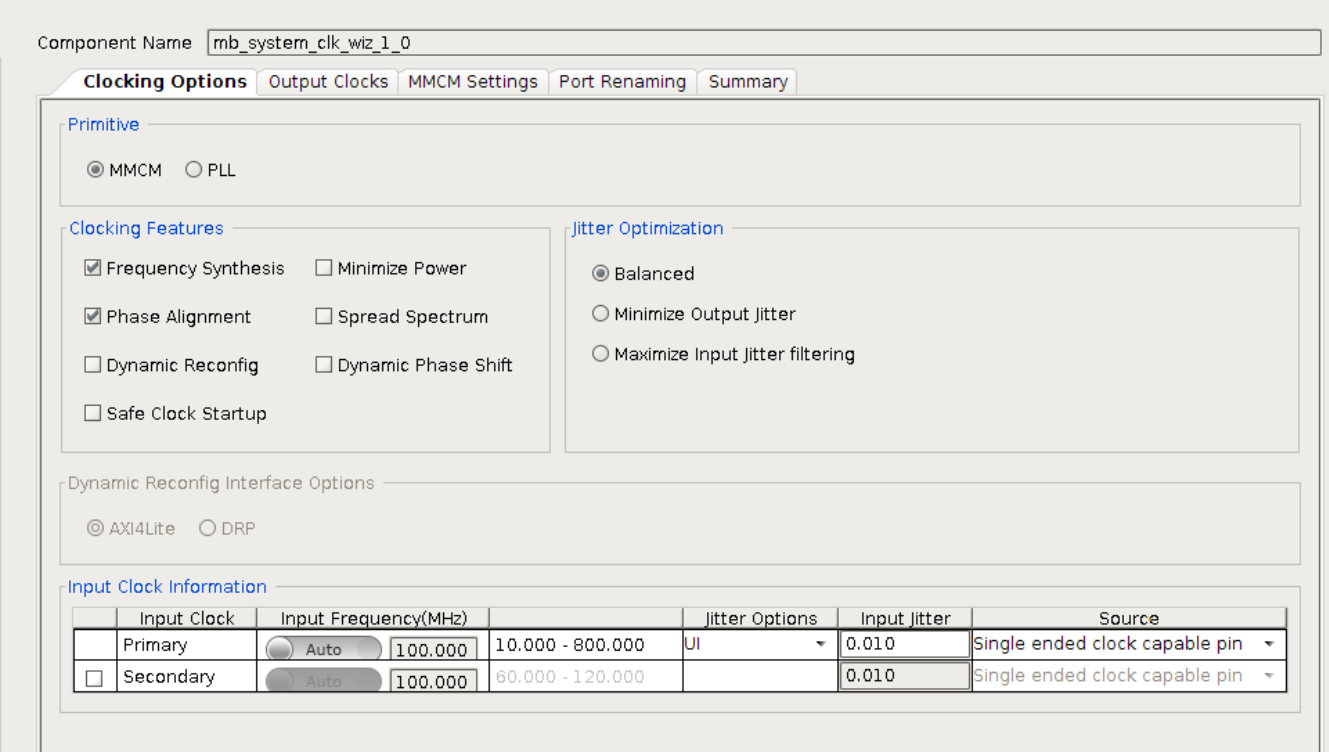
Make Connections

1. Connect the **interrupt** signal of the AXI Uartlite to the **In0** 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 In0 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. Can you suggest some? Double click on the Concat block and change the number of ports to 1 in the Re-customize IP window. Click OK.

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. Click OK.



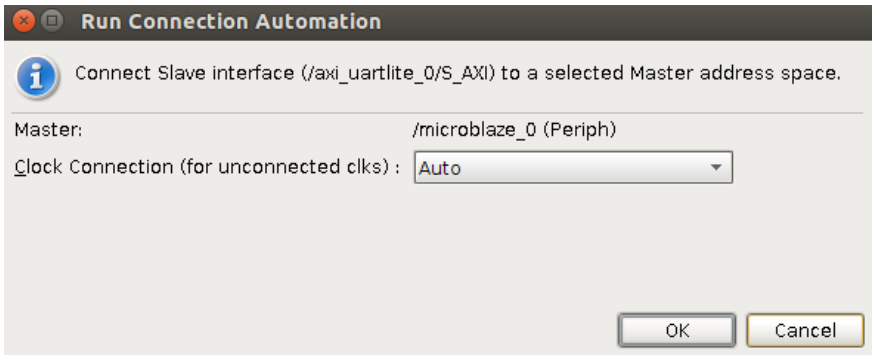
3. Click on **Run Connection Automation** and choose **clk_wiz_1/clk_in1**. Make the input clock external.

4. Click on **Run Connection Automation** and choose **clk_wiz_1/reset**. Make the clock reset active high.

5. Click on **Run Connection Automation** and choose **rst_clk_wiz_1_100M/ext_reset_in**. This makes the external reset external.

6. Click on **Run Connection Automation** and choose **axi_uartlite_0/S_AXI**. Leave the clock connection as Auto. Click OK.

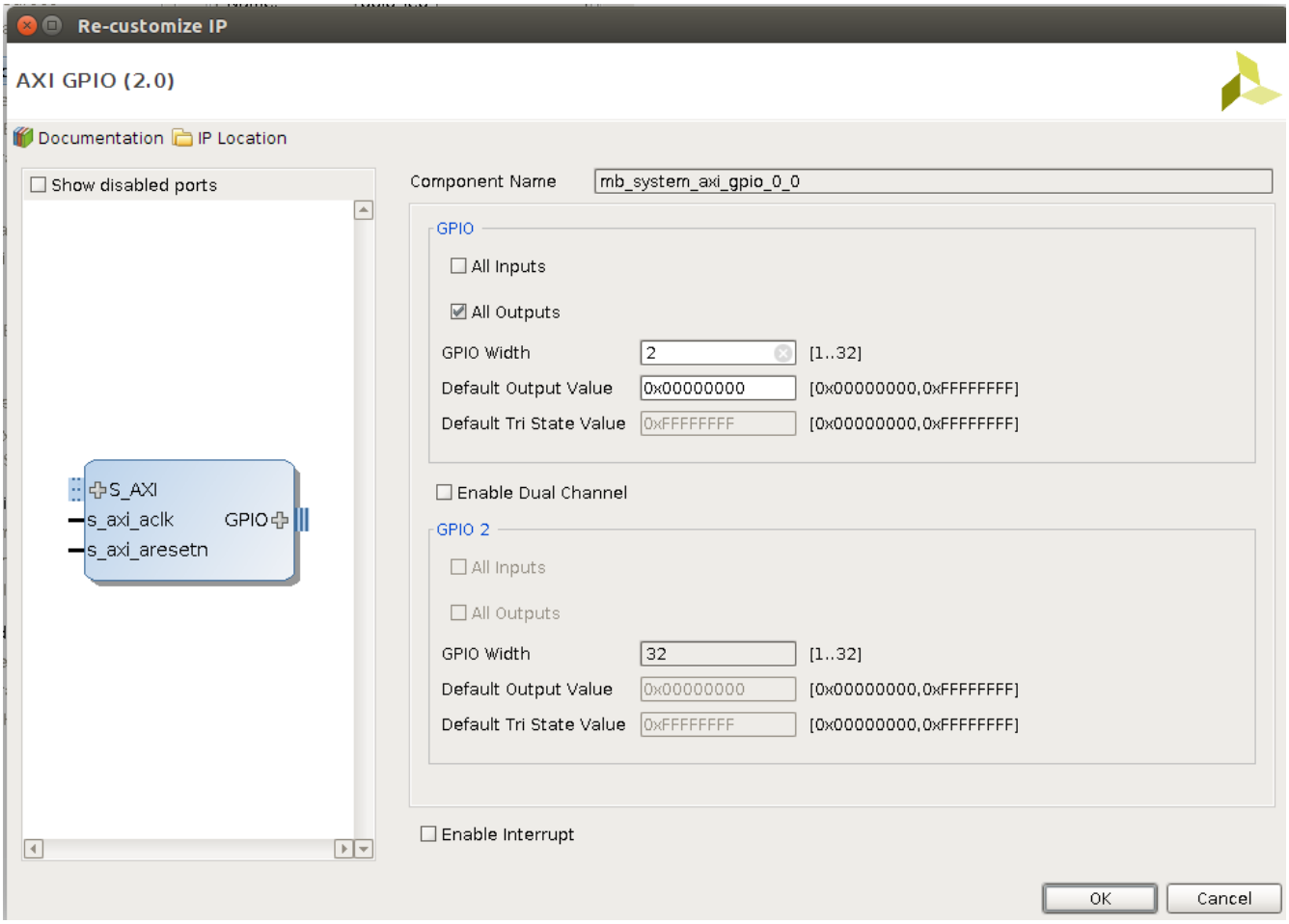
This connects the interface signals on uartlite to the microblaze signals.



7. Click on **Run Connection Automation** and choose **axi_uartlite_0/UART**. Click OK. This makes

the UART interface external.

8. Double click on gpio_led and a configure box pops up. Check **All Outputs** options and this makes the external signals all outputs. 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.



8. Double click on gpio_switch and a configure box pops up. Check **All Inputs** options and this makes the external signals all inputs. Change the **GPIO Width** to 2 as we only need to use 2 switches to control 2 leds.

9. Click on **Run Connection Automation** and choose **gpio_switch/S_AXI**. Leave the clock connection as Auto. Click OK.

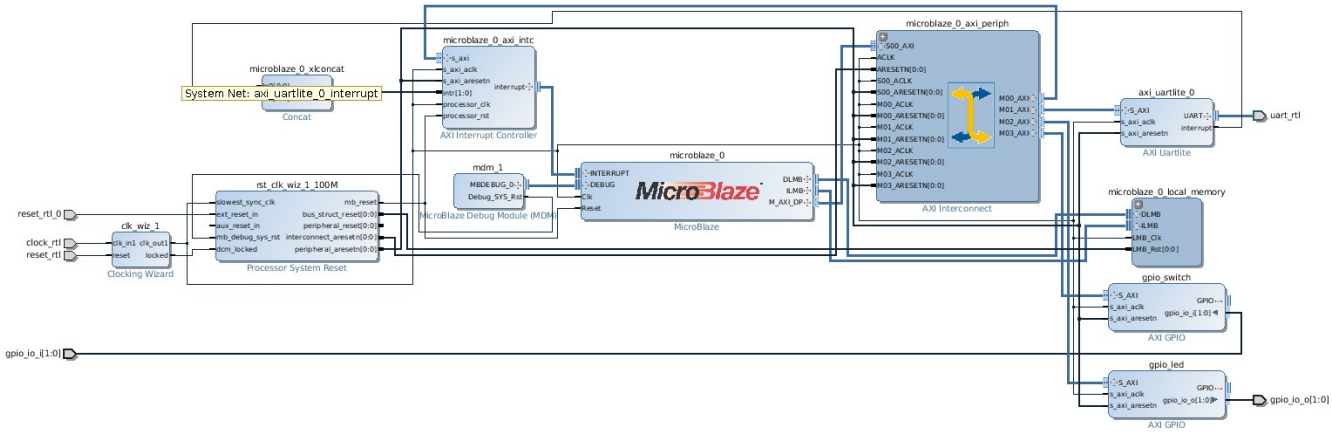
10. Click on **Run Connection Automation** and choose **gpio_led/S_AXI**. Leave the clock connection as Auto. Click OK.

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

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

13. Click on  to validate your design. Or simply just press F6. If you followed every step above, it should say validation successful shown below.

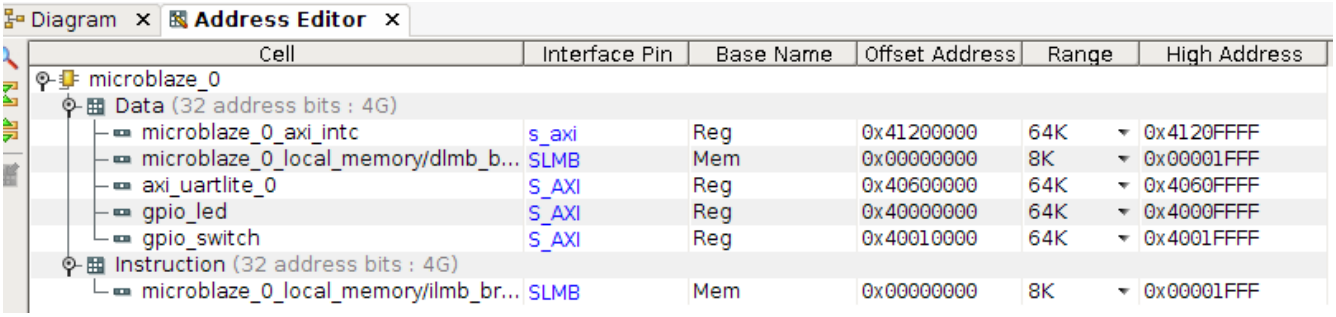
The entire design should look like below except that the blocks may be in different positions.



Memory-Mapping

1. Click on **Address Editor** to map the address. Expand all lines.

2. Map your memory as follows. Make sure that your local memory starts at 0x00000000 and memory size is 8KB. Otherwise, it will complain after you transfer your design to the SDK.



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.

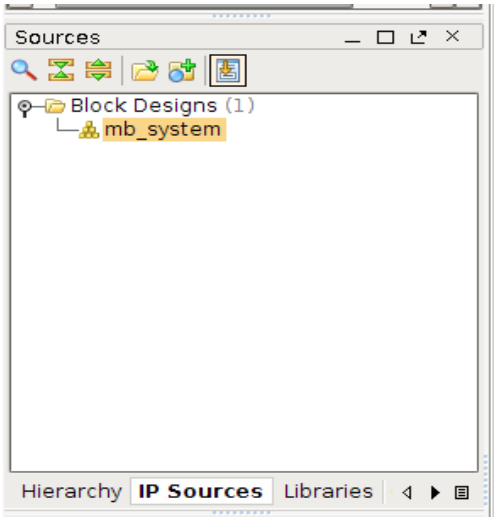
```
## Clock signal

set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clock_rtl }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz

create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock_rtl}];

##Switches

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_i[0] }];
#IO_L24N_T3_RS0_15 Sch=sw[0]

set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_i[1] }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]

set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports { reset_rtl_0 }];
#IO_L7N_T1_D10_14 Sch=sw[5]

set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports { reset_rtl }];
#IO_L17N_T2_A13_D29_14 Sch=sw[6]

## LEDs

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_o[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]

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

```
#IO_L24P_T3_RS1_15 Sch=led[1]

##USB-RS232 Interface

set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 } [get_ports { uart_rtl_rxd }];
#IO_L7P_T1_AD6P_35 Sch=uart_txd_in

set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 } [get_ports { uart_rtl_txd }];
#IO_L11N_T1_SRCC_35 Sch=uart_rxd_out
```

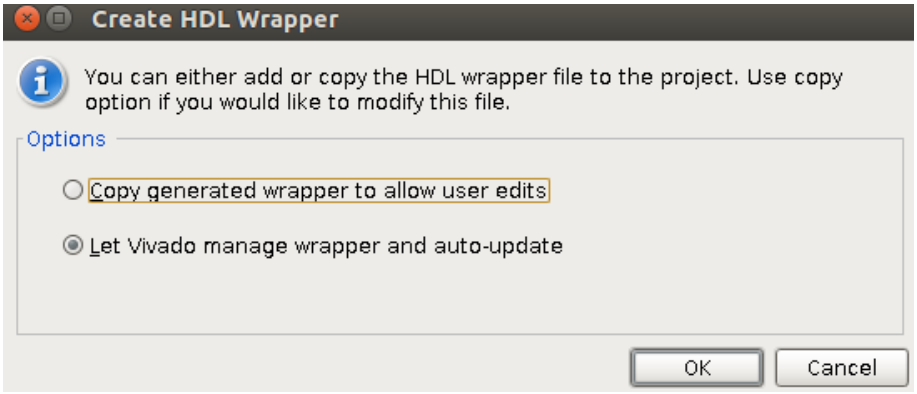
Note that the reset signals are switch 5 and 6 on the board.

Wrap 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.

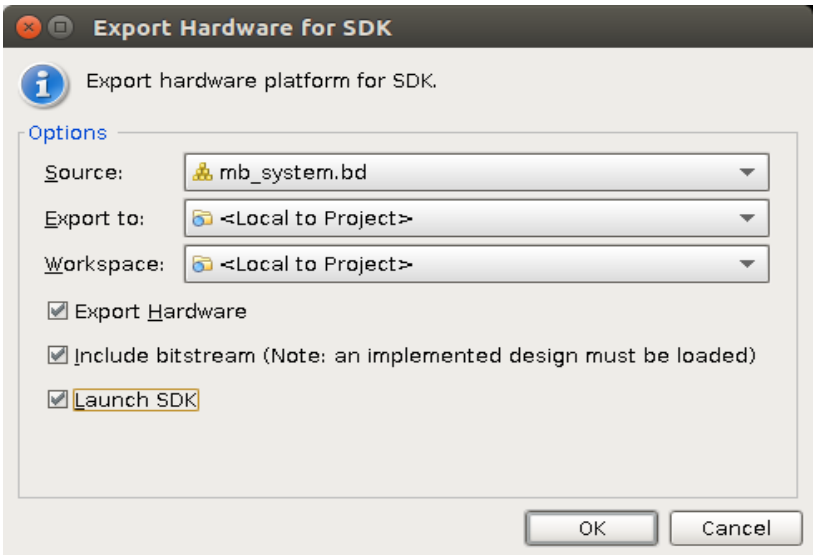
4. Click on **Run Synthesis.**

5. Click on **Run Implementation.**

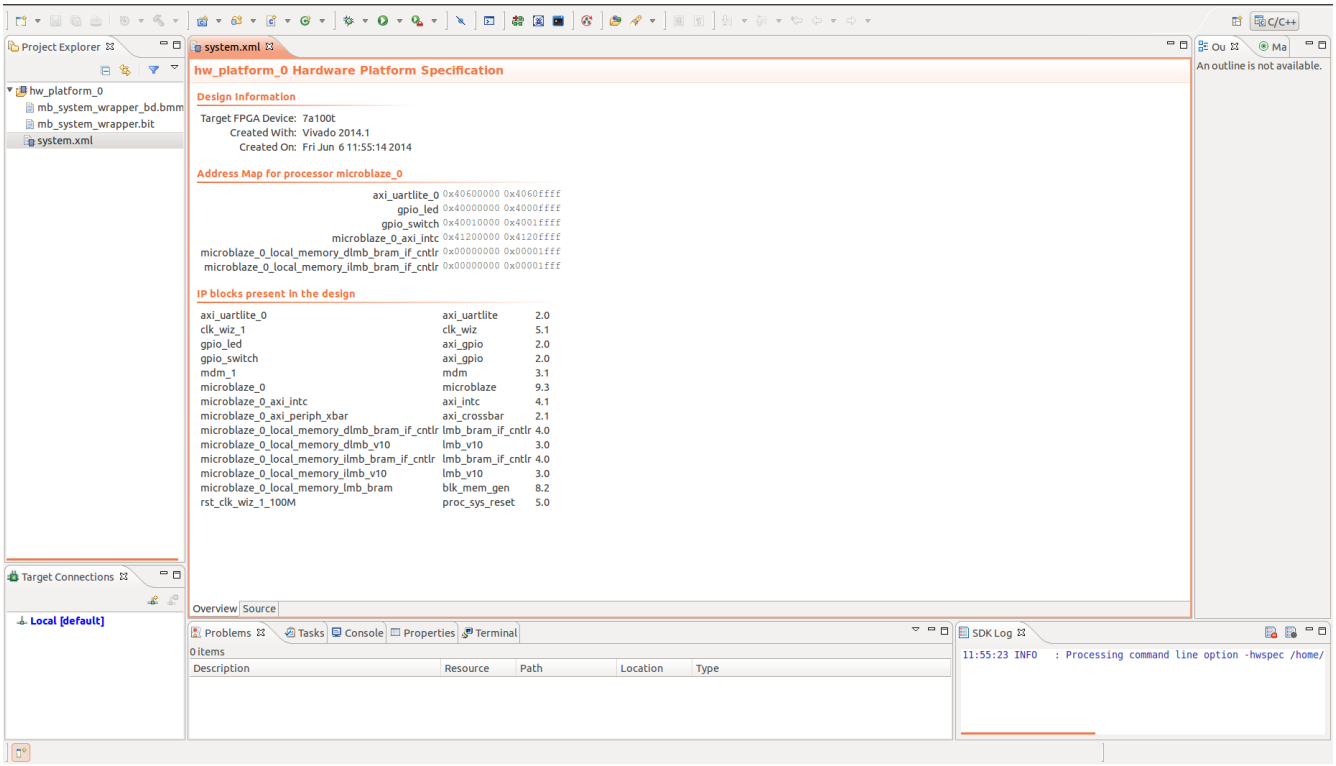
6. Click on **Generate Bitstream.**

3. Export to SDK

After generating the bitstream, click on **File** on the top and choose **Export**, then choose **Export Hardware for SDK.** 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. Also make sure you check **Launch SDK** so that after export, the SDK program will launch automatically.



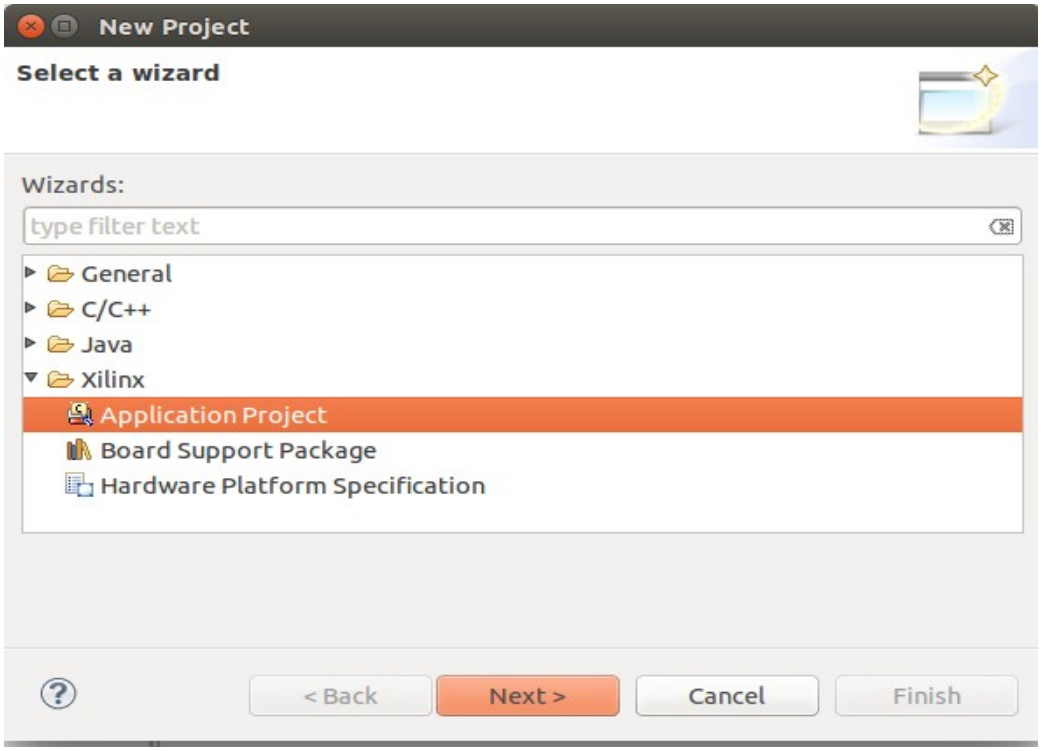
After the SDK first launches, double click system.xml and you should see something like below.



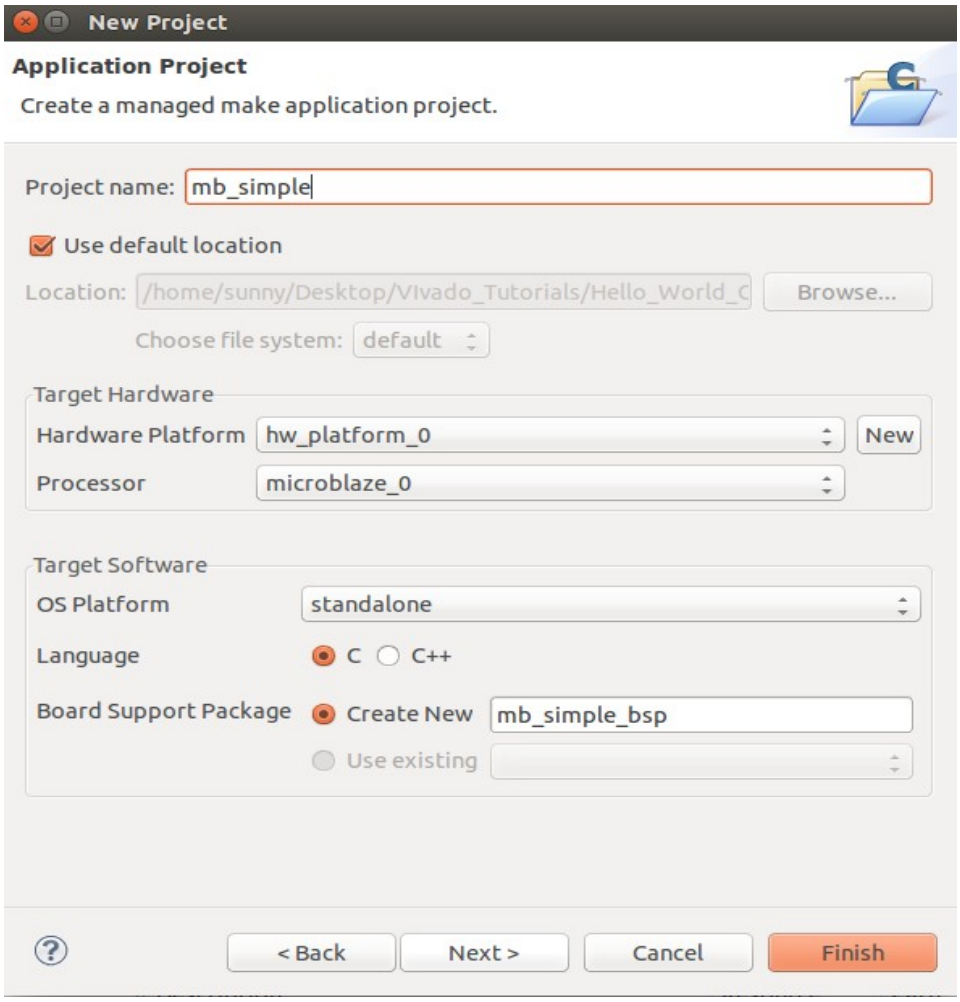
Creating a Hello World program

1. Right click on **hw_platform_0** in Project Explorer -> **New** -> **Project**

2. **Xilinx->Application Project->Next**



3. Enter the **Project name** and choose the **OS Platform** to be **standalone**. Click Next.

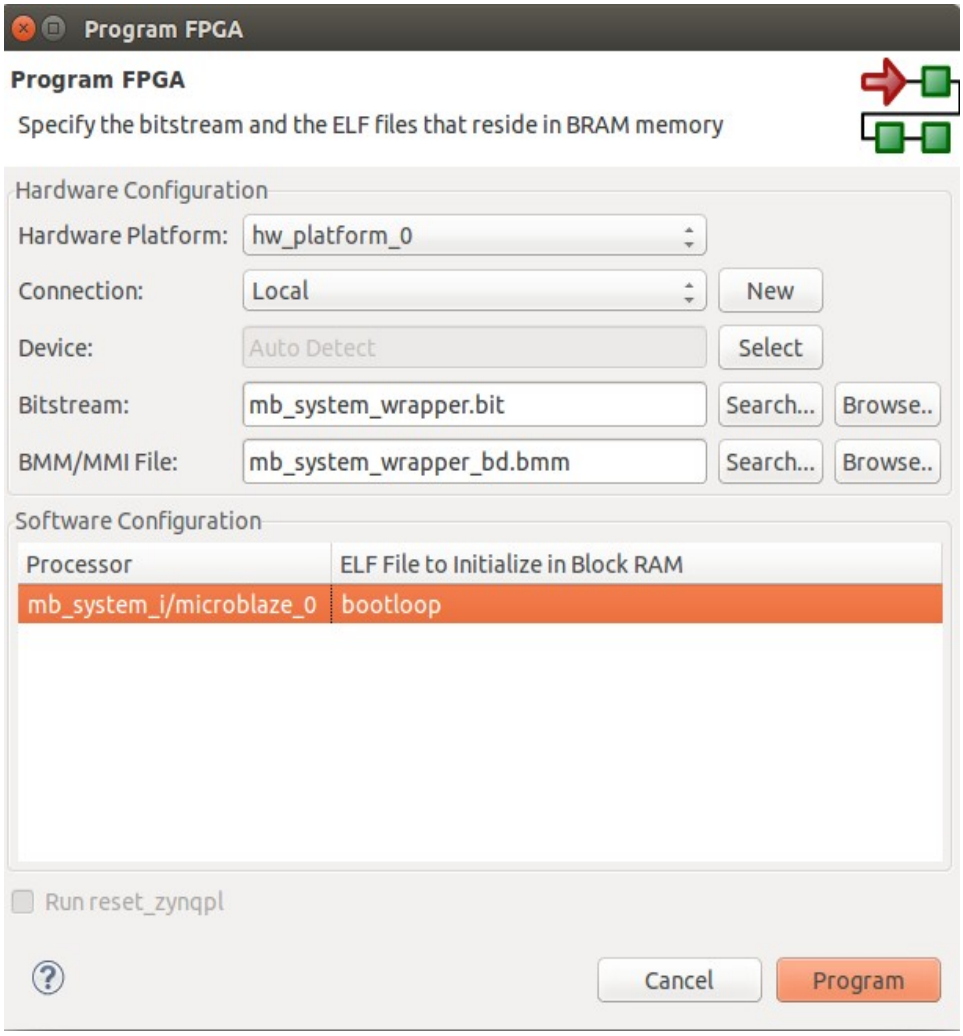


4. Choose **Hello World** as your template. Click Finish.

5. Connect your board to the computer and turn it on.

6. Click **Xilinx Tools** at the top -> **Program FPGA** to program your board.

Note: Make sure you switch on sw5 when you program the FPGA. sw5 is the reset in this design and SDK will give you an error saying the design is under reset if you do not turn it on.



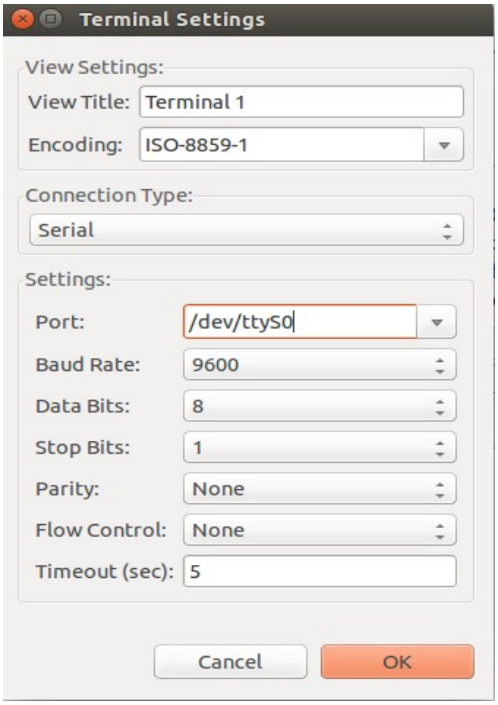
Connect to UART

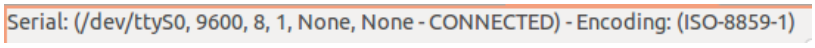
1. Click on **Terminal**

2. Click on  in the terminal window to choose serial connection.

3. Change **Connection Type** to **Serial** and leave others as default. Click OK.

Now the program is connected to the board through serial connection.



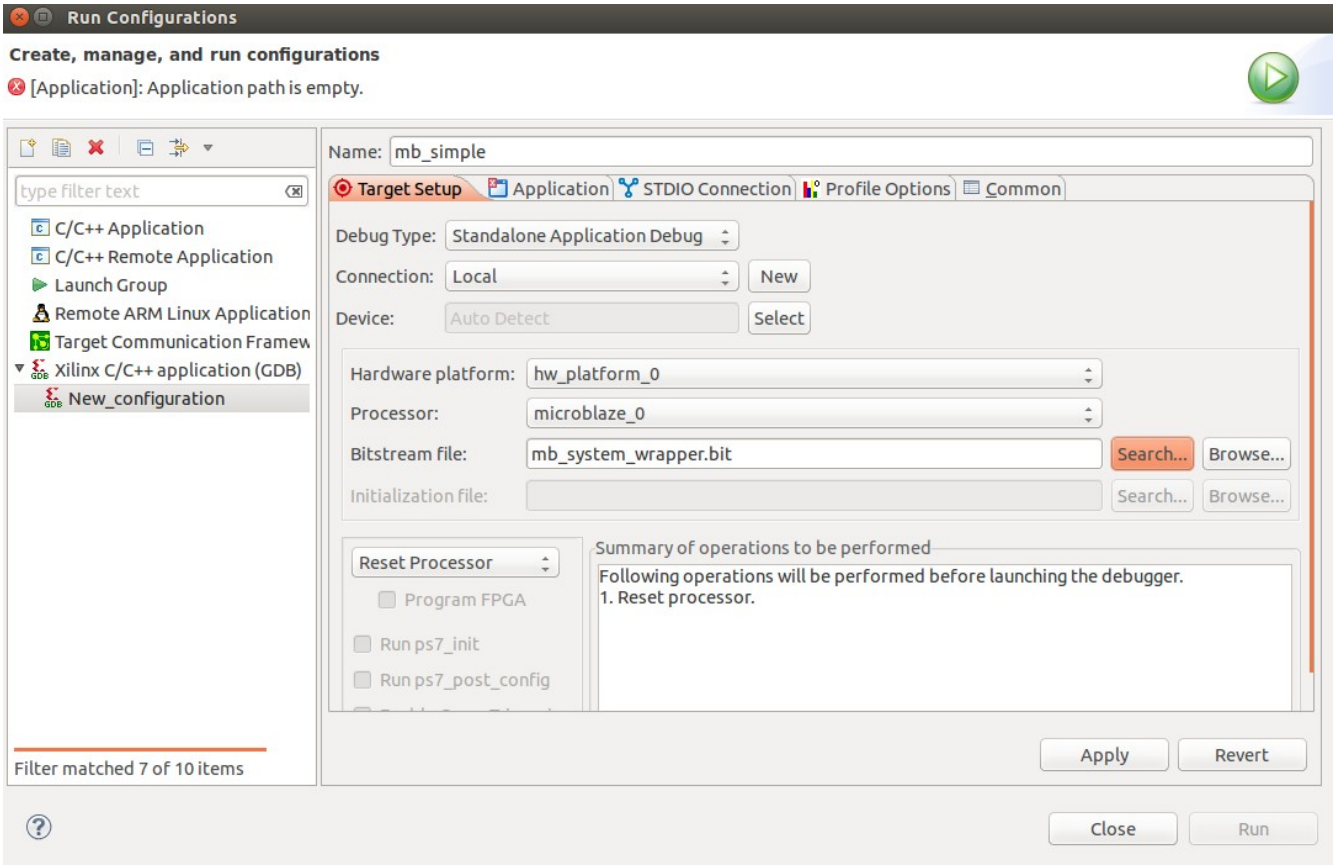


Run a C Program

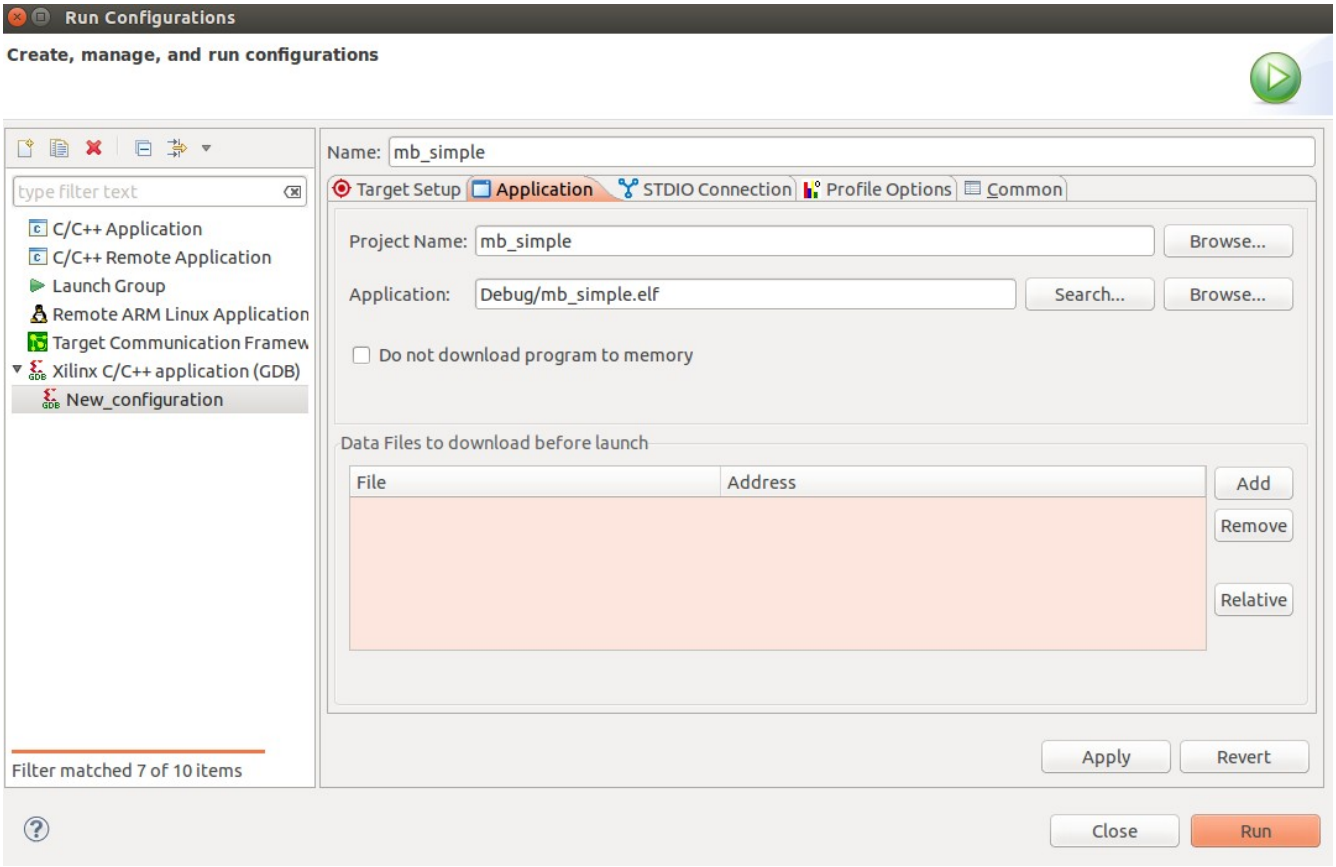
1. Click on **Run** on the top -> **Run Configurations** to create a new run session.

2. Choose **Xilinx C/C++ application(GDB)** and click new on the top left corner to create a new launch.

3. In **Target Setup**, search for your bitstream file. When you search, there will be two bitstream files. One is wrapper.bit and other one is download.bit. Choose **wrapper.bit** and click Ok.

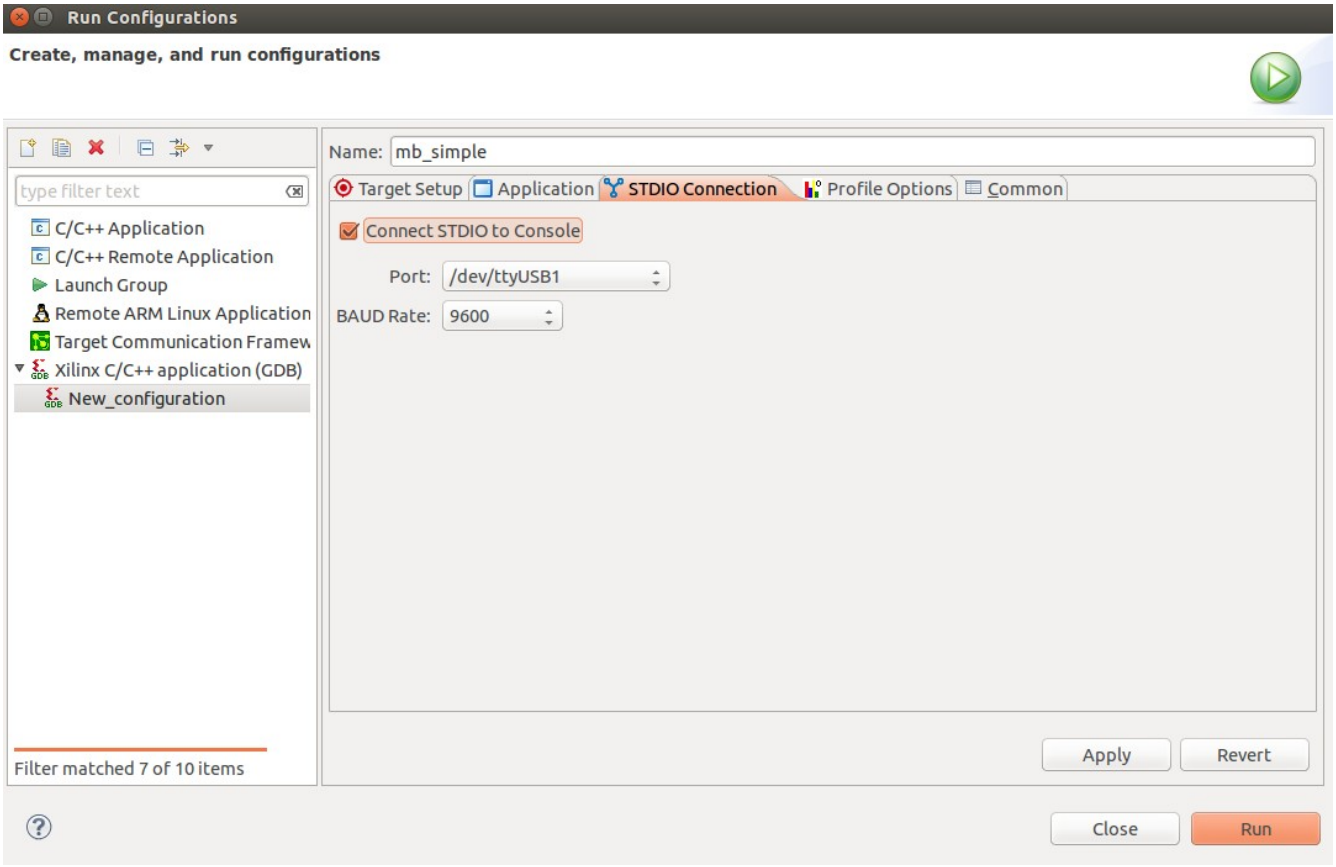


4. In **Application**, browse the Project Name and choose your project. In this example, it is mb_simple. You will notice that the Application file is automatically selected under Debug folder.



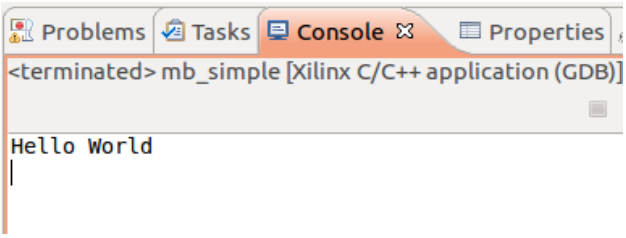
5. In **STDIO Connection**, check **Connect STDIO to Console** if you want to see your program output in the Console instead of Terminal windoe. If you do, choose your USB port.

Note: The USB port depends on your computer. It might be /dev/ttyUSB1 or /dev/ttyUSB0. If you don't see any USB port, check your console where you ran your program. If something like 'Could not create LCK...ttyUSB1 because it already exists' shows up, delete the file and then do Run Configuration again.



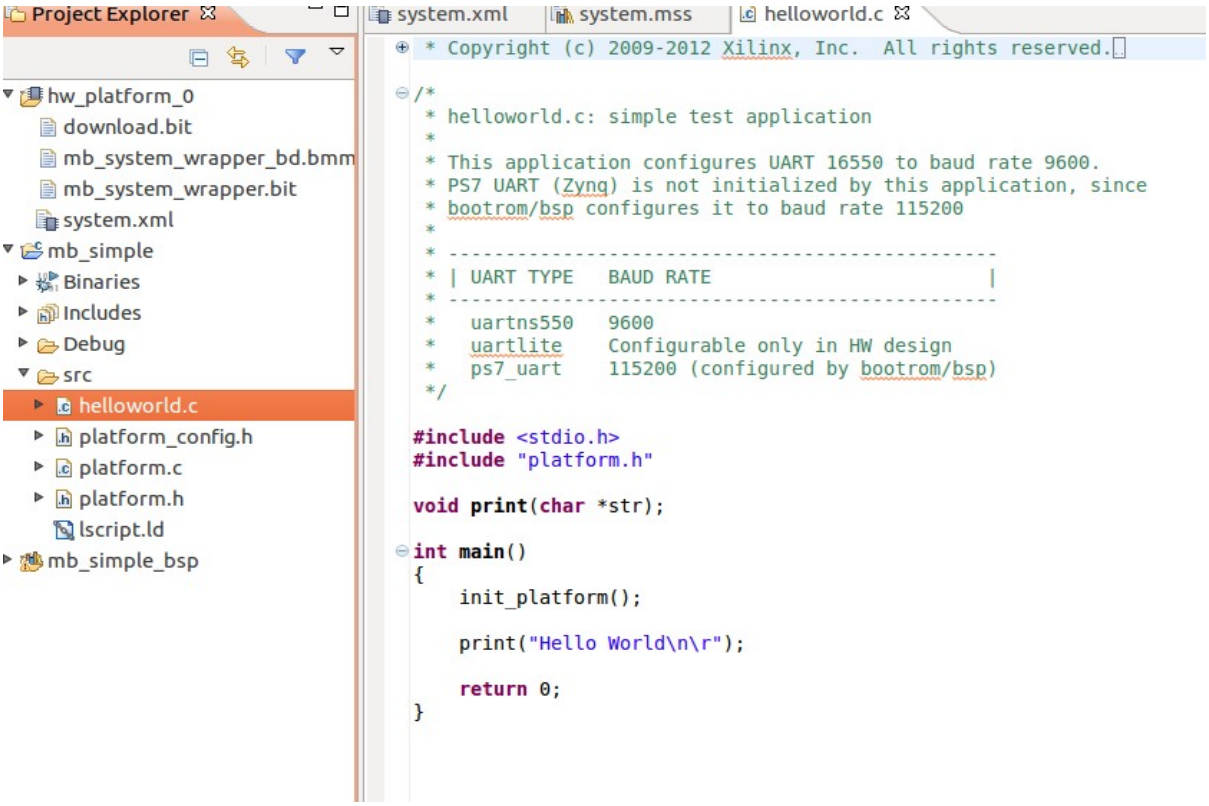
6. Leave everything else as default and click Run.

You will see Hello World is printed in the Console.



Controlling LEDs using Switches

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


Project Explorer
hw_platform_0
download.bit
mb_system_wrapper_bd.bmm
mb_system_wrapper.bit
system.xml
mb_simple
Binaries
Includes
Debug
src
helloworld.c
platform_config.h
platform.c
platform.h
lscript.ld
mb_simple_bsp
system.xml
system.mss
helloworld.c
* Copyright (c) 2009-2012 Xilinx, Inc. All rights reserved.
/*
* helloworld.c: simple test application
*
* This application configures UART 16550 to baud rate 9600.
* PS7 UART (Zynq) is not initialized by this application, since
* bootrom/bsp configures it to baud rate 115200
*
* --
* | UART TYPE BAUD RATE |
* --
* uartns550 9600
* uartlite Configurable only in HW design
* ps7_uart 115200 (configured by bootrom/bsp)
*/
#include <stdio.h>
#include "platform.h"
void print(char *str);
int main()
{
init_platform();
print("Hello World\n\r");
return 0;
}

2. Copy the following code to helloworld.c

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

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

void print(char *str);

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

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

In this example, the base address of the leds is 0x40000000 and the base address of the switches is 0x40010000. The infinite while loop keeps checking the value of the switches and assign them to the leds. This way you can control the leds by switches.

3. Click **Run**.

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

Note: Make sure you have gcc compiler installed in your computer otherwise the program will not compile after you change the code.