ECE532S Digital Systems Design

Tutorial 2 - Creating a Simple MicroBlaze Project

Last Updated: July, 2019

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. In short, a *soft processor* is a parameterizable processor core that can be built within the logic of an FPGA.

In this tutorial, we will build a simple MicroBlaze-based soft-processor system on the Nexsys DDR FPGA. The MicroBlaze system we build will be composed of the following Xilinx IP:

- MicroBlaze Processor
- AXI Timer
- UARTLite
- Debug Module (MDM)

- Processor System Reset
- Interrupt Controller
- Local memory bus (LMB)
- Clock Wizard

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 we develop within the Xilinx Software Development Kit (SDK) in the Vivado Design Suite. The application code developed in the SDK prints "Hello World" on a terminal.

1 Creating a Vivado Project with a Block Design

Invoke the Vivado IDE to create a new project as described in Tutorial 1. From the *Getting Started* page, select **Create New Project**. In the *Project Name* dialog box, type the project name and location; make sure to choose a directory to which you extracted the contents of the provided *zip* file for this tutorial. In the *Project Type* dialog box, select **RTL Project**. In the *Add Sources* dialog box, ensure that the Target language is set to **Verilog**; we don't have any source to add to this project. In the *Add Constrains and Add Existing IP* dialog box make sure to **Add** the *microblaze.xdc* constraints file included in the tutorial *zip* file. Choose **xc7a100tcsg324-1** in the *Default Part* dialog box. Click **Finish** to finish creating the project.

Next, we will create a new *Block Diagram* for our MicroBlaze system. Select **Create Block Design** under the *IP Integrator* section of the *Flow Navigator* pane. Specify any design name and click **OK** to finish the creation of the block diagram. Once the block design has been created,

you should see the interface presented in Figure 1. Block Designs are an alternate way to design a hardware system in the Vivado IDE, presenting a graphical interface where various IPs and Verilog source files can be connected using *click-and-drag* motions rather than typing out the connectivity in the Verilog code. Note, block diagrams are used by Vivado to generate Verilog.
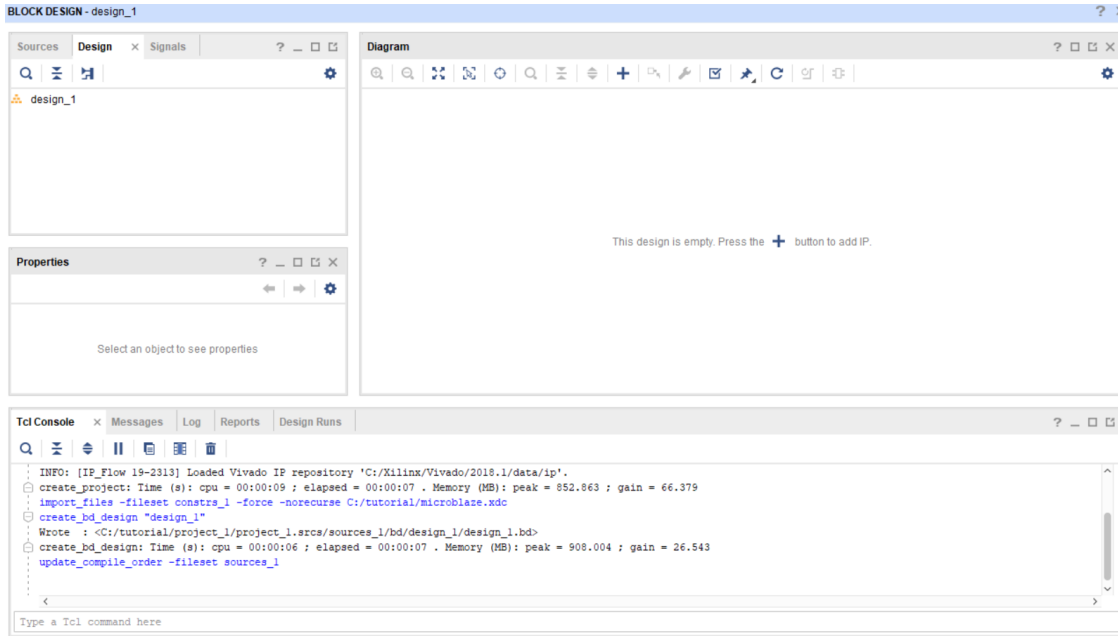

Figure 1: The Block Design editor view

2 Creating a MicroBlaze System

Now that we have a blank block design created, we need to add the IP cores for the functionality we hope to implement. The first IP Core we're going to add is the MicroBlaze soft-processor. Click the **Add IP** (✚) button, either on the toolbar above the block diagram or in the middle of the block design field, to add an IP Core. You can also **Right Click** and select **Add IP** anywhere in the block diagram to add a new IP Core. In the Search field, type **microblaze** to find the MicroBlaze IP, then click **Enter**. Once the microblaze has been added, you should see a new Core within the Block Design, as depicted in Figure 2.

We need to customize the MicroBlaze IP to our specific needs. In general, we can double-click on any IP in order to set its customization parameters. For some IPs, a feature called *Block Automation* is provided which eases the selection of these parameters by only showing the most important parameters. Note, *Block Automation* will also automatically instantiated (create) other IP Cores that are required in order to interface with the IP Core to be customized. We can take advantage of this feature to ease the creation of the full MicroBlaze system. Click the **Run Block Automation** link at the top of the Block Design to run *Block Automation* (Note, this link is visible in Figure 2).

The *Block Automation* wizard you'll see is shown in Figure 3. In the pane on the left side of the *Block Automation* window is the list of IP Cores which are eligible for *Block Automation*; here you can check/uncheck those blocks for which you want to run *Block Automation*. Selecting an IP Core shows the set of customization parameters for that core in the pane on the right side. We should only
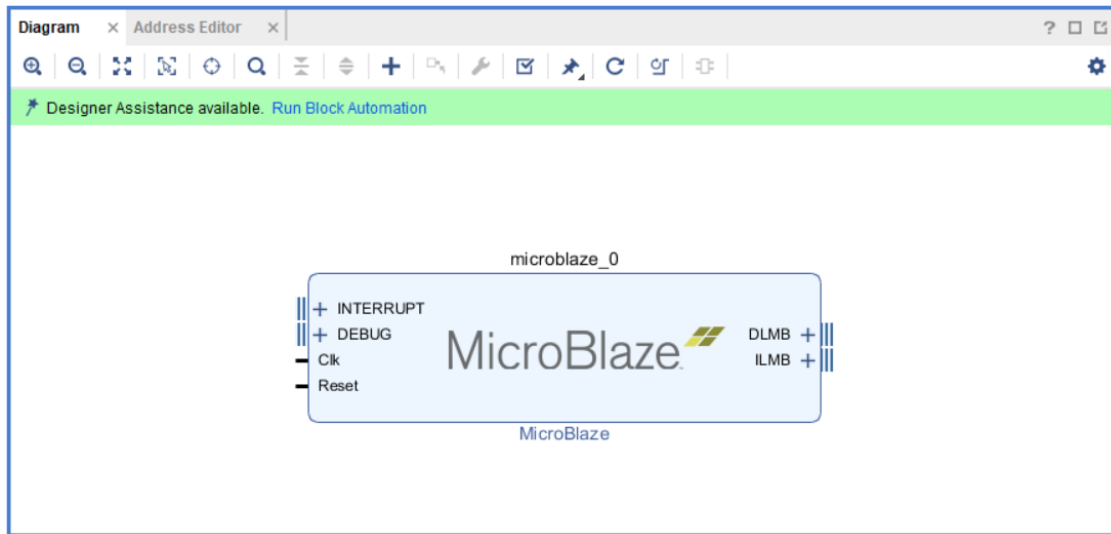
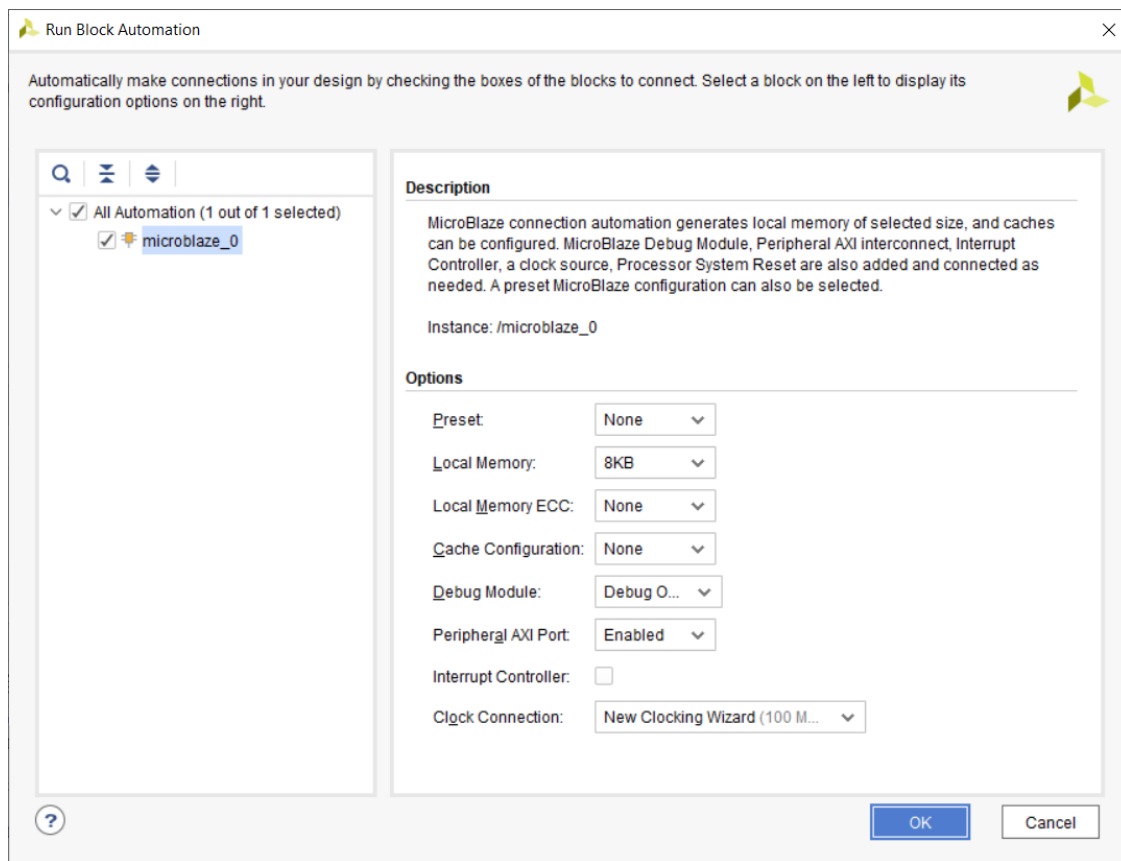Figure 2: The Block Design with the newly added MicroBlaze IP

Figure 3: Block Automation wizard for the MicroBlaze IP

see a single IP Core available for *Block Automation*, the MicroBlaze IP, so select **microblaze_0** (or whatever you're MicroBlaze happens to be named) on the left side. We see a couple of options to customize the MicroBlaze IP Core. From the pulldown menu, set *Local Memory* to **32 KB**.

3

Leave the *Debug Module* option to its default state **Debug Only**. Leave the *Peripheral AXI Port* option as **Enabled**. Check the **Interrupt Controller** option. Select the *Clock Connection* option of **New Clocking Wizard (100 Mhz)**. This will create a clock signal in the block design for the MicroBlaze. Once you've set all these settings, click **OK**.

Once you run Block Automation, you should have a number of IP Cores in your block design (see Figure 4 for reference). These IP Cores have been added as automated connections for the MicroBlaze IP Core. The *MicroBlaze Debug Module (MDM)* was added and automatically connected to the debug port of the MicroBlaze to allow Vivado SDK to communicate to the MicroBlaze (we'll look at that later in the tutorial). The *MicroBlaze Interrupt Controller* was added and automatically connected to the interrupt port of the MicroBlaze. The interrupt controller enables interrupts to be processed by the system; each interrupt you want to respond to would need to be connected to the *Concat* IP Core, which concatenates multiple individual inputs into a bus of wires to connect to the Interrupt Controller. An AXI Interconnect was added to connect the MicroBlaze to the Interrupt Controller; AXI is a memory-mapped interconnect that allows some processor (the MicroBlaze in our case) to communicate to multiple different memory-mapped peripherals (just the Interrupt Controller in our case).
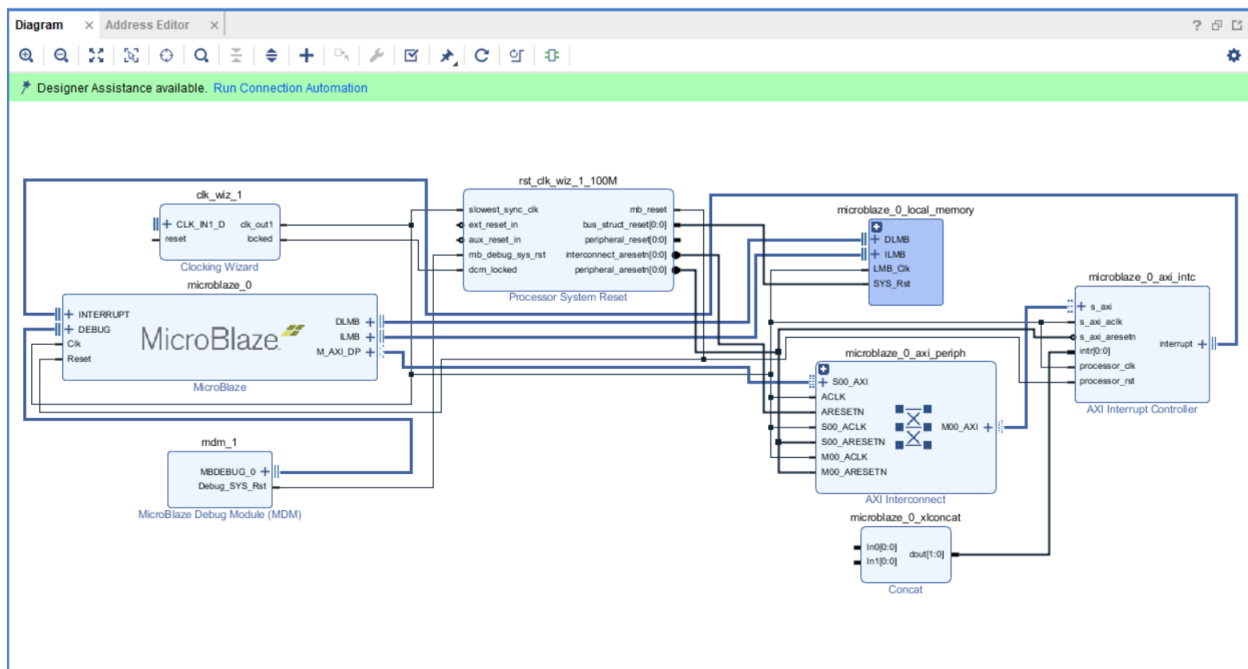

Figure 4: The Block Design after Block Automation

The *Block Automation* also added both program and data memory. To see this memory, note the block that was added called *microblaze_0_local_memory* (or something similar). This block isn't actually an IP Core (note the different colour of the block), but a level of hierarchy instantiated in our block design. **Click** the plus in the corner of this block, or **double-click** the block, to view the contents of this level of hierarchy. There are a couple of IP Cores here that facilitate access to some local memory for the MicroBlaze. The MicroBlaze uses an interface called *Local Memory Bus*, that needs to be decoded by an *LMB BRAM* controller in order to access the contents of a Block RAM (BRAM) component. BRAMs are a form of memory available on Xilinx FPGAs. Note, the Data and Instructions paths are connected to the same block memory, so they share the same memory space; if you wanted the data and instruction paths to be in different memory spaces, you

could add a separate BRAM in this hierarchy.

Finally, *Block Automation* also adds IP Cores to handle the clocks and resets in the system. The *Clock Wizard* IP Core takes a clock from one of the pins of the FPGA and can generate one or more clocks to be used within the FPGA; in our case, a single 100MHz clock is generated for the MicroBlaze and all of the peripherals. The *Processor System Reset* generates separate reset signals for the processor, the peripherals, and the interconnect, according to the AXI standard. We need to make some changes to the clocking configuration, so **Double click** the *Clocking Wizard* and scroll to the bottom of the first tab. Change the *Source* of the *Primary* input clock 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**. See Figure 5 for reference.
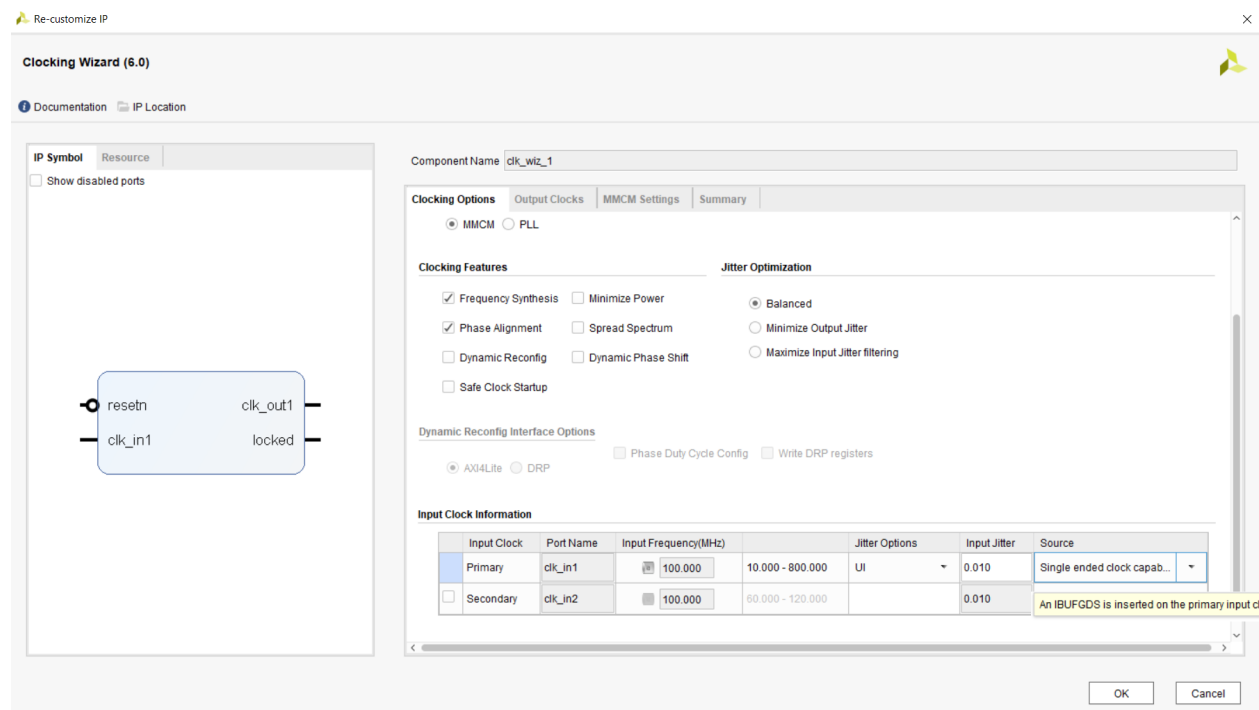

Figure 5: Clocking Wizard block customization interface

The last step we need to do in order to make this a fully functional system is to connect the clock and reset signals to external ports. To do this, we'll use *Connection Automation*, a link to which should have popped up at the top of your block design (🔗 Designer Assistance available. Run Connection Automation). Click **Run Connection Automation** at the top to launch the Connection Automation wizard. *Connection Automation* will allow you to select any unconnected signals and Vivado will try to automatically connect them for you. In our case this includes the *clk* and *resetn* signals of the *Clocking Wizard* and the *ext_reset_n* siganl of the *Processor System Reset* core. **Select** both of the signals for the *Clocking Wizard*, but don't select the *ext_reset_n* signal (we'll connect that manually). Click the **resetn** signal and ensure that *ACTIVE LOW* is selected, and then click **OK**. See Figure 6 for reference.

The *Connection Automation* should have added two input ports to the design, a clock and a reset port. These ports were automatically connected to the clocking wizard. As a final step, **click** on the *ext_reset_n* signal on the *Process System Reset* IP core when the *pen-like* cursor (✏) appears
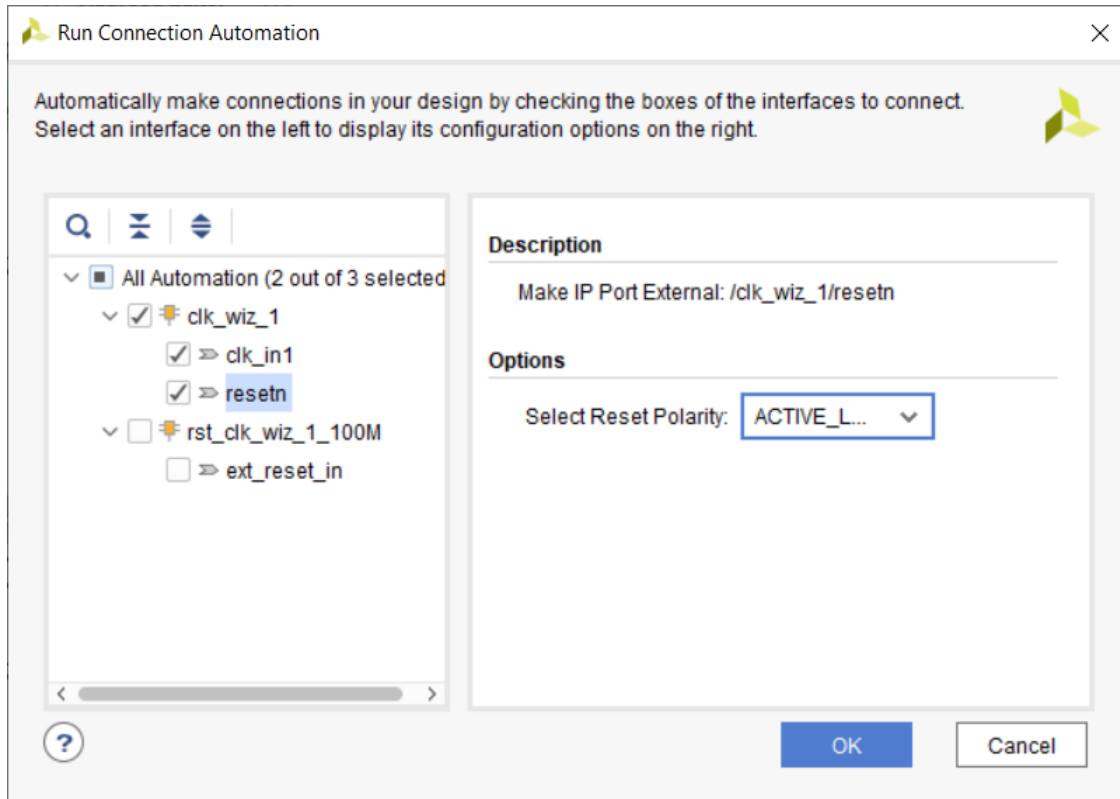
Figure 6: Connection automation for the clocking wizard

and **drag** the cursor to the newly created reset signal; this will connect the *ext_reset_n* signal to the same reset pin we just created. Note, if we had chosen to do *Connection Automation* for the *ext_reset_n* signal, it would have created a separate external pin (we want only a signal external reset). We should now have two ports connected to our design as in Figure 7.

Figure 7: The ports added to the block design

3   Adding Peripherals

We want to add some peripherals to our system to demonstrate that the MicroBlaze can actually interact with other devices and peripherals instantiated within the FPGA or connected to the pins of the FPGA. As a simple example, we'll connect the MicroBlaze system to the switches and LEDs of our board. In addition, we'll add a connection to the UART interface on our board, in order to allow the MicroBlaze to print statements to the console in the Vivado SDK later.

To connect to the LEDs and switches, we need to add AXI GPIO IP cores. GPIO stands for General Purpose I/O, and is used to connect directly to the pins of the FPGA. The AXI GPIO

allows this connection to the pins to be accessible from an AXI interface. **Right click** anywhere in the block diagram, select **Add IP** and search for and select the AXI GPIO. Repeat this step 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. Right click one of the GPIO block and click **Block Properties**. Change the name to **gpio_led**. Right click the other GPIO block and click **Block Properties**. Change the name to **gpio_switch**. See Figure 8 for reference.
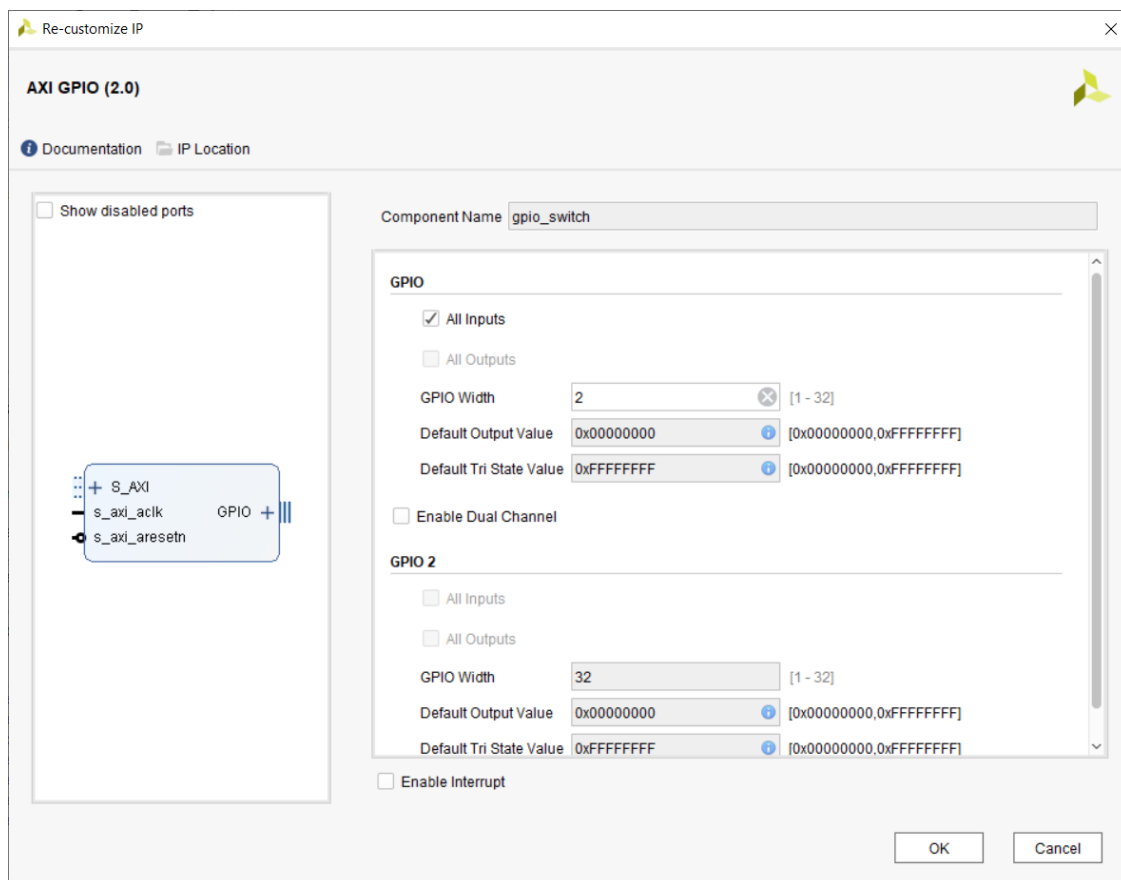

Figure 8: The AXI GPIO block properties window

Figure 9: The AXI GPIO configuration window

Now we need to configure and connect our GPIO cores. Double click on **gpio_led** to bring up the configuration window. Check the **All Outputs** checkbox and change the GPIO Width to

2 as we only need to use 2 LEDs for this tutorial. 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. Note, the switch GPIO should be set to *All Inputs*. See Figure 9 for reference. 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**. This last step will allow the GPIO ports to connect to pins on the FPGA. Once the ports have been made external, we should have a connectivity as per Figure 10.
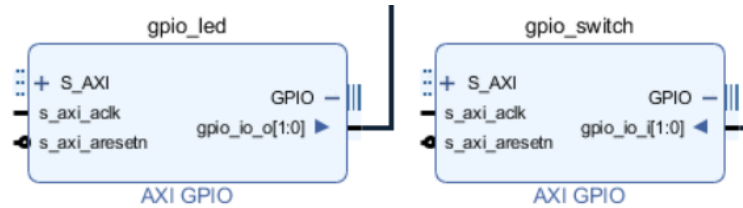

Figure 10: The AXI GPIO connectivity

Next, we'll add the AXI Uartlite peripheral to our project. **Right click** anywhere in the block diagram, select **Add IP** and search for and select the AXI Uartlite. This core has an interrupt functionality that needs to be connected to our MicroBlaze *Interrupt Controller* in order to function properly. Connect the **interrupt** signal of the AXI Uartlite to the **In0** signal of *Concat*, that connects to the interrupt input of the interrupt controller. *Concat* is used to concatenate individual signals into a bus signal. 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**. We should have a connectivity as per Figure 11.
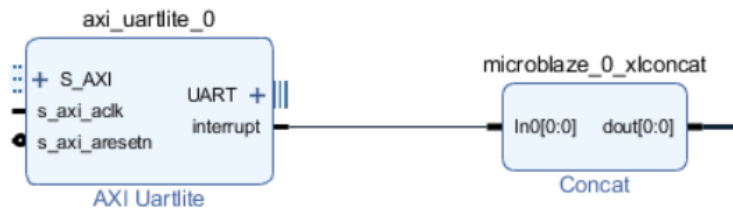

Figure 11: The UART Lite connectivity

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 connect all of the clock and reset signals to the appropriate sources. You should now have a complete system with a MicroBlaze processor, three AXI peripherals, interrupt controller, local memory, debug module and clock and reset generators, as per Figure 12. The *AXI Interconnect* was expanded to add all of the peripherals to the MicroBlaze's memory space. Note, it is often useful to **right click** and select **Regenerate Layout** in order to clean up the layout of your blocks.

An important step in any MicroBlaze system design process is to set the memory-mapping. Note, the *Connection Automation* mapped all of the peripherals for us automatically, but we should
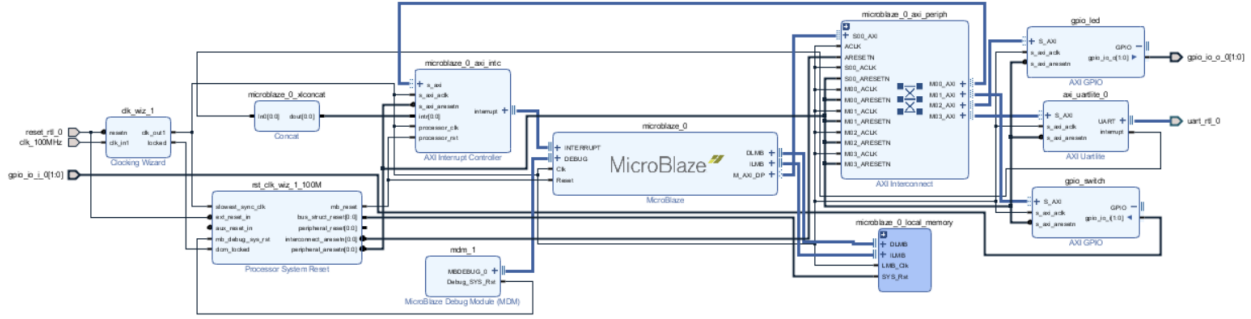
Figure 12: The full block design with a MicroBlaze and three peripherals

still examine it ourselves. Click on the **Address Editor** tab 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 that the gpio led peripheral can be accessed starting from the address 0x4001_0000 from the Data port of the MicroBlaze (see Figure 13). Note, a 64k region was assigned to all of the peripherals, though that size is likely overkill and can be reduced to 4k if you like. The range selected for the BRAMs however is significant, as Vivado uses this range to determine how to size the BRAM inside the block design.

Figure 13: The address map for the data and instruction ports of the MicroBlaze

As a finale step, we must validate the design to ensure all of the connections are made and every peripheral is mapped properly. To do this, press the **Validate Design** button (☑) on the top toolbar in Vivado. If validation is successful, click **OK**, otherwise check the errors and critical warnings to see what's wrong with the design. Save the block design and then close the block design editor view to return to the *Project Manager* view in Vivado.

4 Creating a Wrapper and Adding Constraints

In order to use a block design in our project, we have to create an HDL wrapper which instantiates an instance of our block design. The block design itself cannot be the top level module of the project. In the *Sources* pane, click **Design Sources**. **Right click** your block design and choose **Create HDL Wrapper** to wrap your design. A dialog box pops up with some options for the HDL wrapper; the default options are okay, so simply click **OK**. See Figure 14 for reference. Navigate

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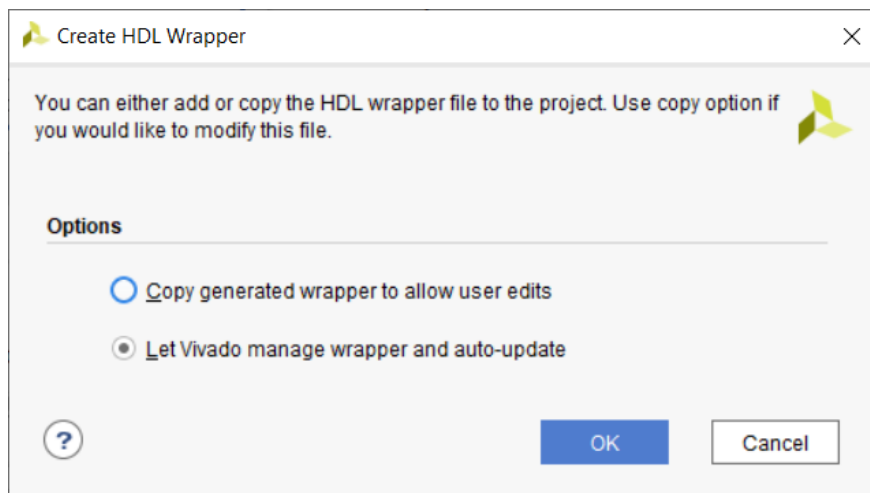

Figure 14: Create HDL wrapper dialog box

Open the newly created HDL wrapper by **double clicking** it in the *Sources* pane. Note that all of the ports that we added to the block design are exposed as top level ports on the verilog wrapper. Within the verilog module, you should see an instance of the block design instantiated. If you want to create your own wrapper, perhaps with some other logic included, this wrapper can be used to determine how to instantiate your block design. The most important feature of the wrapper for the proposes of this tutorial is the name of the top level ports; this is important when creating the constraints file because the names of the ports in the constraints file must match the names in this wrapper.

Open the constraints file you included when you created the project. Verify that the names of all the signals matches the names of the signals in the top level module. As versions of Vivado change, some of these signals may not have the same names as when this tutorial was created. If any of the signal names don't match between the wrapper and the constraints file, change the constraints file to match the verilog wrapper. The contents of the constraints file as of the making of this document are displayed in Figure 15. Now that we have a valid constraints file, we can proceed to build the project, we need to run **Synthesis**, followed by **Implementation**, and the **Generate Bitstream** steps. Note, if you run only the **Generate Bitstream** step, Vivado will run through **Synthesis** and **Implementation** automatically.

```
1    ## Clock signal
2    set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk_100MHz }];
3    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk_100MHz}];
4
5    ## Reset (CPU_RESET)
6    set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports { reset_rtl_0 }];
7
8    ##Switches
9    set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_i_0[0] }];
10   set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_i_0[1] }];
11
12   ## LEDs
13   set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_o_0[0] }];
14   set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { gpio_io_o_0[1] }];
15
16   ##USB-RS232 Interface
17   set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 } [get_ports { uart_rtl_0_rxd }];
18   set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 } [get_ports { uart_rtl_0_txd }];
```

Figure 15: Constraints file for the project

5 Hello World Program in Vivado SDK

After generating the bitstream, click on **File** and choose **Export**, then choose **Export Hardware**. Make sure you check **Include bitstream** (see Figure 16). If the option is gray, make sure to click **Open Implemented Design** under the *Implementation* section of the *Flow Navigator* and repeat the export step. Then, click **File** and select **Launch SDK**, which will launch the Vivado SDK program. Note, Vivado SDK can take some time to load the hardware, so be patient.
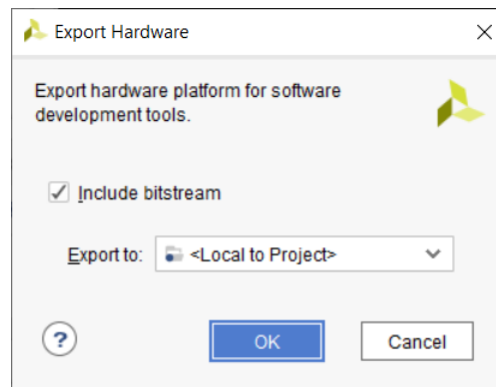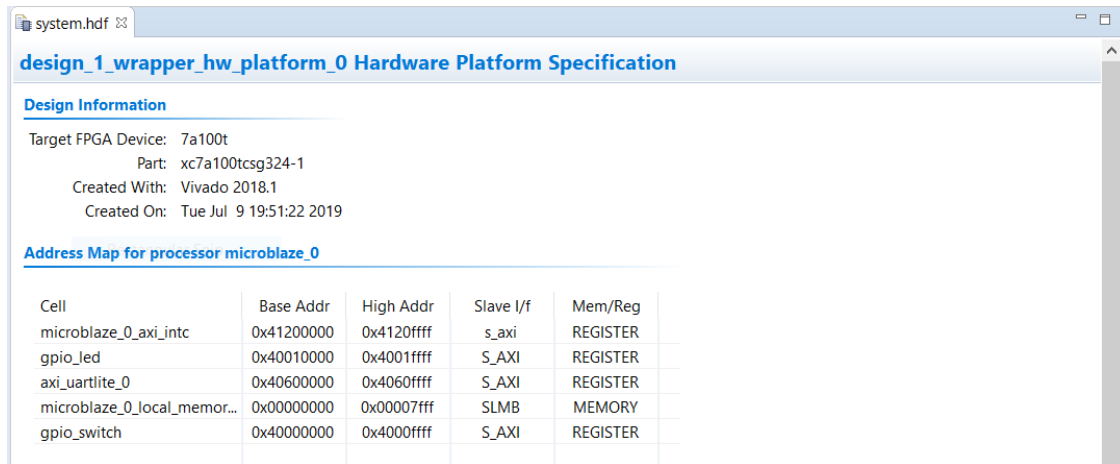

Figure 16: Export hardware with include bitstream checked

After the SDK first launches, double click **system.hdf** to see the information created when you exported the hardware in the previous step. You'll see a description of the peripherals connected to the MicroBlaze processor as well as the memory-map we created using the *Address Editor* (see Figure 17). If there are missing peripherals, or the memory-map doesn't match that which we

11

created in the block design, the hardware must have been exported incorrectly; sometimes this is the case when we update the the block design but forget to re-export the hardware in Vivado.

Figure 17: Export hardware in Vivado SDK

Vivado SDK provides some template programs to help us get started with programming for the MicroBlaze system. We will be creating a *Hello World* application. Select **File** → **New** → **Application Project**. Enter some **Project name** and choose the *OS Platform* to be **standalone**. Click **Next**. Choose **Hello World** as your template and click **Finish**. See Figure 18 for reference.

The created application is opened in the Vivado SDK IDE, with a listing of source and include files on the left hand side. expand both the new application folder, and the folder with ending _ *bsp* to see the folder structure of the application. In the *src* folder of your application directory, you should now see a file called *helloworld.c*, along with files called *platform.c*, *platform.h*, and *platform_ config.h*. The main function is defined in the *helloworld.c* file, so **double click** that file to open it up and view the contents. You should see a program which calls functions *init_ platform()* and *cleanup_ platform()* at the beginning and end of the program, with some functionality included within. These two functions are defined in the *platform.c* file, and call other functions that themselves are defined in the *platform.c* file, or perhaps in some include files from the aforementioned _ *bsp* directory. The _ *bsp* directory, which stands for *Board Support Package*, contains drivers for the peripherals included in your design and some common Xilinx functionalities. You may want to take a look at some of these files if you're curious. These two functions setup the platform for us to use and should always be included in any project.
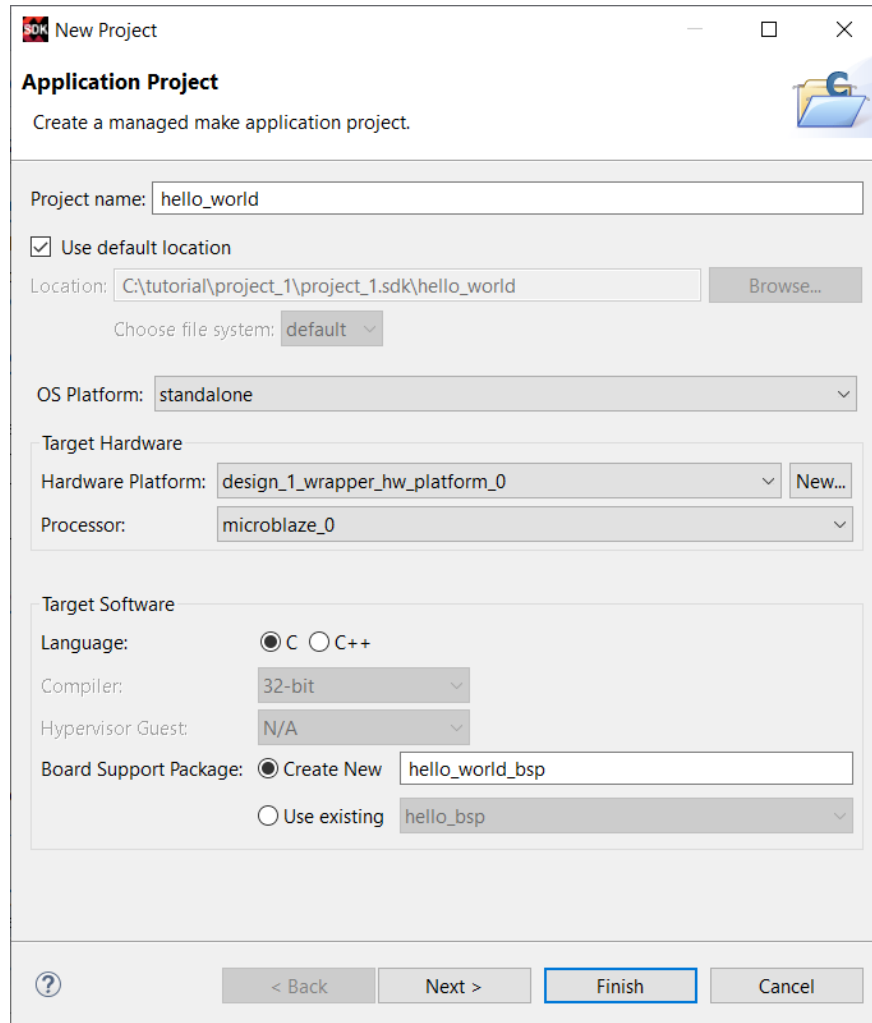
Figure 18: Creating a new application in Vivado SDK

Connect your board to the computer and turn it on. Click **Xilinx Tools → Program FPGA** to program your board with the generated bitstream. If you have multiple boards connected, make sure to select the correct board by clicking **Select...** beside the *Device* field, otherwise the **Auto Detect** option should be fine. The bitstream should be automatically selected in this window. Click **Program** to start programming the board. See Figure 19 for reference. Note, this will program the board with the MicroBlaze system we created.

To run the application on the board, click on **Run → Run Configurations** to create a new run configuration. Choose **Xilinx C/C++ application(GDB)** and then click the **New** button ([]) to create a new launch. You should now see the interface depicted in Figure 20. Open the **Application** tab and click **Browse** next to the *Project Name* entry box, select your newly created hello world program (based on the name you assigned it earlier). You will notice that the application file is automatically selected under the *Debug* folder. In the *Summary* section above, click the **checkbox** in the *Download* column of the application. Click **Apply** to save the run but do not run the application just yet. Click **Close**.
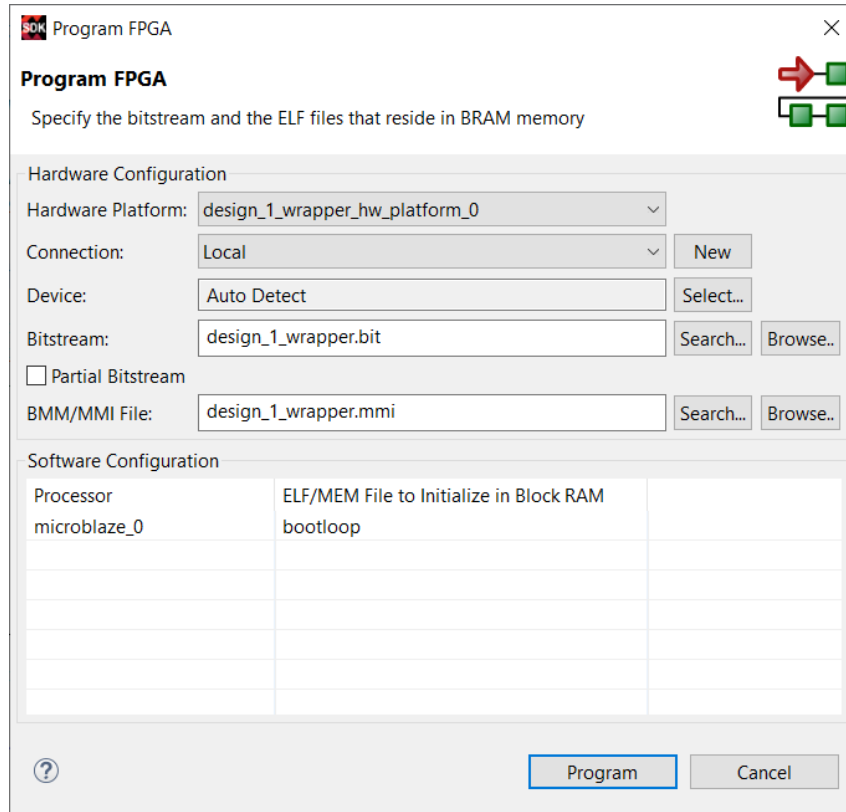
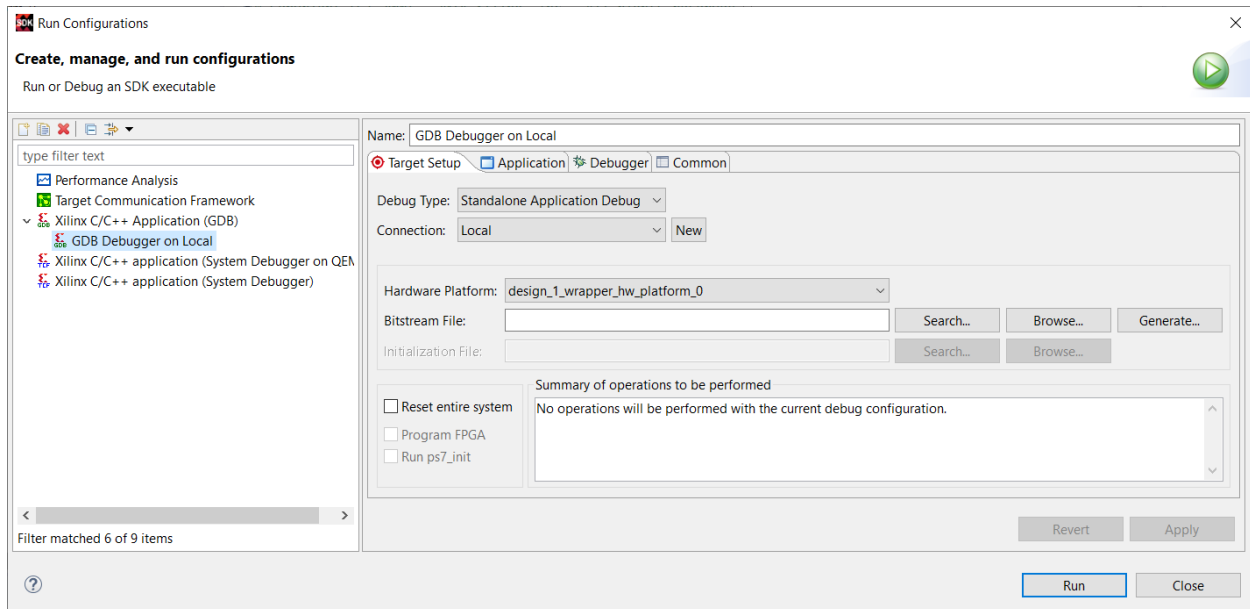Figure 19: The program FPGA dialog in Vivado SDK

Figure 20: Creating a new run in Vivado SDK

To see the output of the print statements in our program, we must connect to the UART port of the board. To do this, select the **SDK Terminal** tab in the console window at the bottom of

Vivado SDK. Now click the **Plus** button () to add a new connection. A dialog will pop up to specify the details of the connection (see Figure 21). On Windows machines, the *Port* field should be a drop down list containing the terminal devices connected to the computer; **select** the device corresponding to the board's UART connection. On Linux machines, you must enter the device file name corresponding to the terminal device in this field (e.g. /dev/ttyUSBx where x is the number corresponding to the connection). For Linux, you must add yourself to the *dialout* group before performing this step. Set the *Baud Rate* to **9600** and click **OK**. If you don't know the port number of the board, you can try different options until you find the correct one, though be careful not to send characters to some other terminal device attached to your computer. If testing different port numbers, remember to disconnect from the previous port before connecting to a new port. Any output from the UART port of the board will be displayed in this *SDK Terminal* tab.
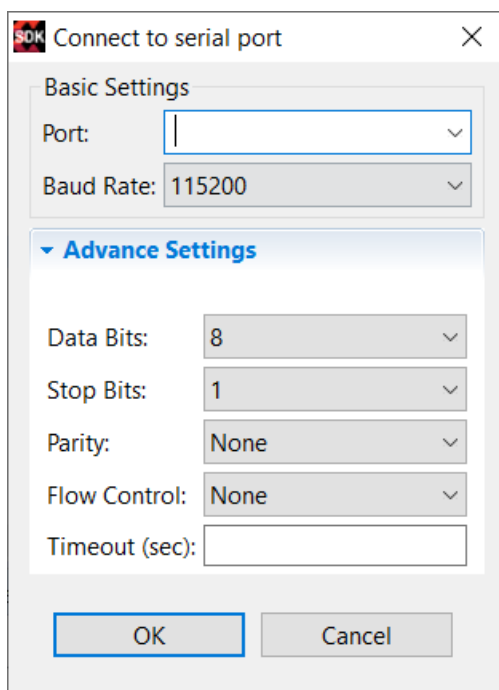

Figure 21: New SDK terminal connection dialog

Re-open the **Run Configurations** window and select the run you recently created listed under *Xilinx C/C++ application(GDB)*. Click **Run** to run the application. If the board is found successfully, there should be no errors. Check the *SDK Terminal* tab to ensure that the *Hello World* text was printed correctly.

6   Program to Access the GPIOs

Open the *helloworld.c* file and modify its contents to match that in Figure 22. This program defines two volatile global pointers to the LED and switch GPIO addresses; accesses to these addresses will access the registers of the AXI GPIO cores in our block design. The macros *XPAR_GPIO_LED_BASEADDR* and *XPAR_GPIO_SWITCH_BASEADDR* are defined in a file called *xparameters.h*, which is found in the *include* folder of the board support package directory. Make sure to check that file to ensure your macros have the same name (if they differ, please change them in the modified *helloworld.c* program to match those given in *xparameters.h*). In this

application, 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.

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"

volatile unsigned int* led = (unsigned int*) XPAR_GPIO_LED_BASEADD;
volatile unsigned int* swt = (unsigned int*) XPAR_GPIO_SWITCH_BASEADD;

int main()
{
    init_platform();

    print("Hello World\n\r");

    while(1)
    {
        *led = *swt;
    }

    cleanup_platform();
    return 0;
}
```

Figure 22: Modified Hello World application using GPIOs

Re-open the **Run Configurations** window and select the run you recently created listed under *Xilinx C/C++ application(GDB)*. Click **Run** to run the application. Try flipping the switches on the board and see what happens to the LEDs. Note, while it may seem that the changes to the LED is instantaneous, in actuality there is some time between changes to the switches and the corresponding changes to the LED values. This delay isn't perceivable since the MicroBlaze is running at 100MHz.

To see this delay, we need to insert some breakpoints. Click **Run → Debug Configurations** to bring up the debug configurations. The run configuration we created earlier should already exist as a debug configuration in this view, **select it** on the left side and click **Debug**. A prompt will come up telling you that Vivado SDK will open the *Debug perspective*, click **Yes**. The window view that appears is the gdb debugger view in Vivado SDK (see Figure 23). Variables and Breakpoints are listed in the pane on the right side, and the program contents and outline are shown in the lower portion of the figure.
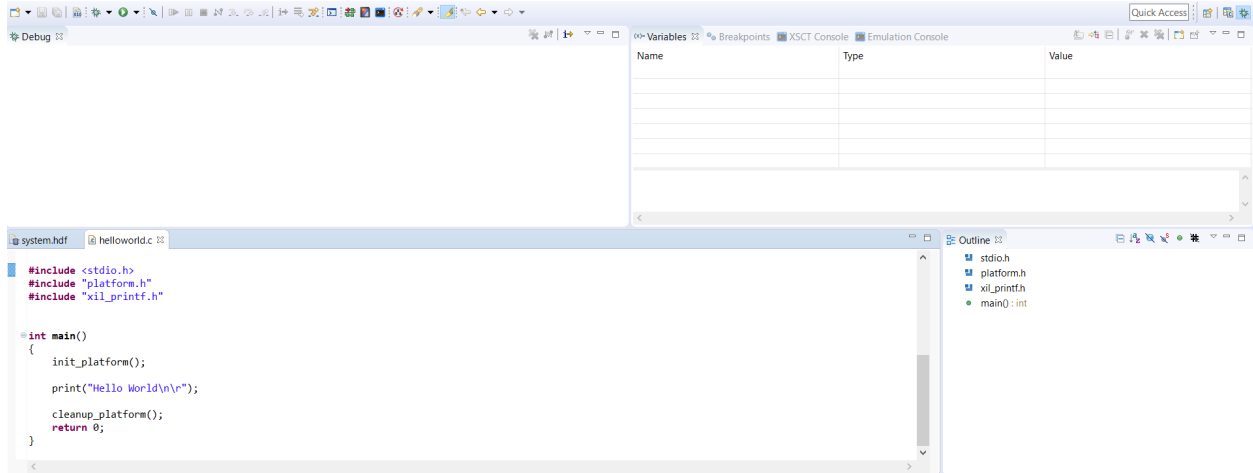
Figure 23: GDB debugging view in Vivado SDK

You can add breakpoints by **double clicking** to the left of the line of code before which you'd like the program to stop, i.e. in the margins of the code. Insert a breakpoint on line 17 (refer to Figure 22). Press the **Resume** button (▮▶) to advance the program to the next break (note, if the program didn't break at the beginning of the main function by default, you will have to stop the program first by pressing the **Stop** button ■). Change the switch values with the program stopped and observer that the LEDs do not change correspondingly. Click **Resume** again to see the LED values update again. We can thus confirm that the LED values are indeed being changed by the software running on the MicroBlaze. The GDB debugger is a useful tool to help debug your applications running on the MicroBlaze system.

7 Summary

The Vivado IP Integrator flow can be used to easily create complex hardware systems, including soft-processor systems with MicroBlaze processors. The project was created with a block design, and a full MicroBlaze system was built in that block design with a number of Xilinx IP cores, including three peripherals. The system was built and a bitstream was generated. The Hardware was exported and Vivado SDK was launched, reading the exported hardware. Example *Hello World* and *GPIO* applications were made and run on the MicroBlaze. The GDB debugger was also used to insert breakpoints into the software and step through the code.